HP COBOL Manual for TNS/E Programs

HP Part Number: 520347-006
Published: August 2010
Edition: H06.08 and subsequent H-series RVUs and J06.03 and subsequent J-series RVUs

Legal Notice

Table of Contents

40 Data References....1037

41 Identification Division....1041

42 Environment Division....1043

43 Data Division....1061

44 Procedure Division....1073

##

List of Figures

List of Tables

List of Examples

1 About This Document

This publication describes the HP implementation of the 1985 version, and portions of the 2002 version, of the COBOL language. It includes information on the ECOBOL compiler (T0356) and program execution, task-oriented information to help an experienced COBOL programmer use HP COBOL for NonStop™ systems, and summaries of compiler and run-time error messages.

Supported Release Version Updates (RVUs)

This manual supports H06.08 and all subsequent H-series RVUs and J06.03 and subsequent J-series RVUs until otherwise indicated in a replacement publication.

Intended Audience

This manual is a complete reference book for experienced COBOL programmers who want to use HP COBOL for NonStop systems (HP COBOL). It explains every feature of the HP COBOL programming language. To understand its explanations, you must have some experience with structured (1974 or 1985) COBOL. If you do not, read an introductory text first.

Acknowledgment

COBOL is an industry language and is not the property of any company or group of companies, or of any organization or group of organizations.

No warranty, expressed or implied, is made by any contributor or by the CODASYL COBOL Committee as to the accuracy and functioning of the programming system and language. Moreover, no responsibility is assumed by any contributor, or by the committee, in connection herewith.

The authors and copyright holders of the copyrighted material used herein

FLOW-MATIC (trademark of Sperry Rand Corporation), Programming for the UNIVAC (R) I and II, Data Automation Systems copyrighted 1958, 1959, by Sperry Rand Corporation; IBM Commercial Translator Form No. F 28-8013, copyrighted 1959 by IBM; FACT, DSI 27A5260-2760, copyrighted 1960 by Minneapolis-Honeywell

have specifically authorized the use of this material in whole or in part, in the COBOL specifications. Such authorization extends to the reproduction and use of COBOL specifications in programming manuals or similar publications.

New and Changed Information

Changes to this manual are itemized for each RVU.

New and Changed Information for 520347–006

- Added a note to the REPORT Phrase under the RECEIVE-CONTROL Paragraph (page 155) in Chapter 7: Environment Division (page 113), indicating that system messages need to be explicitly requested by executing appropriate Guardian procedures in addition to specifying certain classes of system messages in the REPORT clause. In addition, changed the reference cited for system messages from the *Guardian Programmer's Guide* to the *Guardian Procedure Errors and Messages Manual*.
- Revised descriptions of CHECK levels in Table 12-14 (page 550) in Chapter 12 (page 521) and under "CHECK" (page 1001) in Chapter 37 (page 999).
- Added a new section, "Non-Local Jumps in HP COBOL Applications" (page 660), that describes `cobsetjmp()` and `coblongjmp()`, HP COBOL-safe versions of the standard HP C functions `setjmp()` and `longjmp()`. Like `setjmp()` and `longjmp()`, `cobsetjmp()`

and `coblongjmp()` provide the functionality for non-local jumps and are typically used for handling errors and interrupts encountered during program execution.

- Added description of message 149 to Chapter 49 (page 1193).

New and Changed Information for 520347–005

- Added a caution under Chapter 16: Debugging Tools (page 707) on the manner in which the CODECOV (page 551) compiler directive interacts when you are debugging an instrumented application. Also see Generating Instrumented Object Code for Use With the Code Coverage Tool (page 534). Under that section and CODECOV (page 551), placed a reference to the caution in the Debugging section.
- Added new intrinsic functions, TEST-NUMVAL and TEST-NUMVAL-C, to Chapter 15: Intrinsic Functions (page 663). The TEST-NUMVAL Function (page 701) and the TEST-NUMVAL-C Function (page 702) enable testing inputs to the NUMVAL Function (page 687) and the NUMVAL-C Function (page 688), respectively, to validate input strings for correct formation without risking the premature termination of the program.

  Added TEST-NUMVAL and TEST-NUMVAL-C to the table of Integer Intrinsic Functions (page 664).

- Under the RECEIVE-CONTROL Paragraph (page 155) in Chapter 7: Environment Division (page 113), indicated that the upper limit is 2MB for the *`reply-length`* of the REPLY CONTAINS phrase.
- Under the RECORD CONTAINS Clause (page 175) in Chapter 8: Data Division (page 165), expanded the range definition for the *`length-fixed`*, *`length-min`*, and *`length-max`* options of *`contains-phrase-fixed`*.
- Under Size (page 737) in Chapter 21: HP COBOL Limits (page 737), changed the table under Record or table to show “2 MB” for “File assigned to $RECEIVE” and “2 MB for files assigned to $RECEIVE; 32,767 for other files” for “File Section.”
- Added a new item to the Summary of $RECEIVE Rules (page 942) in Chapter 32: Process Initiation, Communication, and Management (page 933).

New and Changed Information for 520347–004

The November 2006 edition of the manual, 520347–004, contains these changes:

- COBOL 2002 enhancements:
  - STANDARD directive and -Wstandard option
  - MIGRATION-CHECK directive and -Wmigration_check option
  - ALLOCATE statement
  - Changes to SET statement
  - FREE statement
  - BASED clause
  - Additional compiler warning and error messages
  - GLOBALIZED directive and -Wglobalized option for improved performance
- CODECOV directive and -Wcodecov option to generate object code for use with the Code Coverage Tool
- Native Inspect support for COBOL programs
- OBJEXTENT directive to allow for larger object files

Document Organization

Table 1-1 Summary of Contents

Chapter	Description
Chapter 2: Introduction (page 49)	Introduces the HP COBOL for NonStop systems (HP COBOL) language and explains how to use it to create TNS/E native processes.
Chapter 3: Source Program Organization and Format (page 53)	Describes how to organize and format a source program into a reference format that the compiler accepts.
Chapter 4: Language Elements (page 69)	Describes the COBOL character set, punctuation characters, and character-strings.
Chapter 5: Data Fundamentals (page 87)	Describes data levels, classes, and categories and how they are organized into data structures and organized, stored, and executed.
Chapter 6: Identification Division (page 107)	Describes the Identification Division that is required in a COBOL program and which contains a PROGRAM-ID paragraph and a DATE-COMPILED paragraph.
Chapter 7: Environment Division (page 113)	Describes the Environment Division that is optional in a COBOL program and which contains a Configuration Section and an Input-Output Section.
Chapter 8: Data Division (page 165)	Describes the Data Division that is optional in a COBOL program. It contains the File Section, the Working-Storage Section, the Extended-Storage Section, and the Linkage Section.
Chapter 9: Procedure Division (page 237)	Describes the optional Procedure Division, but a program without a Procedure Division does nothing except initialize data. The Procedure Division is composed of statements, which specify the actions to be taken by the program.
Chapter 10: Procedure Division Verbs (page 289)	Describes the COBOL verbs that you can use in the Procedure Division, in alphabetic order.
Chapter 11: Source Text Manipulation (page 507)	Describes source manipulation, which comprises the COPY statement, COPY libraries, and the REPLACE statement.
Chapter 12: Program Compilation (page 521)	Describes program compilation including compiler directives that specify the source format, control listing features, control selective compilation of portions of the source code, and request compilation options.
Chapter 13: Program Execution (page 589)	Describes how to run programs in the Guardian environment.
Chapter 14: Libraries and Utility Routines (page 607)	Describes utility routines in dynamic-link libraries (DLLs) named ZCOBDLL and ZCREDLL. (You can also create a user library for an HP COBOL program.)
Chapter 15: Intrinsic Functions (page 663)	Describes intrinsic functions that your program can use, but does not need to declare. An intrinsic function returns a value that is computed at the time of reference during the execution of the object program.
Chapter 16: Debugging Tools (page 707)	Briefly describes HP debugging tools and refers you to appropriate sources for more information.
Chapter 17: ANSI Reference Format (page 711)	Describes ANSI reference format, in which each line has 80 characters (columns). Five margins divide each line into five areas.
Chapter 18: HP Extensions to ISO COBOL (page 715)	Lists the HP extensions to ISO/ANSI COBOL, grouping them according to the sections of this manual that explain them.
Chapter 19: HP COBOL CRE Support (page 719)	Describes the Common Run-Time Environment (CRE), a set of services that supports mixed-language programs.
Chapter 20: Using HP COBOL in the OSS Environment (page 721)	Describes how to run programs in the OSS environment.

Table 1-1 Summary of Contents *(continued)*

Chapter	Description
Chapter 21: HP COBOL Limits (page 737)	Describes the kinds of limits the HP architecture and the method of implementation of the HP COBOL compiler, ECOBOL, imposes on HP COBOL programs.
Chapter 22: Reserved Words (page 743)	Contains alphabetized lists of all reserved words and HP reserved words.
Chapter 23: Creating and Compiling HP COBOL Source Programs (page 757)	Describes various methods of creating and compiling HP COBOL source programs for the Guardian environment, primarily.
Chapter 24: Calling Other Programs and Routines (page 807)	Describes mixed-language programs, which are HP COBOL program that call one or more non-COBOL routines and non-COBOL programs that call one or more COBOL programs. HP COBOL programs can call programs compiled by TNS/E C, TNS/E C++, and EpTAL.
Chapter 25: Issues Related to Linking (page 825)	Describes issues related to linking, mainly in the Guardian environment, and includes topics such as concepts and terms, linking HP COBOL programs, and COBOL segmentation.
Chapter 26: Executing and Debugging HP COBOL Programs (page 835)	Describes how to debug HP COBOL programs that are linked and bound.
Chapter 27: Input and Output Concepts (page 847)	Describes how HP COBOL handles Input and output. The typical batch HP COBOL program obtains data (input), manipulates it, and produces a report (output). The typical interactive HP COBOL program is a server that obtains a request, collects data (input), and sends a reply (output).
Chapter 28: Tape Input and Output (page 855)	Describes how HP COBOL batch programs read or write files that are on magnetic tape (tape files).
Chapter 29: Disk Input and Output (page 873)	Describes how HP COBOL batch programs and transaction-supporting servers use disk files.
Chapter 30: Terminal Input and Output (page 907)	Describes how an application on an HP system uses Pathway/TS to write requester programs in SCREEN COBOL to handle terminal input and output and uses HP COBOL or pTAL server programs to handle database input and output.
Chapter 31: Printer and Spooler Output (page 917)	Describes how a process can write a report to paper: it can write directly to a printer or it can write to a spooler collector.
Chapter 32: Process Initiation, Communication, and Management (page 933)	Describes how to initiate, communicate with, and manage processes, which are unique executing entities created when someone runs object code from a loadfile.
Chapter 33: Fault-Tolerant Processes (page 963)	Describes fault-tolerant processes. A process runs in a fault-tolerant manner when no single point of failure can stop the process or corrupt its data or the files it is manipulating. Processes are not automatically fault tolerant—they must be designed and implemented to be fault tolerant.
Chapter 34: Migrating TNS/R Programs to TNS/E Programs (page 977)	Describes how to migrate TNS/R HP COBOL programs to TNS/E HP COBOL programs.
Chapter 35: Native COBOL Cross Compiler on PC (page 983)	Describes the native COBOL cross compiler, which is an optional cross compiler that runs on PC platforms.
Chapter 36: Commands (page 991)	Describes how to run commands in the Guardian and OSS environments and the ecobol PC command line.
Chapter 37: Compiler Directives (page 999)	Lists and describes compiler directives.
Chapter 38: Source Program Organization and Format (page 1023)	Describes the syntax of the separately compiled source program and the ANSI and TANDEM reference formats.

Table 1-1 Summary of Contents *(continued)*

Chapter	Description
Chapter 39: Language Elements and Expressions (page 1027)	Describes language elements and expressions, including the COBOL character set, character strings, arithmetic expressions, conditional expressions, and concatenation expressions.
Chapter 40: Data References (page 1037)	Describes the syntax of data references.
Chapter 41: Identification Division (page 1041)	Describes the syntax of the identification division.
Chapter 42: Environment Division (page 1043)	Describes the syntax of the environment division.
Chapter 43: Data Division (page 1061)	Describes the syntax of the data division.
Chapter 44: Procedure Division (page 1073)	Describes the syntax of the procedure division.
Chapter 45: Intrinsic Function Calls (page 1117)	Lists and describes intrinsic function calls.
Chapter 46: ZCOBDLL Routine Calls (page 1125)	Lists and describes ZCOBDLL routine calls.
Chapter 47: ZCREDLL Routine Calls (page 1131)	See the CRE Programmer's Guide for ZCREDLL routine calls.
Chapter 48: Compiler Diagnostic Messages (page 1133)	Describes compiler diagnostic messages, including warning, error, and failure characteristics.
Chapter 49: Run-Time Diagnostic Messages (page 1193)	Describes the run-time diagnostic messages that can be reported if an HP COBOL process encounters an error condition.
Appendix A: ASCII Character Set (page 1229)	Contains two tables of the ASCII character set.
Appendix B: Data Type Correspondence (page 1239)	Shows tables that contain the return value size generated by HP language compilers for each data type.

Notation Conventions

COBOL Conventions

This manual uses these conventions throughout:

- "Compiler" means the ECOBOL compiler unless otherwise stated
- "Linker" means the `eld` utility unless otherwise stated
- COBOL Name Conventions
- Syntax Diagram Conventions
- Range Convention
- Example Conventions
- Change Bar Notation

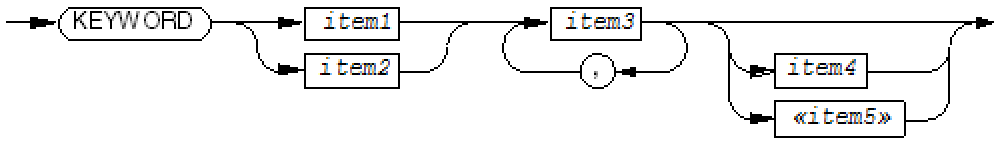
COBOL Name Conventions

This manual uses these COBOL names, and this is what they mean:

Name	Meaning
COBOL	The implementation of the 1985 ISO/ANSI Standard COBOL language
HP COBOL	HP COBOL for NonStop systems, the HP implementation of COBOL with HP extensions
ECOBOL	The HP compiler for the HP COBOL language that produces native TNS/E objects

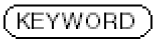
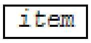
Syntax Diagram Conventions

This manual presents syntax in railroad diagrams. Here is a generic railroad diagram:



VST406.vsd

To use a railroad diagram, follow the direction of the arrows and specify syntactic items as indicated by the diagram pieces:

Diagram Piece	Meaning
 VST412.vsd	Type KEYWORD as shown. You can type letters in uppercase or lowercase.
 VST413.vsd	Replace *item* with a value that fits its description, which follows the syntax diagram.
 VST414.vsd	Type content (punctuation mark, symbol, or letter) as shown. You can type a letter in uppercase or lowercase.
«*item*»	The 1985 COBOL standard classifies *item* as obsolete, so you are advised not to use it. (An obsolete keyword is marked the same way; that is, «KEYWORD».)

Some examples of the meanings of simple diagrams are:

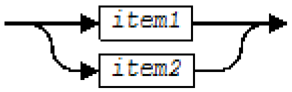
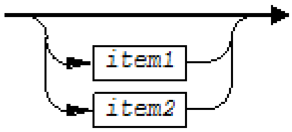
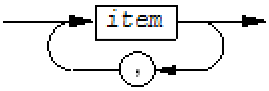
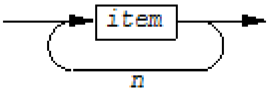
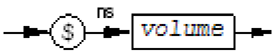
Diagram Piece	Meaning
 VST407.vsd	Choose *item1* or *item2*.
 VST408.vsd	Choose *item1*, *item2*, or neither.

Diagram Piece	Meaning
 VST409.vsd	Specify *item* one or more times, separating occurrences with commas.
 VST742.vsd	Specify *item* at most *n* times.

NOTE: To refer to a particular railroad diagram or figure when giving feedback to HP, use the number at the bottom right corner of that railroad diagram or figure (for example, VST742.vsd).

Spacing rules are:

- If the arrow between two diagram pieces is labelled “ns,” put no spaces between the syntactic items that they represent. For example:



VST420.vsd

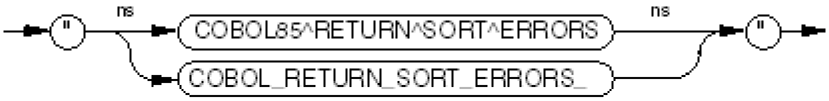
means that you type:

```
$NEWVOL
```

not

```
$ NEWVOL
```

- An “ns” on the top line of a choice structure applies to the lower lines in the choice structure as well. For example:

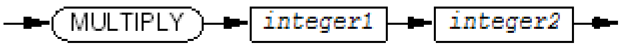


VST635.vsd

means that you type one of:

```
"COBOL85^RETURN^SORT^ERRORS"
"COBOL_RETURN_SORT_ERRORS_"
```

- If two diagram pieces are not separated by a separator character (such as a comma, semicolon, or parenthesis), separate the syntactic items that they represent by at least one space or a new line. For example:



VST410.vsd

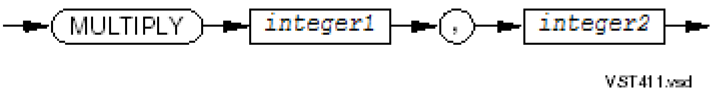
means that you type:

```
MULTIPLY 3 4
```

not

```
MULTIPLY34
```

- If two diagram pieces are separated by a separator character, separating the syntactic items that they represent by spaces is optional. For example:



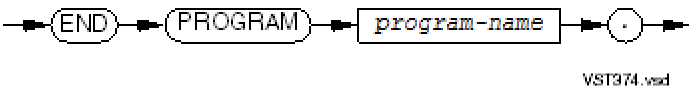
means that you type:

```
MULTIPLY 3,4
```

or

```
MULTIPLY 3, 4
```

- If a diagram piece is immediately followed by a period, putting spaces between the syntactic item and the period is optional. For example:



means that you can type:

```
END PROGRAM SORT.
```

or

```
END PROGRAM SORT .
```

- Explicit spacing rules given for individual railroad diagrams override the aforementioned rules.

NOTE: Except in literals and PICTURE clauses, HP COBOL treats the comma (,) and semicolon (;) as equivalents.

Range Convention

Ranges include their endpoints unless otherwise noted. For example, “x is in the range from 0 through 32,767” means that x is greater than or equal to 0 and less than or equal to 32,767.

Example Conventions

In examples, a modified ellipsis (...) indicates an omission. The code in Example 1-1 can be abbreviated as shown in Example 1-2.

Example 1-1 Code Example Without Ellipsis (...)

```
WORKING-STORAGE SECTION.

 01   MONTH-NAME-TABLE.
      05   FILLER   PICTURE X(9)  VALUE "January".
      05   FILLER   PICTURE X(9)  VALUE "February".
      05   FILLER   PICTURE X(9)  VALUE "March".
      05   FILLER   PICTURE X(9)  VALUE "April".
      05   FILLER   PICTURE X(9)  VALUE "May".
      05   FILLER   PICTURE X(9)  VALUE "June".
      05   FILLER   PICTURE X(9)  VALUE "July".
      05   FILLER   PICTURE X(9)  VALUE "August".
      05   FILLER   PICTURE X(9)  VALUE "September".
      05   FILLER   PICTURE X(9)  VALUE "October".
      05   FILLER   PICTURE X(9)  VALUE "November".
      05   FILLER   PICTURE X(9)  VALUE "December".
 01   MONTH-NAMES REDEFINES MONTH-NAME-TABLE.
      05   MONTH-NAME  OCCURS 12 TIMES   PICTURE X(9).
```

Example 1-2 Code Example With Ellipsis

```
WORKING-STORAGE SECTION.
 01   MONTH-NAME-TABLE.
      05   FILLER   PICTURE X(9)  VALUE "January".
      05   FILLER   PICTURE X(9)  VALUE "February".
      ...
      05   FILLER   PICTURE X(9)  VALUE "December".
 01   MONTH-NAMES REDEFINES MONTH-NAME-TABLE.
      05   MONTH-NAME  OCCURS 12 TIMES   PICTURE X(9).
```

In examples with user input, user input is shown in bold type and it is assumed that the user presses Return after typing the input. For instance, in the example:

```
ENTER RUN CODE
?123
CODE RECEIVED:   123.00
```

the system displays `ENTER RUN CODE` on one line and prompts the user with a question mark (`?`) on the next line, the user types `123` and presses Return, and the system displays `CODE RECEIVED: 123.00`.

Change Bar Notation

A change bar (as shown to the right of this paragraph) indicates a substantive difference between this edition of the manual and the preceding edition. Change bars highlight new or revised information.

Manuals to Which This Manual Refers

Table 1-2 HP Manuals to Which This Manual Refers

Manual	Description
C/C++ Programmer's Guide	Contains information you need about HP C for NonStop systems if you plan to call HP C routines from HP COBOL.
COBOL Manual for TNS and TNS/R Programs	Describes and explains how to use the COBOL85 and NMCOBOL compilers.
Code Coverage Tool Reference Manual	Describes and explains how to use the Code Coverage utilities to generate code coverage reports.

Table 1-2 HP Manuals to Which This Manual Refers *(continued)*

Data Definition Language (DDL) Reference Manual	Describes the Data Definition Language (DDL), with which you can create COPY and SOURCE libraries for your HP COBOL program.
DLL Programmer's Guide for TNS/E Systems	Explains position-independent code (PIC) and dynamic-link libraries (DLLs).
EDIT User's Guide and Reference Manual	Explains how to create and modify source files.
eld Manual	Explains how to use the `eld` utility to link and change the attributes of TNS/E object files.
ENABLE User's Guide	Explains how to use the ENABLE utility to build a file-maintenance application that allows you to modify a disk file.
ENFORM User's Guide	Explains how to use the ENFORM language to query an online database.
enoft Manual	Explains how to use the `enoft` utility to display TNS/E object files.
Enscribe Programmer's Guide	Explains how to use queue files.
Expand Network Management and Troubleshooting Guide	Describes the Expand network, which you must understand if your system is connected to other systems by this means and you want to create partitioned disk files.
FastSort Manual	Describes the FastSort interface routines that give your HP COBOL program access to the utility program FastSort.
File Utility Program (FUP) Reference Manual	Describes the Guardian File Utility Program (FUP), which you can use on files that you use with the ECOBOL compiler.
Guardian Native C Library Calls Reference Manual	Contains information you need about HP C for NonStop systems if you plan to call HP C routines from HP COBOL programs.
Guardian Procedure Calls Reference Manual	Describes Guardian file name syntax and the syntax and programming considerations for using system procedures.
Guardian Procedure Errors and Messages Manual	Describes error codes, error lists, system messages, and trap numbers for system procedures.
Guardian Programmer's Guide	Explains how to use the programmatic interface of the operating system.
Guardian User's Guide	Explains how to run loadfiles (the RUN command) and how to create and modify indexed (key-sequenced) files.
H-Series Application Migration Guide	Explains how to migrate TNS/R programs to TNS/E programs.
Kernel-Managed Swap Facility (KMSF) Manual	Explains how to use the Kernel-Managed Swap Facility (KMSF) to configure and manage swap volumes.
NetBatch Manual	Explains how to use the NetBatch job management system, which is useful for launching compilations to run unattended.
Open System Services Library Calls Reference Manual	Contains information you need if you plan to call HP C routines from HP COBOL in the OSS environment.

Table 1-2 HP Manuals to Which This Manual Refers *(continued)*

Open System Services Programmer's Guide	Describes how to write applications in HP C for the HP NonStop Open System Services (OSS) environment, focusing on how OSS differs from a standard UNIX programming environment (intended for application and system programmers).
Open System Services Shell and Utilities Reference Manual	Describes the syntax and semantics of each command that you can enter interactively to access the OSS command interpreter (the OSS shell), and utilities and other functions that perform general-purpose and program-development operations (intended for all audiences).
Open System Services System Calls Reference Manual	Describes all OSS system calls (file system and kernel functions), their syntax, required external data structures, resulting operations, and source file (header) locations for all literals and symbolic definitions (intended for system and application programmers).
Pathway/TS SCREEN COBOL Reference Manual	Describes the SCREEN COBOL language (HP COBOL is often used to implement servers that works with SCREEN COBOL (TCP) requesters to make up a Pathway/TS application).
Pathway/TS TCP and Terminal Programming Guide	Explains how to write SCREEN COBOL requesters in the Pathway environment.
PS TEXT EDIT and PS TEXT FORMAT User's Guide	Explains how to create and modify source files.
PS TEXT EDIT Reference Manual	Describes the PS TEXT EDIT editor, which you can use to create and modify source files.
pTAL Conversion Guide	Explains how to convert TAL programs to pTAL, which you must do to any TAL programs that your TNS/E HP COBOL programs call.
pTAL Reference Manual	Describes the pTAL language, one of the TNS/E languages in which you can rewrite FORTRAN or Pascal programs that you want your TNS/E HP COBOL programs to call.
Spooler Programmer's Guide	Explains how to use the spooler utility that collects and routes printer output.
Spooler Utilities Reference Manual	Explains spoolers, which can be given Guardian file names and used with HP COBOL programs.
SQL/MP Programming Manual for COBOL	Describes the programmatic interface to HP NonStop SQL/MP for COBOL. You need it if your HP COBOL program contains embedded SQL/MP statements.
SQL/MP Reference Manual	Describes HP NonStop SQL/MP. You might need it if your HP COBOL program contains embedded SQL/MP statements.
SQL/MX Programming Manual for C and COBOL	Describes the programmatic interface to HP NonStop SQL/MX for HP C and HP COBOL. You need it if your HP COBOL program contains embedded SQL/MX statements.
TACL Reference Manual	Describes the HP Tandem Advanced Command Language (TACL), which you need to execute HP COBOL programs from a TACL prompt.
TNS/E Native Application Conversion Guide	Introduces the TNS/E native development and execution environments and explains how to convert exising TNS programs to TNS/E native applications.

Table 1-2 HP Manuals to Which This Manual Refers *(continued)*

TS/MP Pathsend and Server Programming Manual	Explains how to write servers and Pathsend requesters in the Pathway environment
TS/MP System Management Manual	Explains the configuration and operation of servers in the Pathway environment (HP COBOL is often used to implement servers that function as parts of a Pathway applications, using the HP NonStop TS/MP software)

Publishing History

Part Number	Product Version	Publication Date
520347-003	ECOBOL H01 HP Enterprise Toolkit—NonStop Edition (ETK) PC COBOL H01	July 2005
520347-004	ECOBOL H01 HP Enterprise Toolkit—NonStop Edition (ETK) PC COBOL H01	November 2006
520347-005	ECOBOL H01 HP Enterprise Toolkit—NonStop Edition (ETK) PC COBOL H01	August 2009

HP Encourages Your Comments

HP encourages your comments concerning this document. We are committed to providing documentation that meets your needs. Send any errors found, suggestions for improvement, or compliments to `docsfeedback@hp.com`.

Include the document title, part number, and any comment, error found, or suggestion for improvement you have concerning this document.

2 Introduction

This manual describes the HP COBOL for NonStop systems (HP COBOL) language and explains how to use it to create TNS/E native processes. (The *COBOL Manual for TNS and TNS/R Programs* explains how to use the HP COBOL language to create TNS processes and TNS/R native processes.)

HP COBOL Language

The HP COBOL language, the implementation of the COBOL language by the Hewlett-Packard Company, conforms to the standard published by the International Organization for Standardization (ISO) and the American National Standards Institute (ANSI) in 1985. HP COBOL includes a subset of features from the 2002 revision to the ISO standard (ISO/IEC 1989:2002). HP COBOL also provides extensions to the language that enable COBOL programs to act as servers, as in a Pathway application (which uses TS/MP), and to be fault tolerant.

Standard COBOL 1985 is described in *American National Standard for Information Systems Programming Language—COBOL*, which has ANSI document number X3.23-1985 and ISO document number ISO 1989-1985, and the intrinsic function and correction amendments, ANSI X3.23a-1989 and ANSI X3.23b-1993, ISO 1989:1985/Amd.1:1992 and ISO 1989:1985/Amd.2:1994. Standard COBOL 2002 is described in *Information Technology—Programming Languages—COBOL*, which has ISO document number ISO/IEC 1989:2002. HP offers high-level standard COBOL 1985, portions of standard COBOL 2002, and HP extensions.

Table 2-1 Required HP COBOL Modules and Their Levels

Module	Level
Nucleus	2
Sequential I-O	2
Relative I-O	2
Indexed I-O	2
Inter-Program Communication	2
Sort-Merge	1
Source Text Manipulation	2

Table 2-2 Optional HP COBOL Modules and Their Levels

Module	Level
Segmentation*	2 (partial)
Debug*	1
Intrinsic Function	1
Report Writer	Null
Communication	Null

* The 1985 COBOL standard classifies this module as obsolete, so you are advised not to use it.

HP COBOL accepts the syntax for statements in all of these areas; however, certain statements do not apply to NonStop system environments. In such cases, HP COBOL treats these statements as comments after checking that the statements are syntactically correct.

Embedded SQL/MP

The HP COBOL embedded SQL/MP implementation conforms fully to the ANSI Database—Embedded NonStop SQL Standard (ANSI X3.168-1989), with the restrictions and extensions mentioned in *SQL/MP Programming Manual for COBOL*.

SQL/MP is compatible with every HP COBOL compiler (see Table 2-3).

Embedded SQL/MX

The HP COBOL embedded SQL/MX implementation conforms to SQL:1999 standards as described in *SQL/MX Reference Manual* with the restrictions and extensions mentioned in *SQL/MX Programming Manual for C and COBOL*.

SQL/MX is compatible with every HP COBOL compiler except COBOL85 (see Table 2-3).

Summary of Execution Modes

TNS/E systems support three execution modes:

TNS Mode	Accelerated Mode	TNS/E Native Mode
Programs are generated by TNS compilers.	Programs are generated by TNS compilers and then processed by the TNS Object Code Accelerator (OCA).	Programs are generated by TNS/E native compilers.
Programs use TNS process and memory architecture.	Programs use TNS process and memory architecture.	Programs use TNS/E native process and memory architecture.
Programs consist of TNS object code (TNS instructions).	Programs consist of TNS object code (TNS instructions) and accelerated object code (equivalent OCA-generated Itanium instructions).	Programs consist of TNS/E native object code (Itanium instructions).
Millicode routines implement the TNS instructions on Itanium processors.	Programs execute Itanium instructions directly on Itanium processors and use TNS mode to execute instructions for which the OCA could not generate equivalent Itanium instructions.	Programs execute Itanium instructions directly on Itanium processors.

Because of architectural differences between execution modes, you cannot mix TNS object code, accelerated object code, and native object code in one loadfile. A native program can contain only native object code.

HP COBOL Compilers for TNS/E Programs

For HP COBOL source programs, HP provides the compilers in Table 2-3. In this manual, "ECOBOL compiler" means both of these compllers unless otherwise stated.

Table 2-3 HP COBOL TSN/E Compilers (T Number 0356)

Compiler	Description
ECOBOL	Produces native TNS/E object code, which is position-independent code (PIC).
Native COBOL Cross Compiler for TNS/E Programs on PC	Enables you to build native TNS/E objects or NonStop systems applications on a PC

Guardian and OSS Environments

The NonStop operating system offers two operating environments, the Guardian environment and the OSS environment.

The OSS utility `ecobol` calls the ECOBOL compiler.

The OSS environment provides industry-standard application program interfaces (APIs) and utilities to enable you to port existing applications quickly and easily to NonStop systems. The NonStop operating system continues to support Guardian services. Most features of the HP COBOL language and library are available in the OSS environment, and most of them operate as they do in the Guardian environment. For more information on OSS, see Chapter 20: Using HP COBOL in the OSS Environment.

In both the OSS and the Windows environments, the flag `-Wsystype={guardian|oss}` determines whether the native COBOL cross compiler creates loadfiles for the Guardian environment or for the OSS environment. The default is the Guardian environment. For more information on the native COBOL cross compiler, see Chapter 35: Native COBOL Cross Compiler on PC (page 983).

Compiler Input

The only required input to the compiler is:

- A source file containing the program text
- In some cases, compiler directives

The optional input is:

- One or more source library files from which the compiler can copy additional source text specified in COPY statements or SOURCE directives
- One or more user-specified object files that contain object code explicitly called by the program being compiled
- These COBOL external declarations files, which are used in resolving references to routines in the system library:

File	Contents
ECOBEX0	Operating system routines for the latest RVU of the operating system
ECOBEX1	Operating system routines for the next-to-latest RVU of the operating system
ECOBEXT	Operating system routines for the second-next-to-latest RVU of the operating system
ZCOBDLL	Utility routines
ZCREDLL	Utility routines

Compiler Output

The output of the compiler is:

- A listing, including any compiler diagnostic messages (if the file for the listing is a disk file, the compiler creates an EDIT file unless the file already exists)
- An object file, if no compilation errors occurred

The ECOBOL compiler produces either:

- A loadfile
- A linkfile
- A dynamic-link library (DLL)

For information about the COBOL85 and NMCOBOL compilers, see the *COBOL Manual for TNS and TNS/R Programs*.

Combining Separately Compiled Source Programs

You can compile several source files separately and then use the `eld` utility to combine their object code files into a single loadfile (see the *eld Manual* for more information). In the Guardian environment, a loadfile can contain object code from these compilers:

Compiler	Language
ECOBOL	HP COBOL
CCOMP	HP C
CPPCOMP	HP C++
EpTAL	pTAL

You can combine multiple object files in a single loadfile only if the multiple object files are either all TNS object files, all TNS/R native object files, or all TNS/E native object files.

Executing Loadfiles

You can execute loadfiles for these environments:

Guardian Environment

From ...	Use ...	For more information, see ...
A terminal	TACL command RUN or RUND	RUN or RUND Command
An executing program	TS/MP transaction-processing software	*TS/MP System Management Manual*
	CLU_PROCESS_CREATE_ routine	*CRE Programmer's Guide*

OSS Environment

From ...	Use ...	For more information, see ...
A terminal	Name of the executable file (type it and press the return key). The current directory must be in your search path.	Running HP COBOL Programs (page 722)
An executing COBOL process (execute the loadfile as a separate process)	OSS process create function `fork()` or `tdm_fork(2)`	*Open System Services Programmer's Guide*

3 Source Program Organization and Format

The lines of a COBOL source program are organized into divisions. The program ends with an END PROGRAM statement. Individual lines must follow a reference format that the compiler accepts. Any line in a source program can contain a COPY statement, which tells the compiler to insert source text from a specified disk file, called a COPY library.

A run unit can contain more than one source program. Programs in the same run unit can share resources with each other. The programs can be nested, which means that one program contains one or more other programs, which might in turn contain one or more other programs. A nested program can be common, which means that it can be called by any other program contained in the program that contains it. Nested programs are always in the same run unit; therefore, they can share resources.

An initial program is one whose program state is initialized whenever any program in its run unit calls it.

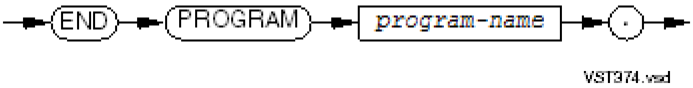
Source Program Components

The components of a COBOL source program must appear in the order shown in Table 3-1.

Table 3-1 Source Program Components

Component	Required or Optional?	Purpose	Explained in Chapter
Identification Division	Required	• To specify the program name (required) • To specify your name, the date, and the program purpose (optional)	5
Environment Division	Optional	To describe the program's equipment and processing options and name the files it uses (you must change this information when you move a COBOL program to an HP system from another type of system)	6
Data Division	Optional	To define the data that the program uses (reserve data storage; define data formats, types, and structures; complete file descriptions)	7
Procedure Division	Optional	To specify the program activity (the data processing)	8
END PROGRAM statement	Depends—see Purpose	To mark the end of the program If you submit only one program to the compiler at a time, the END PROGRAM statement is optional. If you compile several programs (one after another or nested), the END PROGRAM statement is required for each program except the one that ends last. You can use an ENDUNIT directive instead of an END PROGRAM statement.	2

The END PROGRAM statement has this syntax:



program-name

is a COBOL word. It names the same program unit that the PROGRAM-ID does (see PROGRAM-ID Paragraph (page 109)), for example:

```
IDENTIFICATION DIVISION.
PROGRAM-ID. MYPROG.
...
END PROGRAM MYPROG.
```

Reference Format for Source Program Lines

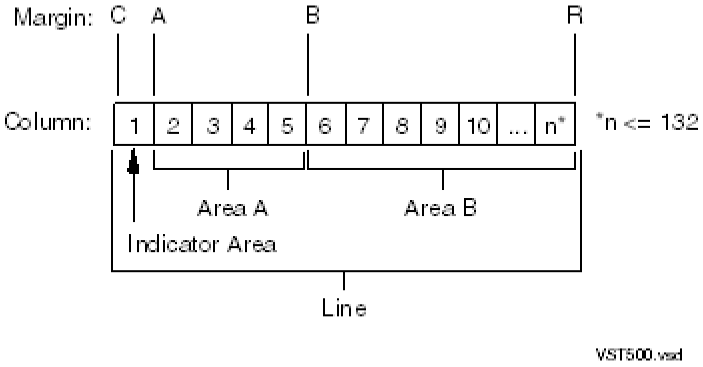
Individual source program lines must follow a reference format that the compiler accepts. There are two choices of reference format: Tandem and ANSI. Tandem reference format, an HP extension to COBOL, is less restrictive than ANSI. The principal differences between the Tandem and ANSI formats are in:

- Margin locations
- Permitted line lengths
- Absence of sequence number and identification field in Tandem reference format

You can write an HP COBOL program completely in either format or in a mixture of both. The default is Tandem reference format. Unless you direct the compiler to accept the ANSI format, the compiler assumes the entire program is in Tandem reference format.

The rest of this topic describes the Tandem reference format. For information on the ANSI reference format, see Chapter 17: ANSI Reference Format.

Figure 3-1 Tandem Reference Format



Lines in Tandem reference format are not restricted to a fixed length and can have up to 132 characters (longer lines are truncated). The Tandem reference format has no identification field. You can supply comments for your program on separate lines. See Indicator Area.

Indicator Area

The indicator area begins at margin C and ends at margin A, using only column 1. It can be empty, or it can contain a single character that describes the type of information on the line.

Table 3-2 Valid Indicator Area Characters (Tandem Reference Format)

Character	Character Name	Meaning and Topic Name
?	Question mark	Compiler Directive (?)
*	Asterisk	Ordinary Comment (*)
/	Slash	Comment for Top of Next Page (/)
D	Uppercase *D*	Debugging Line (D or d)
d	Lowercase *d*	Debugging Line (D or d)

Table 3-2 Valid Indicator Area Characters (Tandem Reference Format) *(continued)*

Character	Character Name	Meaning and Topic Name
-	Hyphen	Continuation Line (-)
	Space	Text Line

Compiler Directive (?)

A compiler directive has a question mark (?) in the indicator area. A compiler directive is an instruction to the compiler (for more information on compiler directives, see Compiler Directives (page 542)).

The compiler interprets any line that has a question mark in column 1 as a compiler directive. (This is true even when a program uses ANSI format, in which case the compiler handles the line as if it began with the indicator area. For details on ANSI format, see Chapter 17: ANSI Reference Format (page 711).)

Ordinary Comment (*)

An ordinary comment has an asterisk (*) in the indicator area. A comment can appear anywhere in a program. The compiler ignores it.

Comment for Top of Next Page (/)

A comment to be printed at the top of the next page has a slash (/) in the indicator area. Like an ordinary comment, this comment can appear anywhere in a program. The compiler advances to the top of the next page and prints the comment at the top of that page.

Debugging Line (D or d)

A debugging line has the letter *D* or *d* in the indicator area. If the program includes a DEBUGGING MODE clause, the debugging line is part of the program; otherwise, it is a comment. A debugging line cannot contain embedded SQL/MP or SQL/MX statements. For information on DEBUGGING MODE, see SOURCE-COMPUTER Paragraph (page 114).

Continuation Line (-)

A continuation line has a hyphen (-) in the indicator area. It is a continuation of the previous line. Always leave area A of a continuation line blank.

In an HP COBOL source program, you can put a blank line between a continued line and its continuation line. Such a blank line has no effect on the continued line or its continuation (see Text Line).

You can continue any word or literal. If you continue a numeric literal, a reserved word, or a user-defined word, the compiler ignores the trailing spaces of the previous line and initial spaces of the continuation line.

The rules for continuing a nonnumeric literal in Tandem reference format are:

- The nonnumeric literal does not contain the trailing spaces of the previous line.
- The first nonblank character in area B of the continuation line must be a quotation mark. The continuation begins with the character immediately following that quotation mark.

The rules for continuing a national literal in Tandem reference format are:

- The national literal does not contain the trailing spaces of the previous line.
- The first nonblank character in area B of the continuation line must be the letter *N* followed by a quotation mark. The continuation begins with the character immediately following the quotation mark.

Example 3-1 Continuation Line in Tandem Reference Format

```
....+....1....+....2....+....3....+....4....+....5....+....6....+....7....+....8
      DISPLAY "THIS IS AN EXAMPLE OF CONTINUING A LITERAL
 -            " IN TANDEM REFERENCE FORMAT."
```

Text Line

The compiler handles any line that begins with a space character as a program text line.

NOTE: If the character in the indicator area is not a question mark, asterisk, slash, the letter *D* (uppercase or lowercase), or a hyphen, the compiler issues a warning and handles the character as if it were a space.

If all characters between margin A and margin R are spaces, then the line is a blank line. Blank lines can occur anywhere in the source text and have no effect on either the syntax or semantics of source programs. If at least one character other than spaces occurs after margin A, that character begins a new lexical element. When a program text line is not continued (that is, when its successor text line does not contain a hyphen in its indicator area), the compiler assumes that a space follows the last nonblank character of the first line. If the final character is a period, comma, or semicolon, the assumed space completes the separator begun by that character; otherwise, the assumed space itself acts as a space separator.

Area A and Area B

HP COBOL ignores the distinction between area A and area B. The lexical elements of a source program can occur anywhere between margin A and margin R. The few exceptions to this are described in Restrictions.

Restrictions

You can format the text of a source program line freely except as noted in these topics:

- Reserved Words
- Period Separators
- Embedded SQL/MP or SQL/MX Statements
- Comment-Entry
- Standard COBOL Practice

Reserved Words

Do not split these reserved words across program lines:

- COPY

  For other restrictions on the format of the COPY statement body, see COPY Statement (page 507).

- DATE-COMPILED

  If you split the reserved word DATE-COMPILED across program lines, the compiler handles the paragraph as a simple comment-entry and does not replace its contents with the date and time of compilation.

- REPLACE

Period Separators

Do not put anything other than spaces after the period separator that follows one of these reserved word sequences:

- DIVISION
- DECLARATIVES
- END DECLARATIVES

Do not put anything other than spaces or a USE statement after the period separator that follow the reserved word SECTION.

Embedded SQL/MP or SQL/MX Statements

When COBOL statements and embedded SQL/MP or SQL/MX statements appear on the same line, these restrictions apply:

- The COBOL statements cannot be COPY or REPLACE statements.
- The COBOL statements must follow the embedded SQL/MP or SQL/MX statement terminator.

Comment-Entry

The restrictions on the comment-entry are:

- Following a comment-entry, the first keyword in the ensuing text (such as ENVIRONMENT DIVISION or another comment-entry) must begin on a program text line that has a space in the indicator area. This keyword must start in area A, and only space characters can precede the keyword.
- You cannot continue a comment-entry with the hyphen convention; however, you can implicitly continue it onto additional program text lines, provided that area A in such lines contains only space characters.

Standard COBOL Practice

Observe these restrictions to conform to standard COBOL practice (HP COBOL does not enforce them).

- Verify that the special constructs DECLARATIVES and END DECLARATIVES (along with each one's terminating period separator) are completely contained in a single program text line with a space character in the indicator area. Start them in area A and precede them only by space characters.
- Start each of these in area A of a program text line that has a space in its indicator area:
  - The keyword or user-defined word naming a division, section, or paragraph
  - The initial keyword of an END PROGRAM statement

  Any characters preceding these words must be space characters.
- Start the level-numbers 01 and 77 and all level indicators (FD and SD) in area A of a program text line with a space character in the indicator area and precede them only by space characters. Other level-numbers can start in either area A or area B.
- When a lexical element that starts in area A of a program text line is followed by a separator, you can start the next element (if any) on that line at any point after the separator; otherwise, start all lexical elements other than those mentioned in the preceding rules in area B. Always start continuations of lexical elements in area B.
- Verify that both of the characters that comprise the pseudo-text separator, ==, appear on the same program text line.

COPY Libraries

Any line in a source program can contain a COPY statement, which tells the COBOL compiler to insert source text from a specified disk file, called a COPY library.

A COPY library is an EDIT or OSS ASCII file divided into one or more sections that begin with a SECTION directive. In the COPY library, you can specify the reference format of the line or lines to be copied.

You can specify the name of a COPY library in the source program or in a command to compile the program:

- The COPY statement itself can include the phrase IN *library-name* (see COPY Statement (page 507)).
- The command that initiates the compilation can specify a *library-name* for the compiler to use whenever a COPY statement does not include a *library-name* (see Starting a Compilation (page 538)).

If you do not specify a *library-name* in the compile command or in the COPY statement, the compiler uses the name COPYLIB.

If you do not fully qualify *library-name* in the compile command or in the COPY statement, the compiler assumes the library file is on the current default volume or subvolume or both.

For details on the COPY statement and COPY libraries, see COPY Statement (page 507).

Nested Source Programs

Source programs can be nested (that is, a source program can contain other source programs). One program can contain another program directly or indirectly. When a program directly or indirectly contains other programs, each program can use the same user-defined names for different objects.

Topics:

- Directly Contained Programs and Indirectly Contained Programs
- Scope of User-Defined Names

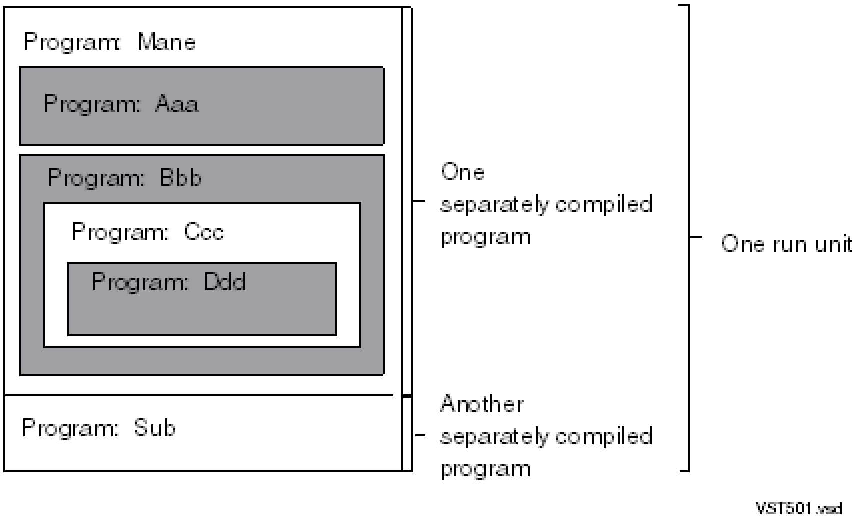
Directly Contained Programs and Indirectly Contained Programs

One program can contain another program directly or indirectly. Suppose a program, L, contains another program, R. Program L directly contains program R when program L does not contain any other program that also contains program R. Program L indirectly contains program R if program L does contain any other program that contains program R.

In Figure 3-2, all the programs are in the same run unit and:

The program ...	Directly contains ...	And indirectly contains ...
Mane	Aaa, Bbb	Ccc, Ddd
Aaa	Nothing	Nothing
Bbb	Ccc	Ddd
Ccc	Ddd	Nothing
Ddd	Nothing	Nothing
Sub	Nothing	Nothing

Figure 3-2 Directly Contained Programs and Indirectly Contained Programs



Scope of User-Defined Names

When a program directly or indirectly contains other programs, each program can use identical user-defined names to name objects independent of the use of these user-defined names by other programs. If program X describes a data item named B, and program X includes another program Y, program Y can describe a different data item named B (or even a file connector named B). The B of program Y is entirely separate from the B of program X. The two Bs have different scopes.

Table 3-3 Scope of User-Defined Names

Type of User-Defined Name	What Can Reference It
Paragraph-name Section-name	Statements and entries in the program that defines it
Library-name Text-name	Any COBOL program
Alphabet-name Class-name Condition-name Mnemonic-name Symbolic character	Statements and entries in either the program that contains the Configuration Section or programs that it contains (when the user-defined word is declared in a Configuration Section)
Program-name Condition-name* Data-name File-name Record-name Index-name	See these topics: • Program-Name • Condition-Name, Data-Name, File-Name, and Record-Name • Index-Name

* When not declared in the Configuration Section.

Program-Name

The PROGRAM-ID paragraph of a program’s Identification Division declares that program’s program-name. Only the CALL and CANCEL statements and the END PROGRAM statement can refer to a program-name. The program-names allocated to programs constituting a run unit are not necessarily unique, but when two programs in a run unit are identically named, at least one of those two programs must be directly or indirectly contained within another separately compiled program that does not contain the other of those two programs.

These rules regulate the scope of a program-name:

- If the program-name, X, is that of a program that does not have the common attribute and is directly contained within another program, Y, then only statements included in Y can refer to X.
- If the program-name, V, is that of a program that does have the common attribute and is directly contained within another program, W, then any programs directly or indirectly contained within W can reference V; however, program V and any programs contained within V cannot refer to V (because V calling V is recursion, and because V cannot cancel itself).
- If the program-name, U, is that of a separately compiled program, then U can be referenced by statements included in any other program in the run unit, except programs it directly or indirectly contains. HP COBOL programs that are compiled at the same time but separated by the ENDUNIT directive or by an END PROGRAM statement are called separately compiled, as if they had been compiled by different executions of the compiler.

Condition-Name, Data-Name, File-Name, and Record-Name

When condition-names, data-names, file-names, and record-names are declared in a source program, only that program can refer to them except when one or more of the names is global and the program contains other programs.

The requirements governing the uniqueness of the names allocated by a single program as condition-names, data-names, file-names, and record-names are explained in the topic COBOL Words (page 73).

A program cannot reference any condition-name, data-name, file-name, or record-name declared in any program it contains.

A program P that declares a global name can reference that name. Any program directly or indirectly contained in P can also reference the global name.

When program B is directly contained within program A, both programs can declare a condition-name, data-name, file-name, or record-name using the same user-defined word. When program B refers to such a duplicated name, the compiler uses this sequence of rules to determine which object is being referenced:

- The set of names the compiler uses to identify a referenced object consists of all names that are defined in program B and all global names that are defined in program A and in any programs that directly or indirectly contain program A. Using this set of names, the compiler applies the normal rules of qualification and any other rules for uniqueness of reference until it identifies one or more objects.
- If the compiler identifies only one object, that object is the reference object.
- If the compiler identifies more than one object, no more than one of them can have a name local to program B. If zero or one of the objects has a name local to program B, these rules apply:
  - If the name is declared in program B, the object in program B is the referenced object.
  - If the name is not declared in program B, and if program A is directly contained within another program, C, the referenced object is:
    - The object in program A if the name is declared in program A and is global.
    - The object in C if the name is not declared in program A or is not global in A, and is declared in C and is global in C; otherwise, the compiler applies this rule to further containing programs until it has found a single valid name.

Index-Name

If a data item is external and/or global and includes a table accessed with an index, that index is also external and/or global (respectively); therefore, the scope of an index-name is identical to that of the data-name that names the table whose index is named by that index-name, and the scope rules for data-names apply. Index-names cannot be qualified.

Common Programs

A common program is one that includes the COMMON clause in its Identification Division. A common program can be called by any program directly or indirectly contained in the program that directly contains the common program (except for the called program itself and the programs that it contains). In Example 3-2: Programs With Shared Data, if program Aaa includes the COMMON clause, the programs Bbb, Ccc, and Ddd can call it.

Programs in the Same Run Unit

A program and the programs that it contains are always in the same run unit, but separate programs (not contained in that program and not containing it) can also be in the same run unit.

Initial Programs

An initial program has an INITIAL clause in its Identification Division. An initial program's program state is initialized whenever the program is called. If program X is an initial program, whenever a program calls X, the program state of X is the same as when X was first called in that run unit. During the process of initializing an initial program:

- The program's internal data items are initialized (see Initializing Data Items (page 188)).
- Files with internal file connectors associated with the program are not in the open mode.
- The control mechanisms for all PERFORM and ALTER statements contained in the program are set to their initial states.

The CANCEL and NOCANCEL directives, which determine whether a program is initialized the first time it is called after having been canceled by a CANCEL statement, do not affect initial programs. Initial programs are initialized every time they are called, whether or not they were cancelled.

How Programs Share Resources

A program can share resources with programs that it contains (directly or indirectly) and programs that are in the same run unit as it is.

The resources that a program can share with programs that it contains are:

- record-names
- data-names
- condition-names
- file-names

The program makes them accessible to the inner programs by declaring them to be global. This enables the inner programs to use the associated data items and file connectors without declaring them. (The opposite of global is local. Local names are only accessible to the program that declares them.)

The resources that a program can share with programs in its run unit are:

- data items
- file connectors

The program makes them accessible to the other programs by declaring them to be external. Any other program in the run unit can use them by likewise declaring them to be external. (The opposite of external is internal. Internal objects are only accessible to the program that declares them—or, if they have global names, to programs within that program—and only one program can declare them.)

Topics:

- Global and Local Names
- External and Internal Objects
- Shared Data
- Shared Files

Global and Local Names

A data-name (which names a data item) or a file-name (which names a file connector) can be either global or local.

Suppose that program X declares data-name Y, and program X contains program Z, which also declares data-name Y. When program Z references data-name Y, it is referencing the Y that it declared, not the Y that program X declared, whether program X declared its Y to be global or not.

Some names are always global; other names are always local; and some names are either global or local, depending on specifications in the program that declares the names:

Name	It is global if a GLOBAL clause is in ...
Record-name	its record description entry or the file description entry for the file-name associated with its record description entry (if the record-name is in the File Section)
Data-name	its record description entry
Condition-name	an entry to which its data description entry is subordinate (that is, if a data-name is declared global, all condition-names subordinate to it are automatically global)
File-name	its file description entry

In some circumstances, a data description, file description, or record description entry cannot specify the GLOBAL clause (see GLOBAL Clause (page 202)).

If a data-name, file-name, or condition-name declared in a data description entry is not global, the name is local.

Global names are inherited by contained programs. Suppose program OUTER declares a name X to be global and contains program INNER, which contains program INNERMOST. As long as program INNER does not define the name X explicitly, a reference to X in INNER refers to the X defined in OUTER. Also, as long as program INNERMOST does not define the name X and program INNER does not define X as a global name, a reference to X in INNERMOST refers to the X defined in OUTER.

Global names, as well as local names, can be associated with external and internal objects.

External and Internal Objects

An **external** object is stored in an area that is associated with the run unit rather than with any particular program within the run unit.

An **internal** object is stored in an area that is associated only with the program that describes the object.

An external object can be referenced by any program in the run unit that describes the object. All such descriptions must be identical, or the results of the references are unpredictable. References to an external object from different programs are always to the same object. In a run unit, there is only one representation of an external object.

External and internal objects can have either global or local names.

Data Records in Working and Extended-Storage

To give a data record described in the Working-Storage or Extended-Storage Section the external attribute, include the keyword EXTERNAL in its data description entry. Only record data description entries can include the EXTERNAL clause. Any data item described by a data description entry subordinate to an entry describing an external record also inherits the external attribute.

If a record or data item does not have the external attribute, it is part of the internal data of the program that describes it.

File Connectors and Their Records

To give a file connector the external attribute, include the keyword EXTERNAL in its file description entry. When a file connector has the external attribute, the records and the data items of the file inherit the external attribute.

If a file connector does not have the external attribute, it is internal to the program that describes the associated file-name.

The data records described subordinately to either of these file description entries are internal to the program that describes the file-name unless the data records themselves are declared to be external:

- A file description entry that does not contain the EXTERNAL clause
- A sort-merge file description entry

Any data items described subordinate to the data description entries for such records are also internal to the program that describes the file-name.

Linkage Section Records

Data records and any subordinate data items in the Linkage Section are representatives of data items defined in other programs. They are considered to be internal to the program describing them and are directly accessible only to that program; however, they are indirectly accessible to programs called by that program.

Shared Data

Two programs in a run unit can refer to common data in these circumstances:

- Any program that has described an external data record can refer to the data content of that record.
- If program B is contained within program A, both programs can refer to data possessing the global attribute and described in either:
  - The containing program A
  - Any program that directly or indirectly contains A
- When a program passes a parameter value by reference, this establishes a common data item—a storage location that each program can access. The called program can refer to a data item in the calling program, using the same identifier or a different identifier.

If several programs define a data item as external (causing its storage location to be a single location outside all programs) and they also define the data item as having a global name, then all such programs and all programs nested within each of them have access to the data item.

Example 3-2 is a listing of a set of programs that share data through the global and external mechanisms. Example 3-3 shows the output produced by executing the programs.

Example 3-2 Programs With Shared Data

```
*  _______________________________________________________________________
* | Program: Mane                                                         |
* |  Data: w (local to Mane)                                              |
* |        y (global throughout Mane and its descendants)                 |
* |        z (global throughout Mane and its descendants, and external)   |
* |   Can call: Aaa, Bbb because both are directly contained              |
* |    _________________________________________________________________  |
* |   | Program: Aaa                                                    | |
* |   |  Data: none                                                     | |
* |   |   Can call: Bbb because Bbb is common                           | |
* |   |             Sub because Sub is separate unit                    | |
* |   |_________________________________________________________________| |
* |   | Program: Bbb (common)                                           | |
* |   |  Data: w (external)                                             | |
* |   |        x (global throughout Bbb)                                | |
* |   |   Can call: Ccc because Ccc is directly contained               | |
* |   |             Sub because Sub is separate unit                    | |
* |   |  ____________________________________________________           | |
* |   | | Program: Ccc                                      |           | |
* |   | |  Data: w (external)                               |           | |
* |   | |        y (local to Ccc)                           |           | |
* |   | |   Can call: Ddd because Ddd is directly contained |           | |
* |   | |             Sub because Sub is separate unit      |           | |
* |   | |  ______________________________________________   |           | |
* |   | | | Program: Ddd                                 |  |           | |
* |   | | |  Data: none                                  |  |           | |
* |   | | |   Can call: Sub because Sub is separate unit |  |           | |
* |   | | |______________________________________________|  |           | |
* |   | |___________________________________________________|           | |
* |   |_________________________________________________________________| |
* |_______________________________________________________________________|
* | Program: Sub                                                          |
* |  Data: w (external)                                                   |
* |        x (local to Sub)                                               |
* |        y (global throughout Sub and any descendants)                  |
* |        z (global throughout Sub and any descendants, and external)    |
* |   Can call: no other routines (calling Mane would be recursion)       |
* |_______________________________________________________________________|
* Accessibility:  Which Routines Can Access Which Data Names?
*                                                      -----
*     Mane  Aaa  Bbb  Ccc  Ddd   Sub      Item
*     ----  ---  ---  ---  ---   ---   -------------
*      L     .    .    .    .     .     w of Mane
*      /     .    L    .    .     .       w of Bbb
*      /     .    .    L    .     .         w of Ccc
*      /     .    .    .    .     L           w of Sub
*      .     .    L    G    G     .     x of Bbb
*      .     .    .    .    .     L       x of Sub
*      L     G    G    /    G     .     y of Mane
*      .     .    .    L    .     .       y of Ccc
*      .     .    .    .    .     L         y of Sub
*      L     G    G    G    G     .     z of Mane
*      .     .    .    .    .     L       z of Sub
*
*    ( L = access because it is locally defined)
*    ( G = access because ancestor declared it global)
*    ( / = can't access global because of local definition)
*    ( . = can't access because no local or global definition)*

* Storage    1.  W of Bbb, w of Ccc, and w of Sub all refer to one location
* Allocation
*            2.  Z of Mane and z of Sub each refer to one location.
*
```

```
*           3.  All other identifiers refer to unique locations.
?main Mane
  IDENTIFICATION DIVISION.
  PROGRAM-ID. Mane.
  DATA DIVISION.
  WORKING-STORAGE SECTION.
    01 w PICTURE 99 VALUE 3.
    01 y PICTURE 99 GLOBAL VALUE 1.
    01 z picture 99 GLOBAL EXTERNAL.

  PROCEDURE DIVISION.
  m.
    DISPLAY "Mane begin"
    MOVE 25 to z
    PERFORM show-me
    CALL Aaa   PERFORM show-me
    CALL Bbb   PERFORM show-me
    CALL Sub   PERFORM show-me
    CALL Sub   PERFORM show-me
    DISPLAY "Mane end"
    stop run
    .
  show-me.
    DISPLAY "in Mane, w=/" w "/  y=/" y "/  z=/" z "/"
    .
  IDENTIFICATION DIVISION.
  PROGRAM-ID. Aaa.
  PROCEDURE DIVISION.
  a.
    DISPLAY "  Aaa begin"
    PERFORM show-me
    DISPLAY "  Aaa adding 2 to y"
    ADD 2 TO y  PERFORM show-me
    CALL Bbb    PERFORM show-me
    DISPLAY "  Aaa end"
    EXIT PROGRAM
    .
  show-me.
    DISPLAY "  y (global from Mane)=/" y "/"
            " z (global from Mane)=/" z "/"
    .
  END PROGRAM Aaa.
  IDENTIFICATION DIVISION.
  PROGRAM-ID. Bbb COMMON.
  DATA DIVISION.
  WORKING-STORAGE SECTION.
    01 x PICTURE 99 GLOBAL   VALUE 0.
    01 w PICTURE 99 EXTERNAL.
PROCEDURE DIVISION.
  b.
    MOVE 1 TO w
    DISPLAY "    Bbb begin"
    PERFORM show-me
    DISPLAY "    Bbb adding 3 to w and to x"
    ADD 3 to w x  PERFORM show-me
    CALL Ccc      PERFORM show-me
    DISPLAY "    Bbb end"
    EXIT PROGRAM
    .
  show-me.
    DISPLAY "    in Bbb, w (ext)=/" w "/"
             "  x (global in Bbb)=/" x "/"
             "  y (global from Aaa)=/" y "/"
    .
  IDENTIFICATION DIVISION.
  PROGRAM-ID. Ccc.
  DATA DIVISION.
  WORKING-STORAGE SECTION.
    01 w PICTURE 99 EXTERNAL.
    01 y PICTURE 99 VALUE 0.
  PROCEDURE DIVISION.
   c.
    DISPLAY "      Ccc begin"
    PERFORM show-me
    DISPLAY "      Ccc adding 4 to w, x, and y"
    ADD 4 to w x y
    PERFORM show-me
    CALL Ddd
    DISPLAY "      Ccc end"
    EXIT PROGRAM
    .
  show-me.
    DISPLAY "      in Ccc, w (ext)=/" w "/"
               "  x (global from Bbb)=/" x "/"
               "  y (local in Ccc)=/" y "/"
    .
  IDENTIFICATION DIVISION.
PROGRAM-ID. Ddd.
  DATA DIVISION.
  PROCEDURE DIVISION.
   d.
    DISPLAY "        Ddd begin"
    DISPLAY "        in Ddd, x (global from Bbb)=/" x "/"
                           " y (global from Mane)=/" y "/"
    MOVE 17 to z
    DISPLAY "        Ddd changing z to 17"
    DISPLAY "        in Ddd, z (global from Mane)=/" z "/"
    DISPLAY "        Ddd end"
    .
  END PROGRAM Ddd.
  END PROGRAM Ccc.
  END PROGRAM Bbb.
  END PROGRAM Mane.
?ENDUNIT
IDENTIFICATION DIVISION.
  PROGRAM-ID. Sub.
  DATA DIVISION.
  WORKING-STORAGE SECTION.
    01 y PICTURE 99 GLOBAL VALUE 2.
    01 w PICTURE 99 EXTERNAL.
    01 x PICTURE 99 VALUE 0.
    01 z PICTURE 99 EXTERNAL.
  PROCEDURE DIVISION.
  s.
    DISPLAY "  Sub begin"
    PERFORM show-me
    DISPLAY "  Sub moving 5 to w"
    MOVE 5 to w   PERFORM show-me
    IF x = 0
       DISPLAY "  Sub adding 5 to w, x, y, and z"
       ADD 5 to w x y z  PERFORM show-me
    END-IF
    DISPLAY "  Sub end"
    EXIT PROGRAM
    .
  show-me.
    DISPLAY "  in Sub, w (ext)=/" w "/ x (local)=/" x "/"
                  "  y (local)=/" y "/ z (ext)=/" z "/"
    .
  END PROGRAM Sub.
```

Example 3-3 Output From Programs With Shared Data

```
Mane begin
in Mane, w=/03/  y=/01/  z=/25/
  Aaa begin
  y (global from Mane)=/01/ z (global from Mane)=/25/
  Aaa adding 2 to y
  y (global from Mane)=/03/ z (global from Mane)=/25/
    Bbb begin
    in Bbb, w (ext)=/01/  x (global in Bbb)=/00/  y (global from Aaa)=/03/
    Bbb adding 3 to w and to x
    in Bbb, w (ext)=/04/  x (global in Bbb)=/03/  y (global from Aaa)=/03/
      Ccc begin
      in Ccc, w (ext)=/04/  x (global from Bbb)=/03/  y (local in Ccc)=/00/
      Ccc adding 4 to w, x, and y
      in Ccc, w (ext)=/08/  x (global from Bbb)=/07/  y (local in Ccc)=/04/
        Ddd begin
        in Ddd, x (global from Bbb)=/07/ y (global from Mane)=/03/
        Ddd changing z to 17
        in Ddd, z (global from Mane)=/17/
        Ddd end
      Ccc end
    in Bbb, w (ext)=/08/  x (global in Bbb)=/07/  y (global from Aaa)=/03/
    Bbb end
  y (global from Mane)=/03/ z (global from Mane)=/17/
  Aaa end
in Mane, w=/03/  y=/03/  z=/17/
    Bbb begin
    in Bbb, w (ext)=/01/  x (global in Bbb)=/07/  y (global from Aaa)=/03/
    Bbb adding 3 to w and to x
    in Bbb, w (ext)=/04/  x (global in Bbb)=/10/  y (global from Aaa)=/03/
      Ccc begin
      in Ccc, w (ext)=/04/  x (global from Bbb)=/10/  y (local in Ccc)=/04/
      Ccc adding 4 to w, x, and y
      in Ccc, w (ext)=/08/  x (global from Bbb)=/14/  y (local in Ccc)=/08/
        Ddd begin
        in Ddd, x (global from Bbb)=/14/ y (global from Mane)=/03/
        Ddd changing z to 17
        in Ddd, z (global from Mane)=/17/
        Ddd end
      Ccc end
    in Bbb, w (ext)=/08/  x (global in Bbb)=/14/  y (global from Aaa)=/03/
    Bbb end
in Mane, w=/03/  y=/03/  z=/17/
  Sub begin
  in Sub, w (ext)=/08/ x (local)=/00/  y (local)=/02/ z (ext)=/17/
  Sub moving 5 to w
  in Sub, w (ext)=/05/ x (local)=/00/  y (local)=/02/ z (ext)=/17/
  Sub adding 5 to w, x, y, and z
  in Sub, w (ext)=/10/ x (local)=/05/  y (local)=/07/ z (ext)=/22/
  Sub end
in Mane, w=/03/  y=/03/  z=/22/
  Sub begin
  in Sub, w (ext)=/10/ x (local)=/05/  y (local)=/07/ z (ext)=/22/
  Sub moving 5 to w
  in Sub, w (ext)=/05/ x (local)=/05/  y (local)=/07/ z (ext)=/22/
  Sub end
in Mane, w=/03/  y=/03/  z=/22/
Mane end
```

Shared Files

Two programs in a run unit can refer to common file connectors in these circumstances:

- Any program that has described an external file connector can refer to that file connector.
- If program G is contained within program H, both programs can refer to a common file connector. They do so by referring to an associated global file-name (or associated global record-name, in the case of the WRITE and REWRITE statements) described in either:
  - The containing program H
  - Any program that directly or indirectly contains H

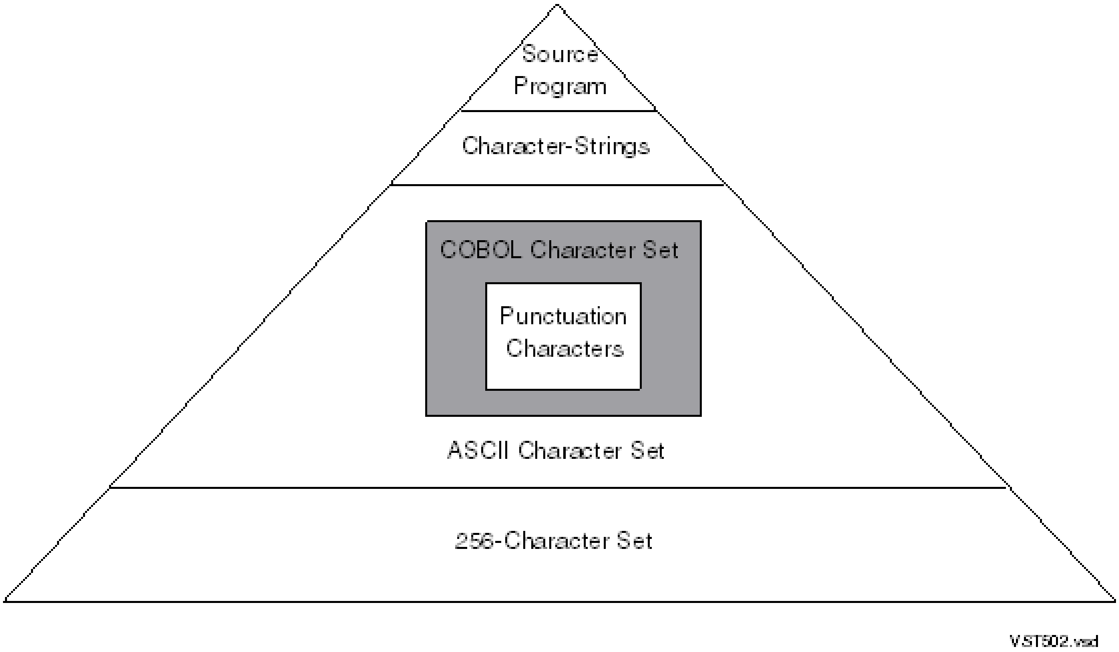
If several programs define a file connector as external (causing its storage location to be a single location outside all programs) and they also define the file connector as having a global name, then all such programs and all programs nested within each of them have access to the file connector.

4 Language Elements

The smallest unit of the COBOL language is a character. You use most characters to form character-strings, and you use a few punctuation characters to form separators. The text of a source program consists of character-strings delimited by separators.

Most character-strings and all separators consist of one or more characters from the COBOL character set, which is a subset of the ASCII character set. The character-strings that are exceptions to this rule are comments and nonnumeric literals, which can contain any of 256 characters (although ASCII characters are recommended).

Figure 4-1 Language Element Relationships



Topics:

- COBOL Character Set
- Punctuation Characters
- Character-Strings

COBOL Character Set

The COBOL character set is a subset of the ASCII character set (which is listed in Appendix A: ASCII Character Set). The COBOL character set has 78 characters; the ASCII character set has 128.

Figure 4-2 COBOL Character Set

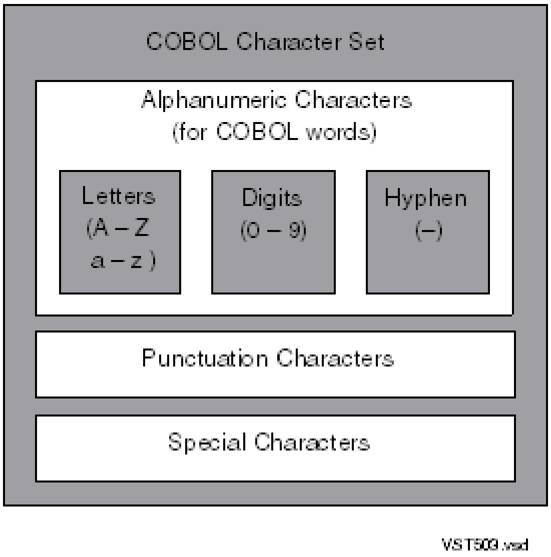


Table 4-1 Alphanumeric Characters (for COBOL Words)

Characters	Name of Character Set
0 through *9*	Digits
A through *Z*	Uppercase letters
a through *z*	Lowercase letters
-	Hyphen or minus sign

Table 4-2 Punctuation Characters

Character	Name of Character
	Space
,	Comma
;	Semicolon
:	Colon
.	Period
"	Quotation mark
(	Left parenthesis
)	Right parenthesis
=	Equal sign

Table 4-3 Special Characters

Character	Name of Character
+	Plus sign
-	Hyphen or minus sign
*	Asterisk
/	Stroke or slash
$	Currency sign

Table 4-3 Special Characters *(continued)*

Character	Name of Character
>	Greater than sign
<	Less than sign

Except in nonnumeric literals, the compiler handles lowercase letters as equivalent to the corresponding uppercase letters.

You can use characters that are not in the COBOL character set in your COBOL source programs in these cases:

- A character other than the dollar sign ($) can represent the currency symbol in PICTURE clauses (see SPECIAL-NAMES Paragraph)
- A question mark (?) precedes a compiler directive (see Indicator Area (page 54))
- Comment-entries, comment lines, and nonnumeric literals can contain any characters in the computer's character set (but some control characters adversely affect the compiler listing)

Punctuation Characters

Punctuation characters belong to the COBOL character set and are listed in Table 4-2. In a COBOL source program, you can use a punctuation character in these contexts:

- Separators
- Comments
- Nonnumeric and National Literals
- Numeric Literals and PICTURE Character-Strings

Separators

A separator is one or more consecutive punctuation characters used to separate character-strings, sentences, or special clauses or to delimit other characters in expressions. The punctuation characters that can be used as separators are:

- Space
- Comma or Semicolon
- Colon
- Period
- Quotation Marks
- Parentheses
- Equal Sign

Every character-string must be followed by a sequence of one or more separators. The syntactic definition of the COBOL language specifies when a sequence can or must contain any of the period, parentheses, colon, or pseudo-text separators. A space separator can always immediately precede or follow any other separator, except where the reference format rules specify otherwise (see Reference Format for Source Program Lines (page 54)).

NOTE: The rules for using the punctuation characters as separators do not apply within comments, nonnumeric literals, numeric literals, or PICTURE character-strings)

Space

A space character is a separator. Anywhere that a space is used as a separator or a part of a separator, more than one space can be used. The compiler handles all spaces immediately following a comma, semicolon, or period separator as part of that separator and not as a distinct space separator.

Comma or Semicolon

A comma (,) or semicolon (;) that immediately precedes one or more spaces acts as a comma separator or semicolon separator, respectively. Except where explicitly prohibited, you can use comma and semicolon separators anywhere that the specifications permit or require space separators.

Colon

The colon (:) is a separator that COBOL uses to distinguish a reference modifier from a subscript. The colon marks the end of the leftmost character position portion of the reference modifier. It is required when shown in a syntax diagram.

Period

A period (.) that immediately precedes one or more spaces acts as a period separator. Period separators are required in certain places by the syntactic definition of COBOL. They cannot appear anywhere else.

Each sentence within the Identification and Procedure divisions and each entry within the Environment and Data divisions must end with a period separator.

Quotation Marks

Quotation marks in balanced pairs enclose nonnumeric literals. You must precede the beginning quotation mark (") with a space and follow the ending quotation mark with a separator space, comma, semicolon, period, or right parenthesis. Except where a literal continues across several lines, each delimiting quotation mark acts as a separator.

Within a simple nonnumeric literal, two consecutive quotation marks represent one quotation mark.

Parentheses

The punctuation characters left parenthesis [(] and right parenthesis [)] serve individually as separators. Parenthesis separators can appear only in balanced pairs delimiting subscripts, reference modifiers, arithmetic expressions, or conditions. The opening member of each balanced pair must be a left parenthesis separator; the closing member must be a right parenthesis separator.

Equal Sign

The pseudo-text delimiter (==) is a separator. It can only appear in balanced pairs delimiting pseudo-text. An opening pseudo-text separator must be immediately preceded by a space. A closing pseudo-text separator must be immediately followed by a space, comma, semicolon, or period separator.

Comments

All characters appearing within a comment-entry or comment line are a part of that entity and are never interpreted as separators in that context.

Nonnumeric and National Literals

Characters appearing within a character-string that represents a nonnumeric or national literal are components of its value and are never interpreted as separators.

Numeric Literals and PICTURE Character-Strings

Several punctuation characters (comma, period, left parenthesis, right parenthesis) are also defined as characters available for the formation of numeric literals and PICTURE character-strings. They are not separators in these contexts but instead represent a part of that language element; however, an exception exists if the apparent last character of such a character-string is a period or comma immediately followed by a space. In this case, the period or comma is always interpreted as part of the separator following the character-string, and not as the last character of the numeric literal or PICTURE character-string.

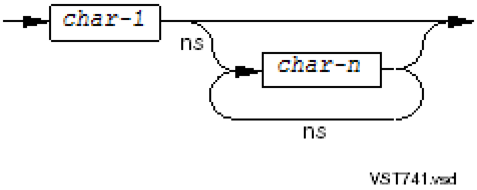
Character-Strings

A character-string consists of one or more characters that form:

- COBOL Words
- Literals
- PICTURE Character-Strings
- Comments

Most character-strings are limited to certain characters, but nonnumeric literals (except for hexadecimal literals beginning with `X`) and comments can contain any characters. Character-strings and separators form the text of COBOL source programs.

COBOL Words



char-1, *char-n*

If the COBOL word is a level-number or segment-number, then *char-1* is a digit.

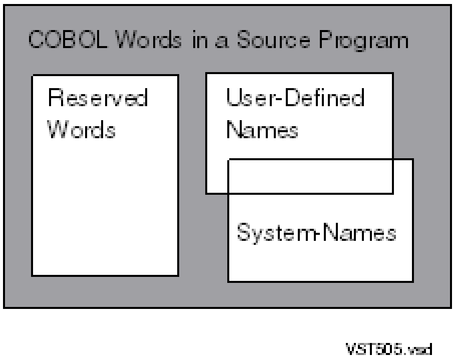
If the COBOL word is a section-name or paragraph-name, then these rules apply:

- *char-1* is a letter (uppercase or lowercase) or a digit.
- *char-n* is a letter (uppercase or lowercase), a digit, or a hyphen (-).
- The last character cannot be a hyphen.
- The maximum length of the word is 30 characters.

If the COBOL word is not a level-number, segment-number, section-name, or paragraph-name, these rules apply:

- *char-1* is a letter (uppercase or lowercase) or a digit.
- If *char-1* is the only character in the word, then it must be a letter.
- *char-n* is a letter (uppercase or lowercase), a digit, or a hyphen (-).
- At least one character must be either a letter or a hyphen.
- The last character cannot be a hyphen.
- The maximum length of the word is 30 characters.

Figure 4-3 COBOL Words in a Source Program



The same COBOL word can be used as a system-name and as a user-defined name within a source program. The class of a specific occurrence of the word is determined by context.

With the exception described in the preceding paragraph, every name that you reference in a COBOL program must be unique, either because no other name has the same spelling (including hyphenation), or because the name is part of a hierarchy of names (such as a data-name defined within a record). In the latter case, you can qualify the name with one or more of the higher-level names. You must specify enough higher-level names to make the name unique, but you need not specify all levels. The most significant name in a hierarchy must have unique spelling, because you cannot qualify it.

Topics:

- Reserved Words
- User-Defined Names
- System-Names
- Qualified Names

Reserved Words

A reserved word is a COBOL word that has a special meaning for the compiler. All reserved words appear in uppercase letters throughout this manual. (Some other words, such as product names, also appear in uppercase letters,)

A reserved word can appear in the source program only where the language syntax requires or permits it. Every reserved word is unique and cannot be used or redefined for any purpose other than those described in this manual.

Table 4-4 Reserved Word Categories

Category	Definition	Examples
Keywords	Required elements of the language construct for which they are defined	MOVE DIVIDE AND OR
Optional words	Elements of language constructs that you can use or omit, at your discretion (their presence or absence does not affect the meaning of the language construct for which they are defined)	ON THEN
Special registers	Data items that the compiler generates automatically when the source program uses the associated language	DEBUG-ITEM GUARDIAN-ERR LINAGE-COUNTER LINE-COUNTER * PAGE-COUNTER * PROGRAM-STATUS

Table 4-4 Reserved Word Categories *(continued)*

Category	Definition	Examples
Figurative constants	Words that name and reference constant values	SPACE ZERO
Special-character words	Required arithmetic and relational operators, used in arithmetic expressions and relation conditions, respectively	+ - * / ** = < > <= >=

* LINE-COUNTER and PAGE-COUNTER are associated with the Report Writer and are not available in HP COBOL.

For a list of all reserved words, see Chapter 22: Reserved Words.

User-Defined Names

A user-defined name is a COBOL word that you compose for your own use. You cannot use a reserved word for a user-defined name. You can compose words for these types of items:

alphabet-name
class-name
data-reference
file-name
index-name
level-number
library-name
mnemonic-name
paragraph-name
program-name
routine-name
section-name
segment-number
symbolic-character
text-name

In general, each user-defined name in each category must identify a unique entity within the source program. Exceptions:

- Level-numbers and segment-numbers have no uniqueness constraints.
- Using an all-digit word as a section-name or paragraph-name does not interfere with its concurrent use as a level-number or a segment-number.
- The same paragraph-name can identify two or more paragraphs if each paragraph appears in a different section of the Procedure Division.
- Using a word as a library-name does not interfere with its concurrent use as the name of an entity in another category (this is an HP extension to COBOL, which does not permit a library-name to duplicate the name of another entity); however, library-names must be unique among themselves.
- Using a word as a text-name does not interfere with its concurrent use as the name of an entity in another category. The same text-name can identify two or more library texts if each text appears in a different library.
- The category data-reference includes record-names, data-names, and condition-names. The same user-defined name can identify two or more entities in any combination of these subcategories; however, a program cannot reference the entities identified by a duplicated name unless the contexts of their definitions permit sufficient qualification to construct a unique reference to each of them (see Qualified Names).

System-Names

Although COBOL does not define any specific system-names, it includes several syntactic definitions that require them. COBOL limits the forms of system-names to those permitted for user-defined names. HP COBOL relaxes this restriction. The use of each particular system-name is limited to contexts appropriate for its category. The few minor restrictions on duplicate usages of system-names are discussed in SPECIAL-NAMES Paragraph.

In HP COBOL, system-names are either resource names or file names in the form used by the operating system.

The types of system-names are:

- Resource names

  A resource name is a mnemonic-name that identifies part of the environment:

Resource Name (mnemonic-name)	Environment Part Identified
CONSOLE	Operator console
MYTERM	Home terminal
CHANNEL-1 through CHANNEL-15	Carriage-control tape channels
SWITCH-1 through SWITCH-15	External switches
DANSK-NORSK DEUTSCH ESPANOL FRANCAIS-QW FRANCAIS-AZ SVENSK-SUOMI UK USASCII	Native character sets

  For more information on resource names, see SPECIAL-NAMES Paragraph.

- Operating system file names

  The NonStop operating system supports two file systems: the Guardian file system and the Open System Services (OSS) file system.

  A Guardian file name identifies one of these:

  — Disk file

  — Input or output device (such as a line printer)

  — Process (such as a requester)

  — Spooler collector

  — Special file name (such as a temporary disk file)

  For details on Guardian file names, see the *Guardian Procedure Calls Reference Manual*.

An OSS file name identifies one of these:

— Disk file

— Special file name (such as a temporary disk file)

For details on OSS file names, see Files in the OSS Environment (page 723)Files in the OSS Environment.

- Special names for operating system files

  A special name is a substitute (in the source program) for a certain operating system file:

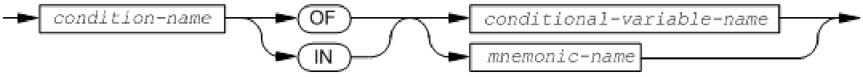
	Operating System File	
Special Name	**Guardian**	**OSS**
#IN	The file named in the IN parameter of startup message of current process.	The default input device (FD 0)—do not use it in SELECT clauses or the SPECIAL-NAMES paragraph as you can in the Guardian environment.
#OUT	The file named in the OUT parameter of startup message of current process.	The default output device (FD 1)—do not use it in SELECT clauses or the SPECIAL-NAMES paragraph as you can in the Guardian environment.
#TERM	Home terminal of current process.	
#TEMP	Temporary disk file on default volume. Created during execution of an OPEN statement, purged during execution of a CLOSE statement. You cannot specify a volume name for #TEMP. The volume used for #TEMP is the current default volume. If you want a temporary file on another volume, either specify only the volume name or specify a TACL ASSIGN command with only the volume name, for example: `ASSIGN a-file, $vol`	
#DYNAMIC	File name specified with the run-time library routine COBOLASSIGN or COBOL_ASSIGN_ during the execution of the current process. Legal only in the SELECT clause of a file-control entry. If you use #DYNAMIC as the file name in an ASSIGN command, TACL accepts it, but does not make the file dynamically assignable, and the file becomes unusable because it has an invalid name.	

For more information on spooler collectors, see the *Spooler Utilities Reference Manual*.

Qualified Names

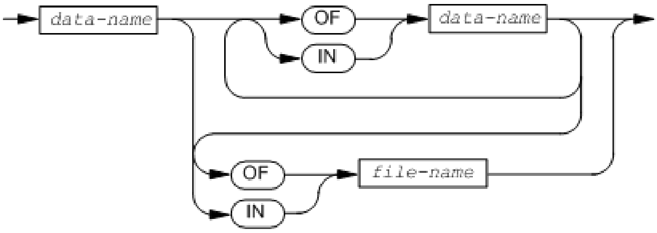
Every name that you reference in a COBOL program must be unique, either because no other name has the same spelling (including hyphenation) or because the name is part of a hierarchy of names (such as a data-name defined within a record). In the latter case, you can qualify the name with one or more of the higher-level names. You must specify enough higher-level names to make the name unique, but you need not specify all levels. The most significant name in a hierarchy must have unique spelling, because you cannot qualify it.

Qualified Condition-Name:



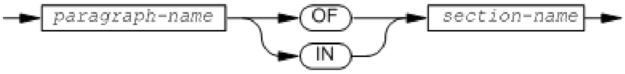
VST001.vsd

Qualified Data-Name:



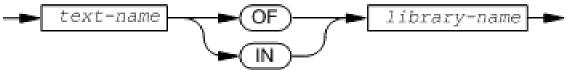
VST002.vsd

Qualified Paragraph-Name:



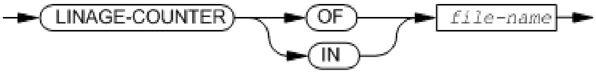
VST003.vsd

Qualified Text-Name:



VST004.vsd

Qualified LINAGE COUNTER:



VST005.vsd

Within the Data Division, you can use file-names from file description (FD) or sort-merge file description (SD) entries and data-names from data description entries for qualification. Within the levels of qualification, file-names (names associated with level indicators FD and SD) are most significant, then data-names for level-01 items, and then data names for level-02 items, and so on to level-49 items. The name of a conditional variable can qualify any of its condition-names.

These rules apply to qualification of names:

- Each qualifier must be at a higher level than the previous one and stay within the same structure of the name it qualifies.
- The same name cannot occur at different levels in a structure; otherwise, the name could qualify itself.
- A data-name used as a qualifier cannot be subscripted; all subscripts that apply to a qualified data-name appear after all qualifiers.
- A name can be qualified even though it does not need qualification. If there is more than one combination of qualifiers that make a name unique, then any one of them will do, including complete qualification (naming all qualifiers). HP COBOL permits a name to be qualified completely, with a name from every level of its structure.
- If a data-name or a condition-name is assigned to more than one element in a source program, the data-name or condition-name must be qualified each time it is referred to in the Procedure, Environment, and Data divisions (except in the REDEFINES clause where, by context, qualification is unnecessary). The name of a data-item can be used as the lowest level qualifier for any of its associated condition-names.
- If a word defined as a status condition-name is assigned to more than one element in a source program, the condition-name must be qualified by the mnemonic-name of its external switch

each time it is mentioned in the Procedure Division. The qualification of status condition-names is an HP COBOL extension.

- A word can be defined as a paragraph-name more than once in a source program. If more than one paragraph in the same section of a source program has the same name, no statement anywhere in the program can refer to any of those paragraphs by name.

  If two or more sections of a source program each contains a paragraph with the same paragraph-name, unqualified references to that paragraph-name are acceptable only within such sections—all other references must be qualified. For example, suppose sections S1 and S2 contain paragraphs named P. All references to P in other sections must be qualified as P OF S1 or P OF S2. Without such qualification, they cause the compiler to report an ambiguous reference. Unqualified references made within S1 to P are assumed to refer to P OF S1, and unqualified references made within S2 to P are assumed to refer to P OF S2. Statements in S1 can refer to P OF S2 and statements in S2 can refer to P OF S1.

  The word SECTION is not part of a section-name when used to qualify a paragraph-name.

- If a text-name is not a member of the default COPY library, it must be qualified by its library-name each time a COPY statement mentions it.
- If more than one file description entry in a program contains a LINAGE clause, the special register LINAGE-COUNTER must be qualified by its file-name each time the Procedure Division mentions it.

In Example 4-1, all data-names except PREFIX are unique.

Example 4-1 Qualified Names

```
FD  TRANSACTION-FILE
       ...
   01  TRANSACTION-REC.
       03 ITEM-NO.
          05 PREFIX X(2).
          05 CODE   X(3).
       03 QUANTITY  S9(8).
   ...
   01  MASTER-REC.
       03 CODE-NO.
          05 PREFIX X(2).
          05 SUFFIX X(3).
       03 DESCRIPTION X(70).
```

Qualification is necessary to refer to either PREFIX item. For example, any of these sentences would move the contents of one PREFIX to the other PREFIX:

```
MOVE PREFIX OF ITEM-NO          TO PREFIX OF CODE-NO.
MOVE PREFIX IN ITEM-NO          TO PREFIX OF MASTER-REC.
MOVE PREFIX OF TRANSACTION-REC  TO PREFIX IN CODE-NO.
```

It is possible to define a collection of program elements such that even complete qualification fails to establish uniqueness for some of their names. In this case, your program can never reference these elements. To avoid this situation, follow these rules:

- Do not define two identical data-names subordinate to a data structure unless their references can be made unique through qualification by intermediate data structure names.
- Do not define two identical record-names subordinate to any file-name. Independent data items (level-77 items) and record items that are not files must have unique names if they are to be referenced.
- Do not associate two identical condition-names with the same data item. If a word used as a condition-name is also used to identify another program element, then it can be referenced if and only if the associated data item can be referenced.

- Do not define two identical status condition-names within the same system-name clause of the SPECIAL-NAMES paragraph. If a word used as a condition-name in such a clause is also used to identify another program element, then the clause must also include a unique mnemonic-name to serve as a reference qualifier.
- Do not put two identical paragraph-names in the same section. If a word used as a paragraph-name is also used to identify another program element, then it can be referenced if and only if its section can be referenced. Sections must have unique names if they are to be referenced.
- Do not locate two identical text-names in the same COPY library.

Literals

A literal is a character-string that has a value implied by either of these:

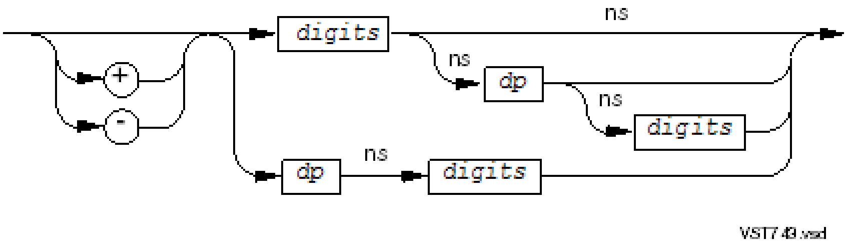
- An ordered set of characters that compose the literal
- A figurative constant (a special type of reserved word)

Topics:

- Numeric literals
  - Decimal Numeric Literals
  - Hexadecimal Numeric Literals
- Nonnumeric literals
  - Simple Nonnumeric Literals
  - Hexadecimal Nonnumeric Literals
- National Literals
- Figurative Constants

Decimal Numeric Literals

A decimal numeric literal is a character-string that has the value of the sequence of its digits.



digits

is a string of one to 18 digits. The total number of digits in a decimal numeric literal cannot exceed 18.

dp

is a decimal point. It is a period (".") unless the program contains the DECIMAL POINT COMMA phrase in the SPECIAL-NAMES paragraph, in which case it is a comma (",").

Decimal numeric literals follow these rules:

- When a decimal numeric literal appears in a context where its value is assumed to be a sequence of characters, its size is equal to the number of digits in its representation.
- A sign character (+ or -) has no effect on the size of a literal. Absence of a sign signifies a positive number.

- COBOL does not permit the decimal point as the rightmost (last) character; HP COBOL relaxes this restriction somewhat:
  - If the apparent last character of a numeric literal qualifies as a decimal point and the immediately following character is not a space (but is a semicolon, comma, or right parenthesis), the compiler interprets that last character as the decimal point.
  - If the apparent last character qualifies as a decimal point but the immediately following character is a space, the compiler interprets that last character and the space together as the separator that follows the literal.
- The compiler interprets a decimal point as an assumed decimal point; that is, the character is not present in the value even if the context of the literal implies that the value is represented as a sequence of characters.
- A decimal numeric literal that has no decimal point is an integer.

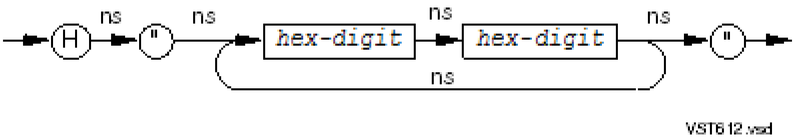
Example 4-2 Integer Decimal Numeric Literals

```
+601
34116
0
15
1234. ,    (an integer followed by a period separator and a comma
            separator)
```

Example 4-3 Noninteger Decimal Numeric Literals

```
+601.1
89.6
0.0051
-.1
1234.,    (a noninteger followed by a comma separator)
```

Hexadecimal Numeric Literals



hex-digit

is one of the characters *0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F, a, b, c, d, e, f*. The maximum number of *hex-digits* in a hexadecimal numeric literal is 16 (eight pairs).

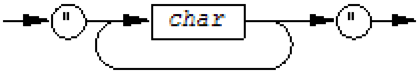
A hexadecimal numeric literal is considered an unsigned integer. It can appear anywhere that a decimal numeric literal can appear.

Example 4-4 Hexadecimal Numeric Literals

```
H"00"
H"0F"
H"0123456789ABCDEF"
H"1003c55b"
```

Simple Nonnumeric Literals

A simple nonnumeric literal is a character-string that has the value of the sequence of its characters.



VST744.vsd

char

is an alphanumeric character or a quotation mark ("). If *char* is a quotation mark, it must be immediately followed by another quotation mark. Each pair of quotation marks represents a single embedded quotation mark. A simple nonnumeric literal can have at most 160 characters, excluding the delimiting quotation marks.

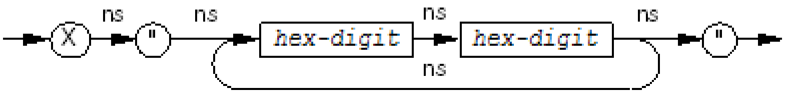
Simple nonnumeric literals follows these rules:

- The literal must be both preceded and followed by at least one separator.
- The delimiting quotation marks are part of the character-string that represents the literal; they are not part of the value of the literal.
- HP COBOL accepts only the double quotation mark ("), not the apostrophe ('), as a quotation mark.
- The literal does not include the second of two consecutive quotation marks that represent a single, embedded one.
- All other characters in the literal represent themselves as a part of the literal's value and have no other interpretation. Lowercase letters are not equivalent to their uppercase counterparts. Punctuation characters are not interpreted as separators.
- The value of a literal is the ordered sequence of characters in its representation (excluding the delimiting quotation marks and interpreting a pair of contained quotation marks as a single quotation mark character). The literal represents a data item of the alphanumeric category whose value is the value of the literal.

Example 4-5 Nonnumeric Literals

```
"THIS IS A NONNUMERIC LITERAL"
"This is ANOTHER one.  "" IS ONE EMBEDDED QUOTATION MARK"
"You don't need to double apostrophes."
". , ; : ( ) == are not separators in literals."
```

Hexadecimal Nonnumeric Literals



VST613.vsd

hex-digit

is one of the characters *0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F, a, b, c, d, e, f*. A hexadecimal nonnumeric literal can have at most 320 hexadecimal digits (160 pairs), excluding the delimiting quotation marks.

A hexadecimal nonnumeric literal can appear anywhere that a simple nonnumeric literal can appear.

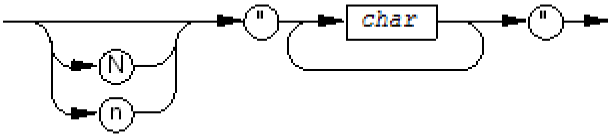
Example 4-6 Hexadecimal Nonnumeric Literals

```
X"00"
X"0F"
X"0123456789ABCDEF"
X"1003c55b"
```

National Literals

A national literal is used for those spoken languages not represented by roman letters and Arabic numbers. An example is the Japanese Kanji alphabet. To use national literals, you must have a special terminal and a special keyboard.

As with other types of literals, the character-string of a national literal has the value of the sequence of its characters. Each character is represented internally by 2 bytes.



VST745.vsd

char

is any character in the national character set, including a punctuation character. A national literal can have at most 160 characters, excluding the delimiting quotation marks.

National literals follow these rules:

- The letter *N* or *n* and the quotation marks are part of the character-string that represents the literal; they are not part of the value of the literal.
- HP COBOL accepts only the double quotation mark ("), not the apostrophe ('), as a quotation mark.
- The value of a national literal is the ordered sequence of characters in its representation, excluding the delimiting quotation marks. The maximum number of characters allowed on a line depends on the column in which the literal begins.
- The literal represents a data item of the national category whose value is the value of the literal.
- Punctuation characters appearing within a character-string that represents a national literal are components of its value and are never interpreted as separators. Each character is represented internally as 2 bytes.

In general, you can use a national literal anywhere you can use a nonnumeric literal. Exceptions are:

- A national literal cannot be compared to a nonnumeric or numeric literal or to a data item not defined as national. Compare national literals only to other national literals or national data items.
- A national literal cannot be specified in these paragraphs, statements, phrases, or clauses:
  - SPECIAL-NAMES paragraph
  - PADDING clause of the SELECT statement
  - RECEIVE-CONTROL paragraph
  - INITIALIZE statement when the REPLACING phrase is used
  - INSPECT statement
  - As *literal-1* or *literal-2* of the REPLACING phrase of a COPY statement
- If national literals and national data items are used for items in a STRING statement (*delim-1*, *delim-2*, *result*, or *delim-store*), all the items must be national literals or national data items.

Figurative Constants

A figurative constant is a character-string that has a value the compiler generates from one of the reserved words in the first column of Table 4-5. The value it generates depends on the context in which the figurative constant appears.

In general, you can use a figurative constant wherever the syntax rules of the language require or permit a literal. The exceptions to this rule are:

- When the literal must be a numeric literal, the only acceptable figurative constant is [ALL] ZERO[[E]S] (which generates the numeric value 0). The other forms of figurative constant always generate a nonnumeric value; therefore, such forms are not acceptable when the context requires a numeric value.
- There are certain contexts in which you cannot use figurative constants whose source form includes the word ALL. When this is the case, the usage considerations for that particular language element's syntax mention the usage restriction.
- The COBOL language includes several constructs where a literal cannot be any figurative constant. When this is the case, the usage considerations for that particular language element's syntax mention the usage restriction.
- When SPACE, QUOTE, or ZERO is applied to a national data item, the class of the figurative literal is national. The 2-byte value of the national character set that corresponds to SPACE, QUOTE, or ZERO when the program is compiled is used. For LOW-VALUES and HIGH-VALUES, the highest and lowest positions in the collating sequence are assumed.

When a figurative constant represents a nonnumeric or national value, that value is a string of one or more characters. The compiler determines the length of the string according to these rules:

- When you associate a figurative constant with a data item (for example, by moving the figurative constant to the data item or by comparing them or using the figurative constant in a VALUE clause), the compiler repeats the figurative constant value character by character until its size equals or exceeds the size of the data item. The compiler then truncates the resultant string from the right end until it has the same number of character positions as the associated data item. This happens prior to and independently of the application of any JUSTIFIED clause for the data item.
- When you use a figurative constant whose source form does not include the word ALL with a DISPLAY, STRING, STOP, or UNSTRING statement, the compiler generates exactly one character.
- When you use a figurative constant whose source form does include the word ALL, there are two possibilities:
  - If a nonnumeric or national literal follows the word ALL, the compiler generates a string having the length and value of the literal.
  - If a nonnumeric or national literal does not follow the word ALL, the compiler generates a string of exactly one instance of the character represented by the reserved word (figurative constant) or the symbolic character that follows the word ALL.
- The length of a figurative constant in a concatenation expression is one.

The result of associating a figurative constant that represents a nonnumeric value with a numeric or numeric-edited data item is defined only when the string the compiler generates from the figurative constant contains only digit characters. In this case, the generated nonnumeric value has the appearance of an integer whose length (in terms of the number of digits it contains) is the same as the length of the associated data item.

Table 4-5 Figurative Constants

Figurative Constant *	What It Represents
ZERO[[E]S]	One or more of the character zero (0), depending on the context
SPACE[S]	One or more spaces, depending on the context
HIGH-VALUE[S]	One or more of the character that has the highest position in the program collating sequence, except in the SPECIAL-NAMES paragraph, where it represents the character that has the highest position in the ASCII character set (the 256th character, which is all binary ones) or in the national character set (default is hexadecimal FFFF)
LOW-VALUE[S]	One or more of the character that has the lowest position in the program collating sequence, except in the SPECIAL-NAMES paragraph, where it represents the character that has the lowest position in the ASCII character set (the first character, the NUL, which is all binary zeros) or in the national character set (default is hexadecimal 0000)
QUOTE[S]	One or more of the character quotation mark (")You cannot use either of these words instead of quotation marks to enclose a nonnumeric literal.
symbolic-character	One or more of the character specified as the value of *symbolic-character* in the SYMBOLIC CHARACTERS clause of the SPECIAL-NAMES paragraph
ALL *literal*	The value of *literal* must contain one or more of these: • A nonnumeric literal • A national literal • A symbolic-character • One of the other reserved words previously defined as figurative constants (except that you cannot precede an ALL literal form of figurative constant with another ALL) When *literal* is a nonnumeric literal or a national literal, this form implies repetition of the literal's value to the extent required by the context. For the other cases, the word ALL is redundant and is used only for readability.

* Singular and plural forms are equivalent and can be used interchangeably.

PICTURE Character-Strings

PICTURE character-strings (character-strings in the PICTURE clause) use the COBOL character set as described in PICTURE Clause (page 202). The editing characters in Table 4-6 specify the editing operations that a process performs on data when storing it in data items. The compiler recognizes these characters as editing characters only within PICTURE clauses. In all other places, these characters follow the rules for character-strings and separators.

Table 4-6 PICTURE Character-String Editing Characters

Character	Editing Operation
B	Space insertion
Z	Zero suppression
0	Zero
+	Plus
-	Minus
CR	Credit
DB	Debit
*	Check protect
$	Dollar sign

Table 4-6 PICTURE Character-String Editing Characters *(continued)*

Character	Editing Operation
,	Comma or decimal point
.	Period or decimal point
/	Slash

Comments

A comment-entry or a comment line can contain any combination of characters in the computer's character set. Both types of comments are character-strings that serve only to document a program. A comment-entry follows a paragraph header in the Identification Division and can continue on additional lines. A comment line is any line anywhere in the source text that has a comment character in the indicator field.

5 Data Fundamentals

Every data item in a COBOL program has a level, class and category. Data items are organized into data structures. How data is represented depends on the machine on which it executes. Not all data storage locations in a COBOL program are the same size. When you direct a COBOL program to store a value of a given size at a location of a different size, the process extends or truncates the value according to a set of alignment rules. Alignment considerations sometimes cause the compiler to allocate unused bytes between data items, which are called implicit FILLER data. COBOL provides several ways to refer to individual data items in a program.

Topics:

- COBOL Character Set
- Data Structures
- Data Representation
- Data Alignment in Receiving Items
- Data Alignment in Memory
- Implicit FILLER Bytes
- References to Data Items

Data Levels, Classes, and Categories

COBOL has two data levels: elementary item and data structure. An elementary item cannot be subdivided; a data structure can be subdivided (see Data Structures).

Data items fall into the classes and categories shown in Table 5-1. The categories are used in PICTURE Clause (page 202).

Table 5-1 Data Levels, Classes, and Categories

Level of Item	Class	Category
Elementary	Alphabetic	Alphabetic
	Numeric	Numeric
	Alphanumeric	Alphanumeric Alphanumeric Edited Numeric Edited
	National	National
	Pointer	Pointer
Data structure	Alphanumeric	Alphanumeric

Data Structures

Data structures—records, files, and tables—are built from elementary items. Level-numbers show the relationship of elementary data items to data structures.

Although the structural forms are defined within a hierarchy, independent structures of any level are permitted. There is no requirement that all data be fully organized to the highest level. Furthermore, multiple structures can refer to the same or overlapping physical data.

Topics:

- Level-Numbers
- Records
- Files
- Tables

Level-Numbers

Level-numbers show the relationship of elementary data items to data structures.

Level-Number	Describes ...
01 through 49	A record (whose level-number starts at 01) and its subordinate data items (whose level-numbers are higher, but not necessarily successive).
66	An elementary item or a data structure introduced by a RENAMES clause. Use level-66 entries to regroup data structures.
77	An entry in the Working-Storage, Extended-Storage, or Linkage section that describes noncontiguous elementary items. These items are not subdivided and are not subdivisions of other items.
88	Condition-name entries, which define a conditional variable to be tested, including a value or range of values assigned to that variable.

A data structure includes all data items described after it and before the next item whose level-number is less than or equal to the level-number of that data structure. In Example 5-1, ADDRESS-RECORD includes everything before PERSONAL-RECORD, OFFICE-NUMBER includes everything before OFFICE-ADDRESS, and so on.

All elementary or data structures immediately subordinate to a given data structure must have identical level-numbers greater than that of the data structure. In Example 5-1, OFFICE-NUMBER and OFFICE-ADDRESS are immediately subordinate to ADDRESS-RECORD, DISTRICT and REGION are immediately subordinate to OFFICE-NUMBER, and so on.

Example 5-1 Level Numbers

```
 FD   BRANCH-OFFICE-FILE
      LABEL RECORDS ARE OMITTED
      RECORD CONTAINS 100 CHARACTERS
      DATA RECORDS ARE ADDRESS-RECORD,
      PERSONNEL-RECORD, MISC-RECORD.
*                          A record follows.
 01   ADDRESS-RECORD.
*                          A data structure follows.
      05   OFFICE-NUMBER.
*                          An elementary item follows.
           10   DISTRICT       PICTURE 99.
*                          Condition-names follow.
                88 NEW-YORK    VALUE 21.
                88 TAMPA       VALUE 43.
                88 OMAHA       VALUE 55.
                ...
           10   REGION         PICTURE 999.
      05   OFFICE-ADDRESS.
           10   STREET         PICTURE X(25).
           10   CITY           PICTURE X(15).
           10   STATE          PICTURE X(2).
           10   ZIP-CODE       PICTURE 9(9).
                ...
*                          A record follows.
 01   PERSONNEL-RECORD.
      05   OFFICE-MANAGER      PICTURE X(35).
      05   NO-OF-EMPLOYEES     PICTURE 9(4).
      05   TAX-GROUPS.
           10   HOURLY.
                15   PART-TIME  PICTURE 99.
                15   FULL-TIME  PICTURE 99.
           10   EXEMPT         PICTURE 9(4).
                ...
```

```
01  MISC-RECORD.
    ...
```

Records

A record is a sequence of character positions. It can be an elementary data item or a data structure. Its data description entry determines its internal structure (see PICTURE Clause (page 202)). Records can be of fixed length or variable length. COBOL programs manipulate logical records and physical records.

Most of the data in a typical COBOL program is in records. When a program reads data from or writes data to a file, the unit of transmission is the record.

Topics:

- Physical Records and Logical Records
- Record Elements
- Record Length

NOTE: In this manual, "record" means logical record unless "physical record" is specified.

Physical Records and Logical Records

COBOL programs manipulate physical records and logical records. A physical record is a physical unit of information whose size and recording mode is convenient to a particular computer for the storage of data on an input or output device (such as 80 characters for a terminal or 132 characters for a printer). The size of a physical record is hardware dependent and bears no direct relationship to the size of the file of information contained on a device. A physical record can contain one or several logical records, or a logical record can span several physical records.

A logical record is a group of related information, uniquely identifiable, and handled as a unit. The number of logical records that can exist in a file is a characteristic of its supporting storage medium, possibly modified by instructions presented to the file system during the file's creation. The logical records of a file have either a fixed size or a size that varies between a maximum and minimum size, depending on record type.

A logical record is defined by a set of data description entries. Each entry has a level-number followed by a data-name and possibly a series of independent clauses. The level-numbers form a structure, dividing a record into smaller and smaller parts.

Example 5-2 Logical Record

```
01  BIBLIOGRAPHY-RECORD.
    03 AUTHOR-NAME.
       05 LAST-NAME           PICTURE X(20).
       05 FIRST-NAME          PICTURE X(20).
    03 TITLE                  PICTURE X(50).
    03 PUBLICATION-INFO.
       05 PUBLISHER.
          07 PUB-NAME         PICTURE X(20).
          07 PUB-LOCATION     PICTURE X(20).
       05 PUBLICATION-YEAR    PICTURE 9999.
```

The concept of logical records also applies to data outside files. You can group data into logical records in all sections of the Data Division.

Once you describe the relationship between logical records and physical records, record manipulation is the responsibility of the HP COBOL or CRE run-time routines and the NonStop operating system.

Record Elements

In COBOL, the lowest subdivisions of a record (that is, those not further subdivided) are called elementary items. Consequently, a record is a series of elementary items, or the record itself can be an elementary item. In Example 5-2, the elementary items are:

- LAST-NAME
- FIRST-NAME
- TITLE
- PUB-NAME
- PUB-LOCATION
- PUBLICATION-YEAR

A data structure is a sequence of one or more elementary items that you can refer to by a group name. Data structures in turn can be combined to make other data structures. An elementary item, then, can belong to more than one data structure. A record is a data structure that does not belong to any larger data structure. In Example 5-2, the data structures that contain more than one elementary item are:

- AUTHOR-NAME
- PUBLICATION-INFO
- PUBLISHER
- BIBLIOGRAPHY-RECORD

Record Length

The length of records can be fixed or variable. You specify the length in the file description entry that precedes any data description entries for the records (see File Section (page 166)). If the file description entry includes a

```
RECORD CONTAINS rec-1 TO rec-2 CHARACTERS
```

or a

```
RECORD IS VARYING IN SIZE FROM rec-1 TO rec-2 CHARACTERS
```

clause, the record length is variable. If the file description entry has no RECORD CONTAINS clause, or if the clause is

```
RECORD CONTAINS n CHARACTERS
```

the record length is fixed.

When a file has the fixed-length record type attribute, every record contains the same number of character positions (bytes) and all input and output operations on the file process this fixed size. A program can specify more than one record description for the file, and some record descriptions can describe different numbers of character positions, but every record existing in the file still has the same fixed length.

When a file has the variable-length record type attribute, different records can have different numbers of character positions. In this case the input and output operations on the file process whatever size is associated with a particular logical record. After a record retrieval operation, the source program has sole responsibility for determining which of the possible record formats or lengths apply. The logical record length associated with a variable-length record file is therefore a maximum that records cannot exceed, not a constant to which all records conform.

The OCCURS clause with a DEPENDING ON phrase defines a variable-size table. For more information, see OCCURS Clause for Variable-Size Tables (page 224).

Files

A file is the highest structural form. Each file is a collection of records, ordered by the file's organization or by record attributes (keys), and maintained upon some storage medium. Use of the record structure is not limited to files; individual records can exist independently of files.

In COBOL, a file is a group of records. A file has both fixed attributes and dynamic attributes. Fixed attributes are determined when you create the file, and you cannot change them subsequently. Dynamic attributes can vary to some extent, depending on the logical specifications that source programs request when they access the file.

Some attributes are language-specific. Regardless of file system or operating system, COBOL files have a certain logical structure (organization and access mode). The operating system provides an additional attribute of a file: an exclusion mode when the file is open.

A COBOL program references a file by its file connector, which contains information about the file's attributes. One component of the file connector is the file position indicator, which determines which record is the next one to be sequentially accessed.

Topics:

- Organization
- Quotation Marks
- Open Mode
- Exclusion Mode
- File Connector
- File Position Indicator

Organization

Organization, which specifies a file's logical structure, is a fixed attribute that is established when you create the file and cannot be changed.

Table 5-2 File Organization

Organization	Description
Sequential	A sequential file is organized so that each record except the first has a unique predecessor record and each record except the last has a unique successor record. These relationships are established by the order in which the records are written and cannot be subsequently changed. The only alterations a program can make to an existing sequential file are record replacement (updating an existing record) and record creation (appending a new record following the last existing record). Under the NonStop operating system, COBOL sequential files are unstructured files or entry-sequenced files.
Line Sequential	A line sequential file (code 180) is a sequential file that is compatible with the system text editor of the OSS environment; therefore, it can also be called an OSS ASCII text file. A line sequential file differs from a sequential file in that each of its records ends with a line-feed character. For its other characteristics, see Line Sequential Files (page 726) Line Sequential Files. Under the NonStop operating system, COBOL line sequential files are line sequential files.
Relative	A relative file is organized as a sequence of record areas, each capable of holding a logical record. The successive record areas are uniquely identified by successive integer values (called relative record numbers), beginning with one for the first record area. A program can select any record area for an operation by providing the value of its relative record number. Because a program can fill or empty individual record areas independently, without regard to the presence or absence of logical records in any of the other areas, any combination of record areas can be full or empty at any given time in the file's existence. Under the NonStop operating system, COBOL relative files are relative files.

Table 5-2 File Organization *(continued)*

Organization	Description
Indexed	An indexed file is organized as a set of records uniquely identified by the values of their prime record key, a data item defined within each logical record of the file. A program can select any record for an operation by providing the value of its prime record key. Under the NonStop operating system, COBOL indexed files are key-sequenced files.
Queue	A queue file is an indexed file (and therefore, a key-sequenced file) that can function as a queue. Processes can queue and dequeue records in a queue file. Queue files contain variable-length records that are accessed by values in designated key fields. Unlike other key-sequenced files, queue files have prime keys but cannot have alternate keys. The prime key for a queue file includes an 8-byte timestamp; you can add a user key if desired. The disk process inserts the timestamp when each record is inserted into the file, and maintains the timestamp during subsequent file operations.

Access Mode

The access mode of a file specifies the manner in which the records of the file are to be manipulated: sequential, random, or dynamic. Access mode is a dynamic attribute, described in the COBOL source program that specifies the manner in which the object program unit operates upon records in the file. The access modes available to a file depend on its organization.

Table 5-3 Relationship Between File Organization and Access Mode

	Access Mode		
File Organization	**Sequential**	**Random**	**Dynamic**
Sequential	Program can read records in the order they were created, one after the other. Program can write records one after another. If the file has alternate keys, the program can read it in more than one sequential order.	Not available	Not available
Line sequential	Program can read records in the order they were created, one after the other. Program can write records one after another.	Not available	Not available
Relative	Program can select records in increasing order of relative record number (ignoring empty record areas).* Relative record numbers begin at 1.	Program can select records from anywhere in the file by specifying relative record numbers. Records need not exist or have contiguous record numbers.*	Program can select records sequentially (with READ NEXT statements) or randomly (with READ statements).*

Table 5-3 Relationship Between File Organization and Access Mode *(continued)*

	Access Mode		
File Organization	**Sequential**	**Random**	**Dynamic**
Indexed	Program can select records in increasing order of prime or alternate record key value (record keys are fields within each record). If records with duplicate alternate key values exist, the order of retrieval is either by prime key or by date of entry. To specify that retrieval order is by date of entry, set a file attribute with FUP.*	Program can select records from anywhere in the file.*	Program can select records sequentially (with READ NEXT statements) or randomly (with READ statements).*
Queue	Program can select records in increasing order of prime record key value (record keys are fields within each record). To specify that retrieval order is by date of entry, set a file attribute with FUP.	Program can select records from anywhere in the file.	Program can select records sequentially (with READ NEXT statements) or randomly (with READ statements).

* If the file has alternate keys, this characteristic also applies to the alternate keys. For a description of Tandem reference format, see Reference Format for Source Program Lines.

Open Mode

The open mode is a dynamic attribute of a file connector; it controls which file operations are permitted. In HP COBOL the status of a file connector always includes an open mode.

Table 5-4 File Open Modes

Mode	File Operations Allowed	Statements Allowed
Locked	None	UNLOCKFILE
Closed	Execution of OPEN statement to associate file connector with its physical file	OPEN
Input	Record retrieval operations	READ CLOSE
Output	Record creation operations (deletes existing records upon opening file)	WRITE CLOSE
Extend	Record creation operations (retains existing records upon opening file)	WRITE CLOSE
I-O	Record retrieval, creation, deletion, and replacement operations (as allowed by file organization and access modes)	READ WRITE REWRITE DELETE* CLOSE

* Relative or indexed file only

A file connector that is open in any mode can be closed by a CLOSE statement, which dissociates the file connector from the physical file and sets the open mode state to Closed or Locked (see CLOSE (page 315)).

An internal file connector that is open in any mode can be implicitly closed by a CANCEL statement (see CANCEL (page 312)).

Exclusion Mode

In HP COBOL, you specify a file's exclusion mode when you open it (see OPEN (page 385)).

Table 5-5 Exclusion Modes

Mode	Other Processes Can Read File	Other Processes Can Write to File
Shared	Yes	Yes
Protected	Yes	No
Exclusive	No	No

File Connector

A COBOL program makes reference to files indirectly. It does not refer to a file by the file's Guardian file name. Instead, it uses an ASSIGN clause in a file-control entry in the Input-Output Section of the Environment Division (possibly overridden at execution time by command interpreter commands) to associate the Guardian file name with a file connector, which is referred to throughout the remainder of the COBOL program through a COBOL file name.

The file connector is an entity that exists at execution time and contains information about the file, such as its open mode, position, and so on. A file connector can be internal or external to a given COBOL program. Unless a file connector is explicitly described as external, it is internal.

An internal file connector is associated with and only accessible to the specific program that describes it or to programs contained within that program. An external file connector is associated with the run unit and accessible to every program in the run unit that describes the file connector.

The execution of an OPEN statement associates a physical file with a file connector. HP COBOL permits the simultaneous association of one physical file with more than one file connector in a run unit in certain cases.

The execution of a CLOSE statement dissociates a physical file from a file connector.

In this manual, "file" is often used as an abbreviation for "file connector" and "file name" as an abbreviation for "file referenced by file name." The context determines the meaning in these cases.

File Position Indicator

The file position indicator is a conceptual component of a file connector, and its setting is a dynamic attribute of the connector. During sequential record retrieval operations, the setting of the file position indicator determines precisely which record is the next one to be accessed.

Normally, the setting reflects:

- A record number for a sequential or line sequential file
- A relative record number for a relative file
- A prime record key value for an indexed or queue file
- An alternate record key value for a file with any organization

Sometimes its setting indicates that:

- The at-end condition exists (due to a prior unsuccessful execution of a READ statement for the file).
- No valid next record has been established.
- An optional file is not present.

The setting of the file position indicator is irrelevant when the open mode is Output or Extend or the access mode is Random, because no sequential record retrieval operations are permitted in these cases.

Tables

NOTE: This topic applies to tables in the Data Division, not to SQL/MP or SQL/MX tables.

You can define a table by including an OCCURS clause in a data description entry. This clause specifies that the data item be repeated a stated number of times. The item is a table element, and the item's name and description apply to each repetition of the element.

A table is a data structure composed of one or more occurrences of a specified data item. The repeated data item is called a table element. The number of occurrences of a table element can be fixed or variable. If the element is a table itself, or if it contains other tables, the table to which the element belongs is multidimensional.

Because table elements do not have individual names, you must reference a table element by the table name and its position in the table. Two methods for giving the position number are subscripting and indexing. For information on subscripting, see Subscripts.

Topics explain how to declare:

- One-Dimensional Tables With Fixed Number of Elements
- One-Dimensional Tables With Variable Number of Elements
- Multidimensional Tables

One-Dimensional Tables With Fixed Number of Elements

You can define a table by including an OCCURS clause in a data description entry. This clause specifies that the data item be repeated a stated number of times. The item is a table element, and the item's name and description apply to each repetition of the element. For example, this entry defines a 1-dimensional table:

```
02  TOTAL  OCCURS 20 TIMES ...
```

Each reference to TOTAL must have exactly one subscript (except in SEARCH statements and some intrinsic functions, which do not allow subscripts).

Example 5-3 One-Dimensional Table With Fixed Number of Elements

```
WORKING-STORAGE SECTION.
 01  MONTH-NAME-TABLE.
     05  FILLER   PICTURE X(9)  VALUE "January".
     05  FILLER   PICTURE X(9)  VALUE "February".
     05  FILLER   PICTURE X(9)  VALUE "March".
     05  FILLER   PICTURE X(9)  VALUE "April".
     05  FILLER   PICTURE X(9)  VALUE "May".
     05  FILLER   PICTURE X(9)  VALUE "June".
     05  FILLER   PICTURE X(9)  VALUE "July".
     05  FILLER   PICTURE X(9)  VALUE "August".
     05  FILLER   PICTURE X(9)  VALUE "September".
     05  FILLER   PICTURE X(9)  VALUE "October".
     05  FILLER   PICTURE X(9)  VALUE "November".
     05  FILLER   PICTURE X(9)  VALUE "December".
 01  MONTH-NAMES REDEFINES MONTH-NAME-TABLE.
     05  MONTH-NAME  OCCURS 12 TIMES   PICTURE X(9).
```

One-Dimensional Tables With Variable Number of Elements

You can specify a fixed or variable number of occurrences of a table element. An element of a table of the latter form is called a variable-occurrence data item. Because the number of elements in such a table can vary during the execution of a program, the size of each data structure that contains the table can also vary. Such data structures are said to have a variable size.

Example 5-4 One-Dimensional Table With Variable Number of Elements

```
WORKING-STORAGE SECTION.
 01  ACTIVITY-TABLE-RECORD.
     03  ACTIVITY-COUNT PICTURE 99.
     03  ACTIVITY-TABLE OCCURS 10 TO 20 TIMES
                        DEPENDING ON ACTIVITY-COUNT
                        INDEXED BY SAVE-INX-1
                                   SAVE-INX-2.
         05  ACTIVITY-ENTRY  PICTURE 999.
```

Multidimensional Tables

The elements of a table can be elementary items or groups of subordinate structures, some of which can also be tables. In Example 5-5, TOTAL-B is a table subordinate to the 1-dimensional table named TOTAL. This means TOTAL-B is a 2-dimensional table; each reference to TOTAL-B must have exactly two subscripts (except in SEARCH statements and some intrinsic functions, which do not allow subscripts). The first subscript specifies the element of the TOTAL table, and the second subscript specifies the element of the TOTAL-B table within that element of TOTAL.

Example 5-5 Multidimensional Table

```
02 TOTAL  OCCURS 20 TIMES.
   03 TOTAL-A ...
   03 TOTAL-B  OCCURS 3 TIMES ...
```

A COBOL table can have a maximum of 7 dimensions. If the description of a data item T-B-S subordinate to TOTAL-B also has an OCCURS clause, T-B-S is a 3-dimensional table.

The outermost table of a multidimensional table can be of variable size, but each subordinate table must be of fixed size.

Data Representation

To define a language independent of computer designs, COBOL describes the structure and representation of data in terms of a standard data format. This format represents numbers as integers and represents noninteger and nonnumeric data as strings of characters. The amount of computer storage space occupied by data is measured in character positions. A data item's description determines how many character positions it has. How data is stored depends on the machine on which it executes.

A character position in an HP computer system is a byte: an 8-bit quantity. Data items represented as strings of characters occupy one byte per character. HP COBOL has other data formats that do not maintain the values of data items in characters but use forms more suitable for certain operations. In such cases, the number of bytes into which a data item fits can differ from the number of characters that express the value of the item.

Table 5-6 Allocation for COMPUTATIONAL Data Items

PICTURE Size in Digits	Byte Allocation
1 through 4	2 bytes
5 through 9	4 bytes
10 through 18	8 bytes

When an arithmetic or MOVE statement stores a value in a COMPUTATIONAL item, the number of digits stored equals the number of *9* s in the item's PICTURE clause. Because MOVE statements lack the SIZE ERROR phrase, they cannot detect the loss of high-order digits.

In general, the set of bytes the compiler allocates for a record is simply the contiguous sequence of bytes necessary to accommodate its constituent elementary data items. In some cases, however, alignment considerations cause the compiler to allocate unused bytes between data items. These bytes are called implicit FILLER bytes (see Implicit FILLER Bytes).

Data Alignment in Receiving Items

Not all data storage locations in a COBOL program are the same size. When you direct a COBOL program to store a value of a given size at a location of a different size, the process extends or truncates the value according to alignment rules:

Category of Receiving Item	Standard Alignment Rule
Numeric	If the receiving data item is numeric, data is aligned by decimal point and zero-filled or truncated on either end of each value, as required. When you do not specify a decimal point in the data item's description, the decimal point is assumed to be immediately after the rightmost character position.
Numeric Edited	If the receiving item is numeric edited, data is aligned by decimal point and zero-filled or truncated on either end of each value, as required, except where editing would replace leading zeros.
Alphanumeric Alphanumeric Edited Alphabetic National	If the receiving data item is alphanumeric, alphanumeric edited, alphabetic, or national, data is aligned at the leftmost character position and space-filled or truncated to the right of each value, as required.

Standard data representation and alignment rules are not always appropriate, so these clauses and directive exist to override them:

Clause or Directive	Effect
JUSTIFIED clause	Right-justifies an alphanumeric data within a receiving data item (see JUSTIFIED Clause (page 228))
SYNCHRONIZED clause	Aligns an elementary data item on the most natural computer storage boundary (see SYNCHRONIZED Clause (page 227))
PORT directive	Aligns BINARY/COMPUTATIONAL data items on byte boundaries unless the SYNCHRONIZED clause applies to them, in which case standard alignment rules apply (see PORT and NOPORT (page 569))

Data Alignment in Memory

How a data item is aligned in memory is determined by:

- Its USAGE clause
- Whether its description includes the SYNCHRONIZED clause
- Whether the program is compiled with the PORT directive
- The machine on which the program executes

For efficiency, a data item that is described as USAGE BINARY or USAGE COMPUTATIONAL is aligned on a 2-byte boundary.

To improve the alignment of its subordinate items, a level-01 (record) or level-77 data item is aligned on an 8-byte boundary.

For TNS processes, a word is 2 bytes (16 bits), so a data item that is aligned on a word boundary is aligned on a byte position that is divisible by 2. For TNS/E processes, a word is 4 bytes (32 bits), so a data item that is aligned on a word boundary is aligned on a byte position that is divisible by 4.

The compiler aligns a data item according to its size if one of these conditions is true:

- The data item is described as USAGE BINARY or USAGE COMPUTATIONAL and the PORT directive is not specified.
- The data item is described as USAGE NATIVE-*n*.
- The data item is described with a SYNCHRONIZED clause.

A data item that is aligned according to its size is usually aligned on a character position that is divisible by 2. This is not a TNS/E word size, but this alignment preserves data compatibility.

If the PORT directive is specified, BINARY/COMPUTATIONAL data items are not aligned on byte boundaries, and program execution can be much slower. The PORT directive does not affect NATIVE-*n* data items; they are always aligned on byte boundaries.

If the PORT directive is not specified, then a data item is aligned on a 2-byte boundary unless it is described with the SYNCHRONIZED clause. If it is described with the SYNCHRONIZED clause, it is aligned on a 2-byte boundary if its size is less than 4 bytes, and on a 4-byte boundary if its size is 4 bytes or larger.

Within a data structure, the compiler might align BINARY/COMPUTATIONAL data items by inserting implicit FILLER bytes.

More information:

Topics	Sources
USAGE clause	USAGE Clause
SYNCHRONIZED clause	SYNCHRONIZED Clause
PORT directive	PORT and NOPORT
Implicit filler bytes	Implicit FILLER Bytes

Implicit FILLER Bytes

When an odd number of character positions precedes a 2-byte-aligned item within a record, the compiler inserts FILLER bytes before the item, completing allocation of the preceding 2 bytes.

When the number of character positions preceding a 4-byte-aligned item within a record is not a multiple of 4, the compiler inserts the number of FILLER bytes needed to complete allocation of the preceding 4 bytes.

These extra bytes are not part of the data item. If a data structure contains two items separated by implicit FILLER bytes, then these bytes are a part of that data structure; however, a data structure always begins with the first character position of its first elementary item, ignoring any FILLER bytes that were generated to align that item properly. The initial character positions of a data structure are never implicit FILLER bytes.

Topics:

- Records
- Tables
- REDEFINES Clause

Records

When a record contains implicit FILLER bytes, their character positions are included in the record's allocation requirements, and they occupy space in external representations of the record.

Tables

When an elementary data item is described with an OCCURS clause, is subordinate to a data structure described with an OCCURS clause, or both, all occurrences of the data item must be aligned uniformly:

1. The first occurrence of the item is aligned to the required storage boundary. If the elementary item also begins a containing table's first occurrence, that table's first occurrence is defined to begin at the first character position of the item.
2. When the aligned item is itself a table, the first occurrence ends on the appropriate storage boundary and the remaining occurrences follow without additional FILLER bytes.
3. When the aligned item (or table of aligned items) belongs to a higher-level table, further adjustment can be necessary.

   If the elementary item is 2-byte-aligned and the containing group occurrence consists of an odd number of character positions, the compiler inserts one byte of FILLER after each group occurrence.

The preceding steps are repeated for each higher-level table.

FILLER bytes are not part of the containing occurrences themselves, but are included in data structures that contain the complete table.

REDEFINES Clause

When a data structure that is the object of a REDEFINES clause contains implicit FILLER bytes, their character positions are included in the character positions redefined.

Automatic or requested alignment of data items described by redefinition of a record's character positions (through use of the REDEFINES clause) follows the rules described previously. When the first data item allocated by a redefinition requires 2-byte or 4-byte alignment, the data item being redefined must begin on the appropriate boundary within its record. HP COBOL does not permit redefinitions that require insertion of implicit FILLER bytes before the first data item of the redefinition. Any bytes inserted at other places within the redefinition are counted when determining its size.

In Example 5-6, MASTER appears to occupy this many bytes:

```
(((2+1) * 5+1 ) * 5+1) * 5 = 405 bytes
```

but it actually occupies this many bytes

```
(((2+1+1) * 5+1+1) * 5+1+1) * 5 = 560 bytes
```

due to the alignment requirement for the COMPUTATIONAL item.

Example 5-6 REDEFINES Clause

```
01   MASTER.
     02   TABLE-1 OCCURS 5 TIMES.
          03   TABLE-2 OCCURS 5 TIMES.
               04   TABLE-3  OCCURS 5 TIMES.
                    05   ITEM-A    PIC 99 USAGE IS COMP.
                    05   ITEM-B    PIC X.
               04   ITEM-3         PIC X.
          03   ITEM-2              PIC X.
```

References to Data Items

To refer to a data item, a statement in a COBOL program must contain a reference that uniquely identifies that data item. In some cases, data items do not have unique names; for example:

- All elements of a table share a single name.
- Items that have the same name can occur in different records.

To make references unique, you can use qualifiers, subscripts, and reference modifiers. A data-name made unique by a combination of qualifiers, subscripts, and reference modifiers is called an identifier.

If an item in a record is tested frequently by a program, assigning a condition-name to the item is a convenient way to refer to the item and show the significance of the item's value. Assigning a condition-name to the item is also good programming practice.

Topics:

- Qualifiers
- Subscripts
- Reference Modifiers
- Identifiers
- Condition-Names

Qualifiers

When the name of a data element exists within a hierarchy of names, you must be able to make references to the name unique by mentioning one or more names defined at higher levels of the hierarchy. The higher-level names are called qualifiers. Although you must provide enough qualifiers to make the name unique, you need not include all levels. For more information on qualifiers, see Qualified Names (page 77).

Subscripts

A subscript, or set of subscripts, identifies an element in a table (created by an OCCURS clause). Whenever a statement in the Procedure Division refers to an individual table element or any item subordinate to a table element, that reference must include a subscript.

If the data item belongs to a table nested within one or more other tables (because more than one OCCURS clause applies to the data-name), a reference to the data item must include a subscript corresponding to each table (except in SEARCH statements and some intrinsic functions, which do not allow subscripts).

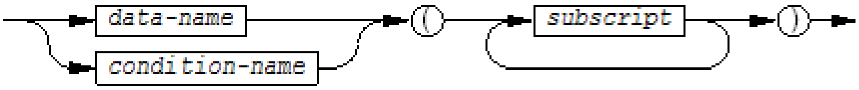
If a condition-name is associated with a data item for which subscripts are required, a reference to that condition-name must also include appropriate subscripts.

NOTE: A program that executes on TNS or TNS/R processors can use indexes instead of subscripts, which might be more efficient. For more information, see the *COBOL Manual for TNS and TNS/R Programs*. For the ECOBOL compiler, there is no difference between indexes and subscripts.

Topics:

- Subscript Syntax
- Number and Range of Subscripts

Subscript Syntax



VST006.vsd

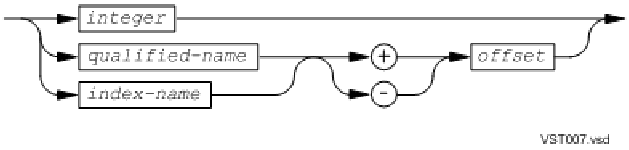
`data-name`

is either a data item described with an OCCURS clause or a data item subordinate to a data item described with an OCCURS clause. If `data-name` is qualified, the subscripts follow the qualifiers.

condition-name

is a level-88 item associated with either a data item described with an OCCURS clause or a data item subordinate to a data item described with an OCCURS clause. If *condition-name* is qualified, the subscripts follow the qualifiers.

subscript



integer

is a nonzero numeric literal. If it is signed, the sign must be positive.

qualified-name

is the identifier of an integer numeric data item. It can be qualified, but cannot have subscripts or reference modifiers.

index-name

is the *index* in the INDEXED phrase of the OCCURS clause that describes *data-name*. Its value is the occurrence number of an element in the associated table. The program must initialize the value of an *index-name* before using it in a subscript.

offset

is an unsigned integer numeric literal. Its value is added to (+) or subtracted from (-) the value of *qualified-name* or *index-name*.

Example 5-7 shows subscripting for two 1-dimensional tables and a 2-dimensional table.

Example 5-7 Subscripting for Tables

```
MOVE TOTAL OF REPORT-MARK (8) TO REPORT-TOTAL-8.
MOVE MONTH-NAME (MONTH-NUMBER + 2) TO REPORT-MONTH.
MOVE MATRIX (ROW COLUMN) TO OUTPUT-DISPLAY-LINE.
```

Number and Range of Subscripts

HP COBOL supports subscripting of up to 7 dimensions.

The lowest legitimate subscript value is 1, which selects the first element of a table. The next sequential elements of the table are selected by subscripts whose values are 2, 3, and so on. The highest subscript value, in any particular case, is the maximum number of elements in the table. Any higher subscript is erroneous, and can cause an error.

You can use the directive CHECK (page 550), with a level-number greater than 1, to include code in the resulting program to perform range checking during execution. If the program attempts to use a subscript that is out of range, the range checking routine reports a fatal error.

If you do not include a CHECK directive, the compilation produces a program in which subscript-out-of-range errors go undetected. These undetected errors can cause corruption of other data, producing errors that can be difficult to locate, even with a symbolic debugger.

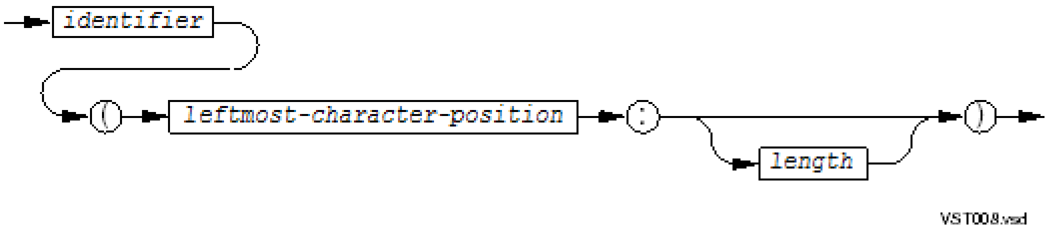
Reference Modifiers

Reference modifiers allow you to reference an arbitrary portion of a data item's value, which is important in some applications. With reference modifiers, you define an elementary item by specifying a leftmost character position within a data item and a length for the new item. The unique data item created by reference modifiers can be used not only as a sending item, but also as a receiving item—you can store a new value into it.

Topics:

- Reference Modifier Syntax
- Rules for Reference Modifiers

Reference Modifier Syntax



identifier

is the name of a data item with USAGE DISPLAY. If it is qualified or subscripted, the reference modifier appears after the qualifiers or subscripts.

leftmost-character-position

is an arithmetic expression. Its value must be a positive nonzero integer less than or equal to the number of characters in *identifier*; it represents the leftmost character of the portion of *identifier* you are selecting.

length

is an arithmetic expression. Its value must be a positive, nonzero integer; it represents the size of the portion of *identifier* you are selecting. The value of the expression

(*leftmost-character-position* + *length*) - 1

must be less than or equal to the number of characters in *identifier*.

If *length* is absent, the defined item begins with *leftmost-character-position* and ends with the last character of *identifier*; thus the length of the defined item is

(*data-name-length* - *leftmost-character-position*) + 1

where *data-name-length* is the length of *identifier*.

Rules for Reference Modifiers

These rules for reference modifiers apply directly if the data item referenced by *identifier* is described as alphabetic or alphanumeric.

If the data item is described as numeric, numeric edited, or alphanumeric edited, it is operated upon for purposes of reference modification as if it were redefined as an alphanumeric data item of the same size as the data item referenced by *identifier*. Any numeric item must be USAGE DISPLAY.

If the data item is described as national, the *leftmost-character-position* and *length* variables refer to the 2-byte pairs representing the characters, not to individual bytes. In all other respects, these rules for reference modifiers given apply directly:

- Reference modifiers for an operand are evaluated immediately after evaluation of any subscripts that are specified for that operand. If no subscripts are specified for the operand, the reference modifiers are evaluated at the time subscripting would be evaluated if subscripts had been specified.
- Each character of the data item referenced by *identifier* is assigned an ordinal number, incrementing by one from the leftmost position to the rightmost position. The leftmost position is assigned the ordinal number one. If *identifier* is described with a SIGN SEPARATE clause, the sign position within that data item is assigned an appropriate ordinal number.

- Reference modifiers create a unique data item, which is a substring of the data item referenced by *identifier*. The program handles this unique data item as an elementary data item without the JUSTIFIED clause.

  When *identifier* references an alphabetic data item, the unique data item has the class and category alphabetic.

  When *identifier* references a data item of any other category, the unique data item has the class and category alphanumeric.
- The unique data item created by reference modifiers can be used not only as a sending item, but also as a receiving item—you can store a new value into it.

Example 5-8 Reference Modifiers

A program contains these data descriptions:

```
01 SPREAD.
   03 LTH PIC 99 VALUE 10.
   03 MM.
      05 FF PIC A(10) VALUE "MARGARINE ".
      05 GG PIC $$9.99.
```

The same program contains these statements:

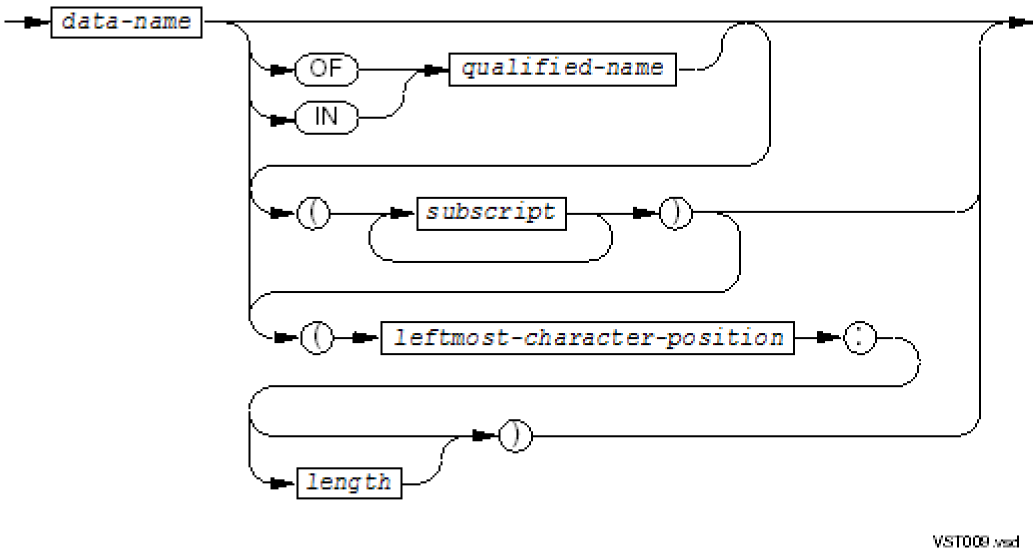
```
MOVE 3.15 TO GG
DISPLAY MM
DISPLAY MM (1:LTH)
MOVE "GONE" TO MM (11:)
DISPLAY MM
```

Executing the program produces these display:

```
MARGARINE
MARGARINE GONE
```

Identifiers

An identifier is a data-name made unique by a combination of qualifiers, subscripts, and reference modifiers.



data-name

is the name of a data item. If it has reference modifiers, it must be USAGE DISPLAY. If it is used as a subscript or qualifier itself, it can be qualified but not subscripted.

qualified-name

is defined in Qualified Names (page 77).

subscript

is defined in Subscript Syntax.

leftmost-character-position

is an arithmetic expression. Its value must be a positive nonzero integer less than or equal to the number of characters in *data-name*; it represents the leftmost character of the portion of *data-name* you are selecting.

length

is an arithmetic expression. Its value must be a positive, nonzero integer; it represents the size of the portion of *data-name* you are selecting. The value of the expression

```
(leftmost-character-position + length ) - 1
```

must be less than or equal to the number of characters in *data-name*.

If *length* is absent, the defined item begins with *leftmost-character-position* and ends with the last character of *data-name* ; thus the length of the defined item is

```
(data-name-length -leftmost-character-position ) + 1
```

where *data-name-length* is the length of *data-name*.

Example 5-9 Identifiers

```
UNIQUE-IDENTIFIER
ITEM-1 OF GROUP-A
ELEMENT OF NAME-TABLE OF MASTER-RECORD (LAST-ACCESSES)
PROD-NAME OF ITEM-X (ITEM-DEX) (1:15)
```

Condition-Names

Often an item in a record is tested frequently by a program. Assigning a condition-name to the item is a convenient way to refer to the item and show the significance of the item's value.

Every condition-name referred to in a COBOL program must be unique or capable of being made unique through qualifiers, subscripts, or a combination of qualifiers and subscripts. If you use qualifiers to make a condition-name unique, you can use the conditional variable as the first qualifier. You can also use the structure of names for the conditional variable as a qualifier. If references to a conditional variable require subscripting, then any of its condition-names also require subscripting.

Example 5-10 defines a condition-name for the conditional variable USE-CODE.

Example 5-10 Condition-Name

```
01 INVENTORY.
   02 PART-NUMBER OCCURS 100 TIMES.
      03 PREFIX              PICTURE 99.
      03 USE-CODE            PICTURE 9.
         88 RESTRICTED-USE              VALUE 1.
      03 SUPPLIER-SUFFIX     PICTURE 99.
```

This IF statement uses the condition-name RESTRICTED-USE to test the value of USE-CODE:

```
IF RESTRICTED-USE IN PART-NUMBER (30)
    PERFORM REPORT-VIOLATION
ELSE ...
```

Using condition-names also makes it easier to modify the program. Suppose the table definition in Example 5-10 changes so that both 1 and 2 mean RESTRICTED-USE. Without the use of a condition-name, the program must examine and possibly change each instance of the testing of the value of USE-CODE. With the condition-name, only the data description entry needs changing:

```
88 RESTRICTED-USE        VALUES ARE 1, 2.
    SET RESTRICTED-USE TO TRUE
```

The SET statement sets USE-CODE to the value 1, the first value in the list.

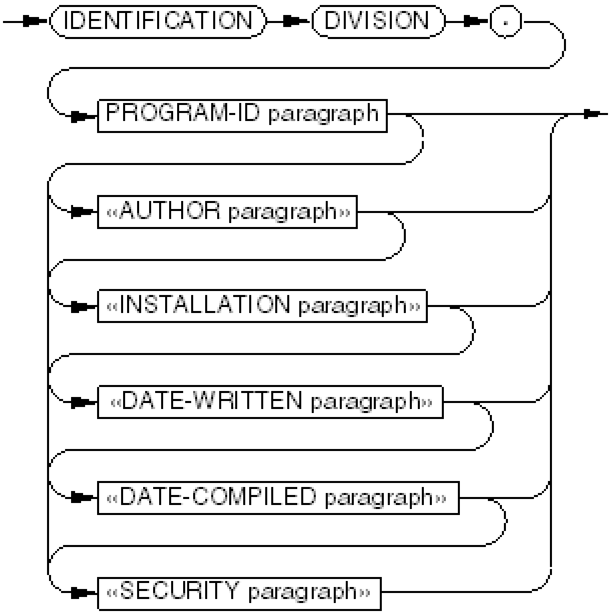
6 Identification Division

The Identification Division is required in a COBOL program. It has one required paragraph (PROGRAM-ID, which specifies the program name), and five optional paragraphs (which specify your name, the date, and the program purpose). The compiler handles the optional paragraphs as comments, except for the DATE-COMPILED paragraph, whose presence causes the compiler to report the compilation time and date in the listing.

Topics:

- Identification Division Syntax
- PROGRAM-ID Paragraph
- DATE-COMPILED Paragraph

Identification Division Syntax



VST010.vsd

IDENTIFICATION DIVISION.

is the division header. It must begin in area A, have a space in the indicator field, and be completely contained on a single program text line.

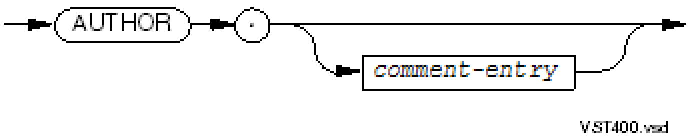
If the order of the paragraphs differs from that shown, the compiler prints a warning message. If any paragraph is repeated, the compiler prints an error message.

PROGRAM-ID paragraph

is defined under PROGRAM-ID Paragraph.

AUTHOR paragraph

NOTE: The 1985 COBOL standard classifies the AUTHOR paragraph as **obsolete**, so you are advised not to use it. Instead, use a comment, as in Example 6-2.



`comment-entry`

is any combination of characters from the ASCII character set.

The first keyword following a *`comment-entry`* must start on a program text line that has a space character in its indicator field. This keyword must start in area A and be preceded by space characters only.

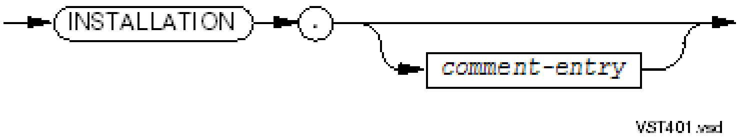
A *`comment-entry`* cannot be continued with the hyphen convention; however, it can be implicitly continued onto additional program text lines if area A of those lines contains space characters only.

The compiler ignores keywords and periods in *`comment-entry`*.

The compiler copies *`comment-entry`* to the listing as is, except the one in the DATE-COMPILED paragraph (see DATE-COMPILED Paragraph).

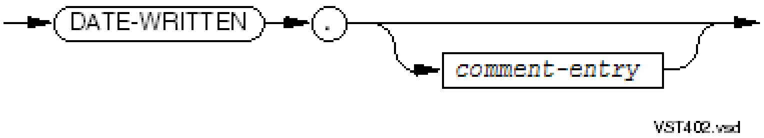
INSTALLATION paragraph

NOTE: The 1985 COBOL standard classifies the INSTALLATION paragraph as **obsolete**, so you are advised not to use it. Instead, use a comment, as in Example 6-2.



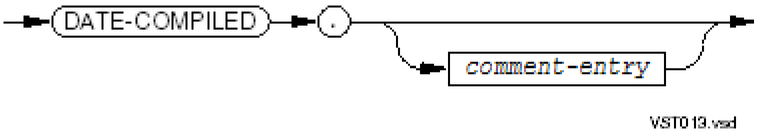
DATE-WRITTEN paragraph

NOTE: The 1985 COBOL standard classifies the DATE-WRITTEN paragraph as **obsolete**, so you are advised not to use it. Instead, use a comment, as in Example 6-2.



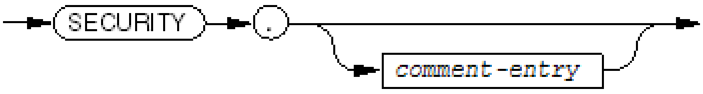
DATE-COMPILED paragraph

NOTE: The 1985 COBOL standard classifies the DATE-COMPILED paragraph as **obsolete**, so you are advised not to use it. Instead, use a comment, as in Example 6-2.



SECURITY paragraph

NOTE: The 1985 COBOL standard classifies the SECURITY paragraph as **obsolete**, so you are advised not to use it. Instead, use a comment, as in Example 6-2.



VST404.vsd

Example 6-1 Identification Division With Obsolete Paragraphs

```
IDENTIFICATION DIVISION.
  PROGRAM-ID. MYPROG
  AUTHOR. JANE DOE
  INSTALLATION. HEADQUARTERS
  DATE-WRITTEN. JANUARY 20, 1992
  DATE-COMPILED. JANUARY 21, 1992
  SECURITY. COMPANY CONFIDENTIAL
```

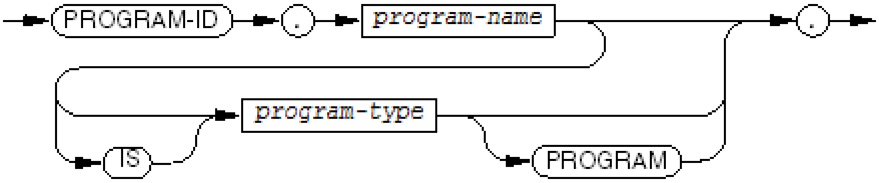
Example 6-2 Identification Division With Comments

```
  IDENTIFICATION DIVISION.
  PROGRAM-ID. MYPROG
* AUTHOR. JANE DOE
* INSTALLATION. HEADQUARTERS
* DATE-WRITTEN. JANUARY 20, 1992
* DATE-COMPILED. JANUARY 21, 1992
* SECURITY. COMPANY CONFIDENTIAL
```

PROGRAM-ID Paragraph

The required PROGRAM-ID paragraph names the COBOL program and (optionally) declares it to be an initial program, a common program, or both.

The name that the PROGRAM-ID paragraph gives the program can differ from the file-name of the source code file or that of the object file. It is the name by which the program is called by another COBOL program's CALL statement. You can also use it to qualify a file-name in the TACL command ASSIGN.

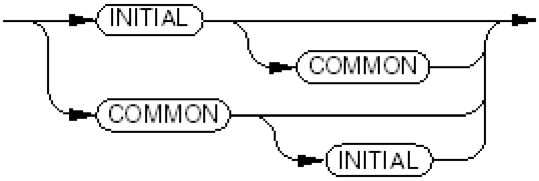


VST011.vsd

program-name

is a COBOL word. It names the program unit.

program-type



VST012.vsd

INITIAL

makes the program an initial program, which means that the program is in the initial program state each time it is executed—that is, all internal program entities (data item values, file connectors, and so on) are in their initial states at the start of each execution of the program.

COMMON

makes the program common, which means that the program must be contained within another program and can be called by any other program contained (directly or indirectly) in the same containing program, but not by the programs it contains.

Usage Considerations:

- A Program Can Be Both Initial and Common
- Storage Allocation For an Initial Program

  For an initial program, all data items are allocated dynamically.

  When a program declares very large data items that do not have initial values assigned, including a NOBLANK directive in the compilation saves significant amounts of compilation time, object program storage space on disk, and run-time initialization time. If you must initialize such items to spaces, use VALUE clauses, INITIALIZE statements, or MOVE statements of the form:

  ```
  MOVE ALL SPACES TO x.
  ```

- Qualification of a File-Name in an ASSIGN Command

  The *`program-name`* can also be used in the command interpreter ASSIGN command, where it serves to qualify a file-name. For example, suppose a COBOL program having the program-name “MAILING-LIST” contains the statement:

  ```
  SELECT IN-FILE  ASSIGN TO "$WEST.CALIF.LIVEONES"...
  ```

  You can reassign the IN-FILE of that program by issuing the command interpreter command

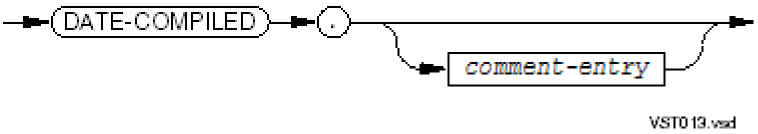
  ```
  ASSIGN MAILING-LIST.IN-FILE, $MIDWST.OKLA.PROSPCTS
  ```

  before executing the object program. See the *Guardian User’s Guide* for more information about the ASSIGN command.

DATE-COMPILED Paragraph

NOTE: The 1985 COBOL standard classifies the DATE-COMPILED paragraph as **obsolete**, so you are advised not to use it. Instead, use a comment, as in Example 6-2.

The DATE-COMPILED paragraph directs the compiler to report the compilation date and time in the listing, which the compiler already does by default.



DATE-COMPILED

must begin in area A, have a space in the indicator field, be completely contained on a single program text line, and end with a period followed by a space.

If you split the reserved word DATE-COMPILED across program lines, the compiler handles the paragraph as a simple comment-entry and does not replace its contents with the date and time of compilation.

comment-entry

is any combination of characters from the ASCII character set.

The first keyword following a *comment-entry* must start on a program text line that has a space character in its indicator field. This keyword must start in area A and be preceded by space characters only.

A *comment-entry* cannot be continued with the hyphen convention; however, it can be implicitly continued onto additional program text lines if area A of those lines contains space characters only.

Except for checking its syntax, the compiler ignores *comment-entry* and replaces it with the date and time of the compilation, which it prints on the listing in this format:

```
DATE-COMPILED.  yy/mm/dd - hh:mm:ss
```

where:

Number	Range	Represents
yy	00 through 99	Year
mm	01 through 12	Month
dd	01 through 31	Day
hh	00 through 23	Hour
ss	00 through 59	Second

Example 6-3 DATE-COMPILED Paragraph

Source text:

```
DATE-COMPILED.  This is the final version.
```

Listing text:

```
DATE-COMPILED.  92/01/08 - 14:45:00
```

7 Environment Division

The Environment Division is optional in a COBOL program. It has two optional sections, the Configuration Section and the Input-Output Section.

The Configuration Section states the type of computer on which to compile the program and the type of computer on which to run the program. This section can also:

- Define the association between program-supplied names and system-supplied facilities such as program switches or character sets
- Provide for substitution of the dollar sign ($) and decimal point (.) characters
- Adjust the character set to allow national alphabet extensions to pass the ALPHABETIC test

The Input-Output Section includes:

- File declarations
- File buffer allocations
- $RECEIVE supplementary declarations for interprocess communication

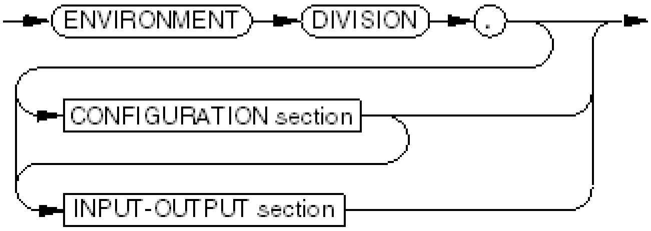
If the program does not need to describe anything previously listed and does not use files, it does not need the Environment Division.

If the program has the Environment Division and you move the program to an HP system from another type of system, you must change the information in the Environment Division.

Topics:

- Environment Division Syntax
- Configuration Section
- Input-Output Section

Environment Division Syntax



VST014.vsd

ENVIRONMENT DIVISION.

is the division header. It must begin in area A.

CONFIGURATION section

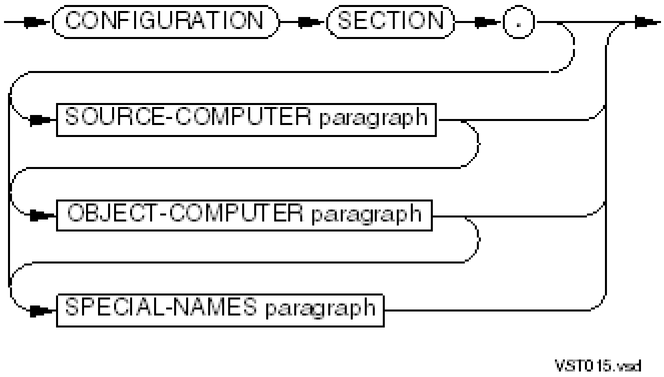
is defined in Configuration Section.

INPUT-OUTPUT section

is defined in Input-Output Section.

Configuration Section

The Configuration Section is forbidden in a contained program and is optional in any other program. The compiler prints a warning if a Configuration Section paragraph is out of order; it reports an error if a paragraph is repeated.



SOURCE-COMPUTER paragraph

is defined in SOURCE-COMPUTER Paragraph.

OBJECT-COMPUTER paragraph

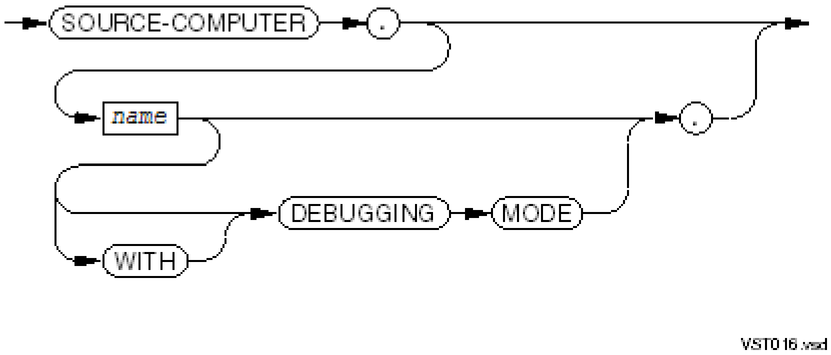
is defined in OBJECT-COMPUTER Paragraph.

SPECIAL-NAMES paragraph

is defined in SPECIAL-NAMES Paragraph.

SOURCE-COMPUTER Paragraph

The optional SOURCE-COMPUTER paragraph names the computer system on which the program is compiled and can specify that debugging lines be compiled.



`name`

is any sequence of ASCII characters except WITH, DEBUGGING, or MODE. It names the computer system on which the program is compiled.

DEBUGGING MODE

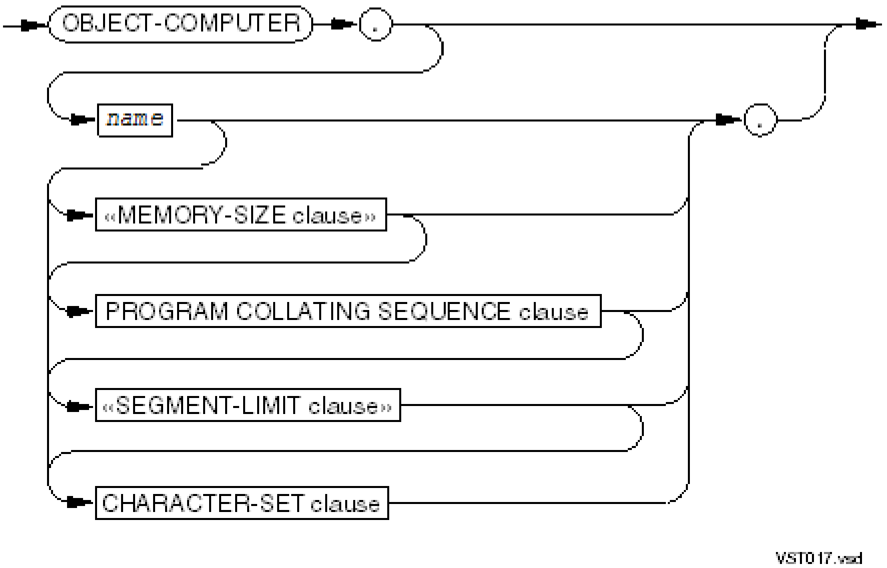
puts any source line that has `D` or `d` in its indicator field into the object program.

The process executes USE DEBUGGING sections only if the PARAM DEBUG ON command is active in the command interpreter environment. The process executes debugging lines whether the PARAM DEBUG ON command is active or not.

NOTE: The COBOL85 compiler also supports USE DEBUGGING declaratives, under the control of the DEBUGGING MODE phrase. The 1985 COBOL standard classifies USE DEBUGGING as **obsolete**, so you are advised not to use it The ECOBOL compiler does not recognize it.

OBJECT-COMPUTER Paragraph

The optional OBJECT-COMPUTER paragraph names the computer system on which the object program can run.



name

is any combination of character-strings except reserved words and separators (except period (.)). It is handled as a comment.

MEMORY-SIZE clause

NOTE: The 1985 COBOL standard classifies the MEMORY-SIZE clause as **obsolete**, so you are advised not to use it.

is checked for proper syntax but otherwise ignored. For more information, see MEMORY-SIZE Clause.

PROGRAM COLLATING SEQUENCE clause

specifies a collating sequence for use in nonnumeric comparisons. For more information, see PROGRAM COLLATING SEQUENCE Clause.

SEGMENT-LIMIT clause

NOTE: The 1985 COBOL standard classifies the SEGMENT-LIMIT clause as **obsolete**, so you are advised not to use it.

is checked for proper syntax but otherwise ignored. For more information, see SEGMENT-LIMIT Clause.

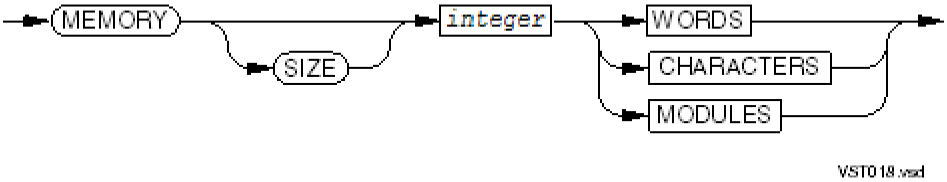
CHARACTER-SET clause

defines the ALPHABETIC class. For more information, see CHARACTER-SET Clause.

MEMORY-SIZE Clause

NOTE: The 1985 COBOL standard classifies the MEMORY-SIZE clause as **obsolete**, so you are advised not to use it.

HP COBOL ignores the MEMORY SIZE clause except to check its syntax.

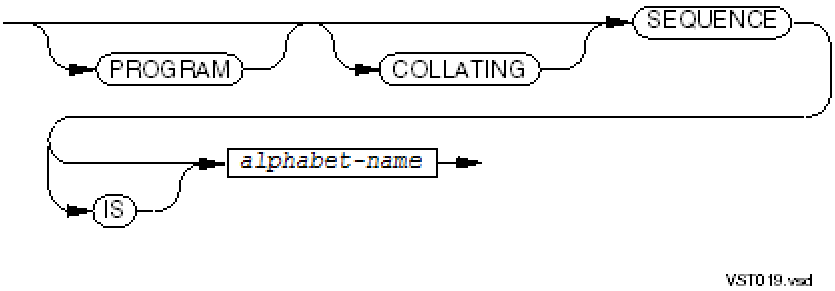


integer
must be in the range 1 through 32,767 but is otherwise ignored.

WORDS, CHARACTERS, MODULES
are ignored.

PROGRAM COLLATING SEQUENCE Clause

The PROGRAM COLLATING SEQUENCE clause enables you to specify an arbitrary collating sequence for use in nonnumeric comparisons and changes the values of the figurative constants HIGH-VALUE, HIGH-VALUES, LOW-VALUE, and LOW-VALUES. This clause has no effect on national data items, which are collated in ascending order of binary value (of the 2-byte pair representing the character).

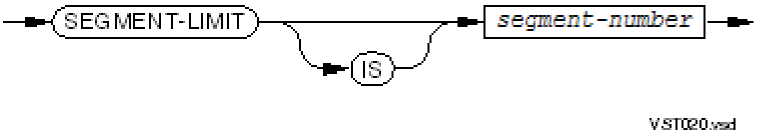


alphabet-name
is defined in an ALPHABET clause in the SPECIAL-NAMES paragraph. For more information, see ALPHABET Clause.

SEGMENT-LIMIT Clause

NOTE: The 1985 COBOL standard classifies the SEGMENT-LIMIT clause as **obsolete**, so you are advised not to use it.

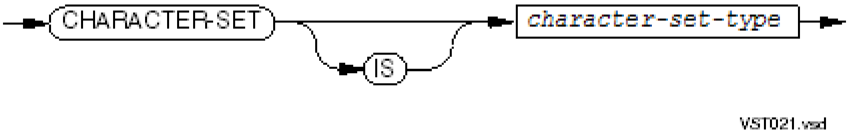
HP COBOL ignores the SEGMENT LIMIT clause except to check its syntax. All segment-numbers, from 01 to 49, are permanent.



segment-number
must be in the range 1 through 49 but is otherwise ignored.

CHARACTER-SET Clause

The CHARACTER-SET clause is an HP COBOL extension that enables you to specify the national character set defining the ALPHABETIC data class.



character-set-type

is a keyword in the left column of this table, representing an alphabet in the right column:

character-set-type	Alphabets
DANSK-NORSK	Danish, Norwegian
DEUTSCH	German
ESPANOL	Spanish
FRANCAIS-AZ	French (AZERTY keyboard)
FRANCAIS-QW	French (QWERTY keyboard)
SVENSK-SUOMI	Swedish, Finnish
UK	United Kingdom
USASCII	United States ASCII

In COBOL, this clause affects only the ALPHABETIC-UPPER, ALPHABETIC-LOWER, and ALPHABETIC tests. In SCREEN COBOL, the CHARACTER-SET IS clause has some additional semantics, including terminal configuration (see the *Pathway/TS SCREEN COBOL Reference Manual*).

The CHARACTER-SET clause does not define the character sets used for national literals or national data items.

The default character set is USASCII. The UK character set designator is handled as a comment, because it contains no extensions to the ALPHABETIC class. When the character set is USASCII (or UK), the ALPHABETIC class test work as Table 7-1 shows.

Table 7-1 How Alphabetic Class Tests Work With Default Character Set

Test	Verifies that each character is ...
ALPHABETIC	A space or a letter between *A* and *Z* or between *a* and *z*
ALPHABETIC-UPPER	A space or a letter between *A* and *Z*
ALPHABETIC-LOWER	A space or a letter between *a* and *z*

If you specify a *character-set-type* other than USASCII or UK, the ALPHABETIC class includes additional characters (see Table 7-2). The French character sets differ only in non-ALPHABETIC characters.

Table 7-2 Additional Characters Accepted by Alphabetic Class Tests With Nondefault Character Sets

	ASCII Character (Decimal Value and ASCII Graphic)									
Character Set	64 @	91 [	92 \	93]	94 ^	96	123 {	124 |	125 }	126 ~
DANSK-NORSK		U	U	U			L	L	L	L
DEUTSCH		U	U	U			L	L	L	L
ESPANOL			U					L		
FRANCAIS-QW	L		L				L	L	L	
FRANCAIS-AZ	L		L				L	L	L	
SVENSK-SUOMI	U	U	U	U	U	L	L	L	L	L

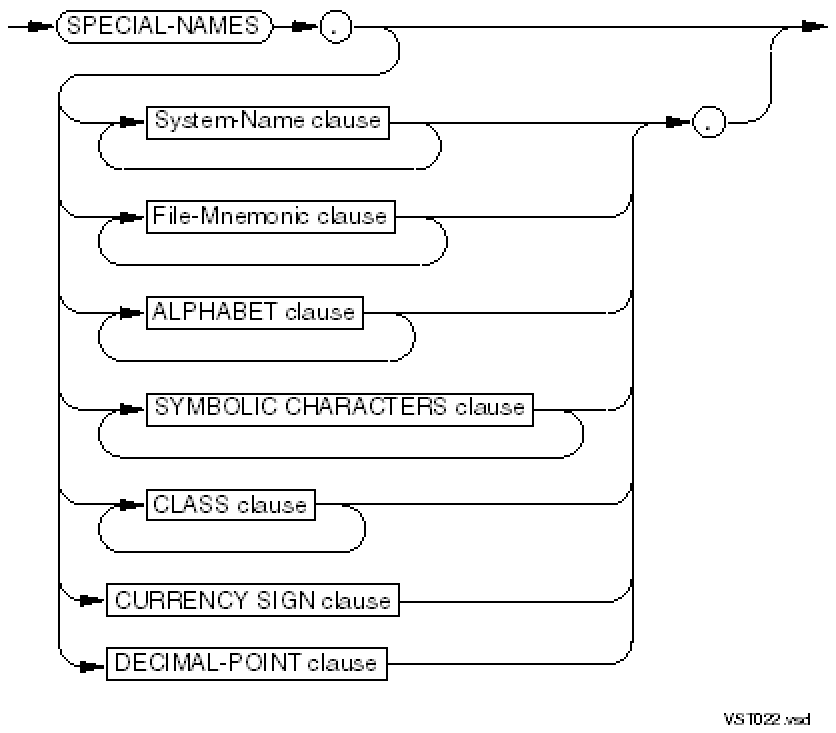
L = This character is ALPHABETIC and ALPHABETIC-LOWER.

U = This character is ALPHABETIC and ALPHABETIC-UPPER.

SPECIAL-NAMES Paragraph

The optional SPECIAL-NAMES paragraph assigns names of your choice to certain system-names. In addition, you can define a currency sign other than the dollar sign ($), and you can exchange the function of commas and periods in PICTURE character-strings and numeric literals.

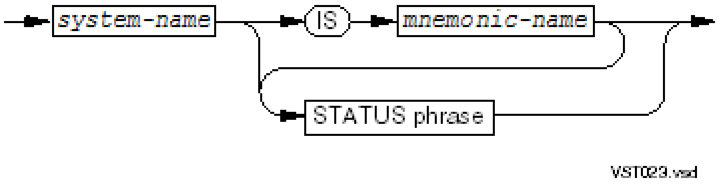
National literals and national data items cannot be used in the SPECIAL-NAMES paragraph.



Topics:

- System-Name Clause
- File-Mnemonic Clause
- ALPHABET Clause
- SYMBOLIC CHARACTERS Clause
- CLASS Clause
- CURRENCY SIGN Clause
- DECIMAL-POINT Clause

System-Name Clause



system-name

is the name of a hardware device, carriage-control tape channel, or external switch.

If the System Name clause has no STATUS phrase, `system-name` must be one of these values:

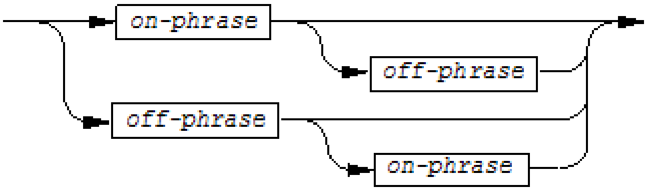
Value	Meaning
CONSOLE	Operator console
MYTERM	Process's home terminal
CHANNEL-1 through CHANNEL-12	Carriage-control tape channel

NOTE: The operator console that CONSOLE specifies is $0, an output-only device. If you post an ACCEPT to this device, a run-time error occurs.

mnemonic-name

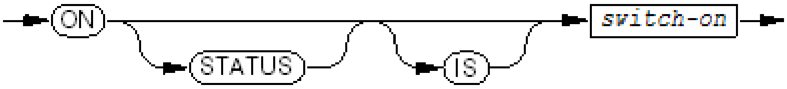
is a name you choose for the hardware device, carriage-control tape channel, or external switch specified by *system-name*.

STATUS phrase



VST746.vsd

on-phrase

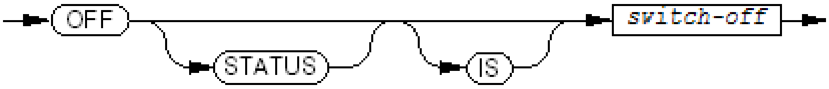


VST025.vsd

switch-on

is a condition-name for testing the settings of switches. You do not have to specify a mnemonic-name for a switch; you can just define this condition-name.

off-phrase



VST026.vsd

switch-off

is a condition-name for testing the settings of switches. You do not have to specify a mnemonic-name for a switch; you can just define this condition-name.

Usage Considerations:

- System-Name Clause Without a STATUS Phrase

  When the System Name clause has no STATUS phrase, *system-name* must be one of these values:

Value	**Meaning**
CONSOLE	Operator console
MYTERM	Process's home terminal
CHANNEL-1 through CHANNEL-12	Carriage-control tape channel

  — *system-name* is CONSOLE

  When *system-name* is CONSOLE, the System-Name clause assigns a mnemonic-name to the operator console. You can then use the mnemonic-name in DISPLAY statements, as in this example:

  ```
  SPECIAL-NAMES.
    CONSOLE IS OPERATOR-CONSOLE ...
    ...
    DISPLAY "MOUNT THE PAYROLL MASTER TAPE"
         UPON OPERATOR-CONSOLE.
  ```

  You cannot use the mnemonic-name in ACCEPT statements, because CONSOLE maps to $0, an output-only device.

  — *system-name* is MYTERM

  When *system-name* is MYTERM, the System-Name clause assigns a mnemonic-name to the home terminal You can then use the mnemonic-name in ACCEPT and DISPLAY statements, as in this example:

  ```
  SPECIAL-NAMES.
    MYTERM IS MY-TERMINAL .
    ...
    DISPLAY "MOUNT THE PAYROLL MASTER TAPE"
         UPON MY-TERMINAL.
    ...
    ACCEPT EMPLOYEE-NAME FROM MY-TERMINAL.
  ```

  — *system-name* is CHANNEL-1 … CHANNEL-12

  When *system-name* is one of CHANNEL-1 … CHANNEL-12, the System-Name clause assigns a mnemonic-name to the channel in a printer carriage-control tape. This assignment allows the ADVANCING phrase of a WRITE statement to specify a channel skip instead of a line count, as in this example:

  ```
  SPECIAL-NAMES.
    CHANNEL-1 IS SIXTEENTH-LINE ...
    ...
    WRITE HEADING AFTER ADVANCING SIXTEENTH-LINE.
  ```

- System-Name Clause With a STATUS Phrase

  When the System Name clause has a STATUS phrase, *system-name* must be the name of an external switch: SWITCH-1 through SWITCH-15.

  In the Guardian environment, set these switches at run time using the TACL command PARAM. In the OSS environment, set these switches with environment variables.

  The System-Name clause assigns a mnemonic-name to the chosen external switch in SET statements in your program. Conditional statements in the program can refer to the switch only through the condition-names you choose for *switch-on* and *switch-off*.

You can give a mnemonic-name to the switch itself, but you must declare at least one STATUS phrase for the switch because only the condition-name of the STATUS phrase can be used for testing within the program. The mnemonic-name has no purpose other than to qualify the condition-name.

You can code either the ON STATUS or the OFF STATUS phrase first, but you can use only one of each phrase for each system-name.

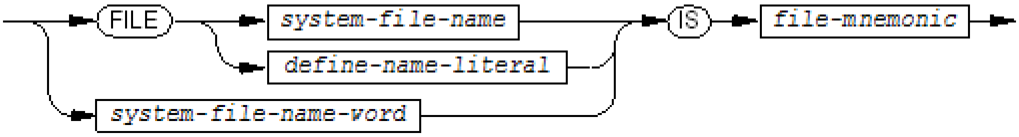
Example 7-1 ON STATUS Phrase

```
SPECIAL-NAMES.
  SWITCH-4 IS TRANSACTION-TRACE,

     ON STATUS IS TRACING-TRANSACTIONS ...
...
  IF TRACING-TRANSACTIONS
     WRITE ...
```

File-Mnemonic Clause

The File-Mnemonic clause is, technically, a variant of the "`system-name IS mnemonic-name`" clause, but its purpose is significantly different. It defines `file-mnemonic` to be either a name that Binder or the linker uses to resolve an external reference in the Procedure Division (see CALL (page 303) and ENTER (page 330)) or a name suitable for use in ACCEPT and DISPLAY statements.

The keyword FILE informs the compiler that the next token is either an operating system file name or (in the Guardian environment) a DEFINE name.



VST027.vsd

`system-file-name`

is a partially or fully qualified operating system file name. It can be represented as a nonnumeric literal (enclosed in quotation marks).

In the Guardian environment, `system-file-name` must be represented as a nonnumeric literal if it has the same spelling as a COBOL reserved word or has the form `subvolume-name.file-name`. If `system-file-name` consists of a single user-defined word or begins with one of the characters in this table, the compiler recognizes it as a file name without its being in quotation marks:

First Character	Meaning
\	System-name qualifier
$	Volume-name or process-name qualifier
#	Special operating system file name qualifier

In the OSS environment, `system-file-name` must have the syntax described for `filename` in OSS Pathnames for OSS Files (page 724).

`define-name-literal`

is a nonnumeric literal that specifies the name of a DEFINE of type MAP. For information about DEFINE names, see DEFINEs (page 601).

system-file-name-word

is a single user-defined word that specifies the name of an operating system file, not qualified by any system name, volume name, or subvolume name. It cannot be a nonnumeric literal. It cannot represent a DEFINE name.

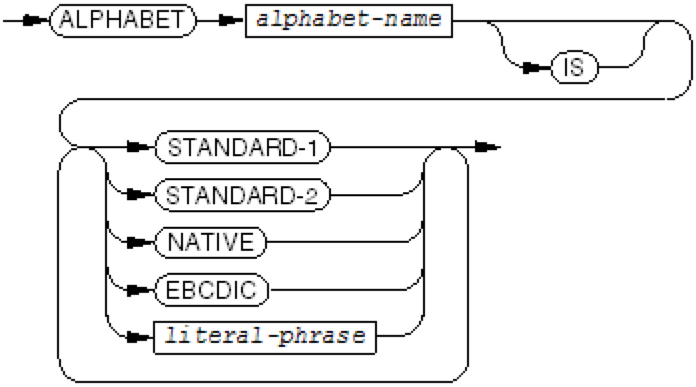
file-mnemonic

is a name you choose as an alias for the operating system file name or (in the Guardian environment) the DEFINE name. You can use it in ACCEPT, CALL, DISPLAY, and ENTER statements.

ALPHABET Clause

The ALPHABET clause provides a way for you to associate a name with a specified character code set, collating sequence, or both. You can use the *alphabet-name* that you define in the ALPHABET clause in:

- PROGRAM COLLATING SEQUENCE clause of the OBJECT-COMPUTER paragraph
- COLLATING SEQUENCE phrase of a SORT or MERGE statement
- CODE-SET clause of a file description entry



VST028.vsd

alphabet-name

is a user-defined word, the name to be associated with the character code set that you define.

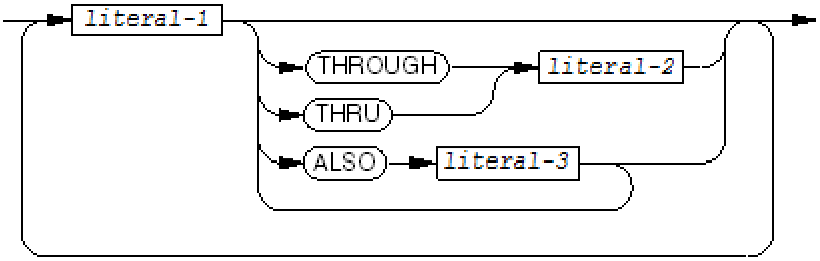
STANDARD-1, STANDARD-2, NATIVE

specify the USASCII character set.

EBCDIC

specifies the Extended Binary-Coded Decimal Interchange Code.

literal-phrase



VST029.vsd

defines an alphabet explicitly. An *alphabet-name* defined by *literal-phrase* cannot be used in the CODE-SET clause in a file description entry. If you use *literal-phrase*, you cannot specify a given character more than once in the ALPHABET clause.

`literal-1`, `literal-2`, `literal-3`

are unsigned integer literals or nonnumeric literals, but not symbolic-character figurative constants.

An unsigned integer literal must have a value in the range 1 through 256. It is the ordinal number of a character position in the computer's character set (the first character has ordinal number 1, even though its representation is octal 00).

A nonnumeric literal is an actual character or set of characters in the computer's character set. If a nonnumeric literal consists of exactly one character, the literal identifies that character. If a nonnumeric literal consists of more than one character, the compiler handles it as an abbreviation for a consecutive series of literal phrases that identify each of the specified characters individually, in the order they appear in the nonnumeric literal.

Each nonnumeric literal in a THROUGH or ALSO phrase must specify exactly one character.

Usage Considerations

- Alphabet-Name in PROGRAM COLLATING SEQUENCE Clause

  When you include an alphabet-name in the PROGRAM COLLATING SEQUENCE clause of the OBJECT-COMPUTER paragraph, you are specifying that the associated collating sequence is to be used in nonnumeric comparisons performed throughout the program.

- Alphabet-Name in COLLATING SEQUENCE Phrase

  When you include an alphabet-name in the COLLATING SEQUENCE phrase of a SORT or MERGE statement, you are specifying that the associated collating sequence is to be used in nonnumeric comparisons performed by that statement.

- Alphabet-Name in CODE-SET Clause of a File Description Entry

  When you include an alphabet-name in a CODE-SET clause of a file description entry, you are specifying that the associated character set is to be used in the external representation of the data for that file.

- Literal-Phrase

  The THROUGH phrase can specify an ascending or descending sequence of characters. THRU is equivalent to THROUGH. The compiler interprets the THROUGH phrase as an abbreviation for a consecutive series of literal phrases, each identifying the next member of a set of contiguous characters in the computer's character set. The set begins with the character identified by `literal-1` and ends with the character identified by `literal-2`. If the character identified by `literal-1` occurs earlier in the collating sequence than the one identified by `literal-2`, the implied literal sequence specifies the characters in their collating order; otherwise, the implied literal sequence specifies the characters in the reverse of their collating order.

  The characters identified in the successive literal phrases, including any implicit literal phrases generated by the interpretation previously described, are assigned successive ascending positions in the collating sequence being defined by the ALPHABET clause. The order of position assignment corresponds directly to the order of the literal phrases.

  When a literal phrase includes one or more ALSO components, each character identified by a `literal-3` is assigned the same position in the collating sequence as the character identified by the phrase's `literal-1`.

  Any characters of the collating sequence that are not identified in the successive literal phrases, including any implicit literal phrases generated by the interpretations previously described, are assigned collating sequence positions greater than that of the last character that is identified. The relative order within the set of these unspecified characters is unchanged from their order in the collating sequence.

- Program Collating Sequence and Figurative Constants

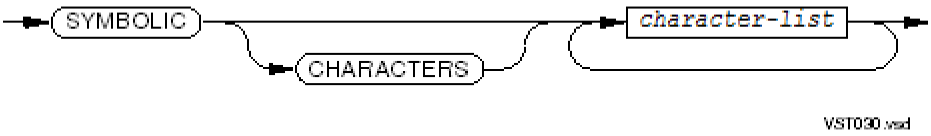
  You can use the literal phrase of the ALPHABET clause to specify an arbitrary collating sequence for use in nonnumeric comparisons, sorting, and the intrinsic functions CHAR and ORD. For information on nonnumeric comparisons, see Arithmetic Operations (page 267). For information on sorting, see SORT (page 449). For information on the intrinsic functions CHAR and ORD, see CHAR Function (page 670)CHAR Function and ORD Function (page 690).

  The character that has the highest ordinal position in the specified program collating sequence is associated with the figurative constant HIGH-VALUE. If more than one character has the highest position, the last character specified is associated with HIGH-VALUE.

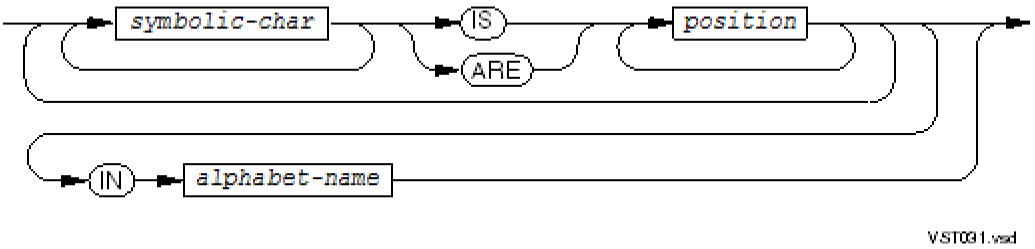
  The character that has the lowest ordinal position in the specified program collating sequence is associated with the figurative constant LOW-VALUE. If more than one character has the lowest position, the first character specified is associated with LOW-VALUE.

SYMBOLIC CHARACTERS Clause

The SYMBOLIC CHARACTERS clause assigns a name to a character, which can then be referenced as a figurative constant. It is especially useful for nonprinting characters that some input interfaces cannot except, such as the end-of-text character EXT.



character-list



assigns the name *symbolic-char* to the character in *position* within *alphabet-name*, enabling you to reference the character as a figurative constant. A *character-list* must have equal numbers of *symbolic-char* and *position*. The first *symbolic-char* corresponds to the first *position*, and so on.

symbolic-char
: is a user-defined COBOL word. No *symbolic-char* can appear more than once in the set of SYMBOLIC CHARACTERS clauses in one program.

position
: is an unsigned integer. Its value is in the range 1 through 256 and is the ordinal position of a character in an alphabet (the first ASCII character has ordinal position 1, although it is represented by octal 00).

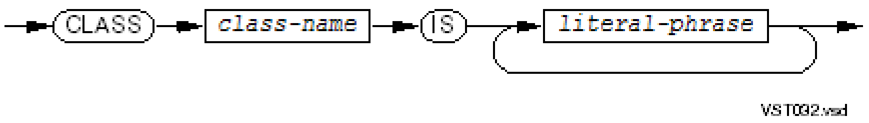
alphabet-name
: is an alphabet that includes the character to which you want to assign *symbol-char*. The default is USASCII.

This SYMBOL-CHARACTERS clause sets C-RETURN to the carriage return character, which has ordinal position 14:

```
SYMBOLIC CHARACTERS C-RETURN IS 14
```

CLASS Clause

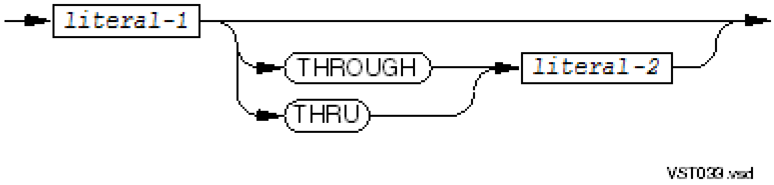
The CLASS clause defines classes other than the NUMERIC, ALPHABETIC, ALPHABETIC-UPPER, and ALPHABETIC-LOWER that the COBOL language defines.



`class-name`

is referenced in the program as a class-condition (see Class Conditions (page 280)).

`literal-phrase`



specifies a string of characters (`literal-1`) or a range of characters (`literal-1` through `literal-2`) that make up the class you are defining.

`literal-1`

is an unsigned integer literal or a nonnumeric literal, but not a symbolic-character figurative constant.

If `literal-1` is an unsigned integer literal, its value must be in the range 1 through 256. The value is the ordinal number of a character position in the computer's character set (the first character has ordinal number 1, even though its representation is octal 00).

If `literal-1` is a nonnumeric literal, its value depends on the presence or absence of `literal-2`. If `literal-2` is present, the value of `literal-1` must be exactly one character, and it can be a character that is either before or after `literal-2` in the ASCII collating sequence. If `literal-2` is absent, the value of `literal-1` can be a string of one or more characters. The compiler handles a multicharacter string as an abbreviation for a consecutive series of literal phrases that identify each of the single characters individually, in the order they appear in `literal-1`.

`literal-2`

is an unsigned integer literal or a nonnumeric literal, but not a symbolic-character figurative constant.

If `literal-2` is an unsigned integer literal, its value must be in the range 1 through 256. The value is the ordinal number of a character position in the ASCII character set.

If `literal-2` is a nonnumeric literal, its value must be exactly one character.

The compiler handles the range `literal-1` through `literal-2` as an abbreviation for a consecutive series of literal phrases that specify all ASCII characters from the lower ordinal position to the higher ordinal position.

The two definitions in Example 7-2 are equivalent.

Example 7-2 Defining a Class of Vowels

```
CLASS VOWEL IS "A" "E" "I" "O" "U" "a" "e" "i" "o" "u"
CLASS VOWEL "AEIOUaeiou"
```

The two definitions in Example 7-3 are equivalent.

Example 7-3 Defining a Class of Special Characters

```
CLASS SPEC-CHAR IS 1 THRU 32
OCT IS "0" THRU "7"

CLASS SPEC-CHAR IS 32 THRU 1
```

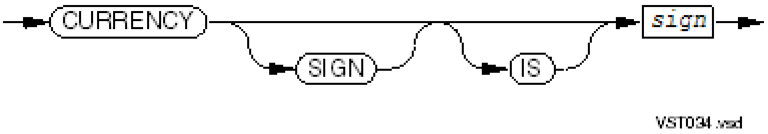
Example 7-4 Defining a Class of Octal Numerics

```
OCT IS "0" THRU "7"
```

The order of specification has no effect on the performance of the program using the definition.

CURRENCY SIGN Clause

The CURRENCY SIGN clause specifies a one-character nonnumeric literal, *sign*, whose value is to be used in the PICTURE clause to represent the currency sign.



sign

is one of these characters (which must be enclosed in quotation marks):

```
! # $ % & ' : < > ? @
E F G H I J K L M O Q T U W Y
[ \ ] ^ _ ~
e f g h i j k l m o q t u w y
```

It represents the currency sign, and cannot be a figurative constant, a control character (ASCII position 0 through 31, or 127), the space character, or any of these characters:

```
0 1 2 3 4 5 6 7 8 9
A B C D N P R S V X Z
a b c d n p r s v x z
* + - , . ; ( ) " ,   =
```

The clause

```
CURRENCY SIGN IS "#"
```

makes the currency sign #, and the picture ordinarily expressed as

```
$$$,$$9.99
```

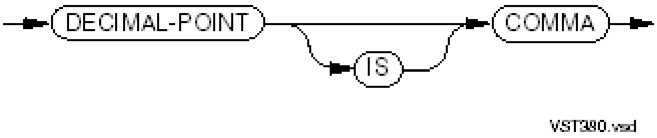
is instead expressed as

```
###,##9.99
```

If *sign* is a lowercase alphabetic character, it appears in any source listing as lowercase, but the compiler converts it to an uppercase character for all further processing. It is uppercase in any numeric edited item produced by the associated PICTURE clause.

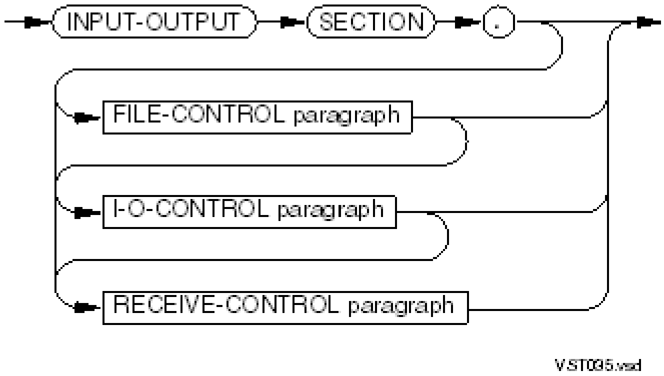
DECIMAL-POINT Clause

The DECIMAL-POINT clause exchanges the functions of commas and periods in PICTURE character-strings and numeric literals.



Input-Output Section

COBOL programs typically handle large amounts of data, stored on such devices as disks and magnetic tapes. Before a program can use the stored data, the operating system file names must be linked to COBOL file names. You connect the names in file-control entries in the Input-Output Section.

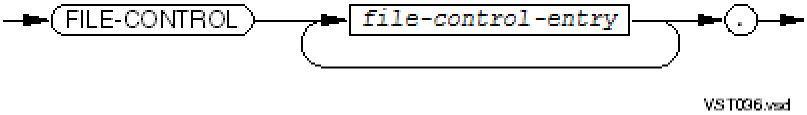


Topics:

- FILE-CONTROL Paragraph
- I-O-CONTROL Paragraph
- RECEIVE-CONTROL Paragraph

FILE-CONTROL Paragraph

If the FILE-CONTROL paragraph is present, it must be the first paragraph in the Input-Output Section.



file-control-entry
has different syntax for sequential, line sequential, relative, indexed, queue, and sort-merge files. See the appropriate topic.

Topics:

- File-Control Entries in General
- File-Control Entries for Sequential Files
- File-Control Entries for Line Sequential Files
- File-Control Entries for Relative Files
- File-Control Entries for Indexed Files
- File-Control Entries for Queue Files
- File-Control Entries for Sort-Merge Files

File-Control Entries in General

The file name of every data file and every sort-merge file described in the Data Division must appear exactly once in the SELECT clause of a file-control entry. Likewise, every file name that appears in the SELECT clause of a file-control entry must also appear in a corresponding data file description entry (with a level indicator of FD) or sort-merge file description entry (with a level indicator of SD).

A file-control entry connects an operating system file name to a COBOL file name, specifies the file's organization and keys, and gives other information needed for input and output. For information about Guardian file names, see the *Guardian Procedure Calls Reference Manual*. For information about OSS file names, see Files in the OSS Environment (page 723).

Each file-control entry consists of a SELECT clause followed by one or more clauses that specify file-related information. Every file-control entry must contain at least the SELECT clause and the ASSIGN clause. The file organization determines which of the other clauses are required or optional. No clause can appear more than once except for the ALTERNATE RECORD KEY clause, which can appear as often as needed to describe the alternate record keys of the file. Code the SELECT clause first, and then code the other clauses in any order.

Table 7-3 Summary of File-Control Entry Clauses

	File Type					
File-Control Entry Clause[1]	**Sequential**	**Line Sequential**	**Relative**	**Indexed**	**Queue**	**Sort-Merge**
SELECT[2]	R	R	R	R	R	R
ASSIGN	R	R	R	R	R	R
RESERVE	O	O	O	O	O	NA
ORGANIZATION	O	O	R	R	R	NA
PADDING CHARACTER	O	NA	NA	NA	NA	NA
RECORD DELIMITER	O	NA	NA	NA	NA	NA
ACCESS MODE[3]	O	O	O	O	O	NA
RECORD KEY	NA	NA	NA	R	R	NA
RELATIVE KEY	NA	NA	O if access mode is sequential, R otherwise	NA	NA	NA
ALTERNATE RECORD KEY	O	NA	O	O	NA	NA
FILE STATUS	O	O	O	O	O	NA
R=Required						
O=Optional						
NA=Not Applicable						

[1] File-control entry clauses have the same syntax for each file type, except as noted.

[2] SELECT clause syntax is the same for all file types except sort-merge.

[3] ACCESS MODE clause syntax is different for all types.

Table 7-4 Descriptions of File-Control Entry Clauses

Clause	Description
SELECT	Defines the file name used to refer to the file in the remainder of the source program. The OPTIONAL phrase applies only to files opened in the INPUT, I-O, or EXTEND mode. When such a file is not required to be present every time the object program unit is executed, you can designate it as optional. Every file name in a SELECT clause must also be in a file description (FD) entry or sort-merge file description (SD) entry in the Data Division. Conversely, every file name in a file description entry or sort-merge file description entry must also be in a SELECT clause.
ASSIGN	Associates the file name used within the program with an operating system file. The operating system file name is the one the operating system uses to refer to the file.
ORGANIZATION	Specifies the logical structure of a file, which was established when the file was created and which you cannot change. The default is sequential organization.
RECORD DELIMITER	Specifies the method of determining the length of a variable-length record on the file's physical medium. It can appear only for a sequential file whose description specifies variable-length records. HP COBOL handles the RECORD DELIMITER clause as a comment.
ACCESS	Specifies the order in which a process can read records from the file. The default is sequential access.When the access mode is sequential, a process can read records from or write records to the file one after the other, in an order that depends on the file's organization (see Table 7-5). When the access mode is random, you use key values to specify the order in which a process reads or writes records. The file-control entry specifies a data item to hold a key value (see Table 7-6).When the access mode is dynamic, records can be processed either sequentially or randomly.
ALTERNATE RECORD KEY	See the paragraphs after Table 7-6.
FILE STATUS	The FILE STATUS clause specifies the identifier (designated in the syntax as *`filestat`*) that is to serve as the file-status data item for the file. When a COBOL run-time I-O routine completes an operation on the file, it stores the status code in the file-status data item before returning control to your program. The identifier *`filestat`* must reference an alphanumeric data item with a size of exactly two characters. The data item must be defined in the Working-Storage Section, Extended-Storage Section, or Linkage Section of the Data Division.

Table 7-5 Sequential Access

File Organization	Records are read by ascending alternate key or processed in the order …
Sequential, Line sequential	that they are written, from the first physical record to the last
Relative	of ascending relative record numbers
Indexed	of ascending record key values within a given key

Table 7-6 Random Access

File Organization	Next record to be processed is determined by the value of …
Relative	RELATIVE KEY or ALTERNATE RECORD KEY data item
Indexed	RECORD KEY or ALTERNATE RECORD KEY data item

The ALTERNATE RECORD KEY clause specifies an alternate record key (designated in the syntax as *`altkey`*) for the file. Each *`altkey`* must reference an alphanumeric or unsigned numeric data item defined in a record description entry associated with its file name. In either case, the collating sequence is ASCII—using tricks to allow `altkey` to be a signed numeric data item causes problems.

An `altkey` cannot reference an item whose size is variable. The presence or absence of the DUPLICATES phrase specifies whether or not the value of this alternate record key can be duplicated among the records in the file.

If more than one record in a file has the same alternate record key value then when a program reads the file according to the alternate record key, the records are presented to the program either in either prime record key order or in the chronological order of their insertion into the file. The 1985 ISO/ANSI COBOL standard specifies insertion order, but HP COBOL provides a mechanism to enable you to choose either order.

Each such file has an INSERTIONORDER attribute that governs this behavior for all its alternate keys. An insertion-ordered alternate key cannot share an alternate key file with other keys of different lengths, or with other alternate keys that are not insertion ordered. The attribute can be changed only by using either a call to the operating system routine SET or by use of the File Utility Program (FUP), not by any COBOL language phrase or clause. In FUP, the BUILDKEYRECORDS and LOADALTFILE commands do not support loading of insertion-ordered alternate key records.

There are size and performance penalties for using insertion-ordered duplicate alternate keys; the size of the alternate file increases and the access time increases as the number of records having duplicate alternate keys increases. For more information see the *File Utility Program (FUP) Reference Manual* or the *Guardian Procedure Calls Reference Manual*.

The data description of each `altkey`, its relative location within the file record, and the specification of the DUPLICATES attribute must correlate with one of the alternate record keys defined when the file was created. The file-control entry can contain at most one ALTERNATE RECORD KEY clause that describes a particular alternate record key of the file. If an alternate record key is not referenced in the source program Procedure Division, then it is not necessary to describe it in the file-control entry. A maximum of 31 alternate record keys can be described for a single file.

No `altkey` can reference an item whose leftmost character position within the file record corresponds to the leftmost character position of the item referenced by any other `altkey` associated with this file.

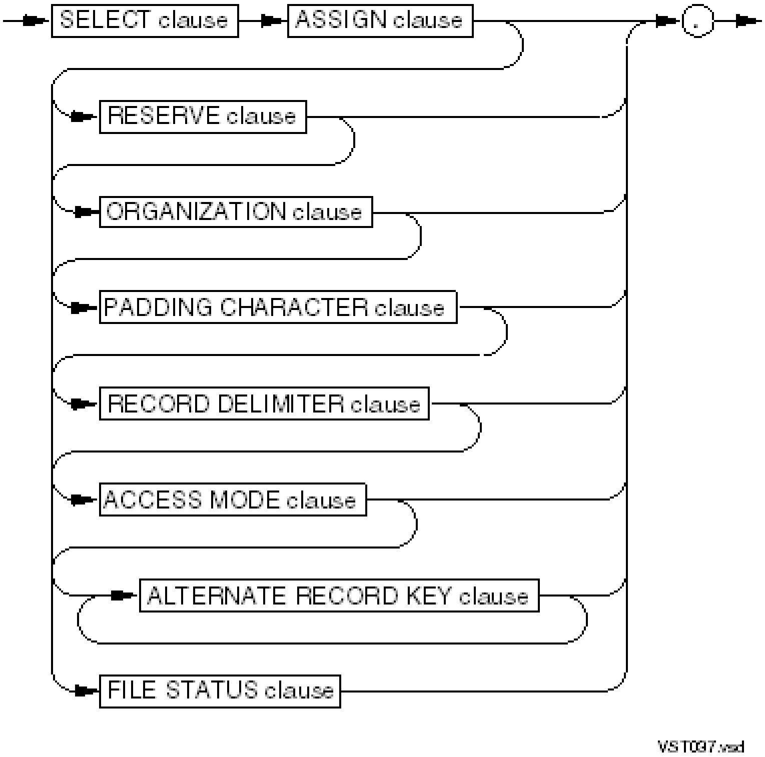
File-Control Entries for Sequential Files

HP COBOL supports two types of sequential disk files:

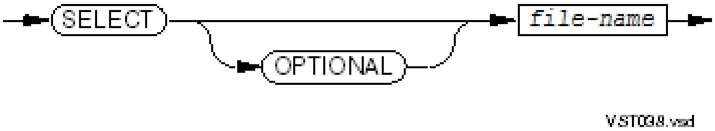
- A disk file created as a sequential file (also called an entry-sequenced file on NonStop systems)
- An unstructured disk file

Also, a COBOL file defined as sequential can be assigned to a tape file, to a device such as a terminal or process, or to $RECEIVE. See OPEN (page 385) and RECEIVE-CONTROL Paragraph.

In HP COBOL, entry-sequenced disk files can have alternate record keys, whose values can identify individual records. A file-control entry for a sequential file includes an ORGANIZATION SEQUENTIAL clause or no ORGANIZATION clause (in which case the file has sequential organization by default).



SELECT clause



OPTIONAL

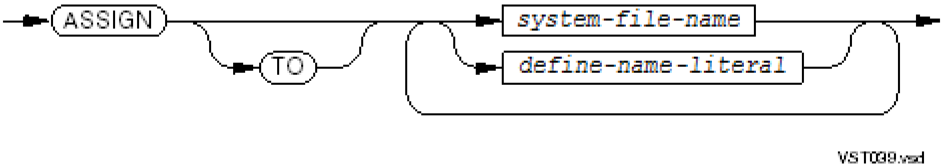
makes the file optional, which means that an OPEN statement with an INPUT or EXTEND phrase can open the file whether or not the file exists. If the file exists, its I-O status code is “00”; if not, its I-O status code is “05”. OPTIONAL does not affect the OPEN statement with an OUTPUT phrase.

When you open a nonexistent optional file for input, the first READ statement for that file uses the AT END option (or USE procedure if the READ statement has no AT END phrase).

`file-name`

is the COBOL file-name (the *`file-name`* in a file description entry).

ASSIGN clause



associates *`file-name`* with *`system-file-name`* or *`define-name-literal`*. Only the first *`system-file-name`* or *`define-name-literal`* has meaning. The compiler ignores subsequent names and literals and issues a warning.

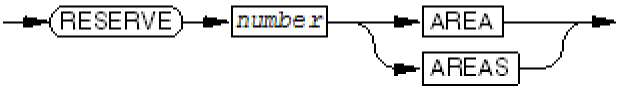
`system-file-name`

is the name of a file that the file system recognizes or one of the special operating system file names described in System-Names (page 75). If *`system-file-name`* does not begin with a dollar sign ($), backward slash (\), or number sign (#), then it must be enclosed in quotation marks unless it forms a COBOL word. For more information about operating system file names, see the *Guardian Procedure Calls Reference Manual*.

define-name-literal

is allowed only in the Guardian environment. It is a nonnumeric literal representing a DEFINE name of class MAP, SPOOL, TAPE, or TAPECATALOG. Quotation marks must enclose *define-name-literal*. For more information about DEFINE names, see DEFINEs (page 601).

RESERVE clause



VST040.vsd

enables or prevents sequential block buffering on input and buffered cache on output, or enables or prevents HP COBOL Fast I-O for both input and output, for a disk file, depending on the value of *number*.

number

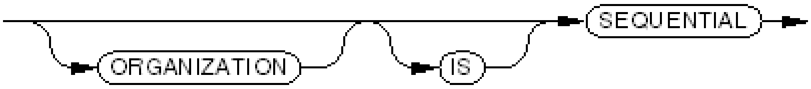
is a numeric literal, an unsigned integer.

number must be in the range 1 through 32, and its value is interpreted:

Value of number	Effect
1	No buffering or HP COBOL Fast I-O
2	Sequential block buffering on input and buffered cache on output if the assigned file qualifies
3 or greater	HP COBOL Fast I-O if the assigned file qualifies; if not, sequential block buffering for input and buffered cache for output if the assigned file qualifies; otherwise normal I-O. *number* is the number of blocks to buffer.

CAUTION: Do not use sequential block buffering for a file opened for shared access. If you do, a process could read data that is not up-to-date while another process alters the file. For information on shared access, see OPEN (page 385).

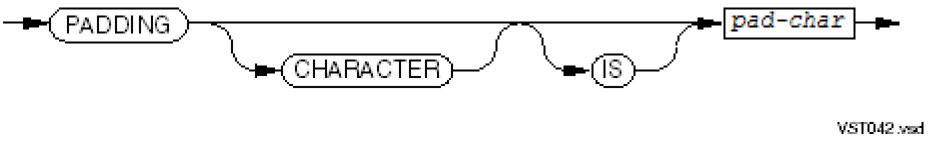
ORGANIZATION clause



VST041.vsd

makes the organization of the file sequential (the default).

PADDING CHARACTER clause

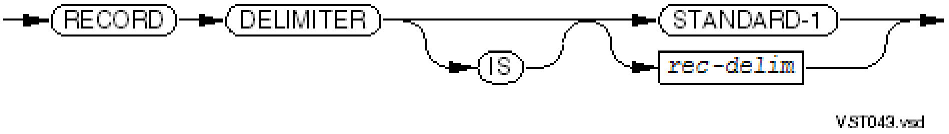


specifies *pad-char*, the character to be used for padding on sequential files on unlabeled magnetic tape when the physical record size (block size) exceeds the logical record size. HP COBOL handles this phrase as a comment, except for checking if *pad-char* is acceptable.

pad-char

is either a nonnumeric literal that represents a single character, or a qualified or unqualified name that references an alphanumeric data item whose value is a single character. If *pad-char* is a data item, it must be defined in the Working-Storage, Extended-Storage, or Linkage Section. *pad-char* cannot be a national literal or national data item.

RECORD DELIMITER clause



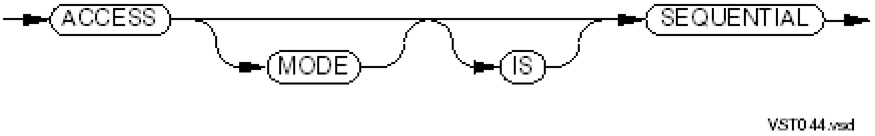
specifies the method of determining the length of a variable-length record on the file's external medium. HP COBOL handles this clause as a comment, except that these requirements are checked:

- This clause can appear only for a file whose description specifies variable-length records.
- STANDARD-1 applies only to magnetic tape files that contain standard label records.
- *rec-delim* must be IMPLICIT (see *rec-delim*, following).

rec-delim

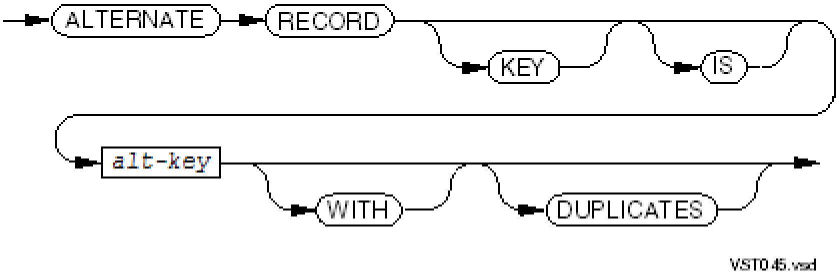
must be IMPLICIT, the only name predefined by HP COBOL for use in this clause. IMPLICIT signifies that the record delimiter is implied by the size of the records on the magnetic tape.

ACCESS MODE clause



makes the access mode of the file sequential (the default).

ALTERNATE RECORD KEY clause



makes *alt-key* an alternate record key.

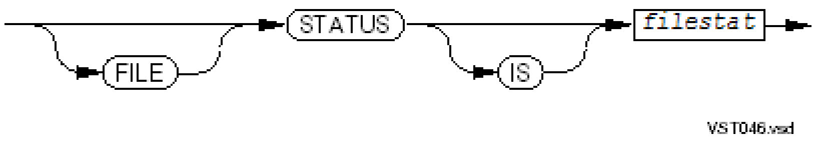
`alt-key`

is an alphanumeric or unsigned numeric data item declared in the record description entry of the file and is used to gain access to records within the file. The size and location of `alt-key` must agree with the size and location of the alternate key within the file, as defined when the alternate key file was established by the Guardian environment File Utility Program (FUP). (For information about FUP, see the *File Utility Program (FUP) Reference Manual*.)

DUPLICATES

means that alternate key values are not necessarily unique.

FILE STATUS clause



defines `filestat` as the file-status data item for the file. When a COBOL run-time I-O routine completes an operation on the file, it stores the status code in `filestat` before returning control to your program (see I-O Status Code (page 257)).

`filestat`

is a 2-character alphanumeric, nonnational data item defined in the Working-Storage Section, Extended-Storage Section, or Linkage Section.

Usage Considerations:

- **EDIT Files**

  A file assigned to an EDIT file can be open for output only if it has the file code 101. It can be given the file code 101 by any of:

  — The application that created it outside the COBOL program

  — An ASSIGN command that was active during run unit initialization (the ASSIGN command must have a CODE phrase)

  — The COBOL_ASSIGN_ routine

  The buffer size of an EDIT file created in the CRE is determined:

  1. If an ASSIGN command includes the BLOCK parameter, that value is the buffer size. (ASSIGN commands are allowed only in the Guardian environment.)
  2. If Step 1 is not true and the file description entry for the EDIT file includes a BLOCK CONTAINS clause, then that value is the buffer size.
  3. If Step 2 is not true, the buffer size is 4,096.

- **Alternate Record Keys**

  Records in a file can be read in the ascending order of alternate key values, even if the file's organization is sequential or relative. The order in which records are obtained using `alt-key` can differ from the order in which the records are stored. You can define up to 31 alternate keys for a file.

  If you include the DUPLICATES phrase in the ALTERNATE RECORD KEY clause, the value of `alt-key` need not be unique for each record in the file. Depending on the INSERTIONORDER parameter of the alternate key file, records with duplicate alternate key values are inserted (or retrieved) in either prime key order (the way NonStop systems software ordinarily works) or in the order in which they were inserted in the file (as specified in the 1985 ISO/ANSI COBOL standard).

  The data description entry for `alt-key` cannot contain an OCCURS clause or be subordinate to an entry that contains an OCCURS clause. The leftmost character position of an `alt-key` item cannot correspond to the leftmost character position of another `alt-key` item in that file.

The file-control entry can contain at most one ALTERNATE RECORD KEY clause that describes a particular alternate record key of the file.

If a file in the file system is defined as having alternate record keys to which the COBOL program does not make any reference, you do not need to specify them in the File-Control paragraph.

An `alt-key` is permitted only for structured disk files and not for unstructured disk files or files that are not disk files. A file having a LINAGE clause in its file description cannot be a disk file (although it can be spooled to a printer by way of a disk). It is, therefore, never legal to define a file with an ALTERNATE RECORD KEY clause and have a LINAGE clause in the file description entry for that file.

- **Sequential Block Buffering**

  Sequential block buffering, enabled by the RESERVE clause when the file is open in INPUT or I-O mode, is a feature of the Enscribe database record manager that enables faster reading of a sequentially structured file by reading a block of records together into a memory buffer. The file's access must be sequential, but its organization can be sequential, relative, or indexed.

  A program requests sequential block buffering (or does not request it) when it opens the file. Each process that opens the file requests or does not request sequential block buffering. One process can request it while another process does not. Nothing stored on the disk keeps a record of whether the file is to be read with sequential block buffering.

  The RESERVE clause acts as a switch. In the CRE, the *`number`* 2 means to use sequential block buffering on input (and buffered cache on output) if the assigned file qualifies.

  Enscribe determines the size of the sequential block buffer. Usually, the buffer size is the file's data block size, but when a process accesses a file by alternate keys, the buffer size is the alternate key file's data block size. You can obtain the data block size for the appropriate file with the FUP command:

  ```
  FUP INFO file-name,DETAIL
  ```

  The size is usually 1,024 characters.

  The size of the buffer space used for reading is 4* *`dbs`*, where *`dbs`* is the file's data block size. The BLOCK CONTAINS clause, if specified, is ignored. Because double buffering is used, the size of the block read is half the size of the buffer. The maximum size of blocks read is 32,768 characters.

  If the process cannot perform sequential block buffering for any reason, it performs normal input-output. The file status code is "07".

NOTE: With the advent of the DP2 disk processing system, normal I/O could be faster than sequential block buffering, depending on the number of records per block. For example, a file containing eight 4K blocks and 1600 records requires one physical I/O operation and 1600 interprocess messages in normal mode, but eight physical I/O operations and only eight interprocess messages in SBB mode; on the other hand, a file with eight 4K blocks and only eight records requires one physical I/O and eight interprocess messages in normal mode, but eight physical I/O operations and eight interprocess messages in SBB mode

- **Buffered Cache**

  Buffered cache is a feature of the DP2 disk process that uses more efficient disk I-O to write disk files.

The RESERVE clause acts as a switch. In the CRE, the `number` 2 means to use buffered cache on output (and sequential block buffering on input) if the assigned file qualifies. The file must be open in the OUTPUT, I-O, or EXTEND mode.

The size of the buffer space used for writing is 29K characters, regardless of the `number` in the RESERVE clause.

This technique buffers records in cache rather than writing them immediately to disk, thereby reducing the number of requests to the disk process. Audited files always use buffered cache; nonaudited files can use buffered cache or write-through cache under DP2. Write-through cache transfers each record to the disk as it is written.

You can use the File Utility Program (FUP) commands SET and ALTER to set the BUFFERED attribute for a file, specifying that any programs that open that file must use buffered cache. If a file has the BUFFERED attribute set, a process can still override the attribute by calling the Guardian routine SETMODE to specify that buffered cache must not be used, but the run-time routines do not do this. If a COBOL program specifies RESERVE 1 AREA for a file that already has the BUFFERED attribute set, the disk process still uses buffered cache.

You must not use buffered cache in applications that require each record to be actually written to disk before execution of the next statement in the program.

- **HP COBOL Fast I-O (Local Buffering)**

  HP COBOL Fast I-O is an enhancement in input-output performance beyond that of sequential block buffering or buffered cache. It is available if a program and the sequential file on which it is to operate meet these criteria:

  — You are not creating an audited file (you can read an audited file, however).

  — The file description includes a RESERVE clause with a `number` specifying the number of blocks to buffer. The `number` must be greater than 2.

  — The file description does not include a LINAGE clause or a CODE-SET clause.

  — The file is not opened with time limits (as with the TIME LIMITS phrase in the OPEN statement).

  — The file's open mode is either INPUT or OUTPUT. If it is INPUT, its exclusion mode is PROTECTED; if it is OUTPUT, its exclusion mode is EXCLUSIVE.

  — The program is not compiled with the NONSTOP directive.

  — Regarding fast I-O and alternate keys, you must adhere to these rules:

    ◦ If the program is compiled with ENV OLD, there must be no alternate keys either in the SELECT statement or in the file itself.

    ◦ If the program is compiled with ENV COMMON, there must be no alternate keys mentioned in the SELECT statement. Any alternate keys that exist in the file must be set to NO UPDATE with FUP.

  With HP COBOL Fast I-O, the run-time routines use an auxiliary block buffer and perform the blocking and deblocking in local storage. This can operate as much as ten times faster than having the operating system perform the blocking and deblocking operations.

  The size of the buffer space used for reading is 2 * `number`*`dbs`, where `dbs` is the file's data block size. The BLOCK CONTAINS clause, if specified, is ignored. Because double buffering is used, the size of the block read is half the size of the buffer. The maximum size of blocks read is 32,768 characters.

  You can obtain the data block size for the appropriate file with the FUP command:

  ```
  FUP INFO file-name, DETAIL
  ```

  The size is usually 1,024 characters.

  The size of the buffer space used for writing is 29K characters, regardless of the `number` in the RESERVE clause.

If the RESERVE clause specifies a `number` that uses HP COBOL Fast I-O and the assigned file qualifies for HP COBOL Fast I-O, but there is not enough buffer space for HP COBOL Fast I-O, then the I-O status code is “07” and sequential block buffering is used.

If you run out of disk space, the corrupt bit is set, the program terminates abnormally, and you cannot open the file. If the file is entry-sequenced, use FUP to clear the corrupt bit. If the file is not entry-sequenced, the file is unusable.

An auxiliary block buffer can be shared by two files in a program as long as both are not open at the same time. The space is allocated as part of the open operation and deallocated as part of the close operation. The SAME AREA clause has no effect on space allocation or deallocation.

Example 7-5 contains two simple sequential files:

— An input file that the COBOL program recognizes as MASTER1-IN and to the operating system as NAMEFILE in a subvolume name SUB1 on a disk drive named $BULK.

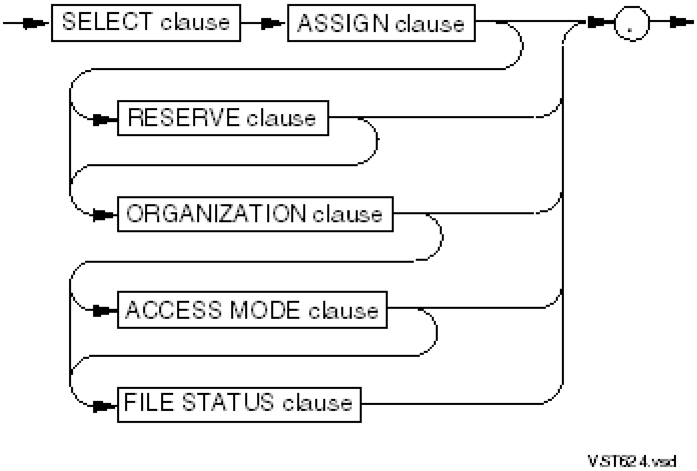
— An output file that the COBOL program recognizes as REPORT-OUT and to the Guardian environment as a DEFINE named =ROSTER-PRINTER.

Example 7-5 FILE-CONTROL Paragraph for Sequential File

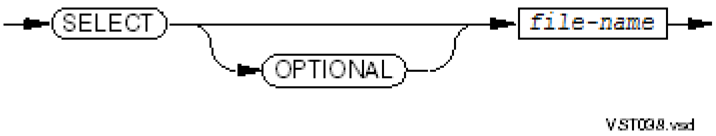
```
INPUT-OUTPUT SECTION.
FILE-CONTROL.
    SELECT MASTER1-IN
      ASSIGN TO "$BULK.SUB1.NAMEFILE"
      ORGANIZATION IS SEQUENTIAL
      ACCESS MODE IS SEQUENTIAL.
    SELECT REPORT-OUT
      ASSIGN TO "=ROSTER-PRINTER"
      ORGANIZATION IS SEQUENTIAL
      ACCESS MODE IS SEQUENTIAL.
      ...
DATA DIVISION.
FILE SECTION.
FD  MASTER1-IN
    LABEL RECORDS ARE OMITTED.
    ...
FD  REPORT-OUT
    LABEL RECORDS ARE OMITTED.
    ...
```

File-Control Entries for Line Sequential Files

Line sequential files are available only in the OSS environment.



SELECT clause



OPTIONAL

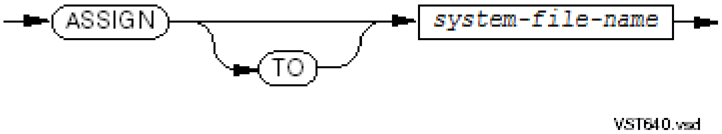
makes the file optional, which means that an OPEN statement with an INPUT, I-O, or EXTEND phrase can open the file whether or not the file exists. If the file exists, its I-O status code is “00”; if not, its I-O status code is “05”. OPTIONAL does not affect the OPEN statement with an OUTPUT phrase.

When you open a nonexistent optional file for input, the first READ statement for that file uses the AT END option (or USE procedure if the READ statement has no AT END phrase).

`file-name`

is the COBOL file-name (the *`file-name`* in a file description entry).

ASSIGN clause



associates *`file-name`* with *`system-file-name`*. Only the first *`system-file-name`* has meaning. The compiler ignores subsequent names and issues a warning.

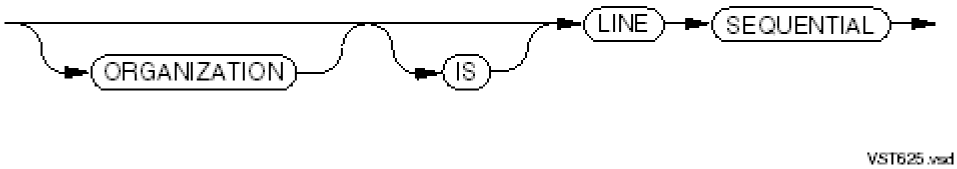
`system-file-name`

is the name of a code-180 file that the file system recognizes or one of the special operating system file names described in System-Names (page 75). If *`system-file-name`* does not begin with a dollar sign ($), backward slash (\), or number sign (#), then it must be enclosed in quotation marks unless it forms a COBOL word. For more information about Guardian file names, see the *Guardian Procedure Calls Reference Manual*. For more information about OSS file names, see the `filename(5)` reference page either online or in the *Open System Services System Calls Reference Manual*.

RESERVE clause

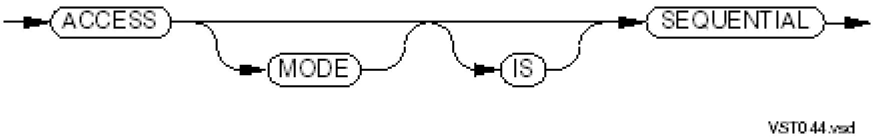
is ignored.

ORGANIZATION clause



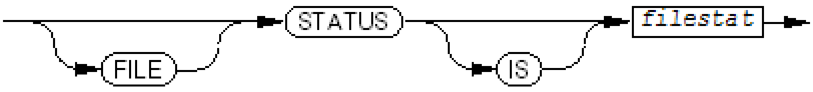
makes the organization of the file line sequential.

ACCESS MODE clause



makes the access mode of the file sequential (the default).

FILE STATUS clause



VST046.vsd

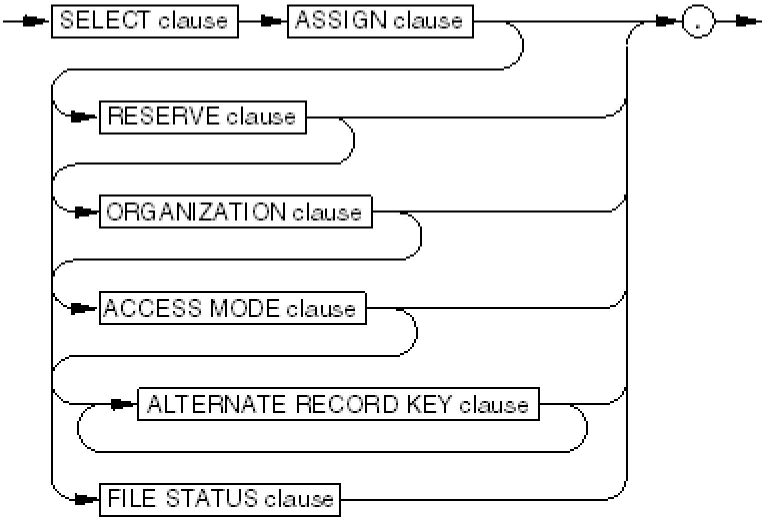
defines `filestat` as the file-status data item for the file. When a COBOL run-time I-O routine completes an operation on the file, it stores the status code in `filestat` before returning control to your program (see I-O Status Code (page 257)).

`filestat`

is a 2-character alphanumeric, nonnational data item defined in the Working-Storage Section, Extended-Storage Section, or Linkage Section.

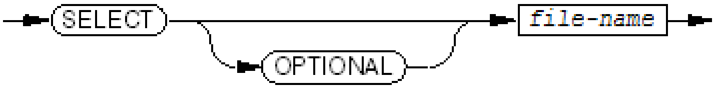
File-Control Entries for Relative Files

Relative organization provides the capability to read records from or write records to a disk file either randomly or sequentially. For each record in a relative file, a positive integer value that specifies the record's logical ordinal position in the file uniquely identifies the record. In HP COBOL you can also use alternate alphanumeric or unsigned numeric keys (with possible duplicate values) within each record. A relative file has a fixed maximum record length, but the records can be of various lengths up to that maximum. A file-control entry for a relative file includes an ORGANIZATION RELATIVE clause.



VST047.vsd

SELECT clause



VST098.vsd

OPTIONAL

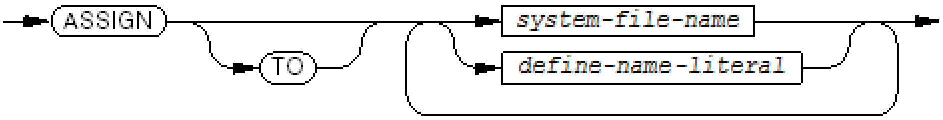
makes the file optional, which means that an OPEN statement with an INPUT, I-O, or EXTEND phrase can open the file whether or not the file exists. If the file exists, its I-O status code is “00”; if not, its I-O status code is “05”. OPTIONAL does not affect the OPEN statement with an OUTPUT phrase.

When you open a nonexistent optional file for input, the first READ statement for that file uses the AT END option (or USE procedure if the READ statement has no AT END phrase).

file-name

is the COBOL file-name (the *file-name* in a file description entry).

ASSIGN clause



VST039.vsd

associates *file-name* with *system-file-name* or *define-name-literal*. Only the first *system-file-name* or *define-name-literal* has meaning. The compiler ignores subsequent names and literals and issues a warning.

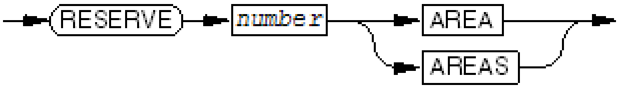
system-file-name

is either the name of a disk file or the special file name #DYNAMIC or #TEMP. Quotation marks must enclose system-file-name unless it is a COBOL word or begins with a dollar sign ($), a backward slash (\), or a number sign (#). For more information about operating system file names, see the Guardian Procedure Calls Reference Manual.

define-name-literal

is a nonnumeric literal. It represents the name of a DEFINE of class MAP that is associated with a disk file. Quotation marks must enclose *define-name-literal*. For information on DEFINE names, see DEFINEs (page 601).

RESERVE clause



VST040.vsd

enables or prevents sequential block buffering on input and buffered cache on output, or enables or prevents HP COBOL Fast I-O for both input and output, for a disk file, depending on the value of *number*. The access mode of the file must be SEQUENTIAL.

number

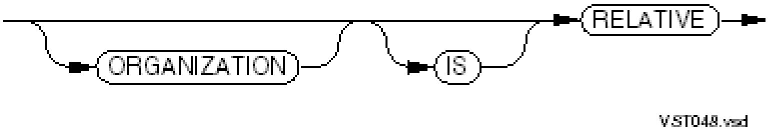
is a numeric literal, an unsigned integer.

number must be in the range 1 through 32, and its value is interpreted:

Value of number	Effect
1	No buffering or HP COBOL Fast I-O.
2	Sequential block buffering on input and buffered cache on output if the assigned file qualifies.
3 or greater	HP COBOL Fast I-O if the assigned file qualifies; if not, sequential block buffering for input and buffered cache for output if the assigned file qualifies; otherwise normal I-O. *number* is the number of blocks to buffer.

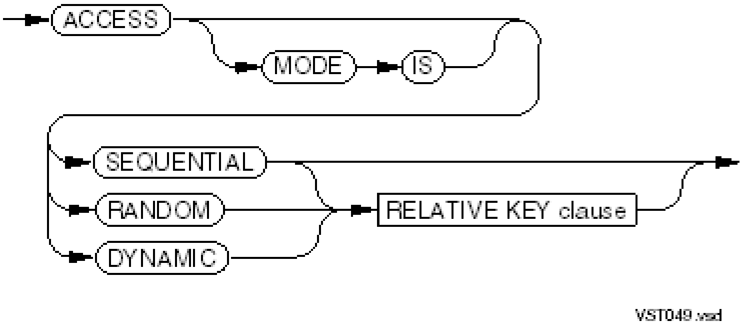
CAUTION: Do not use sequential block buffering for a file opened for shared access. If you do, a process could read outdated data while another process alters the file. For information on shared access, see OPEN (page 385).

ORGANIZATION clause



makes the organization of the file relative. (The default is sequential.)

ACCESS MODE clause



SEQUENTIAL

makes the access mode of the file sequential (the default). That means the records of the file are to be operated upon as if they were sequentially organized.

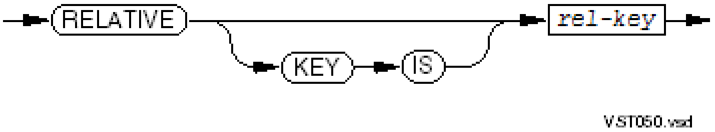
RANDOM

makes the access mode of the file random. That means the records of the file are to be operated upon in any order, as selected by the current value of the relative key or an alternate key.

DYNAMIC

makes the access mode of the file dynamic. That means the records of the file are accessible by either sequential or random access; you can position the file to a certain key value with START and read or write sequentially from there.

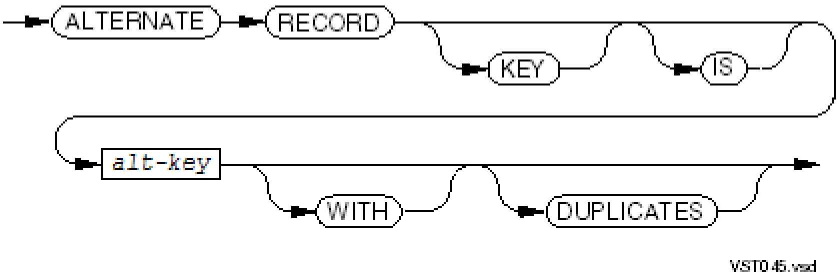
RELATIVE KEY clause



`rel-key`

is an integer data item to be used as a prime key to specify records within the file. It must be large enough to hold the maximum record number. Its definition cannot be in a record description entry for that file. It can be in another record description entry or in the Working-Storage Section, Extended-Storage Section, or Linkage Section.

ALTERNATE RECORD KEY clause



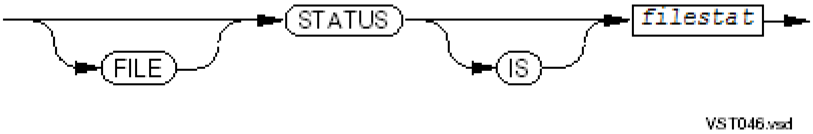
`alt-key`

is an alphanumeric or unsigned numeric data item declared in the record description entry of the file. It is used to access records within the file. Its size and location must agree with the size and location of the alternate key within the file, defined when the alternate key file was established by the Guardian environment File Utility Program (FUP). (For information about FUP, see the *File Utility Program (FUP) Reference Manual*.)

DUPLICATES

means that alternate key values are not necessarily unique.

FILE STATUS clause



defines `filestat` as the file-status data item for the file. When a COBOL run-time I-O routine completes an operation on the file, it stores the status code in `filestat` before returning control to your program (see I-O Status Code (page 257)).

`filestat`

is a 2-character alphanumeric, nonnational data item defined in the Working-Storage Section, Extended-Storage Section, or Linkage Section.

Usage Considerations:

- Relative Key

  Every record in a relative file is uniquely identified by a positive integer called the relative record number. Each record’s number defines its logical position in the file. The first logical record has a relative number of 1, the second logical record has a relative number of 2, and so on. A file is not required to contain records in every logical position; record 3 can exist even when record 2 is missing.

NOTE: COBOL record numbers begin at 1. This is different from the Guardian file-system convention of beginning record numbers at 0. The COBOL run-time I-O routines deduct 1 from the COBOL relative record number to obtain the Guardian file-system record number. When a COBOL program requests record 1, it receives the first record in the file—the record that the Guardian file system designates 0. Remember this difference if other HP products are to operate on a file.

- Alternate Record Keys

  The records within the file can be accessed in ascending order of the `alt-key` value. The order in which records are obtained using `alt-key` can differ from the order obtained using `rel-key`. You can define up to 31 alternate keys for a file.

If you include the DUPLICATES phrase in the ALTERNATE RECORD KEY clause, the value of `alt-key` need not be unique for each record in the file. Depending on the INSERTIONORDER parameter of the alternate key file, records with duplicate alternate key values are inserted (or retrieved) in either prime key order or in the order in which they were inserted in the file.

The data description entry for `alt-key` cannot contain an OCCURS clause or be subordinate to an entry that contains an OCCURS clause. The leftmost character position of an `alt-key` item cannot correspond to the leftmost character position of another `alt-key` item in that file.

The file-control entry can contain at most one ALTERNATE RECORD KEY clause that describes a particular alternate record key of the file.

If a file in the file system is defined as having alternate record keys to which the COBOL program does not make any reference, you do not need to specify them in the File-Control paragraph.

- Sequential Block Buffering

  Sequential block buffering, enabled by the RESERVE clause when the file is open in INPUT or I-O mode, is discussed under File-Control Entries for Sequential Files.

- Buffered Cache

  Buffered cache, enabled by the RESERVE clause when the file is open in OUTPUT, I-O, or EXTEND mode, is discussed under File-Control Entries for Sequential Files.

- HP COBOL Fast I-O (Local Buffering)

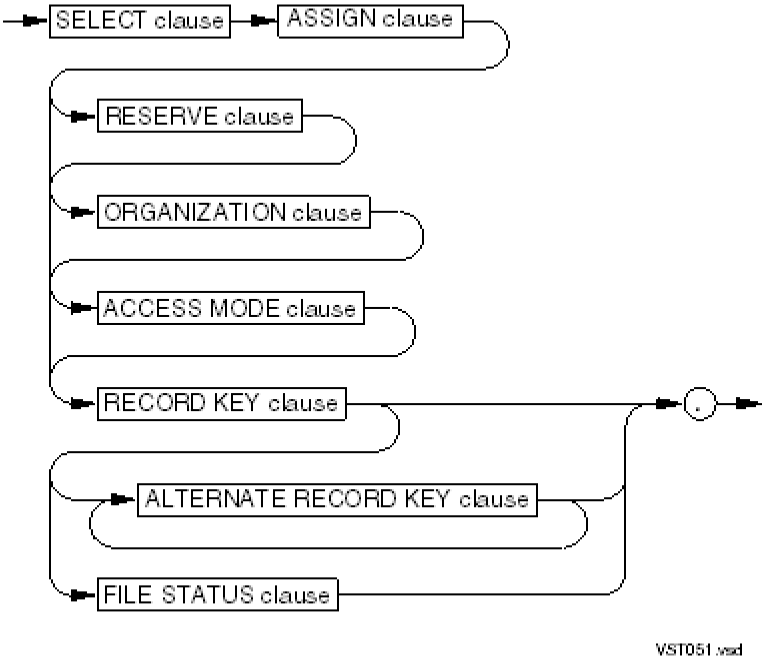
  HP COBOL Fast I-O is an enhancement in input-output performance beyond that of sequential block buffering or buffered cache. It is available if a program and the relative file upon which it is to operate are in the CRE and meet the criteria under File-Control Entries for Sequential Files.

Example 7-6 Relative File Used for Random Access

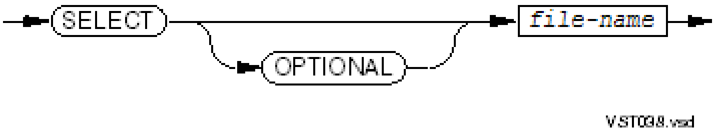
```
INPUT-OUTPUT SECTION.
FILE-CONTROL.
    SELECT INPUT-FILE ASSIGN TO "$DATA.MYDISC.INPUT"
       ORGANIZATION IS RELATIVE
       ACCESS MODE IS RANDOM
       RELATIVE KEY IS INP-RELKEY
       FILE STATUS IS INP-STATUS.
    ...
DATA DIVISION.
FILE SECTION.
FD  INPUT-FILE
    LABEL RECORDS ARE OMITTED.
01  INPUT-REC.
    ...
WORKING-STORAGE SECTION.
01  STATUSES.
    03 INP-STATUS          PICTURE XX VALUE SPACES.
    ...
01  KEYS.
    03 INP-RELKEY          USAGE NATIVE-4.
```

File-Control Entries for Indexed Files

Indexed organization provides the capability to read records from and write records to a disk file either randomly or sequentially. In an indexed file, the value of the prime key of each record uniquely identifies that record. A file-control entry for an indexed file includes an ORGANIZATION INDEXED clause.



SELECT clause



OPTIONAL

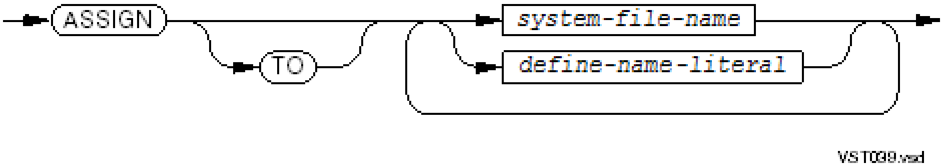
makes the file optional, which means that an OPEN statement with an INPUT, I-O, or EXTEND phrase can open the file whether or not the file exists. If the file exists, its I-O status code is “00”; if not, its I-O status code is “05”. OPTIONAL does not affect the OPEN statement with an OUTPUT phrase.

When you open a nonexistent optional file for input, the first READ statement for that file uses the AT END option (or USE procedure if the READ statement has no AT END phrase).

`file-name`

is the COBOL file-name (the `file-name` in a file description entry).

ASSIGN clause



associates `file-name` with a file designated by `system-file-name`, or `define-name-literal`. Only the first `system-file-name` or `define-name-literal` has meaning. The compiler ignores subsequent names and literals and issues a warning.

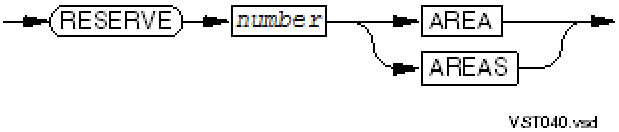
`system-file-name`

is the name of a disk file or either of the special operating system file names #DYNAMIC or #TEMP, described in System-Names (page 75). `system-file-name` must be enclosed in quotation marks unless it begins with a dollar sign ($), a backward slash (\), or a number sign (#), or is a COBOL word. If `system-file-name` is #TEMP, the file cannot have alternate keys. For more information about operating system file names, see the *Guardian Procedure Calls Reference Manual*.

`define-name-literal`

is a nonnumeric literal that represents the name of a DEFINE of class MAP that is associated with a disk file. Quotation marks must enclose any DEFINE name. For information about DEFINE names, see DEFINEs (page 601).

RESERVE clause



enables or prevents sequential block buffering on input and buffered cache on output, or enables or prevents HP COBOL Fast I-O for both input and output, for a disk file, depending on the value of `number`.

`number`

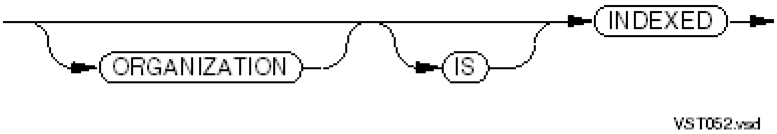
is a numeric literal, an unsigned integer.

`number` must be in the range 1 through 32, and its value is interpreted:

Value of number	Effect
1	No buffering or HP COBOL Fast I-O
2	Sequential block buffering on input and buffered cache on output if the assigned file qualifies
3 or greater	HP COBOL Fast I-O if the assigned file qualifies; if not, sequential block buffering for input and buffered cache for output if the assigned file qualifies; otherwise normal I-O. `number` is the number of blocks to buffer.

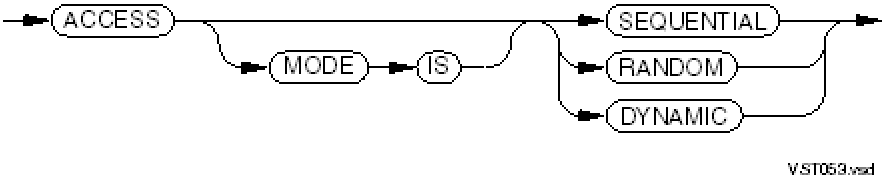
CAUTION: Do not use sequential block buffering for a file opened for shared access. If you do, a process could read outdated data while another process alters the file. For information on shared access, see OPEN (page 385).

ORGANIZATION clause



makes the organization of the file indexed (the default is sequential).

ACCESS MODE clause



SEQUENTIAL

makes the access mode of the file sequential (the default). That means the records of the file are to be operated upon as if they were sequentially organized.

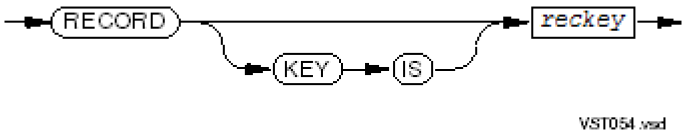
RANDOM

makes the access mode of the file random. That means the records of the file are to be operated upon in any order, as selected by the current value of the record key or of an alternate key. The default access mode is sequential.

DYNAMIC

makes the access mode of the file dynamic. That means the records of the file are accessible by either sequential or random access; you can position the file with START to a certain key value and read or write sequentially from there. The default access mode is sequential.

RECORD KEY clause

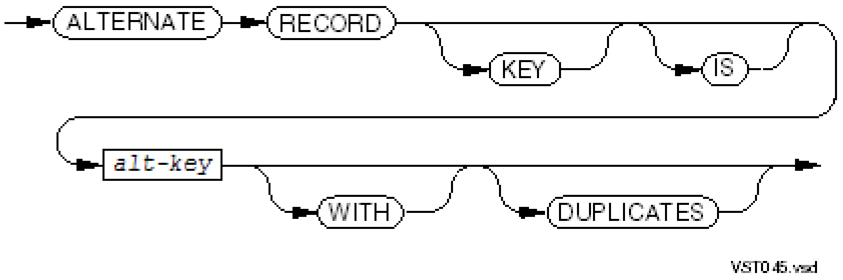


rec-key

is an alphanumeric or unsigned numeric data item defined in the record description entry for the file. It is the prime key for accessing records within the file. If the file was created with FUP or CREATE, the `reckey` size and location must agree with the size and location of the prime key established by FUP or CREATE.

All records in the file are ranked in ascending order of the `reckey` value, which must be unique for each record.

ALTERNATE RECORD KEY clause



makes `alt-key` an alternate record key.

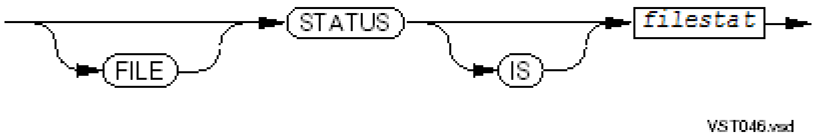
alt-key

is an alphanumeric or unsigned numeric data item declared in the record description entry of the file and is used to gain access to records within the file. The `alt-key` size and location must agree with the size and location of the alternate key within the file, as defined when the alternate key file was established by the operating system File Utility Program (FUP). (For information about FUP, see the *File Utility Program (FUP) Reference Manual*.)

DUPLICATES

means that alternate key values are not necessarily unique.

FILE STATUS clause



defines `filestat` as the file-status data item for the file. When a COBOL run-time I-O routine completes an operation on the file, it stores the status code in `filestat` before returning control to your program (see I-O Status Code (page 257)).

filestat

is a 2-character alphanumeric, nonnational data item defined in the Working-Storage Section, Extended-Storage Section, or Linkage Section.

Usage Considerations:

- Record Key

  The data description entry for *reckey* cannot contain an OCCURS clause.

- Alternate Record Keys

  The records within the file can be accessed in ascending order of the *alt-key* value. The order in which records are obtained using *alt-key* can differ from the order obtained using *reckey*. You can define up to 31 alternate keys for a file.

  If you include the DUPLICATES phrase in the ALTERNATE RECORD KEY clause, the value of *alt-key* need not be unique for each record in the file. Depending on the INSERTIONORDER parameter of the alternate key file, records with duplicate alternate key values are inserted (or retrieved) in either prime key order or in the order in which they were inserted in the file.

  The data description entry for *alt-key* cannot contain an OCCURS clause or be subordinate to an entry that contains an OCCURS clause. The leftmost character position of an *alt-key* item cannot correspond to the leftmost character position of the *reckey* item or another *alt-key* item in that file.

  The file-control entry can contain at most one ALTERNATE RECORD KEY clause that describes a particular alternate record key of the file.

  If a file in the file system is defined as having alternate record keys to which the COBOL program does not make any reference, you do not need to specify them in the File-Control paragraph.

- Sequential Block Buffering

  Sequential block buffering, enabled by the RESERVE clause when the file is open in INPUT or I-O mode, is discussed under File-Control Entries for Sequential Files.

- Buffered Cache

  Buffered cache, enabled by the RESERVE clause when the file is open in OUTPUT, I-O, or EXTEND mode, is discussed under File-Control Entries for Sequential Files.

- HP COBOL Fast I-O (Local Buffering)

  HP COBOLFast I-O is an enhancement in input-output performance beyond that of sequential block buffering or buffered cache. It is available if the files upon which HP COBOL Fast I-O is to operate meet the criteria under File-Control Entries for Sequential Files.

- Performance Penalty for Poorly Organized Indexed File

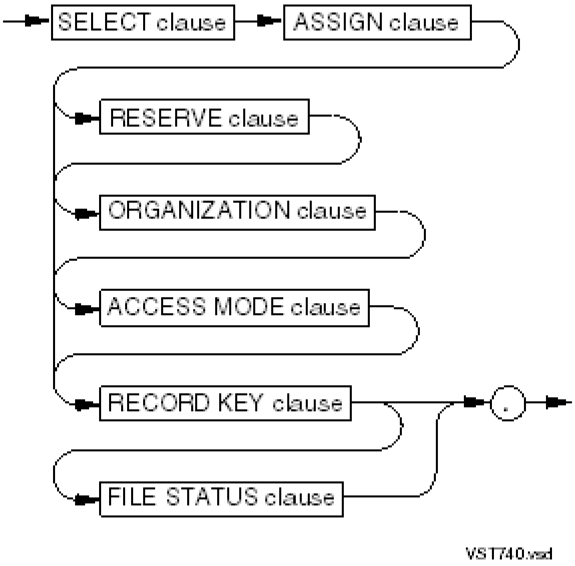
  If a large number of entries have been added to and/or deleted from an indexed file, accessing its records in sequence requires a significant amount of random processing; therefore, performance suffers.

Example 7-7 Indexed File With One Alternate Key

```
INPUT-OUTPUT SECTION.
FILE-CONTROL.
    SELECT RECEIVABLES-MASTER
       ASSIGN TO "=RECMAST"
       ORGANIZATION IS INDEXED
       RECORD KEY IS INVOICE-NUMBER
          ALTERNATE RECORD KEY IS COMPANY-NAME
            WITH DUPLICATES,
          FILE STATUS IS IO-STATUS.
               ...
   DATA DIVISION.
   FILE SECTION.
   FD  RECEIVABLES-MASTER
       LABEL RECORDS ARE OMITTED
       RECORD CONTAINS 39 CHARACTERS.
   01  INVOICE-RECORD.
       05  INVOICE-NUMBER     PICTURE 9(7).
       05  COMPANY-NAME       PICTURE X(15).
       05  INVOICE-DATE       PICTURE 9(6).
       05  INVOICE-AMOUNT     PICTURE S9(9)V99.
           ...
   WORKING-STORAGE SECTION.
   01  STATUSES.
       03  IO-STATUS              PICTURE XX VALUE SPACES.
       ...
```

File-Control Entries for Queue Files

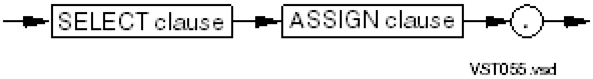
A queue file is an indexed file that can function as a queue. Unlike an ordinary index file, a queue file cannot have alternate keys.



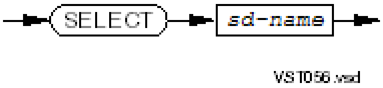
For descriptions of clauses and usage considerations, see File-Control Entries for Indexed Files.

File-Control Entries for Sort-Merge Files

A file-control entry for a sort-merge file defines a scratch file for a sort-merge process.



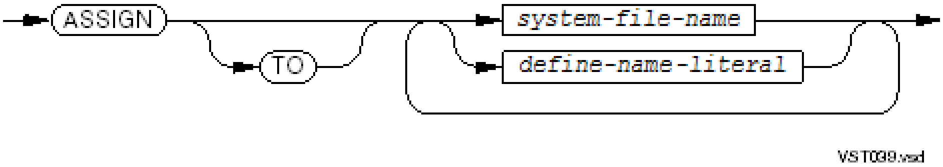
SELECT clause



`sd-name`

is a COBOL file-name (the `file-name` in a sort-merge file description entry).

ASSIGN clause



associates the COBOL file-name (`sd-name`) with a file designated by `system-file-name`, or `define-name-literal`. Only the first `system-file-name` or `define-name-literal` has meaning. The compiler ignores subsequent ones and issues a warning.

`system-file-name`

is the name of a disk file that the file system recognizes. Quotation marks must enclose `system-file-name` unless it is a COBOL word or begins with a dollar sign ($), backward slash (\), or number sign (#). For more information about operating system file names, see the *Guardian Procedure Calls Reference Manual*.

`define-name-literal`

is a nonnumeric literal representing a DEFINE name of type MAP. Quotation marks must enclose `define-name-literal`. For more information about DEFINE names, see DEFINEs (page 601).

Usage Considerations:

- Different Devices for Files

  The sort-merge file is a temporary file used by a SORT or MERGE statement. Programs that define their sort-merge file on a different device than the input or output file of the SORT or MERGE statement must run more efficiently than those defining input or output files on the same device as the sort-merge file.

- Redirecting the Swap File

  The operating system assigns a swap file to swap pages in and out of memory while the compiler is running. The swap file mirrors all of the data areas that the compiler uses. The ideal swap file is a fast device that is neither busy nor mirrored. To redirect the swap file, give `define-name-literal` the value =_SORT_DEFAULTS.

  In Example 7-8, a file-control entry assigns a sort file to a temporary file on the default volume established at run time.

  If you want the temporary file on a different volume than the default, do not use #TEMP. The phrase

  ```
  ASSIGN TO "$FLAG"
  ```

  assigns a temporary file on $FLAG, regardless of the current default volume.

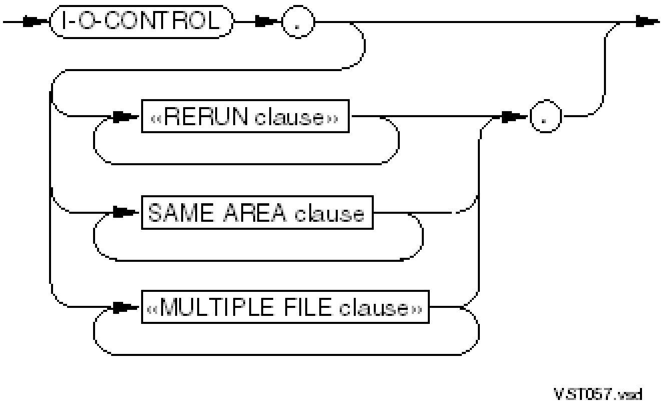
Example 7-8 File-Control Entry for Sort-Merge File

```
INPUT-OUTPUT SECTION.
FILE-CONTROL.
    SELECT SORT-FILE ASSIGN TO "#TEMP".
     ...
DATA DIVISION.
FILE SECTION.
SD  SORT-FILE
    RECORD CONTAINS 40 CHARACTERS.
01  SORT-FIELDS.
     ...
```

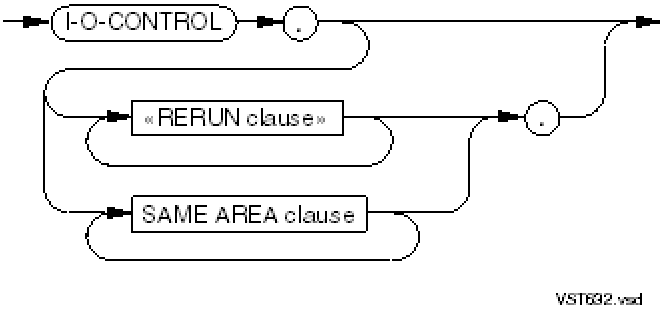
I-O-CONTROL Paragraph

The optional I-O-CONTROL paragraph specifies positioning information for a tape file or the sharing of a memory area by more than one file. If the I-O-CONTROL paragraph is present, it must follow the FILE-CONTROL paragraph, because it refers to files established by file-control entries.

Sequential, relative, or indexed file:

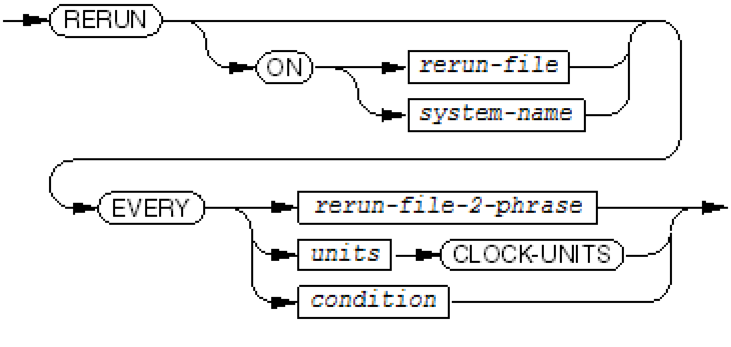


Line sequential file:



RERUN clause

NOTE: The 1985 COBOL standard classifies the RERUN clause as **obsolete**, so you are advised not to use it.

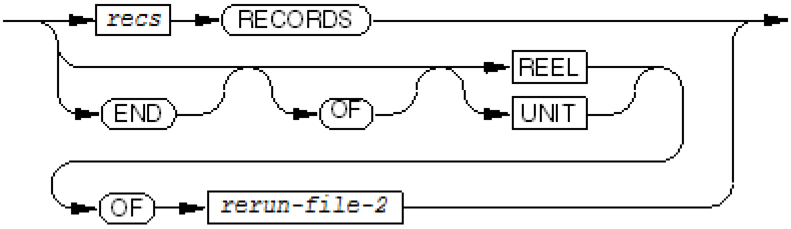


VST058.vsd

rerun-file, *system-name*, *units*, *condition*

are handled as comments.

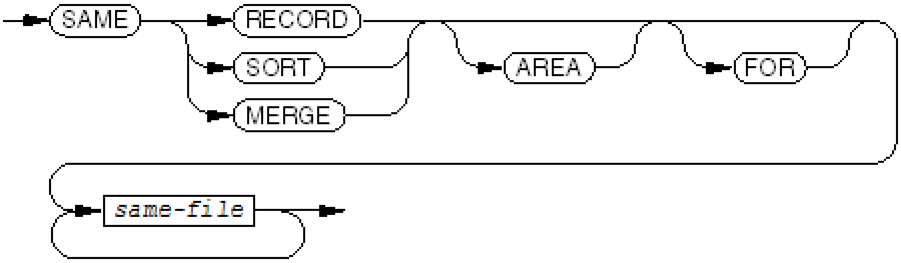
rerun-file-2-phrase



VST059.vsd

is handled as a comment.

SAME AREA clause



VST060.vsd

specifies the files that share the same memory during program execution. These files do not share disk space or tape space.

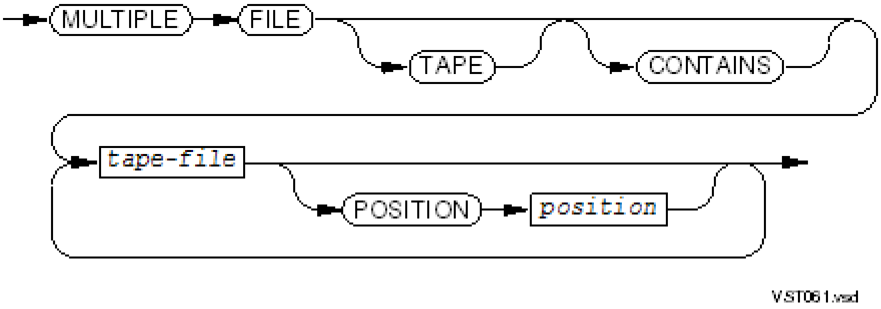
The SAME AREA clause has different meanings for I-O files (sequential, relative, indexed) than for sort-merge files.

same-file

is a file-name.

MULTIPLE FILE clause

NOTE: The 1985 COBOL standard classifies the MULTIPLE FILE clause as **obsolete**, so you are advised not to use it.



specifies a multiple-file tape reel. For more information on multiple files, see MULTIPLE FILE Clause.

`tape-file`

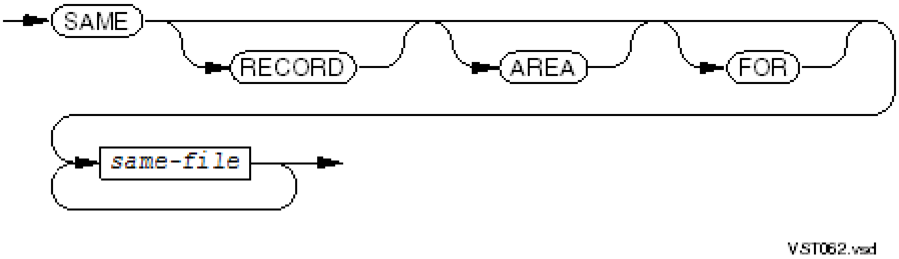
is the name of a sequential file on magnetic tape, defined in a file-control entry and a file description entry.

`position`

is a numeric literal with a value of 1 or more. It defines the relative position of `tape-file` on a tape.

SAME AREA Clause for I-O Files

The SAME AREA clause advises the compiler that two or more files that are not sort-merge files can use the same memory area during processing. Only one of these files can be open at any given time.



RECORD

specifies that two or more files are to use the same memory area for processing the current logical record.

`same-file`

is the `sd-name` in the sort-merge file description entry or the `fd-name` in the file description entry for the file.

The compiler ignores the advice provided by the SAME AREA clause. For information on one way files can share memory, see HP COBOL Fast I-O (Local Buffering) under File-Control Entries for Sequential Files (page 130).

The SAME RECORD AREA clause specifies that two or more files use the same memory area for processing the current logical record. Ordinarily, all the files can be open at the same time, but see the restrictions on the usage of SAME clauses.

A logical record in the SAME RECORD AREA is considered a logical record of each opened output file whose file name appears in this SAME RECORD AREA clause and of the most recently read input file whose file name appears in this SAME RECORD AREA clause. This is equivalent to a redefinition of the area—records are aligned on the leftmost character position.

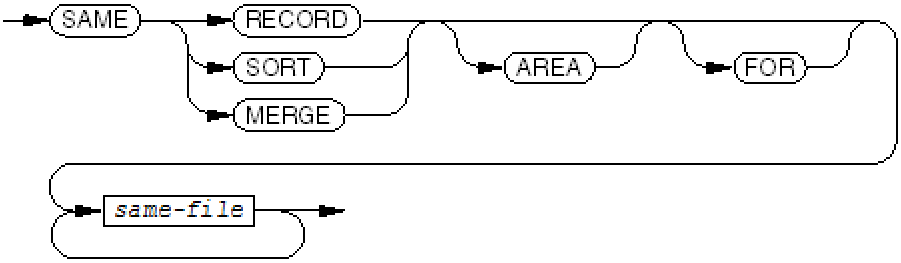
More than one SAME clause can be included in a program; however, there are some restrictions on the usage of SAME clauses:

- A file name must not appear in more than one SAME AREA clause or in more than one SAME RECORD AREA clause.
- If any file names of a SAME AREA clause appear in a SAME RECORD AREA clause, all of the file names in that SAME AREA clause must appear in the SAME RECORD AREA clause; however, additional file names not appearing in that SAME AREA clause can also appear in that SAME RECORD AREA clause. The rule that only one of the files mentioned in a SAME AREA clause can be open at any given time takes precedence over the rule that all files mentioned in a SAME RECORD AREA clause can be open at any given time.

The files mentioned in the SAME AREA or SAME RECORD AREA clause can differ in organization or access.

SAME AREA Clause for Sort-Merge Files

The SAME AREA clause of the I-O-CONTROL paragraph can also specify sort-merge files. HP COBOL ignores the advice provided to the compiler by the SAME AREA clause. The file system automatically allocates and manages all memory areas needed for file processing and for sort or merge operations.



VST060.vsd

RECORD

specifies that two or more files are to use the same memory area for processing the current logical record.

SORT

is ignored by the compiler, but SORT specifies that the compiler can use (and re-use) the same memory area to sort or merge each sort or merge file specified by *same-file*. If SORT is specified, at least one *same-file* must be a sort file.

MERGE

is equivalent to SORT in the SAME AREA clause. If MERGE is specified, at least one *same-file* must be a merge file.

same-file

is the *sd-name* in the sort-merge file description entry or the *fd-name* in the file description entry for the file.

Usage Considerations:

- File Organization and Access Modes

  The files specified in the SAME AREA clause can have different types of organization or different access modes.

- SAME SORT AREA or SAME MERGE AREA Clause

  If the SAME SORT AREA or SAME MERGE AREA clause is used, at least one *same-file* must be a sort or merge file. Files that are not sort or merge files can also be named.

You can include more than one SAME AREA clause in a program; however, if a file name that is not a sort or merge file appears in a SAME AREA clause and one or more SAME SORT AREA or SAME MERGE AREA clauses, then all of the files named in that SAME AREA clause must also be named in each of the affected SAME SORT AREA or SAME MERGE AREA clauses.

This clause specifies that storage be shared:

- The SAME SORT AREA and SAME MERGE AREA clauses advise the compiler that the same memory area can be used in sorting or merging each sort or merge file named and that any memory area used for sorting or merging a sort-merge file can be reused in sorting or merging any other sort-merge files. The compiler ignores the advice provided by this clause.
- Storage areas assigned to files that do not represent sort or merge files can be allocated as needed for sorting or merging the sort-merge files named in the SAME SORT AREA or SAME MERGE AREA clause.
- Files other than sort-merge files do not share the same storage area with each other. If you want these files to share the same storage area, you must also include a SAME AREA or SAME RECORD AREA clause naming the files.
- During the execution of a SORT or MERGE statement referring to a sort or merge file named in this clause, any file named in this clause that is not a sort or merge file must not be open.

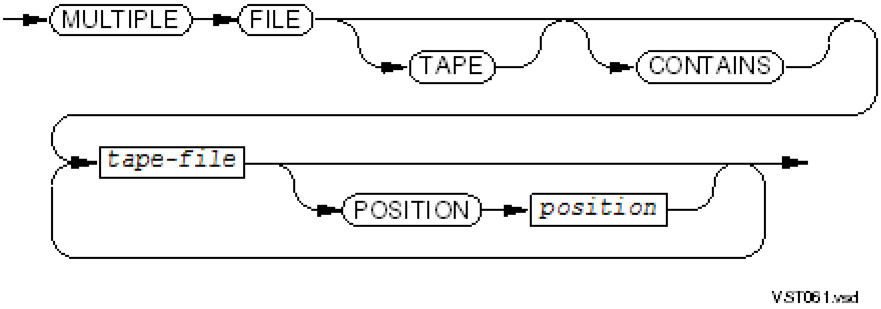
- SAME RECORD AREA Clause

  The SAME RECORD AREA clause specifies that two or more files are to use the same memory area for processing the current logical record. All of the files can be open at the same time. A logical record in the SAME RECORD AREA is considered a logical record of each opened output file whose file name appears in this SAME RECORD AREA clause. This is equivalent to implicit redefinition of the area—records are aligned on the leftmost character position.

  A file name must not appear in more than one SAME RECORD AREA clause.

MULTIPLE FILE Clause

NOTE: The 1985 COBOL standard classifies the MULTIPLE FILE clause as **obsolete**, so you are advised not to use it.



tape-file

is the file name of a file on the tape that the program uses. Regardless of the number of files on a single reel, only files that the program uses need to be defined; however, if any file in the set is not listed, or if the files are not listed in the order they occur on the tape, then each file's position relative to the beginning of the tape must be specified.

POSITION

is unnecessary when you list all the files on the tape in physical order; otherwise, it is required.

`position`

is an integer data item whose value is the ordinal number of a file on the tape. The first file on the tape has position 1; the second, position 2; and so on.

For information on multiple-tape format, see the *Guardian Programmer's Guide*.

RECEIVE-CONTROL Paragraph

You can write server processes in HP COBOL. The system file process named $RECEIVE is the communication mechanism between a requester process and a server process or between the operating system and any process (as if the process were a server). When a COBOL server resumes execution upon the completion of a READ on $RECEIVE, it can require information about the source of the message that was delivered.

To define the two tables used by $RECEIVE, include a RECEIVE-CONTROL paragraph in the Environment Division of your server process. The RECEIVE-CONTROL paragraph used for a run unit is the one for the program that first opens $RECEIVE.

The $RECEIVE mechanism of a server written in HP COBOL uses the receive-control table (sometimes called the Requester table) to record the status of the requesting processes that have opened the server. It uses the reply table to keep copies of the replies it has sent to each requesting process.

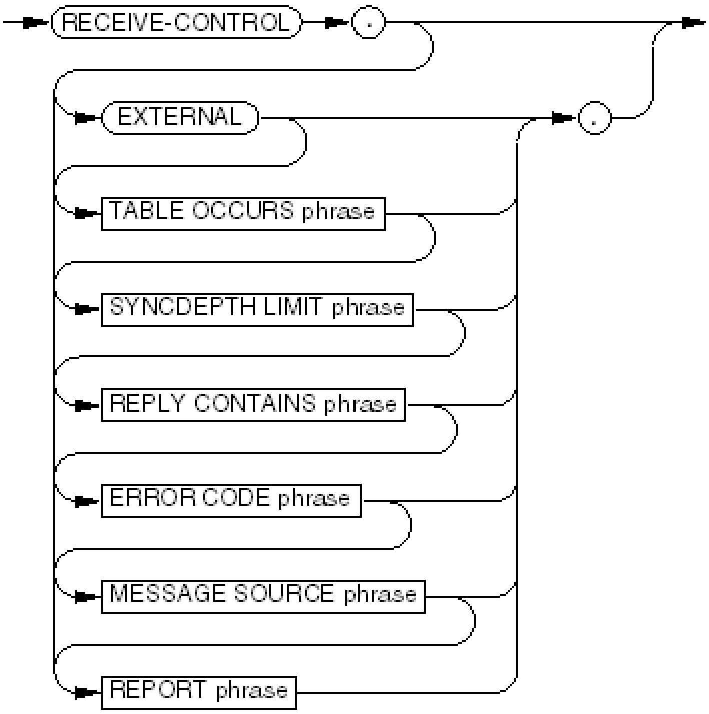
When the requester is running as a process pair and the requester sends a message and the backup process takes over before the requester receives a reply, the server can resend the reply automatically if a CHECKPOINT statement is executed after the server reads the requester's message and before the server writes a reply (or generates one automatically with another READ). In this case, the server does not see the duplicate message. The CHECKPOINT statement can be executed even if the server is not running as a process pair.

In the simplest case, when a single requester and a single server are involved and neither is running as a process pair, the default tables can accommodate the $RECEIVE operations.

The ERROR clause provides a means of specifying that the server itself (and not its run-time routines) takes responsibility for specifying any error number that the $RECEIVE mechanism is to deliver to the requester. The MESSAGE SOURCE phrase provides a means of identifying the sender of any message arriving through $RECEIVE. The REPORT clause provides a means of specifying the classes of messages arriving at $RECEIVE that are to be delivered to the server, rather than handled automatically by the $RECEIVE mechanism.

National literals and national data items cannot be used in a RECEIVE-CONTROL paragraph.

The RECEIVE-CONTROL paragraph for a run unit is the one in the first program to open $RECEIVE.



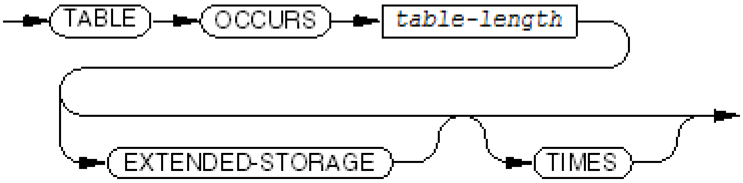
VST063.vsd

EXTERNAL

enables COBOL external files to share communication with $RECEIVE.

EXTERNAL causes the compiler to create a special block, #RECEIVE, which contains the information used when $RECEIVE is opened by a COBOL external file.

TABLE OCCURS phrase



VST064.vsd

establishes the length of the receive-control table, governing the maximum number of requesters that can have this process open concurrently. In a Pathway environment, this number can be decreased at execution time but not increased.

Each entry in the receive-control table records which requester opened the server process and which open operation the requester used (if a requester opens the server process more than once).

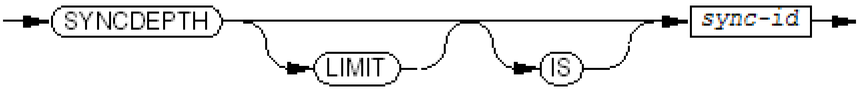
table-length

is an unsigned integer numeric literal in the range 1 through 255. The default value is 1.

EXTENDED-STORAGE

is ignored.

SYNCDEPTH LIMIT phrase



VST065.vsd

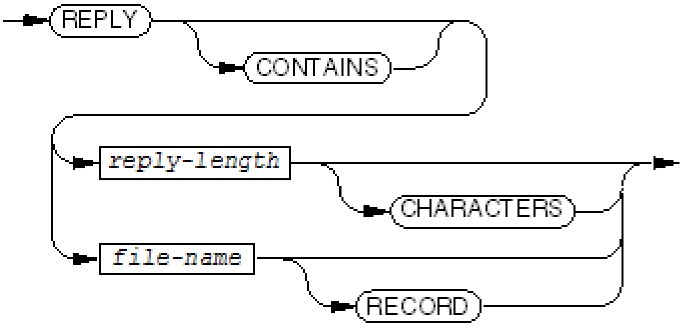
establishes the length of the reply table, controlling how many replies are saved for each requesting process (opener).

sync-id

is an unsigned integer numeric literal in the range 1 through 255. In any requester process that opens the server process, the value in the SYNCDEPTH phrase of the OPEN statement cannot exceed *sync-id*.

When you do not use the SYNCDEPTH LIMIT phrase, *sync-id* defaults to 1. A Pathway server never needs a SYNCDEPTH LIMIT greater than 1.

REPLY CONTAINS phrase



VST066.vsd

specifies the number of characters of a reply message that are to be saved in the reply table. The default is 0.

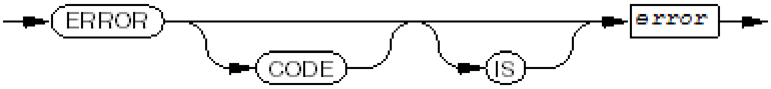
reply-length

is an unsigned integer numeric literal. Its value is the maximum number of characters from a reply message to be saved in the reply table. The upper limit is 2MB.

file-name

is the name of a file whose longest record’s length is the number of characters from a reply message to be saved in the reply table.

ERROR CODE phrase

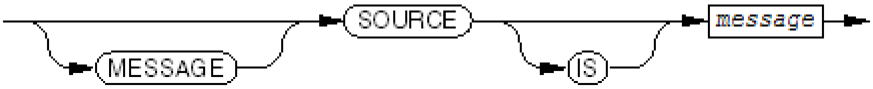


VST067.vsd

error

is an unsigned integer numeric item for storing an error code for a reply message, which you must set. It can be in the Working-Storage Section, the File Section, or the Linkage Section.

MESSAGE SOURCE phrase



VST068.vsd

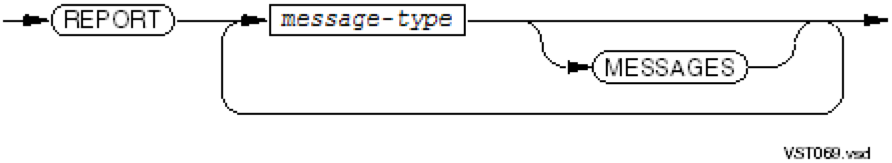
message

is an alphanumeric item that is:

- Defined in the Working-Storage Section, Extended-Storage Section, File Section, or Linkage Section
- Aligned on an even character boundary within a record
- At least 32 characters long
- Not a table
- Not modified by a reference
- Not of a variable size (not a data structure that contains an OCCURS DEPENDING ON clause)

The contents of *message* are updated automatically after a file assigned to $RECEIVE is successfully read.

REPORT phrase



specifies the type of system messages to be passed to the program.

message-type

has one of these values:

BREAK
CLOSE
CONTROL
CONTROLBUF
CPU-DOWN
CPU-UP
DEVICE-INFO
DEVICE-INFO-2-COMPLETION
FILE-GETINFOBYNAME-COMPLETION
FILE-FILENAME-COMPLETION
JOB-PROCESS-CREATION
LOGICAL-CLOSE
LOGICAL-OPEN
MEMORY-LOCK-COMPLETION
MEMORY-LOCK-FAILURE
MESSAGE-CANCELLED
MESSAGE-MISSED
NETWORK
NEWPROCESS-COMPLETION
NEWPROCESSNOWAIT-COMPLETION
NODE-DOWN
NODE-UP
OPEN
PATHSEND-DIALOG-ABORT
POWER-ON
PROCESS-CREATE-COMPLETION
PROCESS-DELETION
PROCESS-TIME-SIGNAL
REMOTE-CPU-DOWN
REMOTE-CPU-UP
RESETSYNC
SETMODE
SETPARAM
SETTIME
STATUS-3270
SYSTEM
TIME-SIGNAL

message-type identifies the class of system message passed to a program. Messages in unnamed classes are handled by the run-time routines in a standard manner.

CONTROL and SETMODE messages are rejected if not requested by a program. BREAK messages are ignored if not requested by a program. SYSTEM as a type signifies a particular subset of the preceding list. The other classes of messages provide the run-time routines with information needed to accept, manage, and delete requesters.

Topics:

- Receive-Control Table
- Reply Table
- ERROR CODE Phrase
- MESSAGE SOURCE Phrase
- REPORT Phrase

Receive-Control Table

The receive-control table, an internal table, is required for $RECEIVE operation. Its purpose is to identify, by the PROCESS-ID, which requesting processes have opened the server process.

Table 7-7 Receive-Control Table Example

Entry Number	Requesting Process
1	REQUESTER PROCESS 1
2	REQUESTER PROCESS 2
3	REQUESTER PROCESS 3
…	…
100*	REQUESTER PROCESS 100

* The number of entries in the receive-control table is defined by `table-length`.

The compiler allocates only one receive-control table because only one file assigned to $RECEIVE can be open for input or input-output at a time. When more than one program unit defines a receive-control table, the compiler reserves space for the largest table.

In the Pathway environment, the value of *`table-length`* for a server must be greater than or equal to the value of the MAXLINKS parameter in the server-class definition for the server.

When the number of active requesters fills the receive-control table, OPEN messages from new requesters are refused with a run-time error message and are not reported to your program. OPEN messages received from backup processes of active requesters are still accepted and reported.

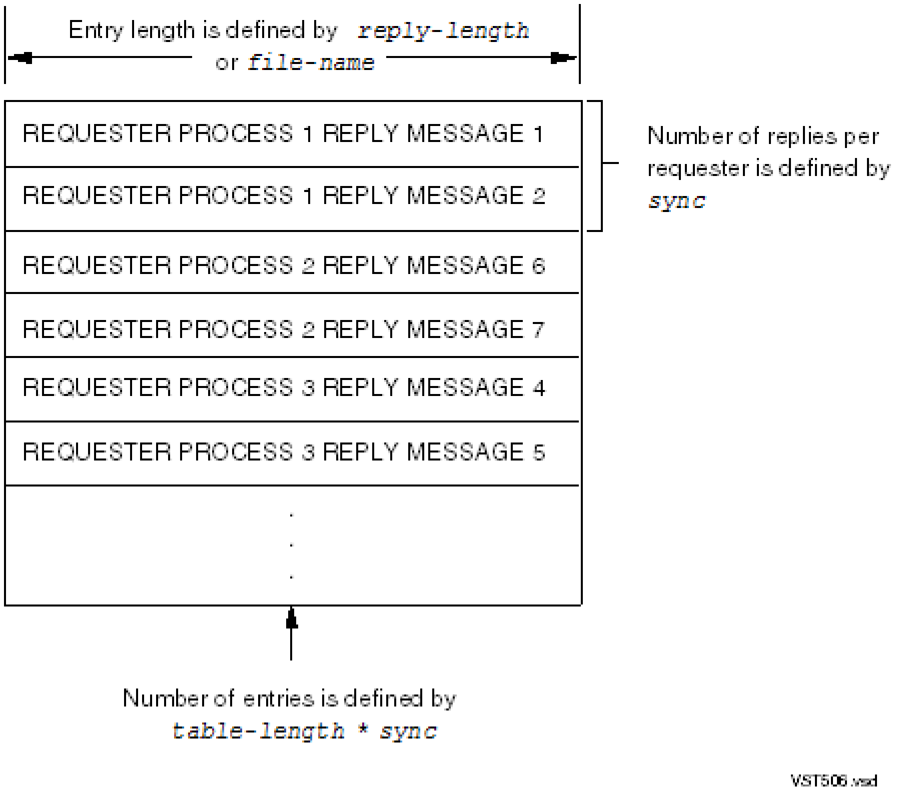
Reply Table

When reply messages are sent back to the requesting processes, the COBOL compiler constructs a second internal table in which to save the replies. The reply message includes the *`sync-id`* and the contents of the reply.

If the reply table exceeds an internal threshold value, the reply table is in the Extended-Storage Section.

NOTE: You do not need a reply table if the requester is not a fault-tolerant process.

Figure 7-1 Reply Table


Entry length is defined by reply-length or file-name
REQUESTER PROCESS 1 REPLY MESSAGE 1
REQUESTER PROCESS 1 REPLY MESSAGE 2
REQUESTER PROCESS 2 REPLY MESSAGE 6
REQUESTER PROCESS 2 REPLY MESSAGE 7
REQUESTER PROCESS 3 REPLY MESSAGE 4
REQUESTER PROCESS 3 REPLY MESSAGE 5
Number of replies per requester is defined by sync
Number of entries is defined by table-length * sync
VST506.vsd

Reply messages saved in the reply table are identified by their requester process. These messages correspond to and answer the specific requester.

For example, if REQUESTER PROCESS 3 in Figure 7-1 failed before it received REPLY MESSAGE 5, and its backup reissued the request based on the checkpoint information, the COBOL fault-tolerant facility would recognize that the request was performed. It would reissue REPLY MESSAGE 5 to REQUESTER PROCESS 3 without re-executing the request.

If a Receive-Control paragraph has no reply table (no SYNCDEPTH LIMIT and REPLY CONTAINS clauses), then replies are not saved. The fault-tolerant facility cannot retransmit them. If a requester's backup process retransmits them, they are not recognized as duplicate requests, because the originals were not saved. Responding to duplicate messages can corrupt the database.

ERROR CODE Phrase

The ERROR CODE phrase of the RECEIVE-CONTROL paragraph provides a way for a server process to appear as a device to its requesters. When the server process sends an error code to a requester, the server process uses file-system error codes to set the ERROR CODE item. How the server process sets the ERROR CODE item depends on the device type of the device the server is simulating.

When the server makes a reply to a READ on $RECEIVE, the operating system passes back the value of the ERROR CODE item separately from the reply-message text. Additionally, the system passes back a condition code (CCE, CCG, or CCL) based on the value of the file-system error number. (See the *Guardian Programmer's Guide* for information about file-system errors and condition codes.)

The requester checks for a condition code other than CCE (successful operation) after its action to send a task message and receive a response. If there is an error (CCG or CCL), the requester must call FILEINFO, a system procedure, to determine the error code (from ERROR CODE item) passed by the operating system. When a requester process is written in COBOL, condition code checking (including the call to FILEINFO) is handled by the COBOL run-time library routines.

When you use ERROR CODE *error*, but the server is not executing a WRITE for each READ on $RECEIVE, the system acknowledges each message from a requester and also passes the value of *error* with each internal reply. If you take this approach, be careful to have the right value in *error* at all times.

The operating system also returns the value of *error* with explicit or system-generated replies to reported system messages (system messages are reported when the REPORT *message-type* MESSAGES option is used). When system messages are not reported, the COBOL run-time library replies to them and generates an error code.

MESSAGE SOURCE Phrase

The MESSAGE SOURCE phrase provides a mechanism through which a COBOL program can discover the sender of each message received. During the successful execution of a read operation on $RECEIVE, the $RECEIVE mechanism assigns a set of values to the storage space designated in the MESSAGE SOURCE phrase.

Example 7-9 MESSAGE SOURCE Phrase

```
01  SOURCE-MESSAGE.
    05  MESSAGE-TYPE       PICTURE  S999 USAGE IS COMPUTATIONAL.
    05  ENTRY-NUMBER       PICTURE   999 USAGE IS COMPUTATIONAL.
    05  FILLER             PICTURE  X(4).
    05  PHANDLE            PICTURE  X(20).
    05  FILLER             PICTURE  X(4).
```

MESSAGE-TYPE

is one of:

Value	Meaning
Zero or greater	Message sent by the requesting process
Less than zero	Message sent by the operating system

ENTRY-NUMBER

is the number assigned to the receive-control table entry for a requesting process. It is in the range set by *table-length*. When certain system messages not associated with any requester, such as CPU-UP and CPU-DOWN, are received, the number is 0.

PHANDLE

is a unique identifier that specifies a process to process-related procedure calls (for more information, see the *Guardian Programmer's Guide*).

REPORT Phrase

Normally, the $RECEIVE mechanism automatically intercepts and processes all messages sent to the run unit by the operating system. By specifying the REPORT clause, you state that the $RECEIVE mechanism is not to intercept certain classes of system messages, but pass them back to the program for processing. The program then supplies the appropriate error code (if the ERROR CODE clause appears), and generates the appropriate response (by performing a write operation or another read operation on $RECEIVE).

The file must have a record length of at least 255 characters.

“System” is an abbreviation for the messages that are generated under program control.

Table 7-8 Message Types That the Program Generates (System)

COBOL Keyword	Message-Type Code
ABEND	Not used—see Table 7-10
CLOSE	-104
CONTROL	-32
CONTROLBUF	-35
MEMORY-LOCK-COMPLETION	-23
MEMORY-LOCK-FAILURE	-24
NEWPROCESSNOWAIT-COMPLETION	-12
OPEN	-103
RESETSYNC	-34
SETMODE	-33
STOP	Not used—see Table 7-10
TIME-SIGNAL	-22

Messages generated by asynchronous hardware events must be explicitly requested.

NOTE: Specifying certain classes of system messages in the REPORT clause does not ensure that the program will receive the system messages. In addition to specifying the system messages in the REPORT clause, these messages must be explicitly requested by executing appropriate Guardian procedures.

Table 7-9 Message Types That the Hardware Generates (Asynchronous)

COBOL Keyword	Message-Type Code
BREAK	-105
CPU-DOWN	-02 or -101
CPU-UP	-03
NETWORK	Not used—see Table 7-10
POWER-ON	-11
SETTIME	-10

See the *Guardian Procedure Errors and Messages Manual* for a description of system messages, their formats, and their implications.

LOGICAL-OPEN and LOGICAL-CLOSE are conventions that enable a program to keep track of its active requesters simply and directly. LOGICAL-OPEN selects only the first OPEN message from each new requester. OPEN messages from requester backups are automatically handled by the run-time routines and not passed to the program. Similarly, LOGICAL-CLOSE selects only the final CLOSE message from a requester. If both are asserted, the program receives a single OPEN message whenever a process or process pair bids for requester access and a single CLOSE message whenever an active requester relinquishes its access. Because the specific message type causes all messages of its class to be passed to the program, the assertion of OPEN (or SYSTEM, which implies OPEN) overrides LOGICAL-OPEN, and the assertion of CLOSE (or SYSTEM) overrides LOGICAL-CLOSE.

Table 7-10 REPORT Clause Message Types

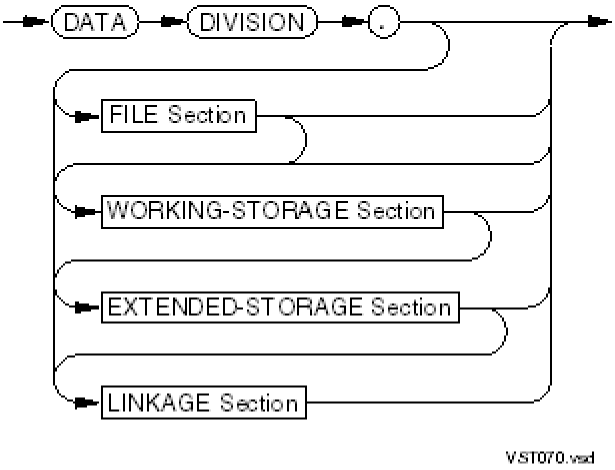
Message Type	Message Type Code	Meaning
BREAK	-20	Break on device
CLOSE	-31	Close
CONTROL	-32	CONTROL system request
CONTROLBUF	-35	CONTROLBUF system request
CPU-DOWN	-02	Local processor failure
CPU-UP	-03	Local processor reload
DEVICE-INFO	-40	Device type inquiry
DEVICEINFO2-COMPLETION	-41	Nowait device type inquiry
FILE-GETINFOBYNAME-COMPLETION	N. A.	Nowait FILE_GETINFOBYNAME_ completion
FILENAME-FINDNEXT-COMPLETION	N. A.	Nowait FILENAME_FINDNEXT_ completion
JOB-PROCESS-CREATION	-09	Job process creation
LOGICAL-CLOSE	N. A.	Logical close
LOGICAL-OPEN	N. A.	Logical open
MEMORY-LOCK-COMPLETION	-23	Memory lock completion
MEMORY-LOCK-FAILURE	-24	Memory lock failure
MESSAGE-CANCELLED	-38	Message cancelled
MESSAGE-MISSED	-13	System message buffer overrun
NEWPROCESS-COMPLETION	-12	NEWPROCESSNOWAIT completion
NEWPROCESSNOWAIT-COMPLETION	-12	NEWPROCESSNOWAIT completion
NODE-DOWN	-08	Lost communication with node
NODE-UP	-08	Established communication with node
OPEN	-30	Open
POWER-ON	-11	Power on
PROCESS-CREATE-COMPLETION	N. A.	Nowait PROCESS_CREATE_ completion
PROCESS-DELETION	-02 -05 -06	Process deletion (processor failure, STOP, ABEND)
PROCESS-TIME-SIGNAL	-26	Process time timeout
REMOTE-CPU-DOWN	-08	Remote processor down
REMOTE-CPU-UP	-08	Remote processor up
RESETSYNC	-34	RESETSYNC system request
SETMODE	-33	SETMODE system request

Table 7-10 REPORT Clause Message Types *(continued)*

Message Type	Message Type Code	Meaning
SETPARAM	-37	Process SETPARAM
SETTIME	-10	Set time
STATUS-3270	-21	3270 device status received
SUBORDINATE-NAME	N. A.	Subordinate name inquiry
SYSTEM	N. A.	All except logical open and close
TIME-SIGNAL	-22	Elapsed time timeout

8 Data Division

The Data Division is optional in a COBOL program. It has four sections: the File Section, the Working-Storage Section, the Extended-Storage Section, and the Linkage Section. Each section contains entries describing data that the program unit being compiled from the source program manipulates. If your program does not use the type of data that the section defines, then the section is optional.



Data Categories and Data Descriptions

Table 8-1 Data Categories

	Category		
	File Data	**Internal Data**	**External Data**
Definition	Data that a process can read from or write to files (including the mapping between the internal program storage and the file storage)	Data that a process develops internally and holds in temporary areas	Data that all programs in the run unit can access
Where Data is Described	File Section	Working-Storage Section Extended-Storage Section Linkage Section	File Section Working-Storage Section Extended-Storage Section
How Data is Described	Data file description entries and sort-merge file description entries, each followed by one or more record description entries	Record description entries and independent data item entries	File descriptions, record description entries, and independent data item entries

Record Description Entries

A record description entry is a set of one or more data description entries. The first data description entry has level number 01. Each additional entry has a level number in the range 02 through 49 or the special level number 66 or 88. Entries with level numbers 02 through 49 and 66 are subordinate data items of the record data item defined in the initial data description entry (which has level number 01). Entries with level number 66 redefine or rename portions of the record. Entries with level number 88 define condition-names.

Data items with level number 77 do not belong to records.

More information:

Record Levels	Sources
01-49	Records (page 89)
66	Descriptions That Rename Items (Level 66)
77	Independent Data Item Description Entries and Descriptions of Noncontiguous Elementary Items (Level 77)
88	Descriptions of Condition-Names for Values (Level 88)

Independent Data Item Description Entries

An independent data item description entry is a set of one or more data description entries. The first entry must have level number 77. Each additional entry has either the special level number 88 (to define condition-names associated with values of the level-77 item) or both the level number 77 and the REDEFINES clause (to make it a redefinition of the preceding level-77 item). Level-77 items are discussed at length in Descriptions of Noncontiguous Elementary Items (Level 77).

Example 8-1 Level-77 Description Entries

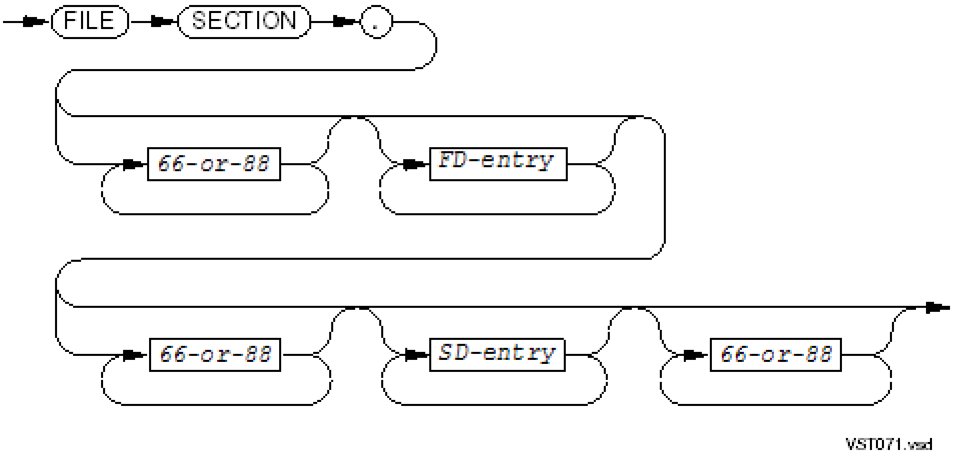
```
77 BUFFER       PIC X(132).
77 ARTIFACT PIC X(10).
   88 CONTAINER VALUES ARE "JAR"  "AMPHORA"  "CANISTER".
   88 WEAPON VALUES ARE "SPEAR"  "BOW"  "KNIFE"  "MISSILE".
77 LOCATION  PIC X(30).
77 OFFICE    PIC X(25) REDEFINES LOCATION.
   88 LOCAL VALUE IS "HEADQUARTERS".
   88 NORTHEAST-USA VALUES ARE "ALLENTOWN" "NEW HAVEN"
                          "CAPE MAY" "WILMINGTON".
   88 SOUTHEAST-USA VALUES ARE "ATLANTA" "MYRTLE BEACH"
```

There is no difference between a level-77 item and an elementary level-01 item, and the latter is preferred.

File Section

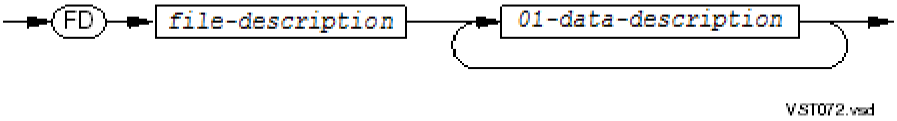
The File Section defines the characteristics of the program's files. Every file name described as a data file (FD) or a sort-merge file (SD) in the File Section must be defined as the same type of file in a corresponding file-control entry of the Environment Division. Conversely, every data file defined in a file-control entry of the Environment Division must be described exactly once in an file description entry of the File Section, and every sort-merge file defined in a file-control entry of the Environment Division must be described exactly once in an sort-merge file description entry of the File Section.

The storage space in the File Section is limited because it is allocated in the lower 64 KB of user data space.



`66-or-88`
: is a level-66 or level-88 description. For syntax, see Descriptions That Rename Items (Level 66) and Descriptions of Condition-Names for Values (Level 88).

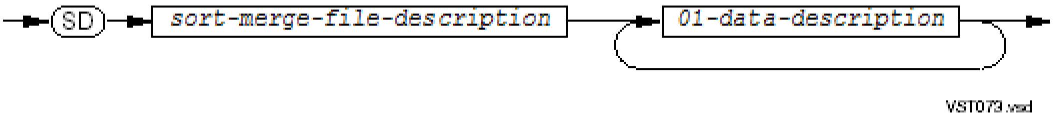
`FD-entry`



`file-description`
: defines the physical aspects of a data file. See File Description Entries.

`01-data-description`
: defines a logical record, specifying the layout of fields within the record and the size and usage of each field. See Data Description Entries.

`SD-entry`



`sort-merge-file-description`
: defines the physical aspects of a sort-merge file. See Sort-Merge File Description Entries.

`01-data-description`
: defines a logical record, specifying the layout of fields within the record and the size and usage of each field. See Data Description Entries.

File Description Entries

The file description (FD) entry is the highest level of organization in the File Section of the Data Division. FD clauses give the size of logical and physical records and the names and attributes of the data records within the file. When a file is to be printed on a printer (either directly, or through a spooler), its file description can include information about how the data is printed on a page.

The characteristics of the file determine which of the clauses are required and which are optional. Clauses that appear can follow in any order, ending with a period after the last clause. No clause can appear more than once. The data description entries of one or more records must follow the file description entry.

Each file description entry must be followed by one or more record description entries. These describe the format or formats of the logical records in the file. When more than one record description entry appears, they can describe record images of different lengths and substructures. All record description entries associated with a single file represent implicit redefinitions of the file's record area. The file name in the file description entry can be used as the final qualifier in references to its record items, their subordinate data items, or condition-names associated with any of these.

When more than one of the source programs compiled into a run unit includes a file description entry defining the same external file name, all of these descriptions must specify the same block size convention, character code convention, labeling convention, linage attributes, and record attributes. For details, see EXTERNAL Clause.

The file description entry describes the logical characteristics of a data file.

The record description entries associated with a file description entry define the possible formats of the logical records for that file. Although different record descriptions define different types of logical records from the perspective of the source program, the corresponding logical records in the file might not have different representations. Conceptually, every record description applies to every logical record in the file; therefore, when the file actually contains different types of logical records, it is your responsibility to code the program such that it determines which of the record descriptions are appropriate for each particular record.

In Example 8-2, two record description entries (level-01 items) follow a file description entry.

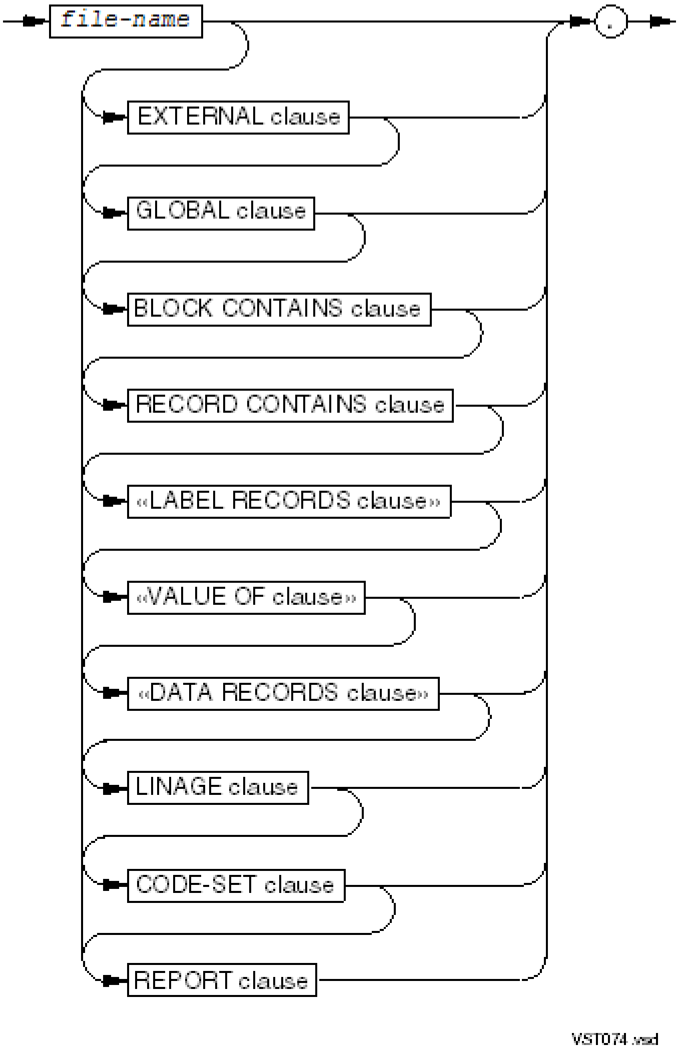
Example 8-2 File Description (FD) Entry

```
FD  INPUT-FILE-MASTER
    GLOBAL
    RECORD IS VARYING IN SIZE FROM 10 TO 256 CHARACTERS

01  OFFICE-DETAILS.
    03  NUMBER    PIC 9(5).
    03  ADDRESS.
        05 LINE-1 PIC X(25).
        05 LINE-2 PIC X(25).
        05 LINE-3 PIC X(25).
    03  HEAD-COUNT PIC 999.
    03  MANAGER-ID-NUMBER 9(5).

01  EMPLOYEE-DETAILS.
    03 ID-NUMBER PIC 9(5).
    03 TITLE     PIC X(25).
    03 MANAGER-ID-NUMBER PIC 9(5).
    03 SALARY-LEVEL PIC X(2).
```

The syntax of a file description entry (*file-description*) depends on the file's access mode:

- File Description Entry for Sequential File
- File Description Entry for Line Sequential File (OSS environment only)
- File Description Entry for Relative, Indexed, or Queue File

File Description Entry for Sequential File



file-name

is the highest-level qualifier for both a file description entry and its data descriptions; therefore, the name must be unique within a program.

EXTERNAL clause

is described in EXTERNAL Clause.

GLOBAL clause

is described in GLOBAL Clause.

BLOCK CONTAINS clause

is described in BLOCK CONTAINS Clause.

RECORD CONTAINS clause

is described in RECORD CONTAINS Clause.

LABEL RECORDS clause

is described in LABEL RECORDS Clause.

VALUE OF clause

is described in VALUE OF Clause.

DATA RECORDS clause

is described in DATA RECORDS Clause.

LINAGE clause

is described in LINAGE Clause.

CODE-SET clause

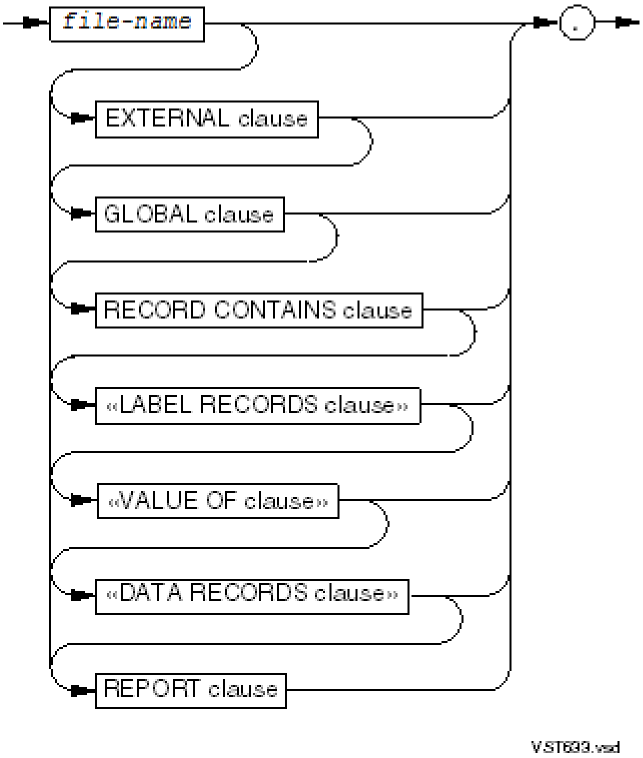
is described in CODE-SET Clause.

REPORT clause

is described in REPORT Clause (page 184).

File Description Entry for Line Sequential File

NOTE: Available only in the OSS environment.



file-name

is the highest-level qualifier for both a file description entry and its data descriptions; therefore, the name must be unique within a program.

EXTERNAL clause

is described in EXTERNAL Clause.

GLOBAL clause

is described in GLOBAL Clause.

RECORD CONTAINS clause

is described in RECORD CONTAINS Clause.

LABEL RECORDS clause

is described in LABEL RECORDS Clause.

VALUE OF clause

is described in VALUE OF Clause.

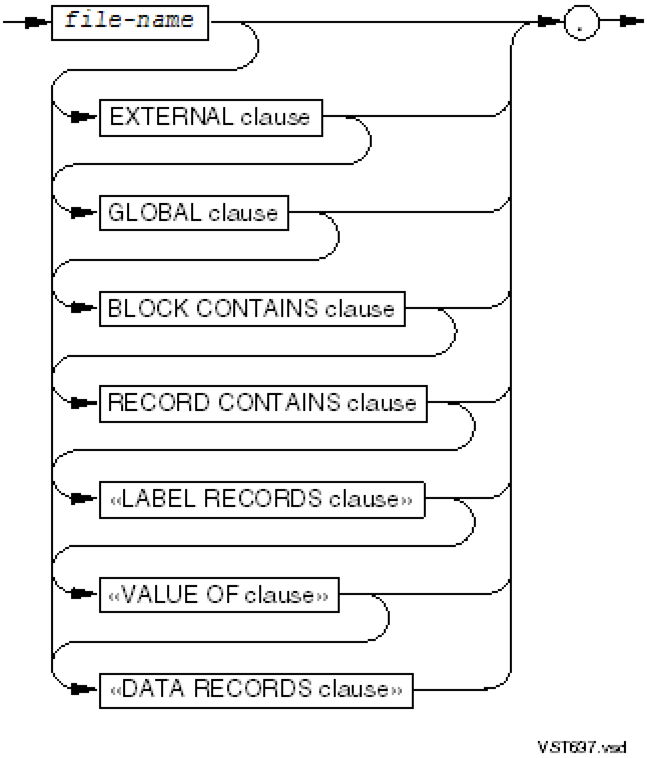
DATA RECORDS clause

is described in DATA RECORDS Clause.

REPORT clause

is described in REPORT Clause.

File Description Entry for Relative, Indexed, or Queue File



file-name

is the highest-level qualifier for both a file description entry and its data descriptions; therefore, the name must be unique within a program.

EXTERNAL clause

is described in EXTERNAL Clause.

GLOBAL clause

is described in GLOBAL Clause.

BLOCK CONTAINS clause

is described in BLOCK CONTAINS Clause

RECORD CONTAINS clause

is described in RECORD CONTAINS Clause.

LABEL RECORDS clause

is described in LABEL RECORDS Clause.

VALUE OF clause

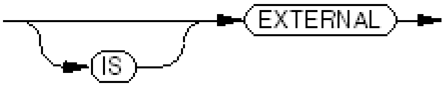
is described in VALUE OF Clause.

DATA RECORDS clause

is described in DATA RECORDS Clause.

EXTERNAL Clause

The EXTERNAL clause gives the file connector referenced by *file-name* the external attribute, meaning that it belongs to the run unit rather than to any single program in the run unit. When more than one program in a run unit specifies a file connector with the same file name and the external attribute, they all refer to the same file connector. For an explanation of external objects (those with the external attribute), see External and Internal Objects (page 62).



VST075.vsd

Usage Considerations:

- Sharing a File Connector

  If more than one of the source programs compiled into a run unit contain file description entries defining the same file name, all entries that include the EXTERNAL clause describe the same external file connector.

  Any file name whose file description entry does not include the EXTERNAL clause refers to an internal file connector, even if the same file name identifies an external file connector in some other program of the run unit.

- EXTERNAL Clause Inherited by Data Items in External Files

  The EXTERNAL clause specifies that the file connector associated with the file name defined by the containing file description entry is an external file connector, and that the data items associated with all of the record descriptions for this file name are external data items; therefore, any other source program compiled into the same run unit can share these resources by including a file description entry that defines the same file name and contains the EXTERNAL clause.

- Other Related Data Items Required to be External

  Any data items referenced in the clauses of a file description entry containing the EXTERNAL clause (such as the LINAGE clause or the VARYING DEPENDING clause) or in the clauses of the corresponding file-control entry (such as the FILE STATUS clause or the ALTERNATE RECORD KEY clause) must be external data items.

- Consistency of External Files Throughout the Run Unit

  All descriptions of the same external file connector, and the file that it references, must be functionally identical. The detailed restrictions are:

  — File-control entries

    All of the file-control entries must specify or imply the same operating system file name, organization, and access mode. If any of the file-control entries includes the OPTIONAL phrase, all of them must do so.

  — Keys

    If any of the file-control entries specifies a relative key, record key, or alternate record key, all of them must specify the same external data item for that purpose. Each alternate record key must have a consistent DUPLICATES attribute.

  — PADDING CHARACTER clause

    If any of the file-control entries includes a PADDING CHARACTER clause, all of them must include a PADDING CHARACTER clause that specifies the same padding attribute. In particular, when the value of a data item supplies the padding character, the same external data item must be specified for this purpose in each of the clauses.

  — RECORD DELIMITER clause

    If any of the file-control entries includes a RECORD DELIMITER clause specifying a system-name, all of them must include a RECORD DELIMITER clause specifying the same system-name.

  — BLOCK CONTAINS clause

    If any of the file description entries includes a BLOCK CONTAINS clause, all of them must include a BLOCK CONTAINS clause that specifies the same block size attribute.

- RECORD CONTAINS clause

  If any of the file description entries includes a RECORD CONTAINS clause, all of them must include a RECORD CONTAINS clause that specifies the same record attributes (fixed-length or variable-length, fixed record size or minimum and maximum record sizes, and the external data item that reflects the current logical record size, if any).

- LABEL RECORDS clause

  NOTE: The 1985 COBOL standard classifies the LABEL RECORDS clause as **obsolete**, so you are advised not to use it.

  If any of the file description entries includes a LABEL RECORDS clause with the STANDARD phrase, all of them must include a LABEL RECORDS clause with the STANDARD phrase.

- LINAGE clause

  If any of the file description entries includes a LINAGE clause, all of them must include a LINAGE clause that specifies the same logical page attributes. In particular, when the value of a data item supplies one of the attributes, the same external data item must be specified for this purpose in each of the clauses.

- CODE-SET clause

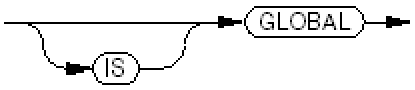
  If any of the file description entries includes a CODE-SET clause that references an alphabet-name associated with a system-name, all of them must include a CODE-SET clause that references an alphabet-name associated with the same system-name.

- Open Mode States

  At the beginning of each execution of the run unit, the open mode state for each external file connector is Closed and the status of all other dynamic file attributes is undefined. If the open mode state of an external file connector is not Closed or Locked when execution of the run unit terminates, it is closed by the execution of an implicit CLOSE statement without any of the optional phrases.

GLOBAL Clause

The GLOBAL clause makes *file-name* a global name, meaning that it is available to every program contained within the program that describes it. The contained programs do not contain a description of the file. For an explanation of global names, see Global and Local Names (page 62).



VST076.vsd

Usage Considerations:

- Referencing Global Items

  A statement in a program contained directly or indirectly within a program that describes a global name can reference that name without describing it again.

- Subordinates of Global Names Are Global Names

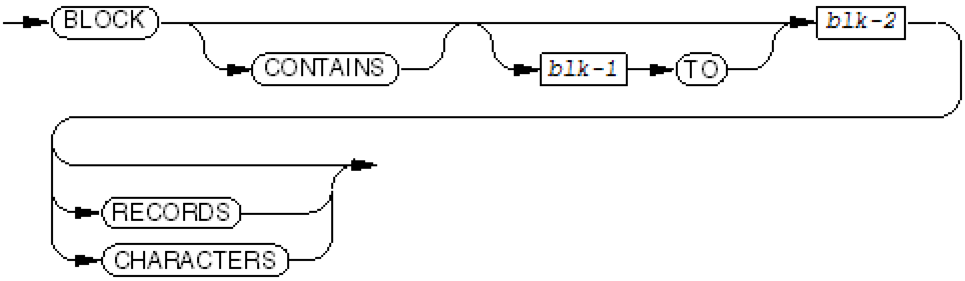
  A file name described using a GLOBAL clause is a global name. All data-names subordinate to a global name are global names. All condition-names associated with a global name are global names.

- SAME RECORD AREA Clause

  If the SAME RECORD AREA clause is specified for several files, the record description entries or the file description entries for these files must not include the GLOBAL clause.

BLOCK CONTAINS Clause

Use the optional BLOCK CONTAINS clause to specify the number of logical records or the number of character positions in a physical record on tape or unstructured disk files only. For these types of files, if this clause is absent, the compiler assigns a block size of one logical record. The operating system handles all other file blocking and deblocking in a way that is transparent to the COBOL program.



VST077.vsd

blk-1

is an unsigned integer literal that specifies the minimum size of a block.

blk-2

is an unsigned integer literal. In the presence of *blk-1*, *blk-2* is the fixed size of a block; otherwise, it is the maximum size of a block.

RECORDS

specifies that the values of *blk-1* and *blk-2* represent numbers of logical records.

CHARACTERS

specifies that the values of *blk-1* and *blk-2* represent numbers of character positions. This is the default.

Usage Considerations:

- Limits

  The minimum number of character positions you can specify is equal to the maximum record size of the file. If you specify more than 32,767 character positions, the compiler reports an error.

  If you specify a number of records greater than 32,767 divided by the maximum record size for the file, the compiler reports an error.

  When *blk-1* is present, its value must be less than or equal to that of *blk-2*. When the RECORDS option is not specified, the value of *blk-1* must be greater than or equal to the minimum record size of the file; otherwise, the value of *blk-1* must be greater than 0.

Although tape devices are capable of handling blocks of 32,767 characters, unstructured disk files are limited to a block size of 4,096. Furthermore, an unstructured disk file can have a BLOCKSIZE attribute that is smaller than 4,096. Other devices have their own block size limitations. See the discussion of the WRITE procedure call in the *Guardian Procedure Errors and Messages Manual* for these limits.

Because the compiler cannot determine the block size of the device that is ultimately associated with a file, you must choose the correct block size and record size or risk getting a run-time error.

- Blocking and Deblocking for Tape and Unstructured Disk

  A BLOCK CONTAINS clause is effective (HP COBOL performs record blocking and deblocking) only when all of these conditions are met, and the file associated at open time is either a tape unit or an unstructured disk file:

  — The file's organization is sequential.

  — The file has fixed-length records. This condition is met only when the file description contains either no RECORD CONTAINS or RECORD VARYING clause or contains a RECORD CONTAINS clause that specifies only a single record size.

  — When the block size is specified in characters, it is a multiple of the number of characters in the logical record size. Also, when a RECORD CONTAINS clause extends the record size, the block size expressed is a multiple of the number of characters in that specified record size.

  — The file is not described with a LINAGE or ALTERNATE RECORD KEY clause.

  — Tape Files

  If the actual file associated with the COBOL file at open time is a tape unit, then blocking/deblocking always occurs. For an unlabeled tape file, the maximum block size is the one specified in the BLOCK CONTAINS clause. For a labeled tape file, the maximum block size specified in the tape label (which must be an exact multiple of the logical record size) overrides the one specified in the BLOCK CONTAINS clause.

  If the file is opened for INPUT, then each physical block on the tape medium must contain one or more complete logical records. The size of a block must not exceed the maximum block size; however, any block can be smaller than the maximum block size (can contain fewer than the potential number of logical records).

  — Unstructured Disk Files

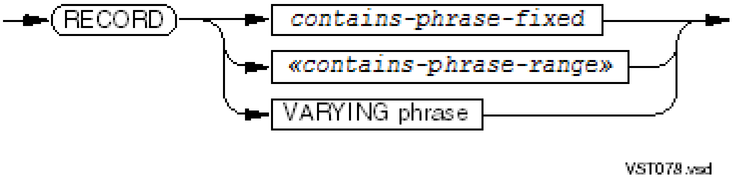
  If the actual file associated with the COBOL file at open time is an unstructured disk file, then blocking/deblocking occurs if the logical record size is an even number of character positions or the file has the odd-length access attribute; otherwise the BLOCK CONTAINS clause is ignored (that is, logical records are physically read or written one at a time). (To have the odd-length access attribute, a file must be created with FUP CREATE and ODDUNSTR must be specified.)

  Odd-length records are written as even-length unless ODDUNSTR is active.

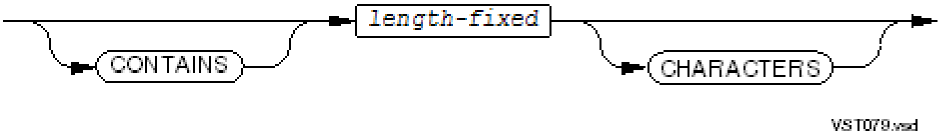
  The compiler issues a warning if blocking is specified when the conditions, other than variable-length records, are not satisfied.

RECORD CONTAINS Clause

Use the RECORD CONTAINS clause to specify whether the records of a file are of fixed or variable length and to document the size of the records.



contains-phrase-fixed



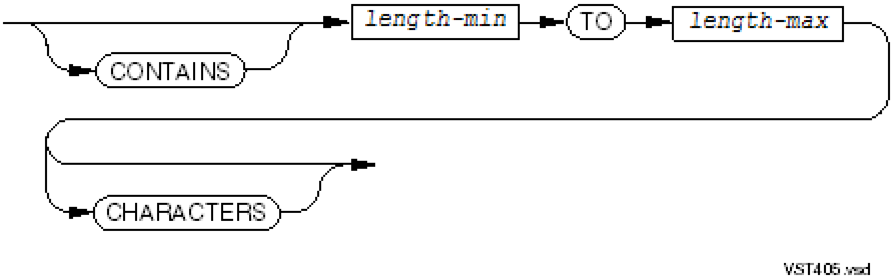
length-fixed

is an unsigned integer literal that specifies the exact length, in characters, of fixed-length records. If the ASSIGN clause specifies $RECEIVE, the value of *length-fixed* is in the range of 0 through 2,097,152 bytes (2MB). Otherwise, its value is in the range of 0 through 32,767 bytes.

«contains-phrase-range»

NOTE: The 1985 COBOL standard classifies *contains-phrase-range* as **obsolete**, so you are advised not to use it. Instead, use a VARYING phrase of the form:

RECORD IS VARYING FROM *length-min* TO *length-max*



length-min

is an unsigned integer literal that specifies the minimum length, in characters, of variable-length records. If the ASSIGN clause specifies $RECEIVE, the value of *length-min* is in the range of 0 through 2,097,152 bytes (2MB). Otherwise, its value is in the range of 0 through 32,767 bytes.

No record description entry for the file described with *length-min* can specify a number of character positions less than the value of *length-min*.

When the RECORD clause appears in a data file description entry, no record key or alternate record key defined for the file can be described as beginning after or extending beyond the number of characters specified by *length-min*.

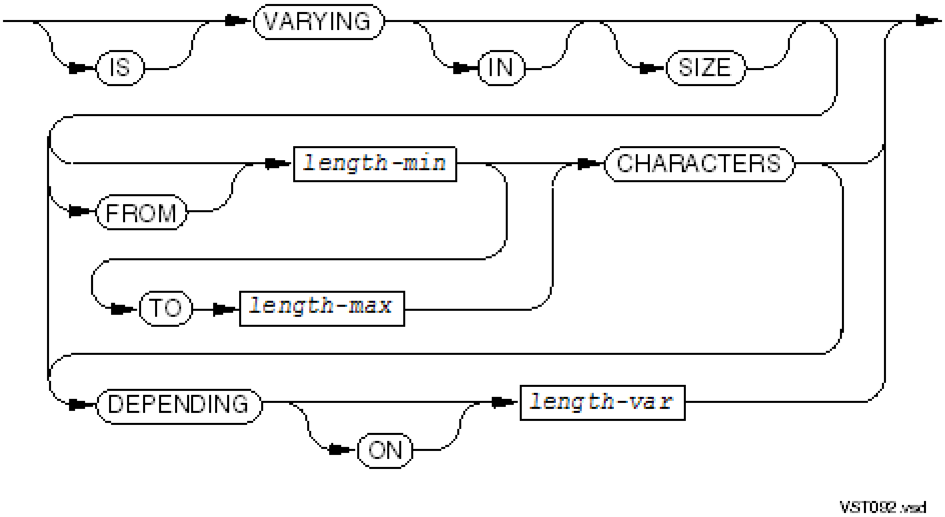
length-max

is an unsigned integer literal that specifies the maximum length, in characters, of variable-length records. Its value must be greater than or equal to the value specified by *length-min* (for the RECORD CONTAINS form of variable-length specification) or greater than the value specified by `length-min` (for the RECORD VARYING form of variable-length specification).

If the ASSIGN clause specifies $RECEIVE, the value of `length-max` has an upper limit of 2,097,152 bytes (2MB). Otherwise, the upper limit is 32,767 bytes.

No record description entry for the file described with `length-max` can specify a number of character positions greater than the value of `length-max`.

VARYING phrase



explicitly declares that the file consists of variable-length records (records of different sizes).

length-min

is as defined previously. The default is the size of the shortest record description entry.

length-max

is as defined previously. The default is the size of the longest record description entry.

length-var

is the data-name of an unsigned integer numeric data item. It tells the REWRITE and WRITE statements how many characters to deliver, and receives the size (in characters) of any record that was read successfully. It can be qualified, but not subscripted or reference-modified.

Record size is the actual number of bytes needed to store the record, determined by the sum of bytes for each fixed-length elementary item plus the maximum number of bytes for any variable-length item. The number of bytes for an item also depends on its USAGE clause, and if synchronization is active, any filler from alignment on storage boundaries.

Usage Considerations:

- Describing Fixed-Length Records

  The RECORD CONTAINS `length-fixed` CHARACTERS form specifies that the file consists of fixed-length records. Both other forms specify that the file consists of variable-length records.

  If the record you write is smaller than the fixed length, the extra characters are undefined.

  If you declare a file for a terminal, you must specify variable-length records, or else each record entered must be exactly the same length.

- Defaulting to Fixed-Length Records

  If no RECORD clause is present, the compiler declares the file to consist of fixed-length records. In this case, the record length is the size of the associated record description entry specifying the greatest number of character positions.

- File With Fixed-Length Records Can Hold Records of Shorter Lengths

  If you describe a file as having fixed-length records, each record description for that file's record area can describe a record of any length from 1 up to the stated fixed length.

  The length you specify in the RECORD CONTAINS clause is the fixed length of the record area that holds individual records read from or to be written to the file during execution. Each record description entry can define the record area in a different way, possibly describing a record of a shorter length than the actual record has in the record area or in the file.

- Example of a File with Fixed-Length Records

  This explicitly declares a file to have fixed-length records:

  ```
  RECORD CONTAINS 256 CHARACTERS
  ```

- Purpose of Variable-Length Records

  If you describe a file as having variable-length records, the individual record descriptions for that file can be of any length from the stated or implied minimum length to the stated or implied maximum length. The specified minimum size can be less than that implied by the record description entry defining the smallest number of character positions. The specified maximum size can be greater than that implied by the record description entry defining the largest number of character positions; however, only those character positions defined in the record descriptions are accessible to the program.

  If the record you write is smaller than the minimum length, the extra characters are undefined.

- Depending Item

  When `length-var` is specified, the data item it references is called the depending item.

  — Governs the length of the record to be written

  The contents of this item, evaluated just before the execution of a REWRITE or WRITE statement for the file, determine the size of the record to be written.

  — Contains the length of the record after it has been read

  After the successful execution of a READ statement for the file, the contents of the depending item indicate the size of the item just read.

  The size is expressed in characters.

  The execution of a DELETE, RELEASE, REWRITE, START, or WRITE statement, or the unsuccessful execution of a READ statement, does not alter the contents of the depending item.

- Establishing the Length of a Variable-Length Record

  When the run-time routines execute a REWRITE or WRITE statement referencing a file of variable-length records, the number of character positions in the record written is determined:

  — If `length-var` is specified, by the content of the data item referenced by `length-var`.

  — If `length-var` is not specified and the record does not contain a variable-occurrence data item, by the number of character positions in the record (as specified in the record description).

  — If `length-var` is not specified and the record contains a variable-occurrence data item, by the sum of the size of the fixed portion and that portion of the table described by the number of occurrences at the time of execution of the output statement.

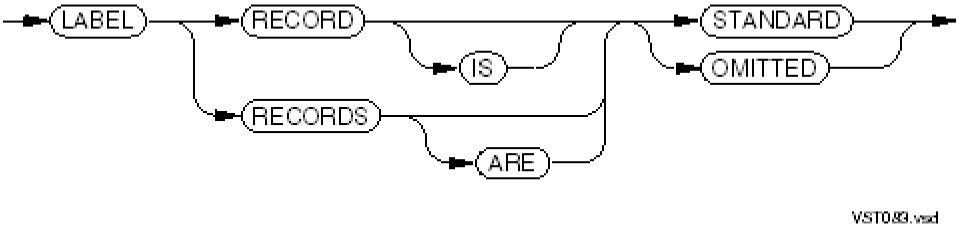
  Example 8-3 Variable-Length Record

  ```
  RECORD IS VARYING IN SIZE FROM 100 TO 144 CHARACTERS
  ```

LABEL RECORDS Clause

NOTE: The 1985 COBOL Standard classifies the LABEL RECORDS clause as **obsolete**, so you are advised not to use it.

On systems that support file labels, this clause specifies whether the file being described has labels or not. HP COBOL does not support labeling or label processing. If you want to use file labels, you must create your own and check them.



STANDARD

specifies that standard system labeling conventions apply to the file.

OMITTED

specifies that no explicit labels exist for the file.

Usage Considerations:

- File Description Entry Without LABEL RECORDS Clause

  When a file description entry does not have a LABEL RECORDS clause, HP COBOL assigns the OMITTED attribute to the file.

- STANDARD or OMITTED

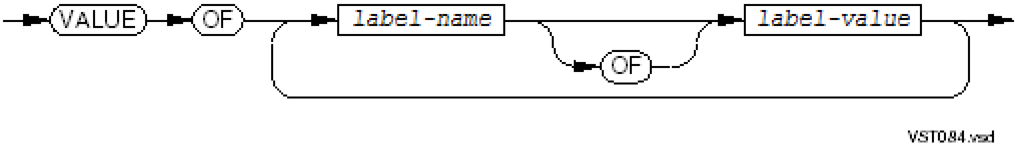
  The choice of STANDARD or OMITTED is significant only for files residing on tape.

- LABEL RECORDS Clause With MULTIPLE FILE TAPE Clause

  When the LABEL RECORDS clause appears in the file description entry of any file name mentioned in a MULTIPLE FILE TAPE clause, the specified labeling convention must apply to all of the file names mentioned in that clause. In practical terms, this means that every file name appearing in a particular MULTIPLE FILE TAPE clause must be described with the LABEL RECORDS STANDARD clause if any of them are so described.

VALUE OF Clause

NOTE: The 1985 COBOL Standard classifies the VALUE OF clause as **obsolete**. HP COBOL does not support the VALUE OF clause. If this clause is present, the compiler checks its syntax and issues a warning.



label-name

is a COBOL word.

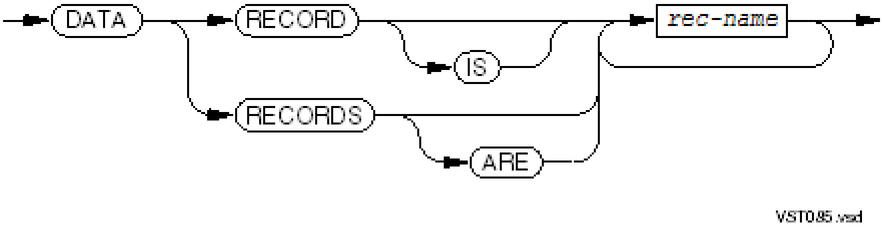
label-value

is either a literal or a COBOL word.

DATA RECORDS Clause

NOTE: The 1985 COBOL Standard classifies the DATA RECORDS clause as **obsolete**, so you are advised not to use it.

Use the optional DATA RECORDS clause to enumerate the names of the records that are defined for the file. Each data-name corresponds to one level-01 name in the record descriptions following the file description. The existence of more than one data-name indicates the file has more than one type of data record. These records can vary in size, format, and so on, and can be listed in any order. All data records within a file share the same memory area.



rec-name

is the name of a record that is to follow the file definition.

Usage Considerations:

- Correspondence

  Each *rec-name* specified in the DATA RECORDS clause must correspond to some level-01 name in the list of record description entries. Order is not important.

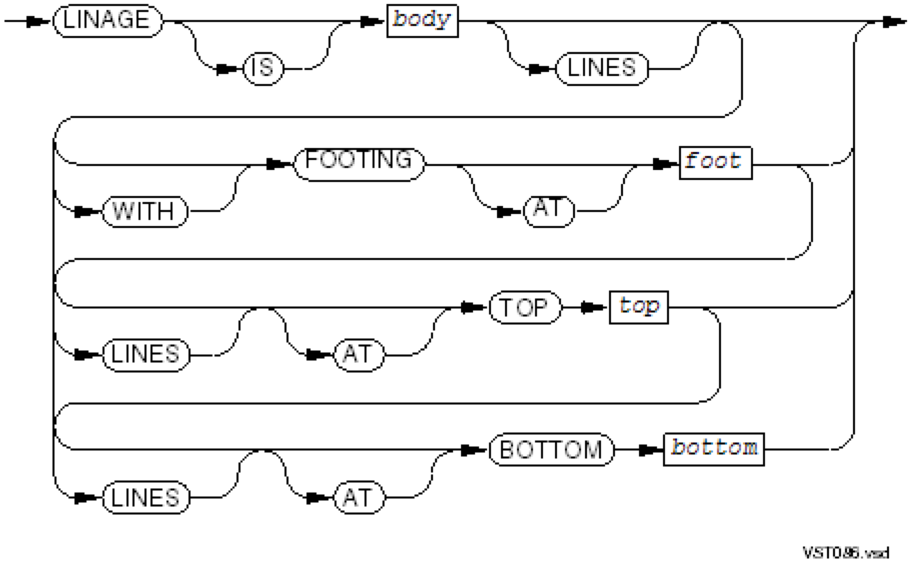
  The converse is not true; level-01 entries can exist among the record description entries without their names appearing in the DATA RECORDS clause.

- FILLER Keyword and the DATA RECORDS Clause

  Although a record description can, except in certain circumstances, be named FILLER or have no name at all (the implicit FILLER), the name FILLER cannot appear as a *rec-name* in the DATA RECORDS clause.

LINAGE Clause

The LINAGE clause controls where data is printed on a page. Top and bottom margins and a body area with an optional footing area within it are defined in terms of the number of lines each has.



body

is either an unsigned integer literal or the data-name of an elementary unsigned numeric integer data item. Its value is the number of lines that can be written on a logical page, so it must be greater than 0.

foot

is either an unsigned integer literal or the data-name of an elementary unsigned numeric integer data item. Its value is the line number within the page body at which the footing area begins, so it must be greater than 0 and not greater than the value of *body*. The default is *body* +1.

top

is either an unsigned integer literal or the data-name of an elementary unsigned numeric integer data item. Its value is the number of lines in the top margin of the logical page. The default is 0.

bottom

is either an unsigned integer literal or the data-name of an elementary unsigned numeric integer data item. Its value is the number of lines in the bottom margin of the logical page. The default is 0.

Example 8-4 LINAGE Clauses

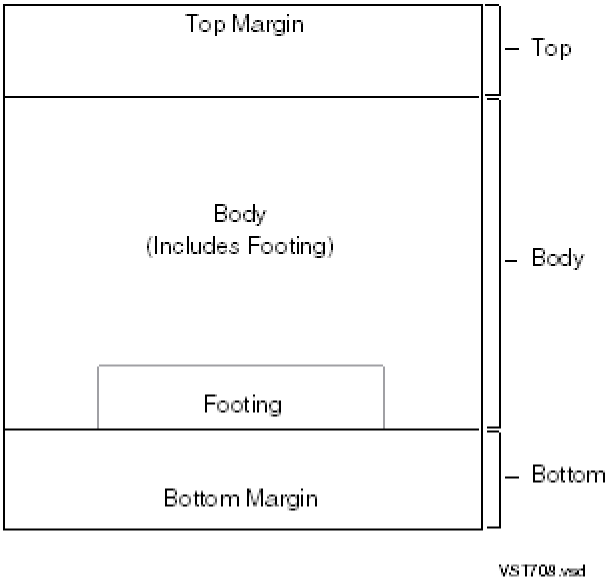
```
LINAGE IS 60 LINES WITH FOOTING AT 51,
   LINES AT TOP 0, LINES AT BOTTOM 3

LINAGE IS NUMBER-OF-TEXT-LINES, LINES AT TOP TEXT-OFFSET,
   LINES AT BOTTOM REST-OF-PAGE
```

Usage Considerations:

- The Logical Page

  The size of a logical page is the sum of the top and bottom margins and the body. The logical page is not necessarily the same size as the physical page.

Figure 8-1 LINAGE Clause Layout



- Restriction

  Because the purpose of the LINAGE clause is to control printing of data, it can be used only with files assigned to line printer devices or spooler processes.

- Initial Positioning of First Logical Page

  Before printing the first logical page, a standard printer file issues a page eject, positioning itself at the fourth line of the physical page. The run-time routines cannot determine if a printer file behaves in the standard way or not; therefore, the run-time routines handle all printer files as if they do; therefore, when the LINAGE clause applies, the OPEN statement operates:

  — If the value of `top` is 3 or more

  If the value of `top` is 3 or more, the run-time routines issue a page eject to establish a known initial position on the physical page. Because the run-time routines now expect the printer position to be at the fourth line (that is, 3 lines have already been skipped), they advance the printer by `top` - 3 lines.

  — If the value of `top` is 2 or less

  If the value of `top` is 2 or less, the preceding strategy does not work, because the run-time routines cannot backspace the printer. The run-time routines assume that the printer is positioned at the first line, omit the page eject, and advance the printer by `top` lines.

  Examples:

		Lines Skipped	
Value of top	Page-Eject Issued?	Additional	Total
0	No	0	0
1	No	1	1
2	No	2	2
3	Yes (skips 3)	0	3
4	Yes (skips 3)	1	4

		Lines Skipped	
Value of top	Page-Eject Issued?	Additional	Total
5	Yes (skips 3)	2	5
6	Yes (skips 3)	3	6

This logic applies only during initial positioning of the first logical page. The printer does not perform page ejects for subsequent pages, and the value of *top* is not modified.

- Handling the First Logical Page Specially

  If you know that the printer file does not behave in the standard way, you can program around the situation described in Initial Positioning of First Logical Page. In simple cases, follow these steps:

  1. In the LINAGE clause, specify a data item for *top*.
  2. Initialize the data item with a value that produces the desired result for the first logical page (see Initial Positioning of First Logical Page).
  3. Open the output file.
  4. Change the value of the data item to the value of the “real” top margin. (You can do this even before the first WRITE statement.)

  In more complex cases, follow these steps:

  1. In the LINAGE clause, specify a data item for *top*.
  2. Initialize the data item to 0.
  3. Open the output file.
  4. Use WRITE statements to position the first logical page properly.
  5. Change the value of the data item to the value of the “real” top margin.

- Effect of *body*, *foot*, *top*, and *bottom*

  The description of the first logical page is determined from *top*, *body*, *foot*, and *bottom* when the file is opened by an OPEN statement with the OUTPUT phrase.

  If literals are used to define all these page areas, all logical pages have the same layout.

  If one or more of the values are data-names, then each time a WRITE statement with the ADVANCING PAGE phrase is executed or a page overflow occurs, the current value of the data-names is used to set up the next logical page.

- Printing Device Does Not Space

  Each logical page continues to the next with no additional spacing from the printing device.

- The LINAGE-COUNTER Special Register

  Every file description that has a LINAGE clause generates a special register called LINAGE-COUNTER. At any given time, LINAGE-COUNTER contains the value of the current line number within the current page body.

  A LINAGE-COUNTER can be read but not modified.

  If more than one file description contains a LINAGE clause, each reference to a LINAGE-COUNTER must be qualified by its file name.

  The value of LINAGE-COUNTER is automatically set to 1 when its file is opened. During execution of a WRITE statement to its file, LINAGE-COUNTER is automatically modified under these conditions:

  — When the ADVANCING PAGE phrase of a WRITE statement is encountered, LINAGE-COUNTER is reset to 1.

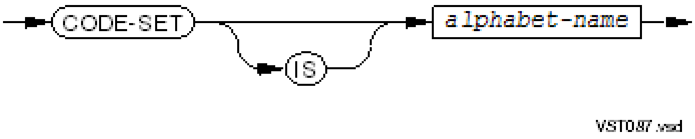
  — When the ADVANCING phrase is specified in a WRITE statement, LINAGE-COUNTER is increased by the number of lines given.

- When the ADVANCING phrase of the WRITE statement is not used, LINAGE-COUNTER is increased by 1.
- When execution of a WRITE statement repositions the file to the first line in the page body of the next logical page, LINAGE-COUNTER is reset to 1.

If the file description entry containing the LINAGE clause includes the EXTERNAL clause as well, the file then has the external attribute. In this case, the LINAGE-COUNTER special register is also an external data item. Execution of a WRITE statement for such a file by any programs that share access to the file modifies the same unique special register.

CODE-SET Clause

The CODE-SET clause specifies the character code convention used to represent data on the external media supporting the file.



alphabet-name

is the character set used to represent data on external media. It determines the way external codes are converted to native character codes during input and output operations. It must be defined in the SPECIAL-NAMES paragraph to be STANDARD-1, STANDARD-2, or NATIVE (all of which designate the ASCII character set), or a system-name predefined by HP COBOL as an alternate alphabet name. The only such system-name is EBCDIC.

Usage Considerations:

- Sequential Files Only

  You can specify the CODE-SET clause only for sequential files that do not have alternate keys. On files associated with the $RECEIVE device, the CODE-SET clause has no effect.

- Action of CODE-SET EBCDIC

  The presence of the CODE-SET clause causes translation between the native USASCII character code convention and the EBCDIC code convention for all input and output operations, regardless of the type of the device associated with the sequential file.

  If you specify CODE-SET EBCDIC in the file description associated with a printer, EBCDIC codes are delivered to the printer (which may or may not be prepared to accept EBCDIC codes).

  The most common use for CODE-SET EBCDIC is for reading or writing tapes for interchange with an EBCDIC-based system.

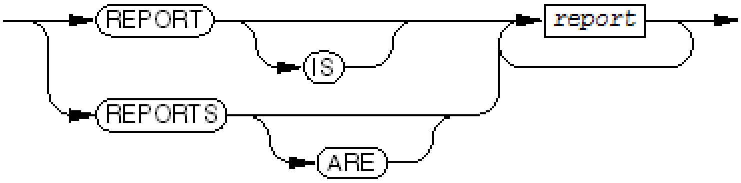
- Restrictions Established by the CODE-SET Clause

  When the CODE-SET clause appears, it places these restrictions on all data items defined in the record description entries associated with the file description entry:

  - Some SIGN SEPARATE clause must apply to every signed numeric data item
  - Every data item must be USAGE DISPLAY.

REPORT Clause

HP COBOL does not support the report-writing feature of ISO/ANSI COBOL. If the REPORT clause is present, the compiler checks it for syntactic validity and reports an error.



VST088.vsd

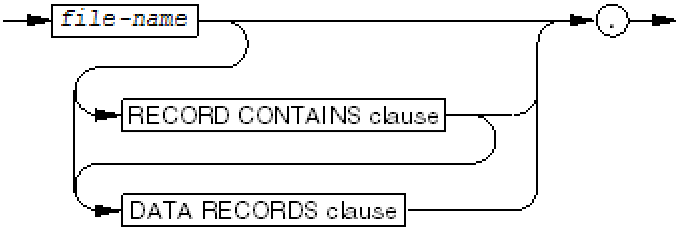
report

is irrelevant.

Sort-Merge File Description Entries

Each SORT or MERGE operation refers to a sort-merge file. The sort-merge file description (SD) describes the size, structure, and names of data records in a sort-merge file.

A sort-merge file description must begin with sort-merge file description followed by a file name. The optional clauses can follow in either order. A period ends the whole entry. One or more record descriptions must then follow. No input-output statements can be executed for this type of file. The `system-file-name` specified in the file-control entry associated with this file identifies the sort-merge scratch file or volume.

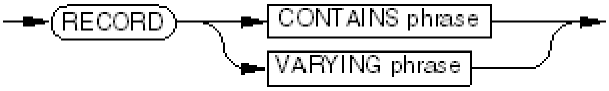


VST089.vsd

file-name

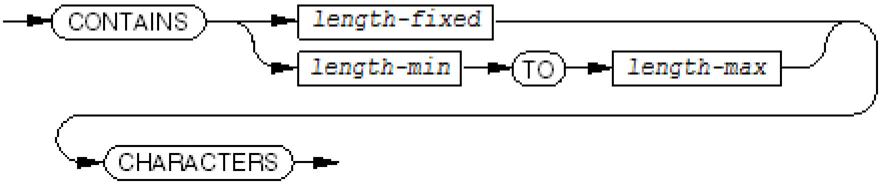
is the highest qualifier for both the sort-merge file description entry and its records. It must be unique within the program.

RECORD CONTAINS clause



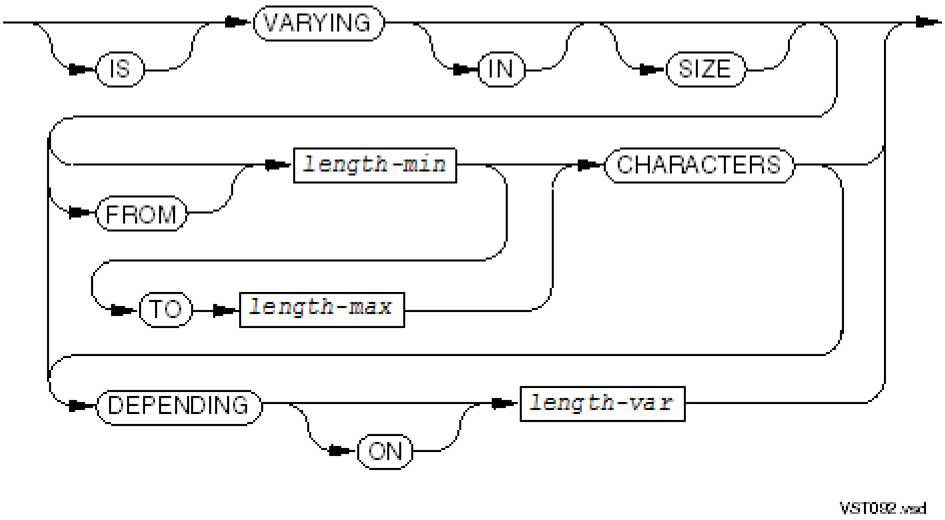
VST090.vsd

CONTAINS phrase



VST091.vsd

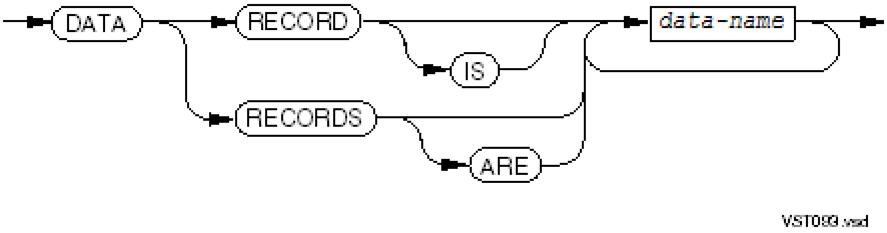
VARYING phrase



length-min, length-max, length-var

are integer values as in the file description entry.

DATA RECORDS clause



data-name

is the name of a record that is declared following the sort-merge file description entry. One or more fields of those records are used as keys in SORT and MERGE statements.

Usage Consideration: Only the DATA RECORD and RECORD CONTAINS (or RECORD VARYING) clauses are valid. For descriptions of these clauses, see File Description Entries.

In Example 8-5, the record description entry associated with the sort-merge file description entry defines CUSTOMER-NAME, CUSTOMER-ADDRESS, and CUSTOMER-ZIP. These data items can be used as sort-merge keys.

Example 8-5 Sort-Merge File Description Entry

```
SD SORT-THIS
   RECORD CONTAINS 80 CHARACTERS
   DATA RECORD IS SORT-TEMPLATE.
01 SORT-TEMPLATE.
   05  CUSTOMER-NAME       PIC X(35).
   05  FILLER              PIC X(35).
   05  CUSTOMER-ADDRESS    PIC X(55).
   05  CUSTOMER-ZIP        PIC 9(15).
       ...
```

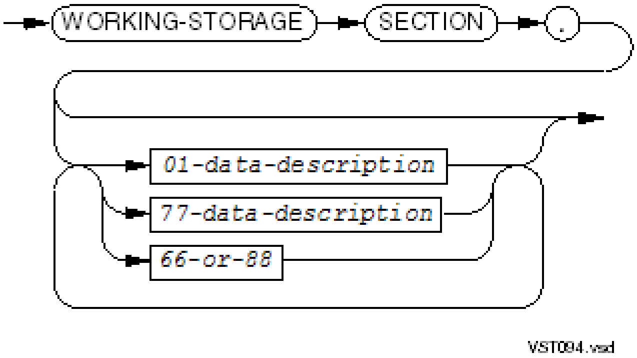
Data Description Entries

Data description entries in the File Section describe record areas associated with files. Each level-01 data description entry is a record description entry that describes the record area for the file named in the preceding file description entry or sort-merge file description entry. If multiple record description entries follow an file description or sort-merge file description entry, each record description entry after the first one is a redefinition of the record area. See Descriptions of Records (Levels 01-49).

You can use level-66 data description entries to rename contiguous items in a record and level-88 data description entries to assign condition-names to values of record items. See Descriptions That Rename Items (Level 66) and Descriptions of Condition-Names for Values (Level 88). You cannot put level-77 data description entries in the File Section.

Working-Storage Section

The Working-Storage Section defines records and miscellaneous data items for the process to use. You can set the initial values of most data items in working storage. When a process does not need local data items or explicit intermediate storage to execute the run unit, you can omit the Working-Storage Section.



01-data-description, 77-data-description

describe data items for the process to use (for details, see Data Description Entries). You can specify initial values for most of these data items (see Initializing Data Items).

There is no difference between level-77 items and elementary level-01 items, and the latter are preferred.

There are limits to the number of records and level-77 items that a program can contain. See Chapter 19: HP COBOL CRE Support (page 719).

66-or-88

is a level-66 or level-88 description. For syntax, see Descriptions That Rename Items (Level 66) and Descriptions of Condition-Names for Values (Level 88).

Data Description Entries

The data description entries for unrelated items (level-77 items) or records or both follow the Working-Storage Section header. Record-names, level-77 item names, and subordinate data-names are not required to be unique within the program except in these cases:

- A data-name that is referenced in the program and cannot be uniquely designated by including qualifying names in the reference cannot be uniquely designated by including qualifying names in the reference (the reference is diagnosed, not the data-name)
- A data-name for a language element whose syntax mandates uniqueness; for example, the level-01 names of external data items

If data items have hierarchical relationships to one another, you must group them into records according to the rules for record descriptions. All clauses available for record descriptions in the File Section are available for record descriptions in the Working-Storage Section. See Descriptions of Records (Levels 01-49).

If elementary data items bear no relationship to any other items, you do not need to group them into records. You can define each as a noncontiguous item with level number 77, followed by a data-name and a PICTURE or USAGE clause. Other data description clauses are optional for noncontiguous items; you can use other clauses to complete an item's description when necessary.

You do not have to put level-01 items (records) and level-77 items in any special order in the Working-Storage Section; however, if you are planning working storage for a program to be run as a process pair, see Checkpointing for a recommended approach that simplifies checkpointing.

You can also use level-66 data description entries to rename contiguous items in a record (see Descriptions That Rename Items (Level 66)) and level-88 data description entries to assign condition-names to values of items (see Descriptions of Condition-Names for Values (Level 88)).

In Example 8-6, the Working-Storage Section has five record description entries.

Example 8-6 Record Description Entries

```
WORKING-STORAGE SECTION.
01  DATA-TO-CHECKPOINT.
    05  USER-INPUTS.
        10  COMMAND-IN         PICTURE X(8)  VALUE SPACES.
        10  TRAN-CODE          PICTURE 999   VALUE ZERO.
        10  REPORT-TO-PRINT    PICTURE 99    VALUE ZERO.
        10  REPORT-NUMBER      PICTURE 99    VALUE ZERO.
    05  PERFORM-FLAGS          PICTURE 9.
        88  BAD                              VALUE 0.
        88  GOOD                             VALUE 1.

01  CURRENT-DATE.
    05  CURRENT-YEAR           PICTURE 99    VALUE ZERO.
    05  CURRENT-MONTH          PICTURE 99    VALUE ZERO.
    05  CURRENT-DAY            PICTURE 99    VALUE ZERO.

01  REPORT-HEADING-1.
    05  FILLER                 PICTURE X(7)  VALUE SPACES.
    05  REPORT-MM              PICTURE 99.
    05  FILLER                 PICTURE X     VALUE "/".
    05  REPORT-DD              PICTURE 99.
    05  FILLER                 PICTURE X     VALUE "/".
    05  REPORT-YY              PICTURE 99.
    05  REPORT-YY              PICTURE 99.
    05  FILLER                 PICTURE X(7)  VALUE SPACES.
    ...
01  MONTH-ABBREVIATIONS
       VALUE "JANFEBMARAPRMAYJUNJULAUGSEPOCTNOVDEC".
    03 MON-ABBR PIC X(3) OCCURS 12 TIMES.

01  CONSTANTS-FOR-I-O.
    05 ERROR-MSG-CONST  PICTURE X(13)  VALUE "*** ERROR ***".
    ...

01  CONSTANTS-FOR-LOGIC.
    ...
```

Initializing Data Items

The value to which a data item is initialized is determined by:

- EXTERNAL clause

  A data item described with an EXTERNAL clause (an external data item) cannot have a VALUE clause and is not affected by events that cause internal data items to be initialized.

Its contents are unpredictable until a program assigns a value to it. Thereafter, it retains the last value assigned to it throughout the current execution of the run unit.

- VALUE clause

  With some exceptions, you can set the initial value of an internal data item by including a VALUE clause in its data description entry (for exceptions, see VALUE Clause).

- BLANK and NOBLANK directives

  When a data item is not described with a VALUE clause, the BLANK or NOBLANK directive determines its initial value (see BLANK and NOBLANK (page 548)).

- BASED clause

  The implicit pointer associated with a BASED data item is initialized to NULL; that is, the BASED item is not associated with any storage.

An internal data item is initialized at each of these times:

- The first time the program is executed.
- The first time the program is called after a CANCEL statement that references it has been executed, unless a NOCANCEL directive affects the program.
- Every time the program is called, if the program is an initial program (see Initial Programs (page 61)). The NOCANCEL directive does not affect initial programs.

Once initialized, an internal data item retains the last value assigned to it throughout the current execution of the run unit, unless an event that causes internal data items to be reinitialized occurs. In that case, the data item is reassigned its initial value if it has a VALUE clause or if a BLANK directive is active; if not, its contents are unpredictable.

Size Limits

In Chapter 21: HP COBOL Limits (page 737), see Data in a run unit.

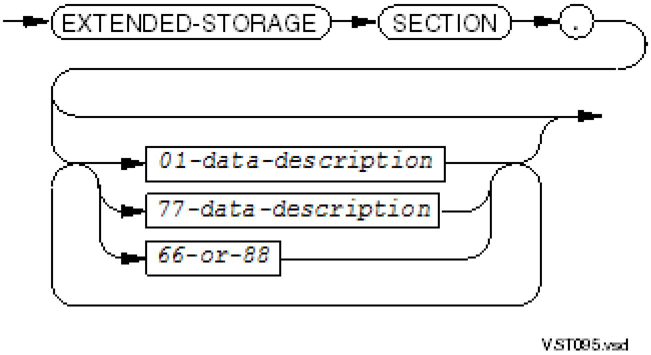
Checkpointing

You do not have to put level-01 items (records) and level-77 items in any special order in the Working-Storage Section; however, when you plan working storage for a program to be run as a process pair, this approach simplifies checkpointing:

- Group all independent data items into a record, and code the record in the same part of the Working-Storage Section as other level-01 items.
- Reserve level-77 items for constants, and code them last.

Extended-Storage Section

NOTE: TNS/E HP COBOL does not need an Extended-Storage Section. The ECOBOL compiler handles any data items that are described in the Extended-Storage Section as if they were described in the Working-Storage Section, without issuing a warning.



01-data-description, 77-data-description

describe data items for the process to use. You can specify initial values for most of these data items. See Data Description Entries and Initializing Data Items.

There are limits to the number of records and level-77 items that a program can contain. See Chapter 21: HP COBOL Limits (page 737).

66-or-88

is a level-66 or level-88 description. For syntax, see Descriptions That Rename Items (Level 66) and Descriptions of Condition-Names for Values (Level 88).

The Extended-Storage Section, if present, must appear after any Working-Storage Section. A program can have both sections, either section without the other, or neither section.

Data Description Entries and Initializing Data Items

Data description entries and data initialization rules are the same in the Extended-Storage Section as they are in the Working-Storage Section. See Data Description Entries and Initializing Data Items.

Addressing

Items described in the Extended-Storage Section are accessed using 32-bit addressing. Extended-storage items are allocated in a single extended data segment, which is managed entirely by the run-time routines. Instructions that involve 32-bit addresses consume more space in the object file and execute at a slightly slower speed than the instructions that involve 16-bit addresses.

Size Limits

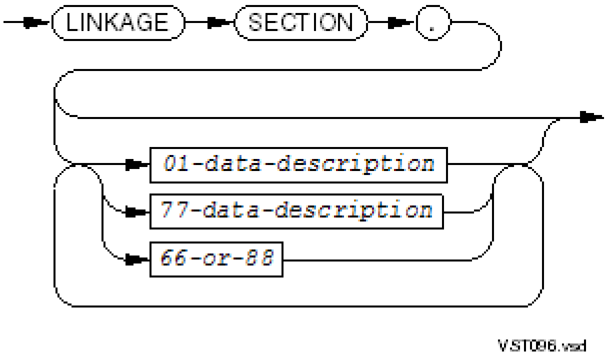
In Chapter 21: HP COBOL Limits, see Data in a run unit.

Checkpointing

If your program is compiled with the COBOL85 compiler and the ENV COMMON directive, the CHECKPOINT statement can process data items declared in the Extended-Storage Section.

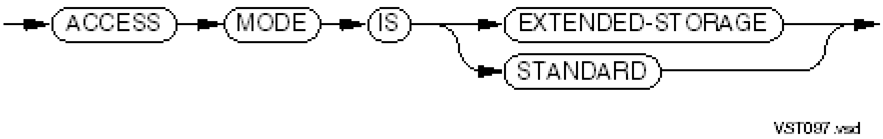
Linkage Section

The Linkage Section describes data passed from a calling program to the program containing the Linkage Section. No local data space is used for these items because they refer to existing items in the calling program.



01-data-description, 77-data-description

are described under Working-Storage Section, except that each data description entry can include this clause:



66-or-88

is a level-66 or level-88 description. For syntax, see Descriptions That Rename Items (Level 66) and Descriptions of Condition-Names for Values (Level 88).

Topics:

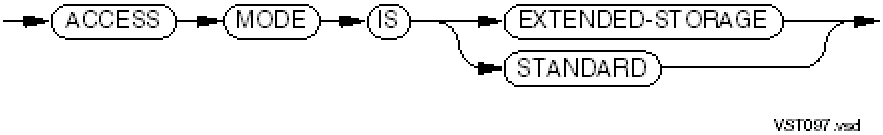
- Data Description Entries and Initializing Data ItemsData Description Entries and Initializing Data Items
- ACCESS MODE Clause
- CALL Statement and USING Phrase
- Index-Names
- Absent Linkage Section

Data Description Entries and Initializing Data Items

Data description entries and data initialization rules are the same in the Linkage Section as they are in the Working-Storage Section, except that you can use the VALUE clause only for level-88 items. A BASED item declared in the Linkage Section has its implicit pointer initialized to NULL every time the program is called. For information on using the VALUE clause in the Linkage Section, see Descriptions of Condition-Names for Values (Level 88). For information on data description entries and data initialization rules, see Data Description Entries, Initializing Data Items.

ACCESS MODE Clause

The ACCESS MODE clause describes the addressing method used to access parameters passed to the program—STANDARD or EXTENDED-STORAGE. You can specify the ACCESS MODE clause only in level-01 through level-49 and level-77 data description entries in the Linkage Section.



EXTENDED-STORAGE

specifies extended (32-bit) addressing. This is the default.

STANDARD

specifies standard (16-bit) addressing.

STANDARD cannot be specified for a program that is to be called with a CALL *identifier* statement. The compiler cannot check for this, so the result of such a call is undefined.

NOTE: If you specify STANDARD in a TNS/E program, the ECOBOL compiler issues a warning and uses 32-bit addressing.

Usage Considerations:

- STANDARD Access Mode and the CALL Statement

  If a formal parameter (in the called program) has STANDARD access mode, the CALL statement (in the calling program) cannot pass it an actual parameter that has EXTENDED-STORAGE access mode (the default) and was declared in the calling program's Extended-Storage Section or Linkage Section. Also, the called program's object must be available to the compiler when the calling program is compiled (see CALL (page 303)).

  The only advantage to STANDARD is slightly more efficient access in the called program.

- Extended Addressing and the CALL Statement

  If a formal parameter (in the called program) has EXTENDED-STORAGE access mode (the default), a CALL statement (in the calling program) can pass it any actual parameter defined in the calling program's Data Division.

- Access Mode Heritability

  Subordinate items inherit the access mode of their parent. If a record has no ACCESS MODE clause or has an ACCESS MODE EXTENDED-STORAGE clause, no element of that record can have STANDARD access mode. If a record has an ACCESS MODE STANDARD clause in its description, no element of that record can have EXTENDED-STORAGE access mode.

CALL Statement and USING Phrase

The Linkage Section is required when parameters are passed from a calling program to a called program, which is signaled by the presence of a USING phrase in the Procedure Division header of the called program.

The data items named in the CALL statement of the calling program correspond with the data items named in the USING phrase of the Procedure Division header of the called program. For details, see CALL (page 303).

Statements within the Procedure Division of the called program can refer to any items defined in the Linkage Section of the program. Global items can be referenced by contained programs. The Linkage Section can contain matched and unmatched items. Matched items are those within a record that is specified in the USING phrase of the Procedure Division header, within a record that is a redefinition of a matched record, a level-77 item or a record that is in the USING phrase, or a redefinition of those. An unmatched item is one not associated with the USING phrase.

The PORT directive determines the action taken for a reference to an unmatched item. If the program was not compiled with the PORT directive, the compiler issues a warning for any unmatched record or level-77 item. A reference to an unmatched item can cause abnormal program termination, data corruption, or other failures. If the program was compiled with the

PORT directive, the SET ADDRESS OF statement can make unmatched items accessible. A reference to an unmatched item for which no address was set can cause abnormal program termination, data corruption, or other failures.

A BASED item must be an unmatched item. No warning is issued for an unmatched BASED item, regardless of the PORT directive.

Example 8-7 Correspondence Between Formal and Actual Parameters

These lines are in the calling program:

```
WORKING-STORAGE SECTION.
01  PARAMETER-TABLE.
    02  ROW-PART       OCCURS 20 TIMES.
        03 COL-PART    PIC 9999 COMPUTATIONAL
                       OCCURS 10 TIMES.
     ...
77 ROW             PIC 99 COMPUTATIONAL.
77 COL             PIC 99 COMPUTATIONAL.
     ...
PROCEDURE DIVISION.
     ...
   CALL "SUBPROG1" USING ROW,
                         COL,
                         PARAMETER-TABLE.
     ...
```

These lines are in the called program:

```
LINKAGE SECTION.
01  PARM-3-IN-OUT.
    04 FORMAL-ROW                OCCURS 20 TIMES.
       07 FORMAL-COLUMN PIC 9999 COMPUTATIONAL
                                 OCCURS 10 TIMES.
77 PARM-R                 PIC 99 COMPUTATIONAL.
   88 ROW-WITHIN-RANGE           VALUE IS 1 THROUGH 20.
77 PARM-C                 PIC 99 COMPUTATIONAL.
   88 COLUMN-WITHIN-RANGE        VALUE IS 1 THROUGH 10.
     ...
PROCEDURE DIVISION USING PARM-R, PARM-C, PARM-3-IN-OUT.
     ...
```

Index-Names

A Procedure Division reference to a Linkage Section data item in the called program refers to a location in the calling program; however, this convention does not extend to index-names. The index-name of a table in the calling program and the index-name of a table in the called program always refer to separate indexes. This remains true even when the names of the indexes are the same in the calling program and the Linkage Section of the called program. An index-name's value can be passed if it is saved in a separate item and then passed. (See the example under CALL (page 303).)

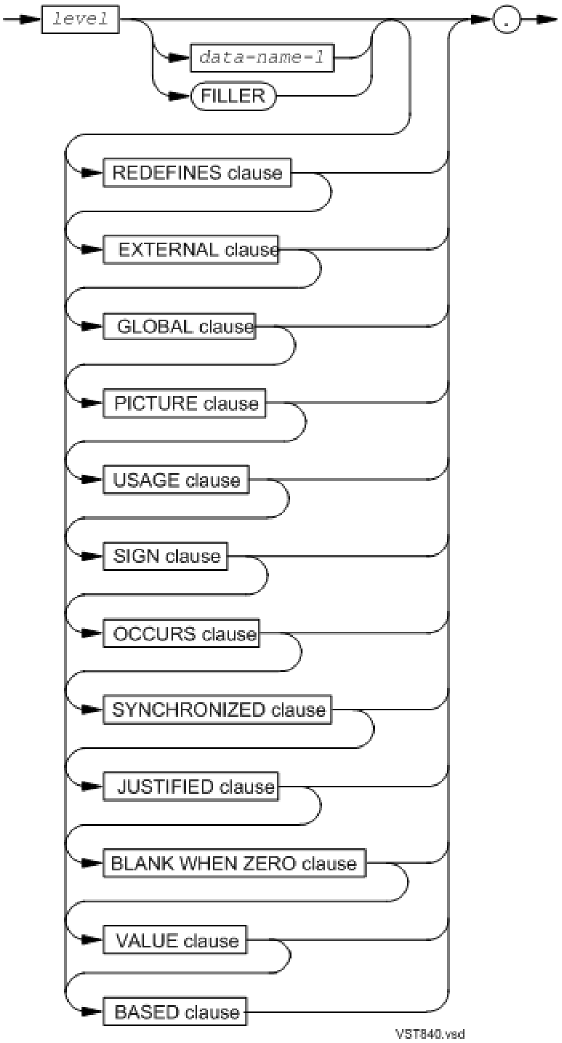
Absent Linkage Section

If there is no Linkage Section in the called program, it looks like a main program to the compiler. If you compile a main program and a subprogram with no parameters in the same compilation session, you must include either a MAIN directive or an empty Linkage Section to tell the compiler which program is the main one (see Main Programs (page 522)). If you do neither, the compiler reports that it found two main programs.

Because the compiler interprets subprograms that have no parameters as main programs, if you want to compile a collection of such subprograms, you need to direct the compiler not to attempt to give any of them the main attribute. To do this, use a MAIN directive specifying a program-name that does not correspond to the name of any program in the compilation.

Descriptions of Records (Levels 01-49)

A data description entry that starts at level-01 is a record description entry. It defines the characteristics of a record, and is followed by subordinate data description entries for items that are part of the record. You can put record description entries in any section in the Data Division.



`level`

is a 1-character or 2-character string whose value is in the range “1” (or “01”) to “49,” which represents the record’s level number.

`data-name-1`

is an alphanumeric data item whose value is the name of the record you are describing.

FILLER

is a place holder for a record to which the program never refers. This is the default.

General Considerations

These considerations apply to all record descriptions:

- Data Items Must be Defined

  You must define every data item the source program uses (except the special registers) with an appropriate data description entry in the Data Division. The forms specified in this topic include:

  - When the data item is a logical record associated with a file, you must define it in a level-01 data description entry associated with the file description entry for the file of which it is a record.
  - When the data item is a logical record that is not associated with a file, you must define it in a level-01 data description entry in the Linkage, Working-Storage, or Extended-Storage Section.
  - When the data item is an independent elementary data item, you must define it in a level-01 or level-77 data description entry in the Linkage, Working-Storage, or the Extended-Storage Section.
  - When the data item is a subordinate part of a record, you must define it in a data description entry (with a level number in the range 02 through 49) associated with the appropriate record description entry.
  - When the data item is a redefinition of another item, you must define it in a data description entry (with the same level number as the item it redefines) associated with the appropriate record description entry or level-77 data description entry.
  - The syntax and semantics of entries with level numbers 66 and 88 are explained in later topics of this section.

- Constraints on Clauses of the Data Description Entry

  Each data description entry must begin with the level number. If a data-name or the keyword FILLER is specified, it must immediately follow the level number. The characteristics of the data item determine which of the clauses are required or optional. Clauses that appear can do so in any order. No clause can appear more than once.

  - Data Name or FILLER Keyword

    If the data-name appears, it specifies the name of the data item being described.

    You can give an entry a name so that the name can be used to refer to the entry. If you do not need to refer to the entry, you can give it the name FILLER, or omit the name entirely. Such unnamed or FILLER items act as place holders in records where not all the fields are referred to or as convenient names for constants, such as in report headings.

    When the level number is 01, the data item is a record, and the data-name is sometimes referred to as a record-name. In this case, the data-name must be specified if the data description entry contains the EXTERNAL clause or if the data description entry is associated with a file description entry that contains the EXTERNAL clause. If neither the data-name nor the keyword FILLER is specified, it is as if the keyword FILLER had been specified.

    When the level number is 77, the data item is an independent (or noncontiguous) elementary data item, and the data-name (not FILLER) must always be specified.

  - REDEFINES Clause

    The REDEFINES clause, when it appears, must immediately follow the data-name or FILLER keyword. If it does not, the compiler issues a warning, because COBOL requires REDEFINES to precede all other clauses.

  - EXTERNAL Clause

    The EXTERNAL clause can be specified only for data description entries within the Working-Storage or Extended-Storage Sections whose level number is 01. It cannot be specified in a data description entry containing a REDEFINES clause.

  - PICTURE Clause

    The PICTURE clause must be specified for every elementary item except these, which must not have PICTURE clauses:

    - Indexes (described only in INDEXED phrases)
    - Data items described as USAGE INDEX, USAGE NATIVE-*n*, or USAGE POINTER
    - Data items declared by RENAMES clauses

  - USAGE Clause

    The USAGE clause cannot be used for national data items.

  - SIGN Clause

    The SIGN clause cannot be used for national data items.

  - SYNCHRONIZED Clause

    The SYNCHRONIZED clause cannot be used for national data items.

  - BLANK WHEN ZERO Clause

    The BLANK WHEN ZERO clause cannot be used for national data items.

  - BASED Clause

    The BASED clause can be used only with level-01 or level-77 data items in the Working-Storage, Extended-Storage, or Linkage Sections. It cannot be specified in a data description entry that includes the EXTERNAL clause or the REDEFINES clause.

- Clauses Applying Only to Elementary Data Items

  The PICTURE, SYNCHRONIZED, JUSTIFIED, and BLANK WHEN ZERO clauses can be specified only for an elementary data item.

- Allocation of Storage

  - Elementary Items

    The description of an internal elementary item in the Working-Storage Section or Extended-Storage Section causes allocation of storage space for that item unless that item redefines another item, in which case that item shares the redefined item’s storage space.

    The description of an external elementary item causes allocation of storage space for that item in space belonging to the run unit, outside the storage area of any individual program. This space is shared by all programs that describe the external data item.

  - Linkage Items

    The description of an elementary item in the Linkage Section does not cause its allocation. In this case, the item is allocated by some other program, or by the run unit, and is made available to the current program with the SET statement or the CALL statement that causes its execution.

  - BASED Items

    The description of a BASED item does not cause its allocation. In this case, the compiler allocates an implicit pointer associated with the item, and the item’s storage is allocated by the ALLOCATE statement, or the item is associated with other storage by the SET statement.

— Data Structures

The description of a data structure in the Working-Storage Section or Extended-Storage Section does not cause any direct allocation, because its data-name merely designates the combined storage spaces of all the elementary items subordinate to it; however, the storage alignment requirements of some elementary items can cause the generation of implicit FILLER items whose storage space also belongs to the containing data structures (for details, see SYNCHRONIZED Clause).

— Data Items in the File Section

Technically, the data description entries in the File Section only establish descriptions of data items in record areas—they are not declarations in the purest sense. If the file description does not establish a record length, the longest data description entry associated with it does so.

— Relationship with File Description Entries

In the File Section, data description entries that describe records follow file description entries or sort-merge file description entries.

The file description or sort-merge file description entry is not part of the data description entry (see Example 8-8).

The record description entry defines a record area associated with the file named in the file description entry.

NOTE: Any File Section record you describe cannot exceed 32,767 bytes. At execution time, records for $RECEIVE files cannot exceed 32 KB (see RECEIVE-CONTROL Paragraph (page 155)). For input and output operations on NonStop systems, the actual record-length limit for individual devices depends on the operating system limits.

Example 8-8 File Description Entry Followed by Record Description Entry

```
FD  SAMPLE-FILE
          LABEL RECORDS ARE OMITTED.
      01  SAMPLE-RECORD.
          05  STORE-ID.
              10  STORE-NUMBER       PIC 999.
              10  STORE-REGION       PIC X.
                  88  NORTHERN               VALUE "N".
                  88  SOUTHERN               VALUE "S".
          05  STORE-MANAGER          PIC X(35).
          05  STORE-ADDRESS.
              10  STREET             PIC X(25).
              10  CITY               PIC X(15).
              10  ZIP-CODE           PIC 9(5).
```

In the other sections of the Data Division, record description entries are data declarations, as in Example 8-9. The level-77 entry is a description of a data item that is not part of a record. See Descriptions of Noncontiguous Elementary Items (Level 77).

Example 8-9 Record Description Entries as Data Declarations

```
01   MEDICATIONS.
     03 BRAND-NAME      PICTURE X(50).
     03 SIZE-IN-MG      PICTURE 9999V99.
     03 CONTRA-COUNT    PICTURE 99.
     03 CONTRAINDICATIONS OCCURS 1 TO 25 TIMES
=                              DEPENDING ON CONTRA-COUNT
                              PICTURE X(50).
77   I-DEX              PICTURE 9999
                       USAGE IS COMPUTATIONAL
                        VALUE IS ZERO.
```

FILLER Keyword

The FILLER keyword, explicit or implicit, substitutes for a data-name when it is not important that an item have a name. It is allowed on elementary or data structures. Commonly, FILLER is used when you build records in the Working-Storage Section or Extended-Storage Section for heading lines or error messages, where most of the text is groups of literals.

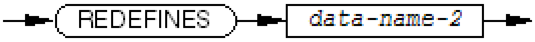
Example 8-10 FILLER Keyword

```
WORKING-STORAGE SECTION.
01   HEAD-1.
     05  FILLER          PIC X(10) VALUE "PART NUM".
     05  FILLER          PIC X(25) VALUE "DESCRIPTION".
     05  FILLER          PIC X(30) VALUE SPACES.
     05  FILLER          PIC X(15) VALUE "UNITS ON HAND"
01   BAD-NEWS.
     02 THE-VALUES.
        05               PIC X(45)
            VALUE "That part number is invalid.".
        05               PIC X(45)
            VALUE "That job code is invalid.".
        05               PIC X(45)
            VALUE "That delivery date is a holiday.".
        ...
     02 THE-MESSAGES REDEFINES THE-VALUES.
        05  ERR-MESSAGE     PIC X(45) OCCURS 15 TIMES.
```

Data items designated as FILLER can be conditional data items. This means that a FILLER item can be followed by one or more level-88 items describing condition-names that have the value TRUE when the FILLER item contains a certain value.

REDEFINES Clause

The REDEFINES clause enables you to describe the same computer storage area in more than one way. This can be quite valuable for tasks such as input data validation, when tests require different descriptions of the data. It is also convenient when some portions of a record are constant, while other parts vary.



VST102.vsd

data-name-2

is the name of the existing data item that is being redefined as *data-name-1*.

In Example 8-11, the two records RECORD-IN and RECORD-TOTAL share the same storage space, but their fields are different.

Example 8-11 REDEFINES Clause

```
WORKING-STORAGE SECTION.
   01  RECORD-IN.
       05  RECORD-CODE         PIC 9.
       05  RECORD-DETAIL       PIC X(30).
       05  RECORD-SUBTOTAL     PIC 9(3)V99.
   01  RECORD-TOTAL REDEFINES  RECORD-IN.
       05  TOTAL-1             PIC 9(5)V99.
       05  TOTAL-2             PIC 9(5)V99.
       05  TOTAL-3             PIC 9(5)V99.
       05  TOTAL-4             PIC 9(5)V99.
       05  TOTAL-5             PIC 9(6)V99.
```

Usage Considerations:

- Determining Size of Shared Storage Area

  When the redefined item is a record item and one or more redefinitions specify a different number of character positions than the redefined item (either greater or lesser), the size of the shared storage area is determined by the requirements of the largest associated record item.

- Elementary items or Data Structures Can Be Redefined

  You can describe an elementary data item and follow that description with a redefinition entry. You can also describe a data structure and follow that description first with descriptions of its component elementary items (with numerically higher level numbers), then with a redefinition entry that redefines the data structure.

  Redefinition continues until the appearance of a level number less than or equal to that of the data-name being redefined (or the ending of the current section of the Data Division).

- Series of Redefinitions

  You can describe a series of redefinition entries with the same level number as that of the redefined entry, provided that each of these entries also contains a REDEFINES clause specifying the data-name of the redefined item.

  You cannot, however, separate a redefinition entry from the redefined entry by any data description entry with a numerically lower level number.

- No Redefinitions of Level-66 or Level-88 Items

  Data items of level 66 (RENAMES) and 88 (condition-name) cannot be redefined.

- Redefined Item Cannot Include OCCURS or REDEFINES

  The description of *data-name-2* cannot include an OCCURS clause or REDEFINES clause; however *data-name-2* can be subordinate to an item containing one or both clauses, and items subordinate to *data-name-2* can contain one or both clauses. Because an OCCURS clause can occur in the description of *data-name-1*, it is common to declare a record first and then to declare an array that redefines the same storage.

  Neither the original definition nor the redefinition can include an item whose size is variable due to an OCCURS clause of a subordinate entry.

- No Subscripting or Qualifying in REDEFINES

  The REDEFINES declaration cannot include any subscripting or qualification on *data-name-2*. Qualification is automatic.

- VALUE Clauses Only in Condition-Name Descriptions

  VALUE clauses are not permitted in the redefinition except as part of a condition-name declared in conjunction with a conditional variable in the redefinition.

- Restrictions on Level Numbers

  The level number of a data description entry with a REDEFINES clause can be 01 in the Working-Storage Section, Extended-Storage Section, or Linkage Section but not in the File Section (where consecutive level-01 items are always multiple definitions of the same storage space). When the level number is other than 01, the redefinition must specify a number of character positions (bytes) that is less than or equal to the number of character positions in the data item being redefined.

- Redefinitions Must All Specify Same Redefined Item

  The REDEFINES clause redefines a storage area, not the data items occupying the area. Multiple redefinition of the same record area is permitted, but each instance of *`data-name-2`* must designate the name of the entry that originally defined the area.

- The EXTERNAL Clause and the REDEFINES Clause

  The redefinition entry cannot contain the EXTERNAL clause.

  When the redefined item is an external record item or is described with a level number other than 01, the redefinition can specify the same or fewer character positions (bytes) than the redefined item, but cannot specify more. When the redefined item is a record (that is, has a level number of 01) without the EXTERNAL clause, there is no such constraint.

- Complete Description of Restrictions on Entry Order

  The first redefinition of a data item must begin immediately after the last data description entry associated with that item. Additional redefinitions can appear immediately after the first one in any convenient order. Each redefinition begins with the data description entry that contains the REDEFINES clause and continues with any other entries needed to define the subordinate data items or condition-names that complete it. That is, the scope of the redefinition continues until the appearance of a level number less than or equal to that of the data-name being redefined, or to the end of the current section of the Data Division; therefore, the set of data description entries that describe a data item and its redefinitions must appear in a continuous sequence within either a single record description or, if the level number of the redefined item is 01 or 77, a single section of the Data Division.

- Multiple Redefinitions

  The REDEFINES clause associates the redefinition data item with the same storage area occupied by the redefined data item. Because more than one redefinition can make reference to the same redefined item, the same storage area can be described in as many ways as required by the logic of the containing source program.

  Although the REDEFINES clause associates multiple data description entries with one storage area, the data items described by these entries are independent in all other respects; therefore, the actual content of the shared storage area at any particular time does not necessarily represent a valid value for all of the associated data items.

- Alignment and the REDEFINES Clause

  The location of data items within a record can be affected by their storage alignment requirements (see SYNCHRONIZED Clause).

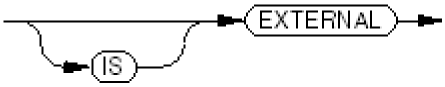
  When a redefinition item is an elementary item, its first character position must coincide with the first character position of the redefined item. The compiler issues a diagnostic message if the item’s alignment requirements make this impossible.

  When a redefinition item is a data structure, the first character position of its initial contained elementary item must coincide with the first character position of the redefined item. The compiler issues a diagnostic message if alignment requirements of the elementary item make this impossible.

- BASED Items and the REDEFINES Clause
  - A redefinition entry cannot contain a BASED clause.
  - A redefined item can be a BASED item.

EXTERNAL Clause

The EXTERNAL clause specifies that a record data item is external. This item and all of its subordinate data items are available to every program in the run unit that describes that record.



VST075.vsd

Usage Considerations:

- Working-Storage Section and Extended-Storage Section Records Only

  Only data description entries within the Working-Storage Section or Extended-Storage Section whose level number is 01 can include an EXTERNAL clause. In this case the data-name must be specified; it cannot be omitted or replaced with the keyword FILLER. Within the same source program, you cannot define a specific data-name in more than one level-01 entry described with an EXTERNAL clause.

  You cannot specify the EXTERNAL clause in a data description entry that contains the REDEFINES clause.

- Not With VALUE Clause

  You must not specify the VALUE clause in any data description entry that includes, or is subordinate to an entry that includes, the EXTERNAL clause. You can specify the VALUE clause for condition-name entries associated with the data description entries, however.

- Not With BASED Clause

  You must not specify the BASED clause in any data description entry that includes the EXTERNAL clause.

- External Record

  The EXTERNAL clause specifies that the data item defined by this data description entry is external. Because this data item is a record data item, it is also referred to as an external record. All data items defined by subordinate data description entries, including any redefinitions, are also external.

  The data contained in the record identified by the record-name is external and can be accessed and processed by any program in the run unit that describes and, optionally, redefines it.

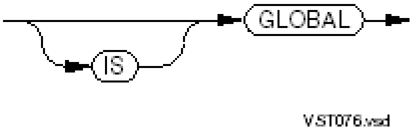
  Within a run unit, if two or more programs describe the same external data record, then the associated data description entries (including all subordinate data-names and data items and their redefinitions) must be identical; however, a program that describes an external record can contain a data description entry including the REDEFINES clause that redefines the complete external record, and this complete redefinition need not occur identically in other programs in the run unit. Also, all descriptions must be in the same section; that is, a description cannot be in the Extended-Storage Section in one program and in the Working-Storage Section of another program.

  When a file description entry contains the EXTERNAL clause, the data items defined in its associated record description entries inherit the external attribute; therefore, the preceding rules apply to these record description entries as well as those defined in the Working-Storage Section and Extended-Storage Section.

Example 3-2: Programs With Shared Data (page 64) contains an example of the use of the EXTERNAL clause.

GLOBAL Clause

The GLOBAL clause specifies that a data-name is a global name. A global name is available to every program contained within the program that describes it, even though the contained programs do not contain a description of it.



Usage Considerations:

- Global Name Can be Used in Contained Programs

  A statement in a program contained directly or indirectly within a program that describes a global name can reference that name without describing it again.

- Only for Level-01 Items

  The GLOBAL clause can be specified only in data description entries whose level number is 01 in the File Section, Working-Storage Section, or Extended-Storage Section.

- Name Uniqueness Within a Single Program

  In the Data Division of any one program, the data description entries for any two data items for which the same data-name is specified must not include the GLOBAL clause.

- Not With SAME RECORD AREA Clause

  If the SAME RECORD AREA clause is specified for several files, the record description entries or the file description entries for these files must not include the GLOBAL clause.

- Global Attribute is Inherited

  A data-name described using a GLOBAL clause is a global name. All data-names subordinate to a global name are global names. All condition-names associated with a global name are global names.

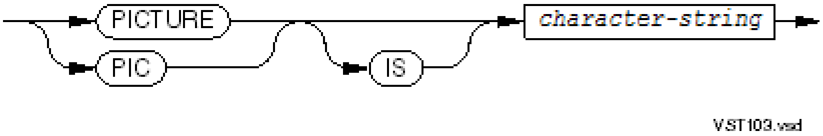
- Use With REDEFINES Clause

  If the GLOBAL clause is used in a data description entry that contains the REDEFINES clause, the global attribute applies only to the subject of the REDEFINES clause.

  Example 3-2: Programs With Shared Data (page 64) contains an example of the use of the GLOBAL clause.

PICTURE Clause

The PICTURE clause defines the characteristics and editing requirements of an elementary item.



character-string

is one or more symbols that determine the category of an elementary item, restrict the values that you can assign to the item, and define editing operations.

These data items do not have PICTURE clauses:

- An index (described only in an INDEXED phrase)
- A data item described as USAGE INDEX
- A data item described as USAGE NATIVE-*n* (where *n* is 2, 4, or 8)
- A data item described as USAGE POINTER
- A data item declared by a RENAMES clause

Usage Considerations:

- Size of *character-string*

  The maximum number of characters in *character-string* is 30. To signify more than one consecutive instance of a picture character, you can write it once, followed by an unsigned integer in parentheses. The integer tells how many times that character is to occur. For example, PICTURE 99999 is equivalent to PICTURE 9(5). While a *character-string* can have at most 30 characters, using the repetition technique lets you define items longer than 30 characters. You cannot use this technique for characters that are limited to one occurrence in a PICTURE *character-string*.

- Composition of *character-string*

  The meaning of each character used in a PICTURE *character-string* is peculiar to the PICTURE clause and is independent of any meaning assumed in other contexts.

  The *character-string* in a PICTURE clause consists of certain allowable combinations of symbols formed from the COBOL character set. The particular combination of symbols in the *character-string* determines the size, category, sign, and editing attributes of the elementary item that it describes. The symbols also specify any restrictions on the values that the program can assign to the item and any editing that COBOL is to perform in conjunction with an assignment.

  Most individual symbols serve two purposes:

  — Represent character positions in the item value

  — Describe the characteristics of these positions

    Some symbols merely contribute attribute information without defining character positions. Within the character-string, the presence of a symbol followed by a repetition factor has the same meaning as the presence of the symbol repeated the specified number of times.

    The symbols are shown in uppercase, but both uppercase and lowercase are effective.

    Table 8-2 lists and explains the symbols used to describe an elementary item.

Table 8-2 PICTURE Character-String Symbols

Symbol	Symbol Counts in Item's Size	Description
A	Yes	Represents a character position for a letter or the space character.
B	Yes	Represents a character position for a space character.
N	No	Indicates that the picture string is a national data item.

Table 8-2 PICTURE Character-String Symbols *(continued)*

Symbol	Symbol Counts in Item's Size	Description
P	No, but determines the maximum number of digit positions in numeric and numeric edited items.	Indicates scaling when the decimal point is not within the number that appears in the data item. One or more `P`s can appear only as a string of contiguous characters to the left or right of all other digit positions in the *`character-string`*. Because `P` implies an assumed decimal point, `V` is redundant. `P` and the insertion character period (.) cannot be in the same *`character-string`*. In some operations that manipulate a data item whose *`character-string`* contains `P`, the algebraic value of the data item is used rather than the actual character representation of the data item. In this algebraic value, the decimal point is in the prescribed location and zero is in each digit position specified by `P`. The size of the value is the number of digit positions represented by the *`character-string`*. These operations are any of: • Any operation requiring a numeric sending operand • A MOVE statement where the sending operand is a numeric or numeric-edited data item, its *`character-string`* contains `P`, and the receiving operand is numeric or numeric-edited • A comparison operand where both operands are numeric In all other operations, the digit positions specified with `P` are ignored.
S	Only if you use a SIGN clause with a SEPARATE phrase	Indicates that the picture string represents a signed numeric value. Only one `S` can appear in a *`character-string`*. If present, it must be the leftmost character. If no `S` is present, the item is unsigned. A negative value becomes positive if it is stored in an item that does not have an `S`. The location (leading or trailing) and representation (embedded or separate) attributes are specified in other clauses of the data description entry, not in the PICTURE clause (see SIGN Clause).
V	No	Represents the assumed decimal point location in noninteger numeric items. Only one `V` can appear in a *`character-string`*, and `V` cannot occur in the same *`character-string`* with an explicit decimal point.
X	Yes	Represents a character position that can contain any character from the ASCII character set.
Z	Yes	Represents a leading numeric character position whose contents are to be replaced by a space when it and all preceding numeric character positions are 0. All `Z`s must precede any `9` s within the *`character-string`*.
9	Yes	Represents a character position for a digit.
0	Yes	Represents a character position where the character zero (`0`) is to be inserted.
,	Yes	Represents a character position where a comma (,) is to be inserted. A comma must not be the last (rightmost) character in the *`character-string`*. See note below.

Table 8-2 PICTURE Character-String Symbols *(continued)*

Symbol	Symbol Counts in Item's Size	Description
.	Yes	Represents the decimal point for alignment purposes. A period (.) is to be inserted at that position. The period must not be the last (rightmost) character in the `character-string` and can only be used once. No `V` can occur in a PICTURE `character-string` containing an explicit decimal point.
/	Yes	Represents a character position where a slash (/) is to be inserted.
+ - CR DB	Yes	Sign-editing symbols. A `character-string` cannot have more than one sign-editing symbol. If it has one sign-editing symbol, the sign-editing symbol represents the position where the sign-control symbol goes.
*	Yes	Represents a leading numeric character position that is to be replaced by an asterisk (*) if its contents and the contents of all preceding numeric character positions are 0. All asterisks must precede any `9` s.
$	Yes	Represents the character position where the currency symbol is to be placed. The dollar sign ($) is used unless a CURRENCY SIGN clause specifies another single character.

NOTE: If you use the DECIMAL-POINT COMMA clause in the SPECIAL-NAMES paragraph in the Environment Division, periods function as commas, and commas function as periods.

Neither the period nor the comma can be the last (rightmost) symbol of a PICTURE `character-string`; with this exception: when the comma, semicolon, or period separator follows the `character-string`. In this case a preceding comma or period character is acceptable as the last symbol in the `character-string`; for example:

999. is “999” followed by a period separator

999., is “999.” followed by a comma separator

- Precedence Rules

  Figure 8-2 shows the order of precedence for the symbols in a PICTURE character-string. An `x` at an intersection indicates that the symbol at the top of the column can precede the symbol at the left of the row. Where two symbols appear in a column or row, they are mutually exclusive within the same character-string.

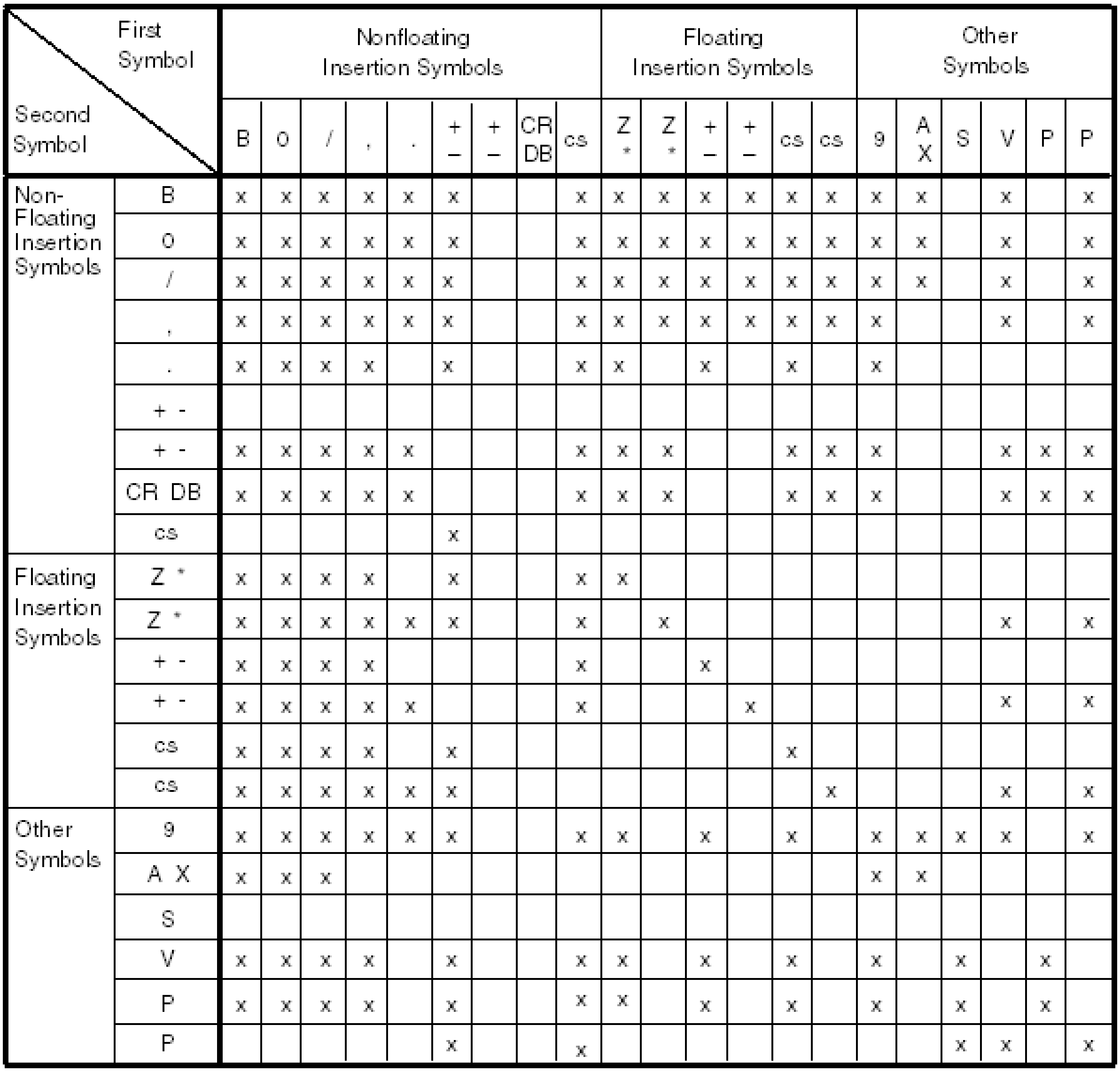
  The comma (,) and period (.) insertion symbols can be reversed by the DECIMAL-POINT COMMA clause (see DECIMAL-POINT Clause (page 126)).

  The symbol `cs` represents the currency symbol.

  The nonfloating insertion symbols plus (+) and minus (-), the floating insertion symbols `Z`, asterisk (*), plus (+), minus (-), and `cs`, and the symbol `P` appear twice. The leftmost column and uppermost row for each symbol represents its use to the left of the decimal point position. The other appearance of the symbol represents its use to the right of the decimal point position.

  Each PICTURE character-string must contain at least one of the symbols `A`, `N`, `X`, `Z`, `9`, or asterisk (*) or at least two of the symbols plus (+), minus (-), or `cs`.

Figure 8-2 Precedence Rules for PICTURE Symbols



Second Symbol	First Symbol	Nonfloating Insertion Symbols									Floating Insertion Symbols						Other Symbols					
		B	0	/	,	.	+ –	+ –	CR DB	cs	Z *	Z *	+ –	+ –	cs	cs	9	A X	S	V	P	P
Non-Floating Insertion Symbols	B	x	x	x	x	x	x			x	x	x	x	x	x	x	x	x		x		x
	0	x	x	x	x	x	x			x	x	x	x	x	x	x	x	x		x		x
	/	x	x	x	x	x	x			x	x	x	x	x	x	x	x	x		x		x
	,	x	x	x	x	x	x			x	x	x	x	x	x	x	x			x		x
	.	x	x	x	x		x			x	x		x		x		x					
	+ -																					
	+ -	x	x	x	x	x				x	x	x			x	x	x			x	x	x
	CR DB	x	x	x	x	x				x	x	x			x	x	x			x	x	x
	cs						x															
Floating Insertion Symbols	Z *	x	x	x	x		x			x	x											
	Z *	x	x	x	x	x	x			x		x								x		x
	+ -	x	x	x	x					x			x									
	+ -	x	x	x	x	x				x				x						x		x
	cs	x	x	x	x		x								x							
	cs	x	x	x	x	x	x									x				x		x
Other Symbols	9	x	x	x	x	x	x			x	x		x		x		x	x	x	x		x
	A X	x	x	x													x	x				
	S																					
	V	x	x	x	x		x			x	x		x		x		x		x		x	
	P	x	x	x	x		x			x	x		x		x		x		x		x	
	P						x			x									x	x		x

cs = currency symbol

VST525.vsd

- Data Item Size

  A data item's size is determined by the symbols of its PICTURE character-string. Each `A`, `B`, `N`, `X`, `Z`, `9`, zero (0), slash (/), comma (,), period (.), plus (+), minus (-), asterisk (*), or currency symbol (usually the dollar sign ($)) counts as one character position. CR or DB counts as two character positions. `S` is one character only if the item is subject to a SIGN clause with a SEPARATE phrase.

  `S`, `V`, CR, and DB cannot appear more than once in a single character-string.

  A PICTURE character-string (or part of a PICTURE character-string) of the format

  `symbol` (*`integer`*)

  where *`symbol`* is any symbol allowed in a PICTURE character-string except `P`, is equivalent to a PICTURE character-string (or substring) of *`integer`* consecutive occurrences of *`symbol`*. For example, PICTURE X(4) is equivalent to PICTURE XXXX and PICTURE A(3)0A is equivalent to PICTURE AAA0A.

The size of a DISPLAY item is determined by the PICTURE character-string symbols. COMPUTATIONAL and BINARY items deviate from the rules for DISPLAY items. See USAGE Clause. The amount of storage given to a data item can exceed its size if the SYNCHRONIZED clause is used; however, any excess bytes so allocated (called “implicit FILLER” bytes) belong not to the synchronized item but to its parent item.

- Trapping Size Errors

  If a data item is described with a PICTURE clause, you must use the SIZE ERROR Phrase (page 254) to trap size errors that occur because the result of a calculation exceeds the maximum number allowed by the PICTURE phrase. The SIZE ERROR phrase is especially important for COMPUTATIONAL and BINARY items, because without the SIZE ERROR phrase, you might not discover until much later that a value larger than that allowed by the PICTURE was stored as the result of a computation.

- Categories of Data Items

  The PICTURE clause can describe these categories of data items:

  — Alphabetic

  — Numeric

  — Alphanumeric

  — Alphanumeric edited

  — Numeric edited

  — National

  The results of most statements in the Procedure Division depend on the categories of the data items. Some statements disallow certain categories for some or all of their operands. In other cases, the same statement can take distinctly different actions when applied to data items of different categories.

  In the remainder of this topic, *9* s and A s within the PICTURE character-string are described as representing character positions that contain only numbers or letters and spaces. For greater efficiency, the HP COBOL compilers do not always enforce this restriction. Characters other than those permitted can be moved into these positions if they appear in the corresponding positions of a sending data item.

  Because the COBOL language considers every data structure to be in the alphanumeric category, manipulations upon data structures ignore all PICTURE constraints of their constituent elementary items, including editing specifications. As an extreme (but quite legal) example, an assignment to a containing data structure can cause any character position of an elementary item to assume any character. If numeric items contain nonnumeric characters, the results of using them in numeric operations are undefined.

- Alphabetic Data Items

  An item is in the alphabetic category when its PICTURE character-string contains only A s. The contents of this type of item are represented externally as some combination of the 26 (uppercase or lowercase) letters of the alphabet and space character.

  Example 8-12 Alphabetic Data Items

```
05  PACKAGE-CODE     PIC AAA.
15  DEPT-ID          PIC A(12).
```

- Numeric Data Items

  An item is in the numeric category when its PICTURE character-string contains only the symbols from the set: *9*, *P*, *S*, and *V*. The number of digits described must be greater than 0 and not more than 18. The contents are represented externally as a combination of 0, 1, 2, 3, 4, 5, 6, 7, 8, and 9. If the *S* is present, the sign of the value is retained.

Example 8-13 Numeric Data Items

```
05  DIVISION-TOTAL      PIC S9(10)V99.
05  FRACTION-AMOUNT     PIC VPP99.
```

The PICTURE character-string must include the symbol *S* if the item is described with a SIGN clause. If the item inherits a SIGN clause from a data structure, the PICTURE character-string cannot include the symbol *S* unless the item category is DISPLAY.

- Alphanumeric Data Items

  An item is in the alphanumeric category when its PICTURE character-string contains both *9* s and *A* s, or only *X* s, or a combination of *9* s, *A* s, and *X* s. A PICTURE character-string of all *A* s or all *9* s is not an alphanumeric item. The item is handled as if the string contained only *X* s. The contents of the item can be any combination of characters.

Example 8-14 Alphanumeric Data Items

```
10  STOCK-ITEM-NAME      PIC X(25).
15  ZONE-ID              PIC A(4)99.
```

- Alphanumeric Edited Data Items

  An item is in the alphanumeric edited category when its PICTURE *character-string* consists of at least one *X* or *A* and at least one *B*, zero (*0*), or slash (/). The contents of the data item can be any combination of characters.

Example 8-15 Alphanumeric Edited Data Items

```
10  PART-NAME          PIC X(5)BX(5).
15  BRANCH-CODE        PIC XX0X.
05  REPORT-VERSION     PIC XX/X.
20  SITE-ID            PIC A(3)0A.
20  SYSTEM-TAG         PIC AA/A.
05  SEAT-NUMBER        PIC AB9.
```

- Numeric Edited Data Items

  An item is in the numeric edited category when its PICTURE character-string contains only combinations of the symbols *B*, slash (/), *P*, *V*, *Z*, zero (*0*), *9*, comma (,), period (.), asterisk (*), plus (+), minus (-), CR, DB, and the currency symbol (usually the dollar sign ($)). The number of digit positions must be in the range from 1 through 18. The PICTURE character-string must have at least one symbol from the listed set other than *9*, *P*, or *V*.

Example 8-16 Numeric Edited Data Items

```
12  R-TOTAL-1        PIC ZZZ,ZZZ.99.
10  ITEM-PRICE       PIC $999.
35  UNIT-PRICE       PIC $$$9.
05  AMOUNT-OWED      PIC 999CR.
10  AMOUNT-LEFT      PIC ***99.
05  BACK-ORDERS      PIC -99.
77  START-DATE       PIC 99/99/99.
12  S-BLIVIT         PIC +$99.99.
03  STARRED-X        PIC ***.**.
77  SUM-X            PIC --B---.---.
```

- National Data Items

  An item is in the national category when its PICTURE character-string begins with *N* or *n* and contains no editing symbols. A national data item is used for languages that are not represented by roman letters and numbers, such as the Japanese Kanji alphabet. Special terminals and keyboards are required to use national data items.

In general, you can use a national data item anywhere you can use an alphanumeric data item. Exceptions are:

- In an ACCEPT statement with DATE, DAY, DAY-OF-WEEK, or TIME
- In a FILE STATUS clause of the SELECT statement
- In an INITIALIZE statement with a REPLACING phrase
- In an INSPECT statement
- In the PADDING clause of the SELECT statement
- In a RECEIVE-CONTROL paragraph
- In the SPECIAL-NAMES paragraph
- As the identifier in a CALL statement
- As the identifier in a CANCEL statement
- In comparison with a nonnational data item or nonnational literal

If national data items and national literals are used for items in a STRING statement (*part-1*, *delimiter*, or *result*) or an UNSTRING statement (*delim-1*, *delim-2*, *result*, or *delimstore*), all the items must be national data items or national literals.

If you use a VALUE clause with a national item, you must precede the value with *N* or *n* with no space between *N* or *n* and the value. For example:

01 *kanji-field* PIC N(4) VALUE IS N"*kanji-value*"

where *kanji-value* is a Kanji literal.

- Editing Characters

Editing is done by inserting, suppressing, or replacing a character. Editing occurs when a value is moved into a data item whose PICTURE character-string contains editing characters. The primary purpose of editing is to easily transform data into reportable form.

The methods of editing are:

- Simple insertion
- Special insertion
- Fixed insertion
- Floating insertion
- Zero suppression

The type of editing that you can perform on an item depends on the item’s category.

Table 8-3 Types of Editing Performed

Data Item Category	Type of Editing
Alphabetic	None
Numeric	None
Alphanumeric	None
Alphanumeric Edited	Simple insertion [*0*, *B*, or slash (/)]
Numeric Edited	All, subject to note following table
National	None

NOTE: Floating insertion editing and editing by zero suppression and replacement are mutually exclusive in a single PICTURE clause, and only one of the two replacement types can be used with zero suppression in a single PICTURE clause.

— Simple insertion

The comma (,), space (B), zero (`0`), and slash (/) are used as the insertion characters. They are counted in the item's size and represent where that character is inserted. The result of simple insertion editing is the appearance of the insertion character within the edited item value in the same position as it appears in the character-string. Here are some examples:

Source Item	PICTURE	Edited Result
123456	PIC 999,999	123,456
123456	PIC 99BBB9999	12 3456
1234	PIC 990099	120034
1234	PIC 99/99	12/34
123456	PIC 99B99B99	12 34 56
1234	PIC 999900	123400
13184	PIC 99/99/99	01/31/84
12345	PIC 999,999	012,345
"ABCD"	PIC ABABABA	A B C D
"1st2nd"	PIC XXXB/BXXX	1st / 2nd

— Special insertion

A period (.) is used as the insertion character and also acts as the decimal point for alignment purposes. The insertion character used for the actual decimal point is counted in the item's size.

An assumed decimal point character (`V`) and an actual decimal point character (.) cannot both occur in the same PICTURE character-string. The result of special insertion editing is the appearance of the insertion character within the edited item value in the same position as it appears in the character-string.

Here are some examples. The caret (^) in each source item is the decimal point location from its PICTURE character-string.

Source Item	PICTURE	Edited Result
1234^56	PIC 9999.99	1234.56
1^23456	PIC 9.99999	1.23456
1^23456	PIC 99.999999	01.234560
123^4	PIC 99.9999	23.4000
12^345	PIC 99.99	12.34

— Fixed insertion

The currency symbol and the editing sign control symbols (+, -, CR, DB) are the insertion characters. Only one currency symbol and one of the editing control symbols can be used in a PICTURE character-string. CR and DB represent two positions when counting an item's size. When used, they must be in the rightmost positions. When a plus (+) or minus (-) is used, it must be in either the leftmost or rightmost character position to be counted in the item's size. When the currency symbol is used, it must be the leftmost character, except when preceded by either a plus (+) or minus (-). Only one currency symbol and only one of the editing sign control characters can be used in the same PICTURE character-string. The result of fixed insertion editing is the appearance of the insertion characters within the item value in the same positions as they appear in the PICTURE character-string.

Table 8-4 Sign Control Symbols

	Result	
Editing Symbol	**Positive or Zero Data Item**	**Negative Data Item**
+	+	-
-	space	-
CR	2 spaces	CR
DB	2 spaces	DB

Here are some examples of fixed insertion editing. The caret (^) in each source item is the decimal point location from its PICTURE character-string.

Source Item	PICTURE	Edited Result
12^34	PIC 99.99+	12.34+
-12^34	PIC 99.99+	12.34-
12^34	PIC +$99.99	+$12.34
-12^34	PIC +$99.99	-$12.34
12^34	PIC $99.99-	$12.34
-12^34	PIC $99.99-	$12.34-
12^34	PIC -$99.99	$12.34
-12^34	PIC -$99.99	-$12.34
12^34	PIC 99.99CR	12.34
-12^34	PIC 99.99CR	12.34CR
12^34	PIC $99.99DB	$12.34
-12^34	PIC $99.99DB	$12.34DB

— Floating insertion

The currency sign and editing sign control symbols plus (+) and minus (-) are floating insertion characters, mutually exclusive in a PICTURE character-string. At least two of any one symbol must be used. Simple insertion characters can be mixed with floating characters.

To indicate floating insertion editing in a PICTURE character-string, you include a string of at least two of the floating insertion characters. This string can also contain any of the simple insertion symbols or have simple insertion characters immediately to its right. Any such simple insertion characters are part of the floating string.

The leftmost character of the floating insertion string marks the leftmost limit of the floating symbol. The rightmost character of the floating string marks the rightmost limit of the floating symbols. The second floating character from the left marks the leftmost limit of numeric data that can be stored. Nonzero numeric data can replace all characters at or to the right of this limit.

In floating insertion editing, you can either represent any or all of the leading numeric positions on the left of the decimal point by the insertion symbol.R or represent all the numeric positions, right or left of the decimal point, by the insertion symbol.

If the character-string contains a decimal point symbol, at least one floating insertion character must appear to the left of it.

If the insertion characters are only to the left of the decimal point, then assignment to the item places a single floating insertion character in the position immediately preceding either the decimal point or the first nonzero digit in the data represented by the insertion symbol string, whichever is farther to the left in the PICTURE character-string. The character positions preceding the insertion character are replaced with spaces.

If all numeric character positions in the PICTURE character-string are represented by the insertion character, the result depends on the value of the assigned data. If the value is 0, the entire data item is set to spaces. If the value is not 0, the result is the same as when the insertion character is only to the left of the decimal point.

To avoid truncation of the edited item, the minimum size of the PICTURE character-string for the receiving data item must be the number of characters in the sending item, plus the number of nonfloating characters being edited into the receiving data item, plus one for the floating insertion character.

Examples of floating insertion editing:

Source Item	PICTURE	Edited Result
123	$$999	$123
1234	$$$$$	$1234
2	$$999	$002
0	$$$$$	(all spaces)
0	$$$99	$00
1234	+++99	+1234
-23	+++99	-23
4	+++99	+04
123	------9	123
-123	------9	-123
1.23	$$$9.99	$1.23
-1.23	$$$9.99	$1.23
0.03	$$$9.99	$0.03
-0.03	$$$9.99	$0.03
1.23	$$$$.$$	$1.23
-1.23	$$$$.$$	$1.23
0.03	$$$$.$$	$.03
-0.03	$$$$.$$	$.03
1.23	---9.99	1.23
-1.23	---9.99	-1.23
0.03	---9.99	0.03

Source Item	PICTURE	Edited Result
-0.03	---9.99	-0.03
1.23	----.--	1.23
-1.23	----.--	-1.23
0.03	----.--	.03
-0.03	----.--	-.03

— Zero suppression

Suppression of leading zeros in numeric character positions is done with a `Z` or asterisk (*) in the PICTURE character-string. These symbols are mutually exclusive in a single PICTURE character-string. When you use the asterisk, you cannot use BLANK WHEN ZERO in the same entry. Each suppression symbol is counted in the item's size. Spaces replace `Z`s; asterisks remain asterisks.

You specify zero suppression and replacement by using a string of one or more `Z`s or asterisks to represent leading numeric character positions that are to be replaced when any of those positions in the data are zero. Any of the simple insertion characters embedded in the string of symbols or to the immediate right of this string are part of the string.

For zero suppression in a PICTURE character-string, you can either represent any or all of the leading numeric character positions to the left of the decimal point by `Z`s or asterisks or you can represent all of the numeric positions, right or left of the decimal point, as `Z`s or asterisks.

If the suppression symbols appear only to the left of the decimal point, any leading zero in the data that corresponds to the symbol is replaced by the replacement character. Suppression terminates at the first nonzero digit in the data represented by the suppression symbol string or at the decimal point, whichever comes first.

If all numeric character positions in the PICTURE character-string are represented by suppression symbols and the value of the data is not 0, the result is the same as if the suppression characters were only to the left of the decimal point. If the value is 0 and the suppression symbol is `Z`, the entire data item is set to spaces. If the value is 0 and the suppression symbol is the asterisk, the data item is set to all asterisks except for the actual decimal point. In this case, the actual decimal point appears in the data item.

Only one of the symbols plus (+), minus (-), asterisk (*), `Z`, and the currency symbol can be used as a floating replacement character within a single PICTURE character-string.

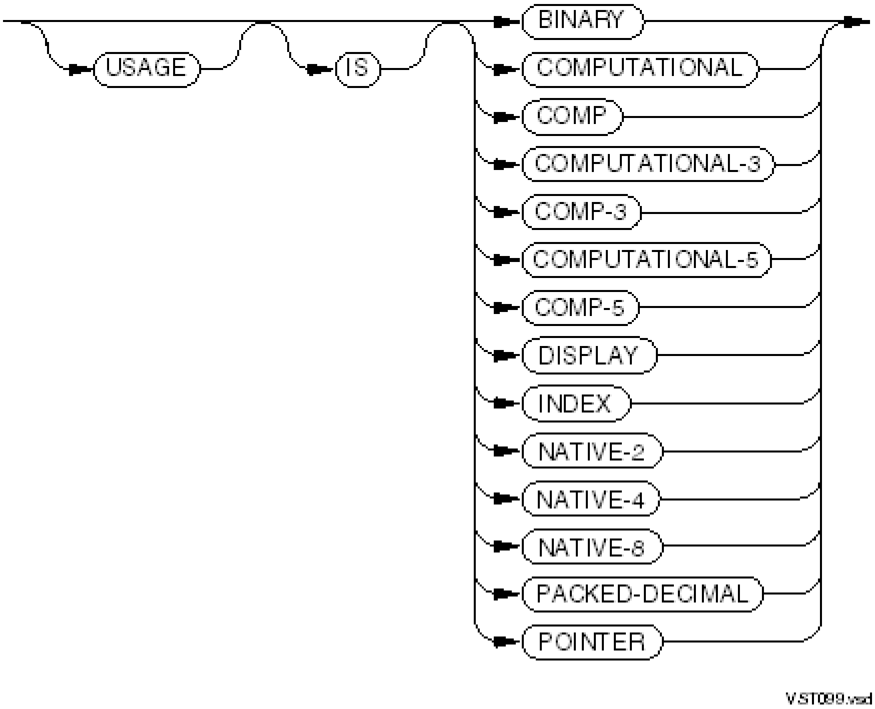
NOTE: The caret (^) in the source item is the decimal point from its PICTURE character-string.

Here are some examples of suppression editing:

Source Item	PICTURE	Edited Result
123456	$ZZZ,ZZZ.99	$123,456.00
1234^56	$ZZZ,ZZZ.99	$ 1,234.56
0	Z,ZZZ.ZZ	8 spaces
12^34	$***,***.99	$*****12.34
0	$***,***.99	$*******.00
0	$***,***.**	$*******.**

USAGE Clause

The USAGE clause determines how a data item is stored in the system (its format), and usually, the number of character positions the data item uses.



BINARY, COMPUTATIONAL, COMP

describe a two's complement binary integer with an implied decimal point.

COMPUTATIONAL-3, COMP-3, PACKED-DECIMAL

describe a numeric data item in radix 10, but with each digit of the value stored in half a computer character (4 bits, called a nibble). The sign is stored in a separate, trailing nibble; that is, at the right-hand (least significant) end of the data item. Any unused nibbles are on the left-hand (most significant) end of the data item and are set to zero. SeeTable 8-5, Table 8-5, and Table 8-7.

COMPUTATIONAL-3/PACKED-DECIMAL is also called binary coded decimal form.

The keyword PACKED-DECIMAL is an element of COBOL, but the data format is not. If your program is compiled with a FIPS directive with NONSTANDARDEXT in the `flag-option-list`, the compiler issues a warning message when it finds a COMPUTATIONAL-3/PACKED-DECIMAL data item.

COMPUTATIONAL-5, COMP-5

describes a two's complement binary integer that occupies 2, 4, or 8 character positions (bytes), depending on its PICTURE clause. (See PICTURE clause under "USAGE Clause".)

The COMPUTATIONAL-5 data type is not an element of COBOL. If your program is compiled with a FIPS directive with NONSTANDARDEXT in the `flag-option-list`, the compiler issues a warning message when it finds a COMPUTATIONAL-5 data item.

DISPLAY

describes a sequence of characters stored in standard data format. Its PICTURE clause determines the number and types of characters in the value, which can be used in any context for its category. DISPLAY is the default for elementary data items.

INDEX

describes an index data item—a data item that occupies four character positions and whose value is the occurrence number of a table element. This value cannot be used in computations. An index data item has no PICTURE clause.

NATIVE-2

describes a signed, two's complement binary integer that occupies 2 character positions (bytes) and can have a value in the range -32,768 through +32,767.

NATIVE-4

describes a signed, two's complement binary integer that occupies 4 character positions (bytes) and can have a value in the range -2,147,483,648 through +2,147,483,647.

NATIVE-8

describes a signed, two's complement binary integer that occupies 8 character positions (bytes) and can have a value in the range -9,223,372,036,854,775,808 through +9,223,372,036,854,775,807.

POINTER

describes a 4-byte data item whose value, which is assigned with the SET statement, is the address of another data item.

NOTE: The NATIVE-*n* and POINTER data types are not elements of COBOL. If your program is compiled with a FIPS directive with NONSTANDARDEXT in the *flag-option-list*, the compiler issues a warning message when it finds a NATIVE-*n* or POINTER data item.

Table 8-5 COMPUTATIONAL-3/PACKED-DECIMAL Digit Representation

	Hexadecimal Digit Representation	
Digit Value	**Left* Nibble (Odd Digit)**	**Right* Nibble (Even Digit)**
0	X"00"	X"00"
1	X"10"	X"01"
2	X"20"	X"02"
3	X"30"	X"03"
4	X"40"	X"04"
5	X"50"	X"05"
6	X"60"	X"06"
7	X"70"	X"07"
8	X"80"	X"08"
9	X"90"	X"09"

* Count even and odd from right to left.

Table 8-6 COMPUTATIONAL-3/PACKED-DECIMAL Sign Digit Representation

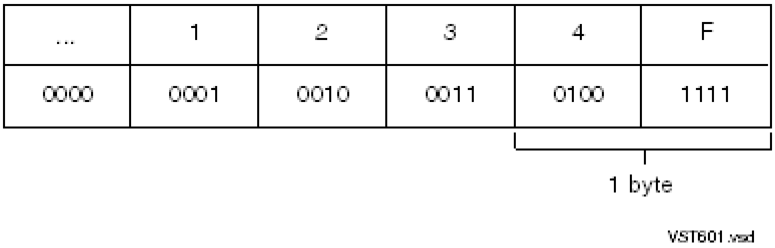
Sign Convention in PICTURE Clause	Sign of Data Item Value	Sign Half-Character (Hexadecimal)
Unsigned	Not applicable	X"0F"
Signed	+	X"0C"
Signed	-	X"0D"

Table 8-7 Numeric Data Storage for the COMPUTATIONAL-3/PACKED-DECIMAL PICTURE Clause

Bytes Required	Number of Digits (Signed or Unsigned)
1	1
2	2-3
3	4-5
4	6-7
5	8-9
6	10-11
7	12-13
8	14-15
9	16-17
10	18

A COMPUTATIONAL-3/PACKED-DECIMAL data item with PICTURE 9999 and value +1234 is stored as shown in Figure 8-3, where `F` represents the nonprinting plus sign.

Figure 8-3 Example of COMPUTATIONAL-3/PACKED-DECIMAL Storage



...	1	2	3	4	F
0000	0001	0010	0011	0100	1111

A COMPUTATIONAL-3/PACKED-DECIMAL data item with PICTURE S9999 and value +1234 is stored as shown in Figure 8-3 except that `F` is replaced by `C` (1100), which represents the plus sign (+).

A COMPUTATIONAL-3/PACKED-DECIMAL data item with PICTURE S9999 and value -1234 is stored as shown in Figure 8-3 except that `F` is replaced by `D` (1101), which represents the minus sign (-).

Usage Considerations:

- National Data Items Cannot Have USAGE Clauses

  Verify that national data items do not have USAGE clauses (inherited, explicit, or implicit) and are not mixed with nonnational data items in group descriptions.

- USAGE Clause Rarely Restricts Actual Usage

  The USAGE clause describes a data item, but rarely restricts how the item is actually used. Exceptions are USAGE INDEX, USAGE POINTER, and rules for certain statements that require the DISPLAY phrase or default to USAGE DISPLAY.

- USAGE Clause Applies to Both Elementary and Data Structures

  Any level of a data description can have a USAGE clause. A USAGE clause at the group level applies to each elementary item in the group, and the USAGE of an elementary item cannot contradict the USAGE clause of a group to which the item belongs. Because data structures are always in the alphanumeric category, their USAGE clauses might not always apply to their own manipulation, for example:

The subitems of a BINARY/COMPUTATIONAL data structure can be used in computations, but the data structure itself cannot.

The subitems of an INDEX data structure are index data items, but the data structure itself is not.

The subitems of a POINTER data structure are pointers, but the data structure itself is not.

- BINARY or COMPUTATIONAL Data Items
  - — Purpose

    The purpose of BINARY or COMPUTATIONAL items is to improve performance and reduce the size of generated code (sometimes by a factor of as much as 100). Use BINARY or COMPUTATIONAL for a value that can be represented in the number of digits specified in the PICTURE clause; for example:

PICTURE	Range of Values
PICTURE 9(4)	0 through 9999
PICTURE S9(4)	-9999 through 9999

  - — PICTURE character-string contents

    A BINARY/COMPUTATIONAL data item's PICTURE character-string can contain only `9` s, the sign character `S`, the decimal point character `V`, and one or more `P` s. The PICTURE clause for an elementary BINARY/COMPUTATIONAL data item must describe the item's category as numeric.

    A BINARY/COMPUTATIONAL data item whose PICTURE character-string has one to four `9` s is stored as 16 bits.

    A BINARY/COMPUTATIONAL data item whose PICTURE character-string has five or more `9` s is stored as 4 or 8 bytes.

  - — Cannot have or inherit SIGN clause

    The description of a group or elementary data item with BINARY/COMPUTATIONAL format cannot include or inherit a SIGN clause, because if such an item is signed, the host computer's architecture determines the representation of the sign.

  - — Cannot have BLANK WHEN ZERO clause

    A BLANK WHEN ZERO clause changes a numeric data item's category from numeric to numeric edited; therefore, the item cannot be BINARY/COMPUTATIONAL.

  - — COMPUTATIONAL data item with value 8224 or 224

    In HP COBOL, if you find that a data item of USAGE COMPUTATIONAL has the value 8224, it might be because it actually consists of 2 bytes, each containing a space character. If it is described as consisting of 3 digits, it appears to have the value 224.

  - — See Trapping Size Errors under PICTURE Clause (page 202).

- COMPUTATIONAL-3/PACKED-DECIMAL Data Items
  - — Purpose

    COMPUTATIONAL-3/PACKED-DECIMAL data items are provided for data compatibility with systems other than NonStop systems. They are not recommended for other purposes.

  - — Inefficiency

    Packed-decimal data format is inefficient because HP hardware does not support it. Avoid using it if possible, especially in benchmarks and performance-critical programs.

- Where you can use them

  You can use COMPUTATIONAL-3/PACKED-DECIMAL data items wherever you can use data items that do not have USAGE DISPLAY.

- SYNCHRONIZED clause

The SYNCHRONIZED clause (with or without the LEFT or RIGHT phrase) has no effect on COMPUTATIONAL-3/PACKED-DECIMAL declarations.

- See Trapping Size Errors under PICTURE Clause (page 202).
- COMPUTATIONAL-5 Data Items
  - Purpose

    The COMPUTATIONAL-5 data type is compatible with the X/Open COBOL specification (see the X/Open CALL Statement (page 811)).

    Signed COMPUTATIONAL-5 data items are equivalent to NATIVE-*n* data items, but unsigned COMPUTATIONAL-5 data items can store a larger range of values than the corresponding NATIVE-*n* data items (see in PICTURE Clause (page 202)).

  - PICTURE clause

    A COMPUTATIONAL-5 data item can have any PICTURE clause containing `9`, `P`, `S`, and `V`. The PICTURE clause determines its size and the range of its values. Bracketed items are optional.

PICTURE Clause	NATIVE-n Equivalent	Range	
		Signed[1]	Unsigned[2]
PIC [S]9(1) - PIC [S]9(4)	NATIVE-2	-32,768 through 32,767	0 through 65,535
PIC [S]9(5) - PIC [S]9(9)	NATIVE-4	-2,147,483,648 through 2,147,483,647	0 through 4,294,967,295
PIC [S]9(10) - PIC [S]9(18)	NATIVE-8	-9,223,372,036,854,775, 808 through 9,223,372,036,854,775, 807	0 through 18,446,744,073,709, 551, 615

[1] NATIVE- *n* items must be in the signed range.

[2] Applies only to PICTURE Clause.

  - SYNCHRONIZED clause causes 2-Byte alignment

    The SYNCHRONIZED clause causes COMPUTATIONAL-5 data items to be aligned on 2-byte or 4-byte boundaries, depending on their size. In this case, LEFT and RIGHT phrases have the same effect.

  - Passing by reference requires SYNCHRONIZED clause

    To pass a COMPUTATIONAL-5 data item to a routine written in a language that requires parameters to be 2-byte-aligned, describe the data item with the SYNCHRONIZED clause.

  - Decimal numeric literal cannot have maximum value of COMP-5 data item

    You cannot specify a decimal numeric literal that is the maximum value allowed for a COMPUTATIONAL-5 data item. The reason is that the maximum value allowed for a COMPUTATIONAL-5 data item has 19 digits and a decimal numeric literal cannot have more than 18 digits. You can use a hexadecimal numeric literal for the maximum value allowed for a COMPUTATIONAL-5 data item.

- DISPLAY Data Items

  If the data description entry for an elementary item does not include a USAGE clause and is not subordinate to any data structure description that includes an explicit USAGE clause, then the effect is as if a USAGE DISPLAY clause had appeared in the elementary item's data description entry; therefore, an elementary data item's usage attribute can be inherited, explicit, or implicit.

  DISPLAY format is required for these data items:

  — Data items subordinate to a data structure that is described with a VALUE clause or has condition-names associated with it

  — Signed numeric data items that are affected by a data structure's SIGN clause

  — Elementary items whose data description entries contain SIGN, JUSTIFIED, or BLANK WHEN ZERO clauses

  — See Trapping Size Errors under PICTURE Clause (page 202).

- INDEX Data Items

  The description of a group or elementary data item with INDEX format cannot include any of these clauses:

  — PICTURE

  — VALUE

  — SYNCHRONIZED

  — JUSTIFIED

  — BLANK WHEN ZERO

  An index data item description cannot be followed immediately by any level-88 items; that is, an index data item cannot serve as a conditional variable.

  The compiler stores each INDEX data item as 4 bytes, aligned on a 4-byte boundary.

- NATIVE-*n* Data Items

  The description of a group or elementary data item with NATIVE-*n* format cannot include any of these clauses:

  — PICTURE

  — SYNCHRONIZED

  — JUSTIFIED

  — BLANK WHEN ZERO

  — SIGN

  If an arithmetic statement includes a SIZE ERROR phrase, the size error condition occurs as defined in SIZE ERROR Phrase (page 254), except that in a NATIVE-*n* receiving item the test is for truncation of significant bits, not significant decimal digits.

  If a NATIVE-*n* data item used as a receiving operand specifies the ROUNDED phrase, decimal (not binary) rounding occurs, as described in SIZE ERROR Phrase (page 254).

  Because COBOL limits literals to a length of 18 digits, the maximum value that you can assign to a NATIVE-8 data item (by initialization with a VALUE phrase or by a MOVE statement, for example) is the 18-digit value of -/+999999999999999999, rather than the maximum storable 19-digit value of -9223372036854775808 or +9223372036854775807.

- How NATIVE-*n* and BINARY/COMPUTATIONAL Formats Differ

  The significant difference between a NATIVE-*n* data item (which occupies *n* character positions by definition) and a COMPUTATIONAL data item that happens to occupy *n* character positions is that the COMPUTATIONAL item has the number of decimal digits declared by its PICTURE clause. Although a NATIVE-*n* data item and a certain COMPUTATIONAL data item both occupy *n* character positions, they cannot necessarily assume the same set of values.

For example, a NATIVE-2 data item and a COMPUTATIONAL data item with PICTURE 9999 both occupy 2 character positions, but any value larger than 9999 is truncated on the left before it is assigned to the COMPUTATIONAL data item, whereas the NATIVE-2 data item can be assigned any value in the range -32768 through +32767.

A program can specify a NATIVE-*n* data item as an operand anywhere that it can specify a BINARY/COMPUTATIONAL item (of equivalent size in character positions), such as in arithmetic expressions, MOVE statements, and so on. In all cases, the value of the item is interpreted as a signed integer.

- How NATIVE-*n* and COMPUTATIONAL-5 Formats Differ

  The difference between a NATIVE-*n* data item and a COMPUTATIONAL-5 data item is that the NATIVE-*n* data item is 2-byte-aligned and the COMPUTATIONAL-5 data item is byte-aligned and requires a PICTURE clause (see PICTURE Clause (page 202)).

- POINTER Data Items

  — Clauses not allowed

  A group or elementary data item described with a USAGE POINTER clause cannot have any other clauses except VALUE IS NULL or VALUE IS NULLS. The clause VALUE IS NULL or VALUE IS NULLS initializes the pointer to a value (all ones) that causes an address fault if the pointer is referenced.

  — How HP COBOL pointers differ from HP C and Pascal pointers

  An HP COBOL POINTER data item is not the same as a pointer in HP C or Pascal. An HP COBOL POINTER data item merely provides a container for an address. You can access an HP COBOL POINTER data item only in a conditional expression, a SET statement, or as a parameter in the USING phrase of a CALL or ENTER statement.

  The statement

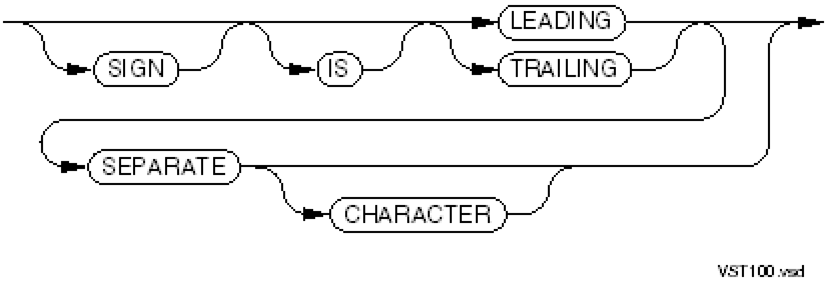
  ```
  MOVE "ABC" TO PTR1
  ```

  where PTR1 is an HP COBOL pointer, does not move the value “ABC” to the address that PTR1 contains.

  For an HP COBOL pointer to accomplish what an HP C or Pascal pointer accomplishes, you must declare a base data item in the Linkage Section. You must not put the base data item in the USING phrase of the PROCEDURE DIVISION heading (that is, you must not pass the base data item as a parameter).

SIGN Clause

The SIGN clause specifies the position and mode of representation of the operational sign for a numeric data item. It can only be used for DISPLAY items with an *S* in the PICTURE character-string. This precludes the use of the SIGN clause with USAGE NATIVE-*n* or USAGE POINTER.



Usage Considerations:

- Function of the SIGN Clause

  The representation of every signed numeric data item includes an operational sign. When the data item is USAGE BINARY or USAGE COMPUTATIONAL, the compiler automatically chooses the appropriate sign convention in accordance with the item's internal representation. When the data item is USAGE DISPLAY, the SIGN clause determines the position and representation of the sign. When no SIGN clause is used, SIGN TRAILING is assumed.

  When the SEPARATE phrase is present, the sign is maintained as a separate character to be considered in the size of the item. A plus (+) for a positive value or a minus (-) for a negative value is placed at the beginning or end of the item's value, depending on the presence of LEADING or TRAILING. If the sign character position contains a value other than plus (+) or minus (-), any operation that uses the item as a numeric sending item has undefined results.

  If the SEPARATE phrase is absent, negative values are represented by a modification of the high-order bit of the high-order or low-order digit of the stored number (depending on the presence of LEADING or TRAILING). The sign is not considered in the size of the item. For positive values of the data item, the value of the sign bit is 0; for negative values, the value of the sign bit is 1.

NOTE: If you move a data item with an included sign to another item with a group move, the character with a negative sign is not a legitimate COBOL character. This can cause problems. Use an item with an included sign only in operations that expect numeric data.

- Use of SIGN With Elementary or Data Structures

  The SIGN clause can appear in the description of an elementary item or a data structure. When it appears in the description of an elementary item, it specifies the sign attributes for that item. The SIGN clause never applies to a data structure, whose category is always alphanumeric. When it appears in the description of a data structure, it specifies the sign attributes of every signed numeric item belonging to that group, except as noted.

- A Lower-Level SIGN Clause Overrides an Upper-Level One

  A SIGN clause appearing in the data description entry of a data structure or a numeric data item overrides the SIGN clause of any data structure to which that item is subordinate; therefore, when more than one SIGN clause apparently applies to a data item, the one specified at the lowest level in the hierarchy takes precedence over the ones specified at a higher level.

- Data Structure With SIGN Clause Must Include DISPLAY Numeric

  When the data description entry of a data structure includes the SIGN clause, at least one subordinate elementary item must be numeric and be described with a PICTURE character-string containing `S`. All such items must be USAGE DISPLAY.

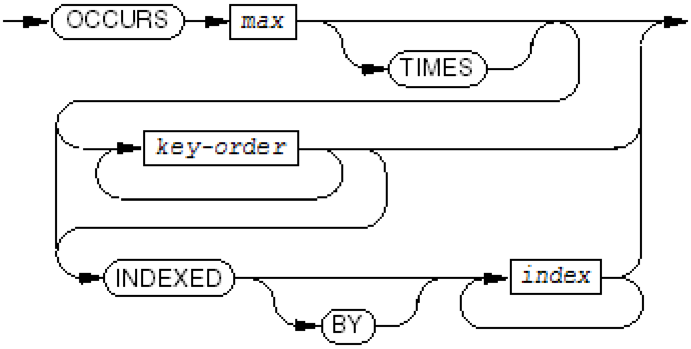
- Elementary Item With SIGN Clause Must Be DISPLAY Numeric

  When the data description entry of an elementary item includes the SIGN clause, its category must be numeric, it must be described with a PICTURE character-string containing `S`, and it must be USAGE DISPLAY.

- The SIGN clause cannot be used for national data items.

OCCURS Clause for Fixed-Size Tables

The OCCURS clause defines tables (sets of repeated items), eliminating the need for separate item entries. These tables can contain a fixed number of elements or can vary within given limits. The OCCURS clause without a DEPENDING phrase defines a fixed-size table.

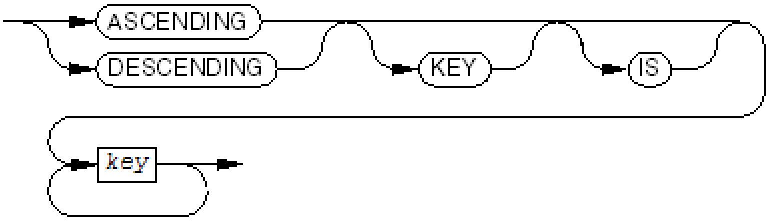


VST104.vsd

max

is an integer literal that specifies the number of elements in the table.

key-order



VST105.vsd

determines whether the table elements are arranged in ascending or descending order of key values.

key

is either the name of the entry containing the OCCURS clause or the name of an entry subordinate to the entry containing the OCCURS clause.

index

is a name for a compiler-created item used to select an element from the table. Each *index* must be unique in the program because you cannot qualify it.

Usage Considerations:

- The OCCURS Clause Describes a Table

  The OCCURS clause specifies that the data-name with which it is associated (the one following the level number in the same data description entry) identifies a table containing multiple occurrences of the elementary item or data structure described by its entry. When the data description entry does not define a name (that is, FILLER is specified either explicitly or implicitly), the table is anonymous and its elements cannot be referred to directly. Except for the OCCURS clause itself, all of the entry's clauses apply to each occurrence of the item described.

- The INDEXED Phrase Declares Index-Names

  The presence of an INDEXED phrase serves also to declare one or more index-names for the table. The names thus declared must not be separately declared elsewhere. These, and only these, index-names can be used as the subscript when an index-name is used to select a particular occurrence of this table.

- KEY Phrase

  Each key (*key* in the syntax diagram) must reference a data item that is subordinate to the table, except that the first key in a list can reference the table itself. No OCCURS clause can apply to a key unless it also applies to the table. As a consequence, a key cannot be or belong to a table that is subordinate to the table of which it is a key. An OCCURS statement can have a maximum of 31 keys. The representation of the reference to the key in this context does not include subscripts.

The KEY phrase indicates that within the table occurrences, the values referenced by the keys are arranged in a consistent order.

When the key appears in an ASCENDING phrase, the values are arranged in ascending order of occurrence numbers (that is, the value associated with an occurrence is never less than the value associated with any preceding occurrence).

When the key appears in a DESCENDING phrase, the values are arranged in descending order (that is, the value associated with an occurrence is never greater than the value associated with any preceding occurrence).

This characteristic is relevant only during the execution of a SEARCH ALL statement that references the table. All of the KEY phrases taken together define a single list, within which the keys appear in decreasing order of search significance.

- Subscripts

  Each appearance of a table's data-name must include an appropriate subscript, except when the entire table is desired (this only occurs in the SEARCH statement and some intrinsic functions). References appearing in a REDEFINES clause or a KEY phrase of the OCCURS clause are not considered operands and must not include subscripts.

- References to Subordinates Must be Subscripted

  If the data-name is a data structure, then all items belonging to the group must also be subscripted whenever they are used as operands; however, subordinate data-names used in the KEY phrase, or as objects of a REDEFINES clause, are not considered operands and must not be subscripted or indexed.

- OCCURS Limited to Certain Level Numbers

  The OCCURS clause must not be specified for a data description entry having a level number of 01, 66, 77, or 88.

- OCCURS Forbidden for Redefined Data Elements

  The OCCURS clause cannot appear in a data description entry whose data-name is specified in a REDEFINES clause of some subsequent data description entry; therefore, an entire table cannot be redefined. It is possible, however, to redefine subordinate data items of table elements. (See REDEFINES Clause.)

- Multidimensional Tables

  Data items subordinate to the subject of an entry described with an OCCURS clause can themselves contain an OCCURS clause; therefore, tables can consist of multiple occurrences of subordinate tables for a maximum of seven levels. A data description entry containing either type of OCCURS clause can be followed by subordinate entries containing the OCCURS clause for fixed-size tables; however, a data description entry with an OCCURS DEPENDING clause (discussed in the next topic) cannot be subordinate to a group entry described with either type of OCCURS clause.

- Order of Subscripts in References

  Up to seven subscripts can be used with one data item. When more than one subscript is used, they are written to the right of the table name, in the order of more inclusive to less inclusive dimensions of the data organization (that is, the first subscript matches the table having the lowest level number).

  All qualification must precede any subscripts (see VEHICLE in Example 8-17).

- Index is a Variant of Subscript

  In COBOL, an index is a variant of a subscript. The program can define an indexed table within a nonindexed table (or the reverse). Both indexes and subscripts can be augmented with an increment or a decrement. That is, if a table is described:

  ```
  01  A-TABLE.
      03  ROWE OCCURS 20 TIMES INDEXED BY R.
          05  KOLUMN OCCURS 10 TIMES.
              07 ELEMENT    PIC X.
  ```

  then a statement in the program could refer to any of these (assuming X, R, and Y had acceptable values):

  ```
  ELEMENT ( R , 5 )
  ELEMENT ( 3 , Y - 3 )
  ELEMENT ( R + 1 , X + 2 )
  ```

- Applying Index-Names to Other Tables

  If you use an index-name specified in the INDEXED phrase of one table to refer to an element of another table in the program, the compiler reports an error. You must use the SET statement to the index-name associated with one table to the occurrence-number designated by the index-name associated with a different table.

  In Example 8-17, MY-TABLE is appropriate for the SEARCH ALL statement. The order of the table is governed by FIRST-ITEM. It can just as well be declared to be governed by MY-TABLE, FIRST-A, FIRST-B, FIRST-B-1, FIRST-B-2, or SECOND-ITEM.

Example 8-17 Fixed-Size Table

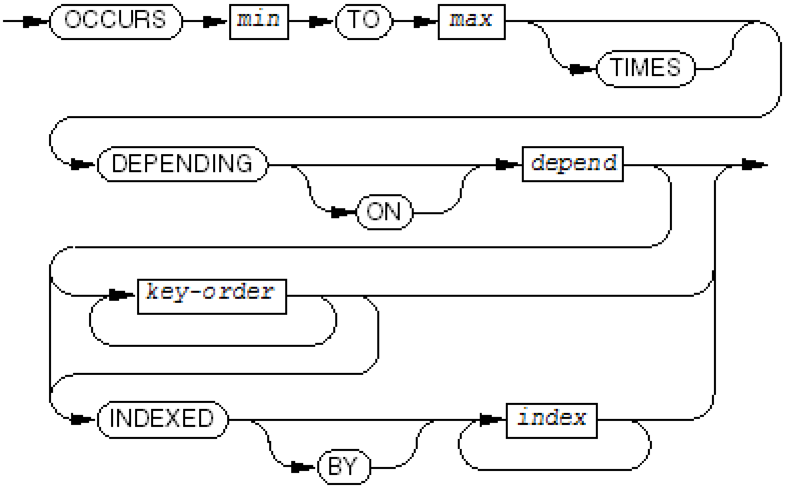
```
01  MY-TABLE-RECORD.
    02  MY-TABLE OCCURS 100 TIMES
                 ASCENDING KEY IS FIRST-ITEM
                 INDEXED MY-INDEX.
        05  FIRST-ITEM.
            08 FIRST-A  PICTURE 99.
            08 FIRST-B.
               10 FIRST-B-1 PICTURE 99.
               10 FIRST-B-2 PICTURE 99.
        05  SECOND-ITEM PICTURE X(6).
01  VEHICLE.
    03  MODEL OCCURS 9 TIMES.
        05 STYLE OCCURS 12 TIMES.
           07 COLOR OCCURS 15 TIMES PICTURE 9(10).
        05 LOCATION              PICTURE X(25).
```

These identifiers specify elements of the table VEHICLE in Example 8-17:

```
MODEL (3)
STYLE OF MODEL (3, 11)
COLOR OF STYLE OF MODEL (3, 11, 14)
```

OCCURS Clause for Variable-Size Tables

The OCCURS clause with a DEPENDING phrase defines a variable-size table.



VST106.vsd

min

is an integer literal that specifies the minimum number of table elements. Its value is in the range 0 through *max*.

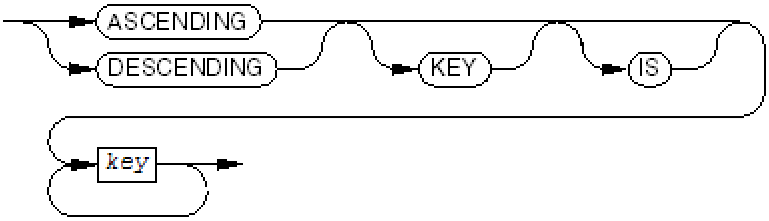
max

is an integer literal that specifies the maximum number of table elements.

depend

is an integer data item that controls the size of the table. As *depend* increases or decreases, the number of table elements increases or decreases. When the table size decreases, elements beyond the element whose ordinal position is specified by the new value of *depend* are lost—even if the next statement increases the table to include them. When the table grows, you must assign values to the new elements prior to using them.

key-order



VST105.vsd

determines whether table elements are arranged in ascending or descending order of key values.

key

is either the name of the entry containing the OCCURS clause or the name of an entry subordinate to the entry containing the OCCURS clause.

index

is a name for compiler-created item used to select an element from the table. Each *index* must be unique in the program because you cannot qualify it.

Usage Considerations

- See the usage considerations for OCCURS Clause for Fixed-Size Tables.
- Depending Item

  The depending item (`depend` in the syntax diagram) must reference an integer numeric data item that is capable of containing the value of `max`. The storage space associated with the depending item cannot be contained within or overlap the storage space associated with the table whose occurrences it controls; therefore, the depending item cannot be described as a subordinate of the variable-occurrence data item. If the table is external, the depending item must also be external.

  At any point during execution of the object program-unit, the current value of the depending item represents the number of occurrences in the table; therefore, its value must never be less than `min` or greater than `max` during execution of a statement that references the table, any of its subordinate items, or any data structure that contains the table.

  You can use the CHECK compiler directive to verify that the depending item has an acceptable value.

- Concept of a Variable-Size Item

  Any data structure containing a variable-occurrence table has a variable size. When such an item is referenced as an operand, its value normally includes only those table elements whose occurrence numbers are less than or equal to the value of the depending item at the start of the operation; however, when the data structure is used as a receiving operand and the depending item is itself a subordinate of the data structure (so that the operation changes its value), the size of the receiving operand is the maximum size of the data structure; that is, the operation proceeds as if the depending item contained the value of `max`.

- Changing the Depending Item Affects Table Values

  Whenever the value of the depending item is increased, occurrences of the table are dynamically appended to each affected variable-size data structure; however, the portion of the item's value contained in these new occurrences has an unpredictable content unless the same or a subsequent operation assigns a known value to that portion.

  Whenever the value of the depending item is reduced, the values of table members whose occurrence numbers exceed the new limit are lost. Even after a subsequent increase in the value of the depending item restores these occurrences to the table, their values are not predictable and they must not be used as operands until known values are assigned.

- OCCURS DEPENDING Incompatible with REDEFINES

  A variable-occurrence data item cannot be specified as a redefinition or as part of a redefinition; that is, an OCCURS DEPENDING clause cannot appear in a data description entry that includes a REDEFINES clause or that is subordinate to a data description entry that includes a REDEFINES clause.

- OCCURS DEPENDING Cannot Be Nested and Must Be Last

  A data description entry with an OCCURS DEPENDING clause can only be followed, within its record description, by descriptions of subordinate items; that is, only one table with a variable number of occurrences can appear in a single record description entry, and the data items it contains must be the last data items in the record. Note, however, that a fixed-occurrence data item can be subordinate to a variable-occurrence data item.

Example 8-18 Variable-Size Table

```
01  ACTIVITY-TABLE-RECORD.
    03  ACTIVITY-COUNT PICTURE 99.
    03  ACTIVITY-TABLE OCCURS 10 TO 20 TIMES
                       DEPENDING ON ACTIVITY-COUNT
                       INDEXED BY SAVE-INX-1
                                  SAVE-INX-2.
        05  ACTIVITY-ENTRY  PICTURE 999.
```

Example 8-19 Fixed-Size Table as an Element of a Variable-Size Table

```
01  DISPOSAL-COMPANIES.
    03 H-COUNT PICTURE 99.
    03 HAULER OCCURS 1 TO 12 TIMES DEPENDING ON H-COUNT.
       05 VEHICLE OCCURS 15 TIMES.
          07 V-NUMBER PICTURE 9(5).
          07 CAPACITY-CU-FT PICTURE 9(4).
          07 SERVICE-SCHEDULE PIC X.
          ...
```

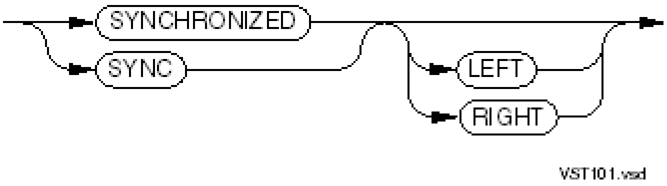
SYNCHRONIZED Clause

The SYNCHRONIZED clause forces alignment of an elementary item on the most natural computer storage boundary. The elementary item cannot be a pointer or a national data item.

Efficiency improves if data is aligned on natural computer memory boundaries, because additional object code is sometimes required to store or retrieve data when 2 bytes of the same word belong to different data items. Also, when a data item is stored within 2-byte boundaries, the complexity of code to process it can decrease.

The SYNCHRONIZED clause is usually unnecessary, because the automatic alignment is also the most natural: each level-77 item of the Working-Storage Section and Extended-Storage Section and each level-01 item is aligned on a physical 2 8-byte boundary. (A word is 16 bits and a TNS/E word is 32 bits.)

Both automatic and forced alignment can cause the compiler to generate implicit FILLER data.



Usage Considerations:

- LEFT and RIGHT

  The optional keywords LEFT and RIGHT have no effect in HP COBOL.

- Where You Can Use the SYNCHRONIZED Clause

  You can use the SYNCHRONIZED clause only in the data description entry of an elementary item that does not have a USAGE POINTER clause. You cannot imply or specify the SYNCHRONIZED clause for national data items.

- DISPLAY Data Items That the SYNCHRONIZED Clause Does Not Affect

  The SYNCHRONIZED clause does not affect DISPLAY items at levels other than 01 or 77. They are composed of one or more character positions and stored as a corresponding number of 8-bit bytes aligned on byte or 2-byte boundaries, whether or not they are described with the SYNCHRONIZED clause.

- COMPUTATIONAL or BINARY Data Items

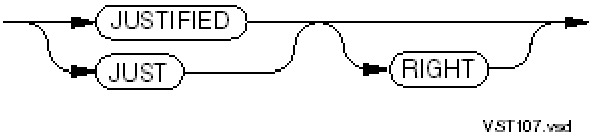
  If the program is not compiled with the PORT directive, then a data item is aligned on a 2-byte boundary unless it is described with the SYNCHRONIZED clause. If it is described with the SYNCHRONIZED clause, it is aligned on a 2-byte boundary if its size is less than 4 bytes, and on a 4-byte boundary if its size is 4 bytes or larger.

  If the program is compiled with the PORT directive, and the COMPUTATIONAL or BINARY data item is not described with the SYNCHRONIZED clause, it is byte-aligned.

JUSTIFIED Clause

The JUSTIFIED clause causes nonstandard positioning of data within a receiving item. It can appear only in the data description of an elementary item that meets these criteria:

- Its category is alphabetic or alphanumeric.
- It is not subordinate to any data structure that has associated condition-names.
- It is not subordinate to any data structure that is described with a VALUE clause.



When a receiving data item is described with the JUSTIFIED clause, the standard alignment rules do not apply. Instead, a sending item too big for the receiving item is truncated on the left. If the sending item is smaller, its rightmost character is aligned with the rightmost character of the receiving field and the value is extended to the left with space characters.

Usage Considerations:

- Effect of RIGHT on Compilation

  The optional keyword RIGHT has no effect on the compilation, and serves only to provide emphasis.

- Trailing Spaces Are Not Ignored

  Trailing spaces in a data item are not ignored when the data item is moved to a right-justified field. For example,

  ```
  77 F1   PIC X(3) JUSTIFIED RIGHT.
  77 F2   PIC X(2) VALUE "A".
  MOVE F2 TO F1.
  ```

  gives F1 the value

  ```
  " A "
  ```

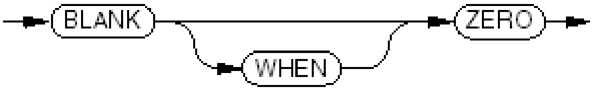
  not the value

  ```
  "  A"
  ```

NOTE: The JUSTIFIED clause has no effect on initialization by the VALUE clause.

BLANK WHEN ZERO Clause

The BLANK WHEN ZERO clause fills an item with spaces when its value is 0.



VST246.vsd

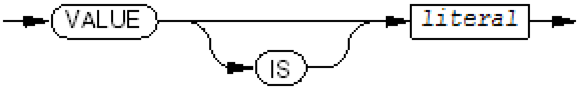
You can use the BLANK WHEN ZERO clause only with elementary numeric or elementary numeric edited data items. This clause revises the category of a numeric item to numeric edited; therefore, the item must be USAGE DISPLAY.

NOTE: The BLANK WHEN ZERO clause has no effect on initialization by the VALUE clause.

The BLANK WHEN ZERO clause cannot be used for national data items.

VALUE Clause

This form of the VALUE clause assigns an initial value to an elementary data item or data structure described in the Working-Storage Section or Extended-Storage Section. For the form of the VALUE clause that assigns an initial value to a condition-name in any section of the Data Division, see Descriptions of Condition-Names for Values (Level 88).



VST108.vsd

`literal`

is a literal or a figurative constant whose value is the initial value of the data item.

Usage Considerations:

- Restrictions on Use of the VALUE Clause

  Do not use a VALUE clause in the description of:

  - A data item described with an EXTERNAL clause
  - An elementary data item described with a USAGE INDEX or REDEFINES clause or subordinate to a data structure described with a VALUE or REDEFINES clause
  - A data structure that has a subordinate item described with a JUSTIFIED or SYNCHRONIZED clause, a USAGE clause other than USAGE DISPLAY (the default), or (except for a condition-name) a VALUE clause

- Restrictions on the Value of `literal`

  - Numeric data items

    If the data item is numeric, `literal` must be a numeric literal or the figurative constant ZERO, ZEROS, or ZEROES. The keyword ALL cannot appear in the figurative constant.

    If `literal` is a numeric literal, its value must not require the truncation of nonzero digits; that is, the number of significant fraction digits in `literal` cannot exceed the number of fraction digits in the numeric data item, and the number of significant integral digits in `literal` cannot exceed the number of integral digits in the numeric data item.

    If `literal` is a signed numeric literal, it must either be NATIVE-*n* or have an associated signed numeric PICTURE character-string. Initialization follows standard alignment rules for numeric data items.

  - Nonnumeric data items

    If the data item is nonnumeric, `literal` must be nonnumeric or a figurative constant. The keyword ALL can appear in the figurative constant.

    The value of `literal` cannot exceed the size indicated by the data item's PICTURE clause. Editing characters in the PICTURE clause are included when the size of the item is determined, but do not effect initialization; therefore, the value of `literal` must conform to the edited form. Initialization follows standard alignment rules for alphanumeric data items. The BLANK WHEN ZERO and JUSTIFIED clauses do not affect initialization.

— Pointer data items

  If the data item is described with a USAGE POINTER clause, `literal` must have the value NULL or NULLS.

— Data structures

  If the data item is a data structure, `literal` must be either a figurative constant or a nonnumeric literal. A data structure is initialized without consideration for its subordinate elementary or data structure items.

  If the data structure is described with one or more OCCURS clauses, every occurrence of the repeated item is initialized with the specified value. If an OCCURS clause has a variable number of occurrences, the initialization proceeds as if the data structure has its maximum number of occurrences.

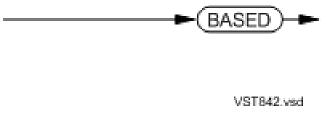
Example 8-20 VALUE Clauses

```
01   MAIN-HEADING.
     05   FILLER      PIC XX     VALUE SPACES.
     05   FILLER      PIC X(8)   VALUE "DIVISION".
     05   FILLER      PIC XX     VALUE SPACES.
     05   FILLER      PIC X(6)   VALUE "REGION".
          ...
01   COUNTERS.
     05   NO-OF-READS    PIC 9(5)    VALUE ZEROS.
     05   NO-OF-WRITES   PIC 9(5)    VALUE ZEROS.
```

BASED Clause

The BASED clause specifies that the record is a BASED item. The compiler does not allocate storage for a BASED item; instead you associate the BASED item with another data item or with dynamically allocated storage at run time. Only level-01 and level-77 items in the Working-Storage, Extended-Storage, and Linkage Sections can be BASED items.

The compiler allocates an implicit pointer for each BASED item. The initial state for this implicit pointer is NULL. Use the SET ADDRESS statement to associate the BASED item with another data item. Use the ALLOCATE statement to associate the BASED item with dynamically allocated storage.



Usage Considerations:

- BASED Clause Cannot be Used With EXTERNAL Clause

  You cannot specify the BASED clause in a data description entry that includes the EXTERNAL clause.

- Use With VALUE Clause

  You can use the VALUE clause with a BASED item, but the VALUE clause has no effect unless the ALLOCATE statement with the INITIALIZED phrase is used with the item.

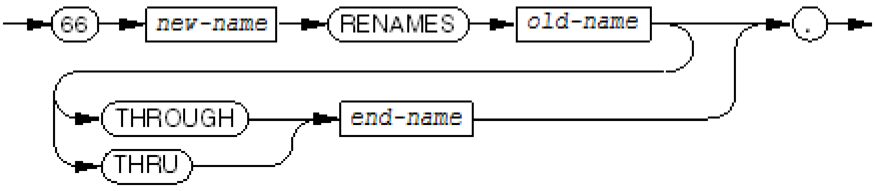
- Use in Linkage Section
  - A BASED item declared in the Linkage Section cannot be listed in the USING phrase of the procedure division header, and has no corresponding actual parameter in the calling program-unit.
  - The lifetime of the implicit pointer for a BASED item in the Linkage section ends when the program returns control to its calling program; the next time this program is called, the implicit pointer is again initialized to NULL.
- Based Item Cannot be Transferred to Backup Process

  A BASED item cannot be transferred to a backup process by the CHECKPOINT statement.
- Effect of CANCEL Statement

  When a CANCEL statement restores a previously-called program to its initial state, each BASED item's implicit pointer is set to NULL. Note that this action does not release any dynamic memory addressed by the implicit pointer; any such dynamic memory might become inaccessible.

Descriptions That Rename Items (Level 66)

The RENAMES clause assigns a new data-name to an item or to two or more contiguous items in a record. RENAMES does not cause allocation of storage. You can put level-66 items in any section of the Data Division.



VST109.vsd

new-name

is the alias for an elementary item or a data structure.

old-name

is either the name of a data structure or the name of the first of several items to be given an alias. It can be qualified.

end-name

is the last group name or last item name included in the alias name. It can be qualified.

The level-66 data description entry in Example 8-21 renames several items in a record so that you can refer to the items by one name.

Example 8-21 Level-66 Data Description Entry

```
05   CARD-CODES.
     10  STORE-CODE      PIC 9.
     10  STATE-CODE      PIC 9(4).
05   ACCOUNT-NUMBER      PIC 9(6).
05   CHECK-DIGIT         PIC 9.
66   CARD-NUMBER RENAMES CARD-CODES THRU CHECK-DIGIT.
```

Usage Considerations:

- RENAMES Stands Alone

  Because RENAMES merely renames a group of existing data items and does not redescribe any of their characteristics, no other clauses can be used with it. One or more RENAMES entries can be written for a logical record. They can occur in any order but must immediately follow all other data description entries for the record.

- Restrictions on Names

  old-name and *end-name* must be data areas within the same logical record and must be different. No part of the storage area referenced by *end-name* can occupy character positions preceding the beginning of the storage area referenced by *old-name*. The area referenced by *end-name* can overlap that referenced by *old-name*, but must extend beyond it; therefore, *end-name* cannot be subordinate to *old-name*.

  old-name and *end-name* cannot be the names of data entries with level number 01, 66, 77, or 88.

  Neither *old-name* nor *end-name* can have an OCCURS clause in its data definition or be subordinate to an item whose description has an OCCURS clause.

- Level-66 item is Elementary or Group

  When *end-name* appears, *new-name* is a data structure including all elementary items starting with *old-name* (if it is an elementary item) or the first elementary item in *old-name* (if it is a data structure) and concluding with *end-name* (if it is an elementary item) or the last elementary item in *end-name* (if it is a data structure).

  When *end-name* does not appear, *new-name* merely renames *old-name* and is a data structure only if *old-name* is a data structure. The new item inherits all characteristics of the old item.

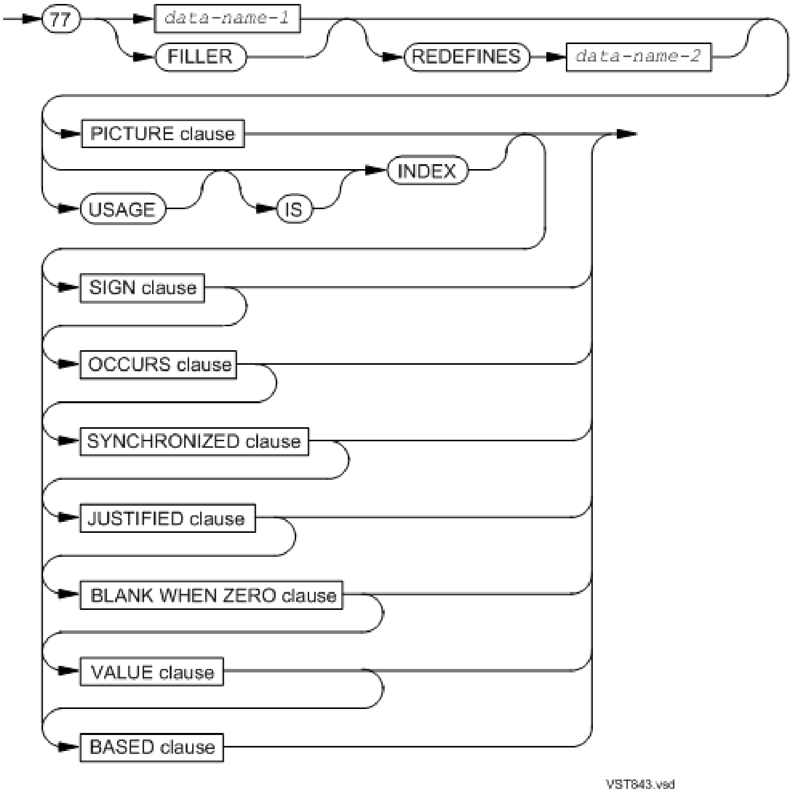
- Restrictions on Data Area

  If *end-name* appears, its data area can overlap that of *old-name* but must extend beyond it; however, no part of the data area described by *end-name* can occupy character positions preceding the beginning of the area described by *old-name*. Because no part of the data area described by *old-name* can occupy character positions following the end of the area described by *end-name*, *end-name* cannot be subordinate to *old-name*.

  No item within the renamed area can have a variable size.

Descriptions of Noncontiguous Elementary Items (Level 77)

In the Working-Storage or Extended-Storage Section, you do not need to use the record structure to define elementary items that bear no hierarchical relationship to one another. Instead, you can classify and define them as noncontiguous elementary items. Use a separate data description entry that begins with the special level number 77 for each of these items



`data-name-1`
: is the name of the noncontiguous elementary item.

FILLER
: is a place holder for a data item to which the program never refers.

`data-name-2`
: is the name of a data item being redefined. See REDEFINES Clause.

PICTURE clause
: is as described in PICTURE Clause.

INDEX
: describes an index data item—a data item that occupies four character positions and whose value is the occurrence number of a table element. This value cannot be used in computations. If a level-77 description contains INDEX, then it cannot include the SYNCHRONIZED, JUSTIFIED, or BLANK WHEN ZERO clause.

SIGN clause
: is as described in SIGN Clause.

OCCURS clause
: is as described in OCCURS Clause for Fixed-Size Tables or OCCURS Clause for Variable-Size Tables.

SYNCHRONIZED clause
: is as described in SYNCHRONIZED Clause. Do not use the SYNCHRONIZED clause with INDEX.

JUSTIFIED clause
: is as described in JUSTIFIED Clause. Do not use the JUSTIFIED clause with INDEX.

BLANK WHEN ZERO clause
: is as described in BLANK WHEN ZERO Clause. Do not use the BLANK WHEN ZERO clause with INDEX.

VALUE clause

is as described in VALUE Clause. Do not use the VALUE clause with INDEX.

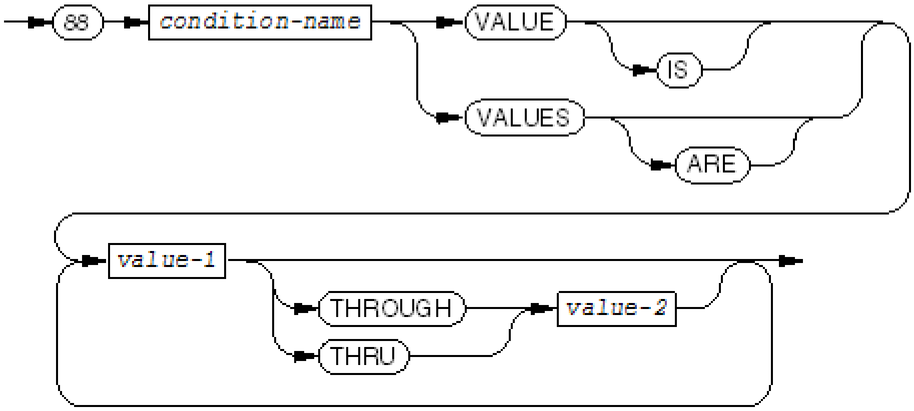
BASED clause

is as described in BASED Clause. Do not use the BASED clause with the EXTERNAL clause or the REDEFINES clause.

Descriptions of Condition-Names for Values (Level 88)

You can assign a name to a specific value, set of values, or range of values that a data item can have and use that name as a condition in a conditional statement. The name is called a *condition-name*, and the data item associated with it is called a *conditional variable*.

To do this, you must put one or more level-88 items (each including a condition-name and a VALUE clause specifying a value or a range of values for that condition-name) immediately after the data description of the conditional variable. You can put level-88 items in any section of the Data Division.



VST112.vsd

condition-name

is the name of the condition value.

value-1

is a literal. It is either a single value or the first in a range of values tested by the condition.

value-2

is a literal and is the final value in a range of values tested by the condition.

Example 8-22 Condition-Names for Values (Level 88)

Declaration:

```
05   RETURN-CODE     PIC 99.
     88   END-OF-FILE          VALUE 01.
     88   ERROR-ON-READ        VALUE 02.
     88   PERMANENT-ERROR      VALUE 03.
     88   ERROR-ON-WRITE       VALUE 04.
```

Statement using one of the condition-names:

```
IF END-OF-FILE
   PERFORM END-UP-OPERATION
   CLOSE FILE-IN
END-IF
```

Definition of an item that has a range of values:

```
05   tax-code              PIC 99.
     88   tax-range     VALUES ARE 00, 03, 07 THROUGH 11.
```

Statement testing if tax-code is 00, 03, 07, 08, 09, 10, or 11:

```
IF NOT TAX-RANGE
   PERFORM TAX-ERROR-ROUTINE.
```

Example 8-23 VALUE Clauses for a Level-88 Data Item

```
01   ZIP-CODE.
     03 ZIP-FIRST-3 PICTURE 999.
        ...
        88 NEW-YORK     VALUE IS 090 THRU 098,
                                 100 THRU 149.
        88 PENNSYLVANIA VALUE IS 150 THRU 196.
        ...
```

Usage Considerations:

- Condition-Name Description Location

  All condition-name entries for a particular conditional variable must immediately follow the entry describing that variable.

- Cannot Use With Certain Descriptions

  A condition-name can be associated with any data description entry, even if specified explicitly or implicitly as FILLER, with these exceptions:

  — Level-66 items

  — Level-88 items

  — Index data item

  — Data structure having any subordinate item described with the JUSTIFIED or SYNCHRONIZED clause, or which are not USAGE DISPLAY

  As the syntax presentation shows, the only clause permitted in a condition-name entry is the VALUE clause containing the value, values, or range of values associated with the condition-name; therefore, the characteristics of the condition-name are implicitly those of the condition variable.

- Setting a Condition-Name to TRUE

  You can set the value of a condition-name to TRUE with the SET TO statement. See Nonpointer Data Items (page 448).

- Numeric data items

  If an item is in the numeric category, all literals in the VALUE clause must be numeric literals or figurative constants ZERO, ZEROS, or ZEROES (the keyword ALL cannot appear in this context), and they must be in the range of values set by the PICTURE character-string of the conditional variable. A signed numeric literal only applies to a signed numeric conditional variable.

A numeric literal must not have a value that would require the truncation of nonzero digits; therefore, the number of significant fraction digits in the numeric literal cannot exceed the number of fraction digits in the data item, and the number of significant integral digits in the numeric literal cannot exceed the number of integral digits in the data item.

- Nonnumeric data items

  If an item is not in the numeric category, all literals of VALUE must be nonnumeric literals or figurative constants (the ALL form of figurative constant is legal in this context), and must not exceed the size of the conditional variable.

- Multiple Values

  More than one value or a range of values can be given for a condition-name entry. Whenever THROUGH is used, the left-hand literal must be less than the right-hand literal.

- Overlap

  The values of different condition-names associated with the same conditional variable are permitted to overlap; therefore, it is possible to construct sets of condition names associated with one conditional variable in which more than one condition-name has the same truth value.

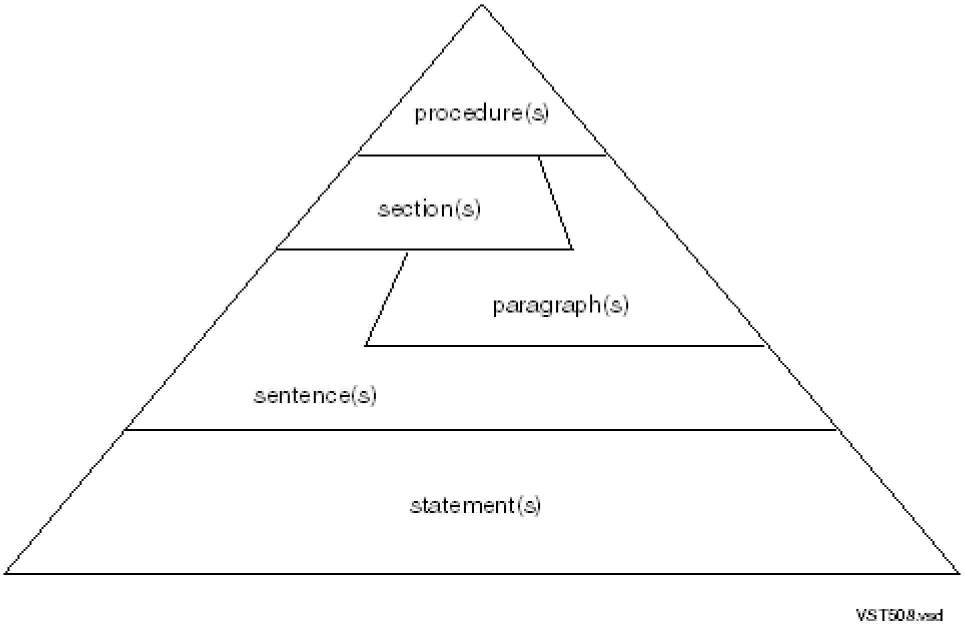
9 Procedure Division

The Procedure Division is optional, but a program without a Procedure Division does nothing except initialize data.

The Procedure Division is composed of statements, which specify the actions to be taken by the program. The program does its work by executing the statements in their appropriate sequence. The way you organize the statements not only governs the order in which they execute but also can contribute to your program's readability and maintainability.

You can organize statements into sentences, sentences into paragraphs, and sentences and/or paragraphs into sections. A paragraph, a group of successive paragraphs, a section, or a group of successive sections can be executed as a unit called a procedure.

Figure 9-1 Relationship of Statements, Sentences, Paragraphs, and Sections

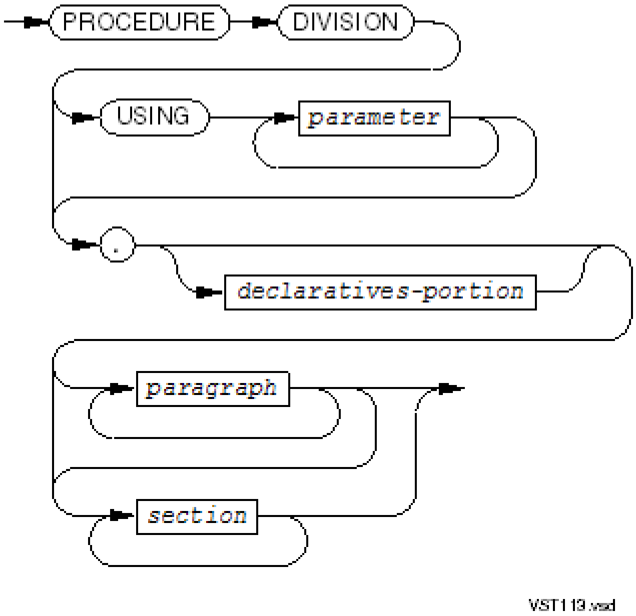


The Procedure Division can optionally include a Declaratives Portion. The Declaratives Portion is a set of sections at the beginning of the Procedure Division. These sections are executed if specified conditions arise during the execution of statements outside the Declaratives Portion.

Topics:

- Procedure Division Components and Syntax
- References to Data Items
- Common Semantic Rules
- Common Phrases
- Input-Output
- Arithmetic Operations
- Conditional Expressions
- Concatenation Expressions

Procedure Division Components and Syntax



USING

marks the beginning of a parameter list in a called program. The CALL statement in the calling program must contain a corresponding USING phrase.

parameter

is the identifier of a data item in the Linkage Section. See Linkage Section (page 191) and CALL (page 303).

declaratives-portion

is described in Declaratives Portion.

section

is described in Sections

Example 9-1 Procedure Division

```
PROCEDURE DIVISION.
DRIVER SECTION.
DRIVE.
   PERFORM 100-INITIALIZATION
   PERFORM 200-PROCESS-REQUESTS UNTIL JOB-IS-DONE
   PERFORM 300-TERMINATION.
100-INITIALIZATION SECTION.
OPEN-FILES.
   OPEN INPUT MESSAGE-IN
   OPEN OUTPUT MESSAGE-OUT
   OPEN INPUT BASE-FILE.
FINISH-UP-INIT.
   MOVE "READY" TO WATCH-WORD.
    ...
200-PROCESS-REQUESTS SECTION.
READ-DOLLAR-RECEIVE.
   READ MESSAGE-IN
    ...PROCESS-INPUT-DATA.
    ...
300-TERMINATION SECTION.
CLOSE-AND-QUIT.
   CLOSE MESSAGE-IN
         MESSAGE-OUT
         BASE-FILE
   STOP RUN.
```

Topics:

- Statements
- Sentences
- Paragraphs
- Sections
- Procedures
- Declaratives Portion

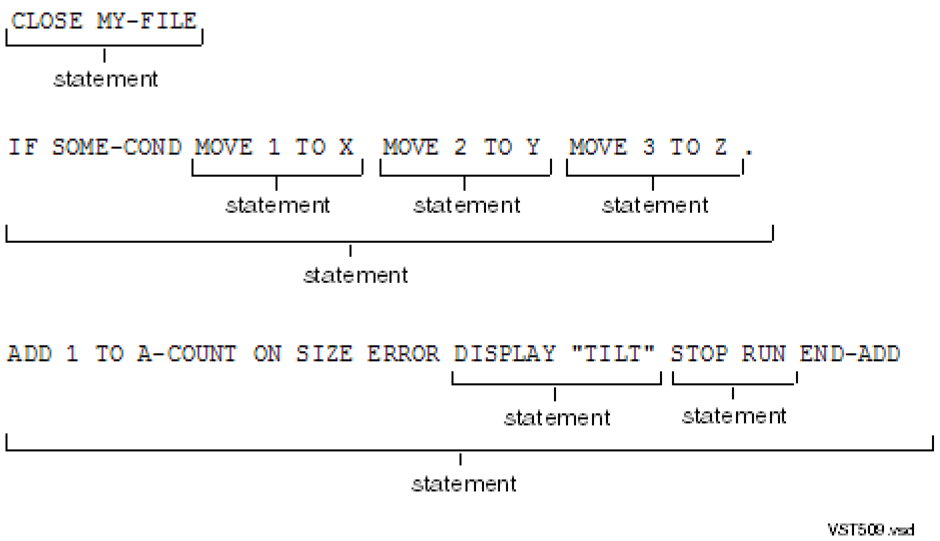
Statements

A statement is a syntactically valid combination of words and symbols beginning with a COBOL verb. Where a statement ends depends on its context.

Table 9-1 Where Statements End

Statement is ...	Statement ends ...
in a sequence of statements and is not the last one	immediately before the verb of the next statement
inside another statement	either immediately before the keyword that begins the next portion of a containing statement or at the period separator that terminates its containing sentence
isolated or the last statement in a sequence	See Scope of Statements.

Figure 9-2 Statement Examples



```
CLOSE MY-FILE

IF SOME-COND MOVE 1 TO X  MOVE 2 TO Y  MOVE 3 TO Z .

ADD 1 TO A-COUNT ON SIZE ERROR DISPLAY "TILT" STOP RUN END-ADD
```

Table 9-2 Statement Types

Statement Type	Definition
Imperative	Specifies an unconditional action for the process to take
Conditional	Specifies that the truth value of a condition is to be determined and that the subsequent action of the run unit depends on this truth value
Delimited-scope	Terminates in its explicit scope terminator
Compiler-directing	Causes the compiler to take some specific action during compilation

Topics:

- Imperative Statement
- Conditional Statement
- Delimited-Scope Statement
- Scope of Statements

Imperative Statement

An imperative statement specifies an unconditional action for the process to take. Each imperative statement begins with an imperative verb.

Table 9-3 Imperative Verbs

ACCEPT
ADD[2]
ALLOCATE[4]
ALTER
CALL[6]
CANCEL
CHECKPOINT[3]
CLOSE
DELETE[1]
DISPLAY
DIVIDE[2]
ENTER[5]
EXIT[7]
FREE[4]
GO TO
INITIALIZE
MERGE
MOVE
MULTIPLY[2]
OPEN
PERFORM
READ[8]
RELEASE
REWRITE[1]
START[1]
STARTBACKUP[3]
STOP
STRING[6]
SUBTRACT[2]
UNLOCKFILE[3]
UNLOCKRECORD[3]
UNSTRING[6]

Table 9-3 Imperative Verbs *(continued)*

COMPUTE[2]	INSPECT	SET	WRITE[9]
CONTINUE	LOCKFILE[3]	SORT	

1 Without INVALID KEY or NOT INVALID KEY phrase or else with a scope terminator

2 Without SIZE ERROR or NOT SIZE ERROR phrase or else with a scope terminator

3 HP extension

4 ALLOCATE and FREE are recognized as verbs only when the STANDARD 2002 directive is in effect.

5 In HP COBOL, analogous to CALL but used only to call non-COBOL routines

6 Without OVERFLOW, EXCEPTION, or NOT EXCEPTION phrase or else with a scope terminator

7 Limited to appearing alone in a paragraph conditional statement. This is because it can include phrases that are or are not executed depending upon the value of a a condition (for example, the delimited-scope IF statement) or upon the occurrence of an exception (for example, the delimited-scope READ statement).

8 Without AT END, NOT AT END, INVALID KEY, or NOT INVALID KEY phrase or else with a scope terminator

9 Without INVALID KEY, NOT INVALID KEY, END-OF-PAGE, or NOT END-OF-PAGE phrase or else with a scope terminator

An individual delimited-scope statement or a sequence of two or more imperative statements is considered to be an imperative statement under the rules for statement formation; therefore, when imperative-statement appears in a statement, it refers to one or more consecutive imperative statements and/or delimited-scope statements.

Example 9-2 Imperative Statement

```
MOVE "Birnham Wood" TO DUNSINANE
```

Conditional Statement

A conditional statement specifies that the truth value of a condition is to be determined and that the subsequent action of the run unit depends on this truth value.

Any of these statements is a conditional statement (unless it ends with an explicit scope delimiter):

- An EVALUATE, IF, RETURN, or SEARCH statement
- A READ statement with the AT END or INVALID KEY phrase
- A WRITE statement with the INVALID KEY or END-OF-PAGE phrase
- A DELETE, REWRITE, or START statement with the INVALID KEY phrase
- An arithmetic statement (ADD, COMPUTE, DIVIDE, MULTIPLY, or SUBTRACT) with the SIZE ERROR phrase
- A STRING or UNSTRING statement with the OVERFLOW phrase
- A CALL statement with the EXCEPTION phrase (or the OVERFLOW phrase, which is obsolete for the CALL statement)

The NOT AT END, NOT INVALID KEY, NOT END-OF-PAGE, NOT SIZE ERROR, and NOT EXCEPTION phrases also make their parent statement a conditional statement. There is no NOT OVERFLOW phrase.

Any conditional statement can be preceded by an imperative statement or any sequence of statements specified as equivalent to an imperative statement by the rules given in Imperative Statement.

Unless it is contained within an IF statement, a conditional statement must be the last or only statement in a sentence; however, any statements listed previously can be written as a delimited-scope statement. For clarity and convenience, you are encouraged to avoid conditional statements entirely and use equivalent delimited-scope statements instead.

In Example 9-3, the period ends both the IF and the AT END phrase. If there were an operation that the program had to perform after the read operation succeeded, but only when DONE-WITH-MASTER was false, you would have to package the READ statement in a separate paragraph. Then you could end the AT END phrase with a period that did not also end the IF statement.

Example 9-3 Conditional Statement

```
IF NOT DONE-WITH-MASTER
   READ MASTER-FILE
        AT END MOVE DONE-VALUE TO MASTER-FLAG.
```

Compare Example 9-3 to Example 9-4.

Delimited-Scope Statement

A delimited-scope statement is any statement that terminates in its explicit scope terminator. An explicit scope terminator is a reserved word used to delimit the scope of a conditional statement or in-line PERFORM statement.

Table 9-4 Explicit Scope Terminators

END-ADD	END-EVALUATE	END-RETURN	END-SUBTRACT
END-CALL	END-IF	END-REWRITE	END-UNSTRING
END-COMPUTE	END-MULTIPLY	END-SEARCH	END-WRITE
END-DELETE	END-PERFORM	END-START	END-UNSTRING
END-DIVIDE	END-READ	END-STRING	

The form of each explicit scope terminator includes the verb from the statement that it terminates. Explicit scope terminators can appear only as specified in the general formats for statements. When the appropriate explicit scope terminator follows a conditional statement, it is considered to be a delimited-scope statement instead.

In Example 9-4, if there were no END-READ scope terminator, the PERFORM would be part of the AT END phrase of the (conditional) READ statement.

Example 9-4 Delimited-Scope Statement

```
IF NOT DONE-WITH-MASTER
   READ MASTER-FILE
        AT END MOVE DONE-VALUE TO MASTER-FLAG
   END-READ
   PERFORM PROCESS-MASTER-RECORD
     UNTIL DONE-WITH-MASTER
END-IF
```

Under the rules for statement execution, a delimited-scope statement is handled in the same manner as a conditional statement. This is because it can include phrases that are or are not executed depending upon the value of a condition (for example, the delimited-scope IF statement) or upon the occurrence of an exception (for example, the delimited-scope READ statement).

Compiler-Directing Statement

A compiler-directing statement is a COPY, REPLACE, or USE statement. It causes the compiler to take some specific action during compilation. A sentence that contains a compiler-directing statement cannot contain any other statements.

NOTE: In some implementations of COBOL, the ENTER verb marks the beginning of an embedded routine in some other language. For this reason, ENTER is classified as a compiler-directing verb in those implementations. The ENTER statement as implemented by HP COBOL is similar to a CALL statement but calls a routine compiled from a language other than COBOL.

The COPY statement directs the compiler to include additional source text at that point in the program.

The REPLACE statement directs the compiler to replace source program text.

The USE statement directs the compiler to include logic in the object program that calls the designated declarative section if the condition described in the USE statement arises during execution.

Example 9-5 Compiler-Directing Statement

```
COPY NEIGHBORS-ANSWER
```

Scope of Statements

When the formation rules for one statement cause it to contain other statements, you must show the compiler the scope of both the contained and the containing statements by using scope terminators. Scope terminators either explicit or implicit.

Explicit scope terminators are defined and listed under Delimited-Scope Statement.

Implicit scope terminators occur at the end of a sentence and at the end of a contained statement.

- At the end of any sentence, the separator period that terminates the scope of all previous statements not yet terminated is an implicit terminator.
- Within any statement containing another statement, the next phrase of the containing statement following the contained statement is the implicit terminator of the scope of any unterminated contained statement.

The detailed rules for scope termination are:

- A conditional statement is composed of other statements. When no delimited-scope statements occur in the conditional statement, the next phrase of the conditional statement terminates the contained statement. In this example, the ELSE phrase implicitly terminates the statement of the THEN phrase and the separator period implicitly terminates the ELSE phrase:

  ```
  IF CREDIT
  THEN  PERFORM POST-CREDIT
  ELSE  PERFORM POST-DEBIT.
  ```

- When the last statement in a sentence is a conditional statement, the period separator that terminates the sentence implicitly terminates the conditional statement. In this example, the paragraph BOTTOM-LINE consists of a single sentence. The period separator that ends the sentence ends the IF statement:

  ```
  BOTTOM-LINE.
     MOVE NET-SALES TO NET-S OF SALES-REC-DISPLAY
     MOVE NET-PROFIT TO NET-P OF SALES-REC-DISPLAY
     IF NET-PROFIT < 0
        DISPLAY "Get Cracking!" UPON DIST-MGR-TUBE.
  ```

- When statements are contained within other statements, a period separator that terminates the sentence also implicitly terminates the scope of any contained statements not yet terminated otherwise. In this example, the period separator terminates both the READ and the IF statements:

```
IF AUXILIARY-INPUT-TAPE
   READ AUX-TAPE
      AT END      PERFORM AUX-TAPE-ENDED
      NOT AT END  PERFORM PROCESS-AUX-REC.
```

- A conditional statement can be nested within an IF statement, but not within any other form of conditional statement. The preceding example shows a conditional READ statement nested within an IF statement. It is not possible, for example, to have a READ statement with an AT END phrase and no END-READ terminator (making it a conditional statement) within another such READ statement's AT END phrase. Both must be provided with scope terminators as shown in this example:

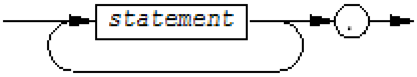
```
READ PRIMARY-FILE
  AT END READ SECONDARY-FILE
           AT END DISPLAY "End of second file"
         END-READ
END-READ
```

- When a delimited-scope statement is contained within another delimited-scope statement with the same verb, each explicit scope terminator terminates the statement begun by the most recently preceding, and as yet unpaired, occurrence of that verb. In this example, the first END-IF ends the second IF statement:

```
IF A NOT > B
   PERFORM PHASE-1
   IF A = B
      PERFORM PHASE-1X
   ELSE
      PERFORM PHASE-1Y
   END-IF
   PERFORM PHASE-2
ELSE
   PERFORM PHASE-5
END-IF
```

Sentences

A sentence is one or more statements terminated by a period separator.



VST114.vsd

statement

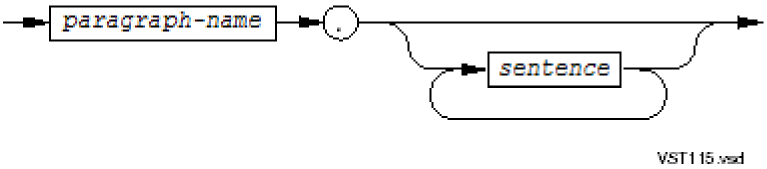
is described in Statements.

A sentence that contains a compiler-directing statement cannot contain any other statements. When the compiler acts upon the COPY statement, that statement and its terminating period are logically replaced by whatever text the statement specified. The COPY sentence can therefore appear to be within another sentence, but once the compiler acts upon the COPY statement, the COPY sentence itself is replaced by text from a COPY library and the period separator has disappeared.

When you use the delimited-scope statements of COBOL, you can write an entire paragraph with only one required period—the one at the end of the paragraph.

Paragraphs

A paragraph groups related sentences and statements together and identifies them by one name. Using the name, GO TO, PERFORM, SORT and MERGE statements can transfer control to the paragraph.



paragraph-name

is either a COBOL word consisting of up to 30 alphanumeric characters or an integer of up to 30 digits.

sentence

is described in Sentences. A sentence ends with a period; therefore, a paragraph ends with the period at the end of its last sentence.

Example 9-6 Paragraph With One Sentence

```
CHK-REPORT-YY.
   IF CURRENT-YY IS LESS THAN 0
   OR GREATER THAN 99
      DISPLAY "REPORT YEAR IS NOT BETWEEN 00 AND 99, "
         "REENTER YEAR"
      ACCEPT CURRENT-YY
      GO TO CHK-REPORT-YY.
```

Example 9-7 Paragraph With Several Sentences

```
CONVERT-REPORT-DATE-TO-SERIAL-DAY.
   MOVE CURRENT-YY TO REPORT-SERIAL-YEAR.
   MOVE 0 TO DIVIDE-RESULT
             LEAP-YEAR.
   DIVIDE REPORT-SERIAL-YEAR BY 4
      GIVING DIVIDE-RESULT
      REMAINDER LEAP-YEAR.
   IF LEAP-YEAR EQUAL TO 0
      MOVE 1 TO LEAP-YEAR
   ELSE
      MOVE 0 TO LEAP-YEAR.
   ADD CURRENT-DD
          DAYS-TO-DATE(CURRENT-MM)
          GIVING REPORT-SERIAL-DAYS.
   IF REPORT-SERIAL-DAYS IS GREATER THAN 59
      ADD LEAP-YEAR TO REPORT-SERIAL-DAYS.
```

Usage Considerations:

- Paragraph Headers Not Restricted to Area A

  The HP COBOL compilers accept paragraph headers that do not begin in area A; the compiler recognizes a statement by its initial verb. When the compiler recognizes the end of a sentence (by detecting the terminating period) and is prepared to accept another sentence or a paragraph, it accepts any legal paragraph header or section header whether it begins in area A or after it.

- End of a Paragraph

  A paragraph ends immediately before the next paragraph header or section header, at the end of the Procedure Division, or when it is the last paragraph in the Declaratives Portion of the Procedure Division, at the keywords END DECLARATIVES.

- Multiple Paragraph Headers (Null Paragraphs)

  A paragraph header can be followed immediately by another paragraph header:

```
CHECK-THE-INPUT.
GET-THE-FIRST-RECORD.
    READ IN-FILE ...
```

In this case, CHECK-THE-INPUT is a null paragraph, so

```
GO TO CHECK-THE-INPUT
```

is equivalent to

```
GO TO GET-THE-FIRST-RECORD
```

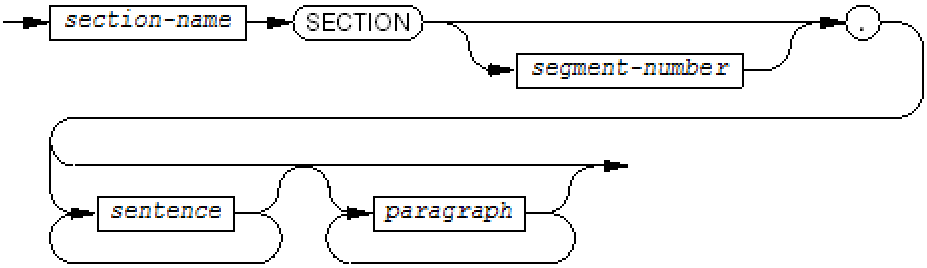
but PERFORM CHECK-THE-INPUT returns control immediately to the statement following the PERFORM.

- Paragraph Form Only for Use With the ALTER Statement

  There is a simpler form of the paragraph used in conjunction with an ALTER statement. For details, see ALTER (page 302).

Sections

A section groups related sentences and paragraphs together and identifies them by one name. Using the name, GO TO, PERFORM, SORT and MERGE statements can transfer control to the section.



VST116.vsd

section-name

is either a COBOL word consisting of up to 30 alphanumeric characters or an integer of up to 30 digits.

segment-number

is a numeric literal of 1 or 2 digits.

sentence

is described in Sentences.

paragraph

is described in Paragraphs.

Usage Considerations:

- Section Format
  - Beginning

    After the period separator that precedes the first paragraph of a section, do not put anything on the same text line except space characters or a USE statement.
  - End

    A section ends at the next section header, at the physical end of the Procedure Division, or at the keywords END DECLARATIVES.
- Standard COBOL Format

  To conform to the COBOL standard, observe these rules. (HP COBOL compilers do not require that you follow these rules.)
  - If you use sections, put all paragraphs in sections

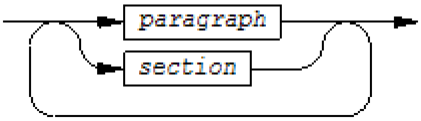
    If the Procedure Division contains sections, put every paragraph in a section. (This means that if the Procedure Division contains a Declaratives Portion, which always contains at least one section, then each paragraph in the Procedure Division must be in a section.)
  - Start each section on a separate line and follow section header immediately by paragraph header

    Start each section on a separate line, with the `section-name` beginning in area A and nothing but space characters preceding it.
- Independent Segments

  An independent segment is a section whose segment-number is greater than 49. Independent segments are relevant only to the ALTER statement, which you are advised not to use because the 1985 COBOL standard classifies it as an obsolete element. (Use a MOVE statement and a conditional GO TO statement instead.)

Procedures

A procedure is a paragraph, a group of successive paragraphs, a section, or a group of successive sections executed as a unit under the control of a PERFORM statement.



VST415.vsd

paragraph

is described in Paragraphs.

section

is described in Sections.

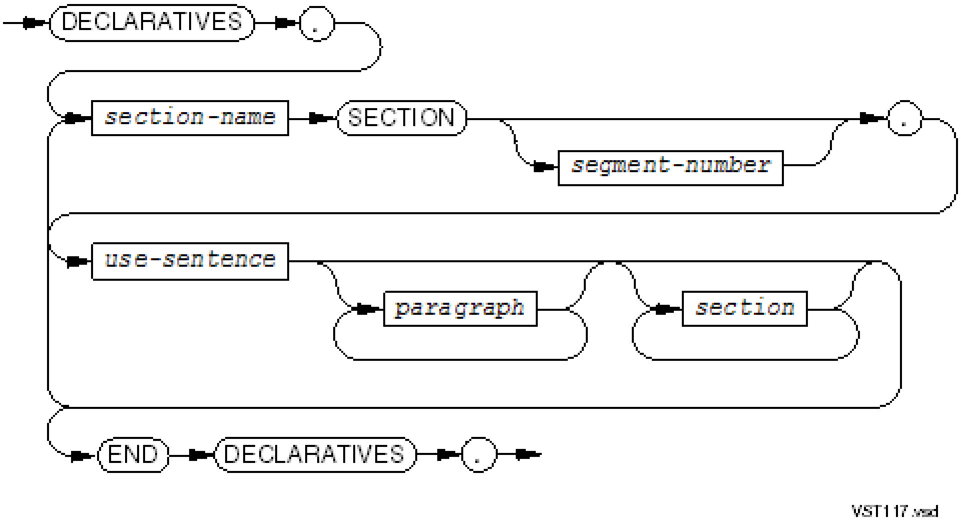
There is a distinct difference between procedure-name and procedure. A procedure-name refers to a paragraph or section in the source program. A procedure-name is either a section-name or a paragraph-name (which can be qualified by a section-name). A procedure is a paragraph, a group of successive paragraphs, a section, or a group of successive sections to be executed as a unit.

The term procedure can also refer to the set of paragraphs or sections executed under control of a PERFORM *procedure-name* or a PERFORM *procedure-name* THROUGH *procedure-name* statement. When the term is used in this dynamic sense, one procedure can contain or overlap another procedure. The rules of COBOL do not require that the relationships between members of this set of be obvious; that is, the set of paragraphs executed by a PERFORM statement is determined by control flow, not solely by the order of the paragraphs in the source file.

Declaratives Portion

The optional Declaratives Portion is one or more sections at the beginning of the Procedure Division, bracketed by the keyword DECLARATIVES and the keywords END DECLARATIVES, that are executed individually when certain conditions arise during execution of statements in the rest of the Procedure Division.

The Declaratives Portion is reserved for debugging routines and input-output error routines specified by USE statements. When used, this area must be coded immediately after the Procedure Division heading.



section-name, *segment-number*

are described in Sections.

use-sentence

is a sentence containing only a USE statement.

paragraph

is described in Paragraphs.

section

is described in Sections.

Example 9-8 Procedure Division with a Declaratives Portion

```
PROCEDURE DIVISION.
DECLARATIVES.
 IN-FILES-USE SECTION.
   USE AFTER STANDARD ERROR PROCEDURE ON IN-FILE.
 IN-FILES-PARA.
   ...
END DECLARATIVES.
BEGIN-MY-PROGRAM SECTION.
   ...
```

Execution of the Procedure Division

Execution of a run unit begins with the first procedure (that is, section or paragraph) of the Procedure Division, excluding any declarative sections. Except where specific rules indicate otherwise, sections are executed in the order in which they appear within the source program.

Similarly, paragraphs are executed in the order in which they appear within their sections, sentences are executed in the order in which they appear within their paragraphs, and statements are executed in the order in which they appear within their sentences.

Topics:

- Statement Execution
- Sentence Execution
- Paragraph Execution
- Section Execution
- Procedure Execution
- Declaratives Portion Execution

Statement Execution

An individual imperative statement is always executed in its entirety. If the statement is followed within its sentence by another statement, control then passes to that statement; otherwise, control passes to the next executable sentence.

A conditional statement or a delimited-scope statement is not necessarily executed in its entirety. Instead, the truth value of the condition in the statement causes the object program to select between alternate paths of control. Because the specific possibilities depend upon the particular statement, they are discussed along with the individual statement descriptions.

The compiler-directing statements COPY, REPLACE, and USE do not participate in execution; therefore, control is never transferred to or from these statements.

Sentence Execution

A sentence containing only imperative statements is always executed in its entirety. Normally, control then passes to the next sentence in the paragraph that contains an imperative or conditional statement. Exceptions to this rule are:

- When the last statement in the sentence is a GO TO statement, control is transferred unconditionally to the specified paragraph or section.
- When the last statement in the sentence is a STOP RUN statement, execution of the run unit, and thus the program, terminates.
- When the last statement in the sentence is an EXIT PROGRAM statement, control is transferred in accordance with the rules for that statement.

If the sentence is not followed within its paragraph by any other sentence that contains an imperative, conditional, or delimited-scope statement, and none of the preceding cases applies, then control passes to the first executable sentence in another paragraph. The paragraph to which control passes is determined by the rules described under Paragraph Execution.

A sentence containing a conditional or delimited-scope statement is not necessarily executed in its entirety. After executing any initial imperative statements, the execution of the conditional or delimited-scope statement causes the object program to select the appropriate path of control.

Control is never transferred to or from sentences that consist of a compiler-directing statement.

Paragraph Execution

Execution of a paragraph begins with the execution of the first sentence containing an imperative, conditional, or delimited-scope statement. Except for the nested execution of other procedures or programs (for example, by the execution of PERFORM or CALL statements) control remains within the paragraph until either:

- The execution of a GO TO, STOP RUN, or EXIT PROGRAM statement transfers control out of the paragraph.
- The final executable sentence of the paragraph completes without explicitly transferring control to some other procedure. In this case, control passes to the next paragraph of the same section if one exists or to the next section in the source program, except in these situations:
  1. The rules for the implicit transfer of control among procedures, described later, can cause control to revert to some other paragraph or section instead.
  2. If no such reversion occurs, and no next section exists, execution of the program terminates.

During the execution of a paragraph, control passes to successive sentences unless this order is modified by a conditional or delimited-scope statement or a GO TO, STOP RUN, CALL, ENTER, or EXIT PROGRAM statement. Execution of a paragraph that does not have any sentences involves only the passing of control described for item 2.

Section Execution

Execution of a section normally consists of serial execution of its paragraphs; however, this order can be modified by explicit transfer statements (for example, a GO TO statement) and the rules for implicit transfer of control. Execution of the section terminates when either:

1. The execution of a GO TO, STOP RUN, or EXIT PROGRAM statement transfers control out of the section.
2. The final paragraph of the section completes without explicitly transferring control to some other section. In this case, control passes to the next section of the source program, except in these situations:
   - The rules for the implicit transfer of control among procedures, described later, can cause control to revert to some other section instead.
   - If no such reversion occurs, and the section is the last one in the source program, the execution of the program terminates.

Execution of a section that does not have any paragraphs involves only the passing of control described for FIX_THIS_LINK.

Implicit Transfer of Control

The mechanism that transfers control from statement to statement in the sequence in which they appear in the source program is an implicit transfer of control. This mechanism applies unless it is overridden by an explicit transfer of control or the absence of a next executable statement to which control can be passed.

Explicit Transfer of Control

An explicit transfer of control consists of execution of one of these statements:

- Conditional
- CALL
- ENTER
- Delimited-scope statement
- EXIT PROGRAM (when executed in a called program)
- GO TO
- PERFORM

COBOL provides these types of implicit transfers of control that override the statement-to-statement mechanism:

1. When a paragraph is being executed under control of another COBOL statement (for example, MERGE, PERFORM, SORT, and USE), and it is the last paragraph in the range of the controlling statement, an implied transfer of control occurs from the last statement in the paragraph to the control mechanism of the controlling statement. If several controlling statements are active, the transfer is to the last statement executed. Furthermore, if a paragraph is being executed under the control of a PERFORM statement that causes iterative execution and that paragraph is the first one in the range of the PERFORM statement, an implicit transfer of control occurs between the control mechanism associated with the PERFORM statement and the first executable statement in the paragraph for each iterative execution of the paragraph.
2. When a SORT or MERGE statement is executed, an implicit transfer of control occurs to any associated input or output procedures. Another implicit transfer of control occurs after the execution of such a procedure, as described in item 1.
3. When the execution of any COBOL statement causes the condition described in the USE statement of a declarative section, an implicit transfer of control occurs to that section. Another implicit transfer of control occurs after the execution of the declarative section, as described in item 1.

The statement-to-statement transfers of control ignore the existence of COPY, REPLACE, and USE statements. Although control never passes to a USE statement itself, the existence of a USE statement generates the control mechanism for implicit transfers of control to and from the section in which it appears.

The term next executable statement refers to the next COBOL statement to which a process is to transfer control according to the rules given previously and the rules associated with each language element in the Procedure Division. There is no next executable statement in these situations:

1. The execution of the last statement in a program does not cause an explicit transfer of control, and the paragraph in which it appears is not being executed under the control of some other COBOL statement. Following the execution of such a statement, execution of the program terminates. When it is a called program, control reverts to the calling program as if an EXIT PROGRAM statement were executed; otherwise, execution of the run unit terminates as if a STOP RUN statement were executed.
2. The program contains no Procedure Division. Execution of such a program proceeds as if it contained a Procedure Division with a single paragraph consisting of a CONTINUE statement. Control then passes from the program as described in item 1.
3. An EXIT PROGRAM statement is executed within a called program. In this case, control reverts to the calling program.
4. A STOP RUN statement is executed within any program. In this case, execution of the run unit terminates.

5. The last statement in the Declaratives Portion of a program completes execution without causing an explicit transfer of control, and its paragraph is not the designated end of a PERFORM … THROUGH procedure group. Following the execution of such a statement in these circumstances, control reaches the end of the Declaratives Portion, causing execution of both the program and the run unit to terminate abnormally.

   Both the program and the run unit terminate abnormally because execution of a declarative section is expected to be done by a PERFORM statement with these characteristics:

   - An explicit PERFORM statement in the Declaratives Portion or somewhere in the rest of the Procedure Division
   - An implicit PERFORM statement executed due to either the use of the COBOL debugging module or an input-output statement's encountering an error

   In the latter case, if the error was recoverable, control returns to the statement following the input-output statement; if the error was not recoverable, execution of the run unit terminates.

   If you transfer control directly to a procedure in a declarative section from a GO TO statement anywhere in the Procedure Division, there is no next executable statement when execution reaches the end of the declarative section.

Procedure Execution

A procedure is a paragraph, a group of successive paragraphs, a section, or a group of successive sections executed as a unit. See Paragraph Execution and Section Execution.

Declaratives Portion Execution

The sections of the Declaratives Portion are executed individually when certain conditions arise during execution of statements in the rest of the Procedure Division (see Section Execution).

Common Semantic Rules

The semantic rules about operand identification and overlapping operands apply to several statements. Their explanations use these terms:

Term	Definition
Sending data item	A data item whose value is to be used in an operation
Receiving data item	A data item to which the result of an operation is to be assigned
Intermediate data item	A conceptual signed numeric data item used as a temporary repository for the result being developed during the execution of an arithmetic operation

Operand Identification

An operand in a statement is either an identifier (which specifies a data item directly) or a condition-name associated with a conditional variable (which specifies a data item indirectly). In either case, the particular data item must be identified before the operand can be used in executing the statement. Operand identification proceeds in this order:

1. Qualifiers

   If an operand contains qualifiers, the compiler uses them to determine the correct interpretation of the operand name.

2. Subscripts

   If an operand contains subscripts, the run-time routines evaluate them from left to right.

3. Size

   If an operand has variable size, the run-time routines determine its appropriate size. The appropriate size is usually the operand's current actual size, but if the operand is a receiving item, its appropriate size is sometimes its maximum size. For details, see OCCURS Clause for Variable-Size Tables (page 224).

4. Reference Modifier

   If the operand contains a reference modifier, the run-time routines evaluate the reference modifier.

Unless the rules for a statement state otherwise, identification of each operand in a statement occurs exactly once as the first operation (or series of operations) in the execution of that statement. The identification of an operand only determines which data item is specified, it does not evaluate or affect the content of the data item.

Overlapping Operands

When a sending and a receiving data item in a statement overlap—that is, when they share part or all of their storage areas but are not defined by the same data description entry—the result of the statement is undefined. Results are also undefined for some statements in which sending and receiving data items are defined by the same data description entry. Such cases are noted in the descriptions of those statements.

Common Phrases

These phrases are common to several different statements:

- CORRESPONDING Phrase
- ROUNDED Phrase
- SIZE ERROR Phrase
- FROM Phrase
- INTO Phrase

CORRESPONDING Phrase

For the purposes of this discussion, *d1* and *d2* are identifiers in a statement containing a CORRESPONDING phrase and:

- Both *d1* and *d2* designate group (but not level 66) data items.
- The data description entries of *d1* and *d2* do not contain USAGE INDEX clauses.
- The data description entries of one or both of *d1* and *d2* can include REDEFINES or OCCURS clauses or be subordinate to items whose data description entries include these clauses.

A pair of data items, one subordinate to *d1* and one subordinate to *d2*, correspond if they follow these rules:

- Both data items have the same data-name and the same potential set of qualifiers up to, but not including, *d1* and *d2*. Neither data item has the data-name FILLER.
- The data description entries of the items do not contain REDEFINES, RENAMES, OCCURS, or USAGE INDEX clauses.
- In a MOVE statement, at least one of the data items is an elementary data item, and it is legal to move the sending item to the receiving item.
- In an ADD or SUBTRACT statement, both items are elementary numeric data items.

If a data item does not qualify as a corresponding item, then none of its subordinates qualify either.

In Example 9-9, assume that the records A and H obey all of the preceding rules. The statement

```
MOVE CORRESPONDING A TO H
```

is equivalent to this series of MOVE statements:

```
MOVE  B OF A        TO    B OF H
MOVE  G OF F OF A   TO    G OF F OF H
```

Example 9-9 CORRESPONDING Phrase

```
01  A                   01  H
    03  B                   05  F
        07  C                   08  G
        07  D               05  B
    03  E                   05  C
    03  F                       09  E
        04  G                   09  D
                                09  G
```

ROUNDED Phrase

When the number of fraction digits in the result of an arithmetic operation exceeds the number of fraction digits in the receiving item, the excess digits must be deleted by either truncation or rounding. Also, when one or more low-order integer positions in a receiving item are represented by *P* in the item's PICTURE character-string, equivalent digits in a result value must be deleted by truncation or rounding.

When the ROUNDED phrase is not specified, excess digits are deleted by truncation; that is, the digits are simply discarded. If rounding is specified, the absolute value of the retained portion of the value is increased by 1 whenever the most significant digit of the excess is greater than or equal to 5.

In these examples, the caret (^) represents the assumed decimal point:

Actual Result	PICTURE Character-String of Receiving Item	Rounded Value
5^71489	9V999	5^715
7650^	99PP	77
3^141592654	9V99	3^14

SIZE ERROR Phrase

The size error condition indicates a problem in computation, such as a loss of precision. The SIZE ERROR and NOT SIZE ERROR phrases of the ADD, COMPUTE, DIVIDE, MULTIPLY, and SUBTRACT statements enable you to test whether the arithmetic operation caused the size error condition to arise. You cannot test for the size error condition with an IF statement.

The situations that cause the size error condition are:

- Exponentiation

  If the program attempts to raise zero to the zero power, or if no real number exists as the result of the exponentiation, the arithmetic operation is terminated and the size error condition arises.

- Division

  If the program attempts to divide any value by zero, the arithmetic operation is terminated and the size error condition arises.

- Overflow

  If, after decimal point alignment, the absolute value of a result exceeds the largest value that can be contained in the receiving data item, a size error condition arises.

The size error condition applies to intermediate and final results. If the ROUNDED phrase is specified, rounding occurs before checking for a size error. When a size error condition occurs, the subsequent action depends on whether the SIZE ERROR and NOT SIZE ERROR phrases are specified.

Usage Considerations:

- SIZE ERROR Phrase and TRAP2 Directive

  NOTE: The ECOBOL compiler, which has traps set by default, ignores the TRAP2 directive and issues a warning.

  The SIZE ERROR phrase catches size errors in COMPUTATIONAL arithmetic that the TRAP2 directive (which is the default) does not; however, the SIZE ERROR phrase generates more code than the TRAP2 directive does.

  In Example 9-10, C=A+B causes a size error that the TRAP2 directive does not catch, but that the SIZE ERROR phrase does catch.

  Example 9-10 SIZE ERROR Phrase and TRAP2 Directive

```
 WORKING-STORAGE SECTION.
   77 A PIC 99 COMP VALUE 99.
   77 B PIC 99 COMP VALUE 88.
   77 C PIC 99 COMP.
   77 D PIC 99 COMP.
 PROCEDURE DIVISION.
   ...
* TRAP2 directive does not catch this size error:
   COMPUTE C = A + B.
   COMPUTE D = C - B.
   DISPLAY D.
* ON SIZE ERROR phrase catches this size error:
   COMPUTE C = A + B
     ON SIZE ERROR DISPLAY "Too Big"
   NOT ON SIZE ERROR DISPLAY "Acceptable".
   ...
```

- COMPUTATIONAL Data Items

  If an arithmetic computation stores a value in a COMPUTATIONAL data item, and the decimal representation of that value exceeds the number of decimal places in the item's PICTURE clause (as in C = A + B in the preceding example), a later attempt to use that value (as in D = C - B in the preceding example) can cause arithmetic overflow.

  HP COBOL can retrieve a number larger than 9,999 from a PICTURE S9(4) COMPUTATIONAL field (even though the storage unit is 2 bytes and can therefore accommodate values from -32,767 through +32,767), but arithmetic overflow is likely.

  The largest value HP COBOL can use in a 4-byte COMPUTATIONAL item is 999,999,999 (not 2,147,483,647).

  The largest value HP COBOL can use in an 8-byte COMPUTATIONAL item is 999,999,999,999,999,999 (not 9,223,372,036,854,775,807).

  If you need the full capacity of 2-byte, 4-byte, and 8-byte storage, describe the item as USAGE NATIVE-2, NATIVE-4, or NATIVE-8 (with no PICTURE) or COMPUTATIONAL-5.

- SIZE ERROR and Multiple Results

  When a statement includes more than one receiving item, the decision to assign a result value or cause a size error condition is determined independently for each such item; therefore, if the size error condition does not occur for a particular receiving item, that item is assigned its correct result value even when the size error condition exists for one or more other receiving items of the same statement (unless the run terminates abnormally because of an arithmetic overflow).

- Values Undefined When SIZE ERROR Phrase is Absent

  When a size error condition occurs during execution of a statement for which the SIZE ERROR phrase is not specified, a trap 2 (arithmetic overflow) occurs unless a NOTRAP2 directive has suspended arithmetic overflow trapping. If a NOTRAP2 directive has suspended arithmetic overflow trapping, results are undefined.

NOTE: The ECOBOL compiler, which has traps set by default, ignores the NOTRAP2 directive and issues a warning.

Use the NOTRAP2 directive only during the process of conversion from COBOL 74 to HP COBOL. It is provided to enable programs that do not include SIZE ERROR phrases to be compiled and executed in HP COBOL without investing programmer time in analyzing the potential for overflow problems. The availability of this directive might be discontinued in HP COBOL after a period of time.

- Values Remain Untouched When SIZE ERROR Phrase is Present

  When a size error condition occurs during execution of an arithmetic statement for which the SIZE ERROR phrase is specified, the value of the affected receiving data items is not altered. That is, these data items retain whatever values they held before the size error condition arose. Unaffected data items receive new values as expected. The program has no mechanism by which it can determine which data items were the cause of the size error condition.

- SIZE ERROR, NOT SIZE ERROR, and Transfer of Control

  After completion of the execution of the arithmetic operations, if the size error condition occurred, the imperative statement in the SIZE ERROR phrase is executed. If execution of the imperative statement causes explicit transfer of control (due to a procedure branching or conditional statement) control is transferred according to the rules for that statement; otherwise, when the imperative statement has been executed, control passes to the end of the arithmetic statement and the NOT SIZE ERROR phrase (if any) is ignored.

  When no size error condition occurs during the execution of an arithmetic statement, the SIZE ERROR phrase (if any) is ignored, and control passes to the NOT SIZE ERROR phrase (if one is present) or to the end of the arithmetic statement. If execution of the imperative statement causes explicit transfer of control (due to a procedure branching or conditional statement), control is transferred according to the rules for that statement.

- CORRESPONDING Phrase and the Size Error Condition

  When the CORRESPONDING phrase appears in an ADD or SUBTRACT statement and any of the individual operations produces a size error condition, the imperative statement in the SIZE ERROR phrase is not executed until all the individual additions or subtractions are completed.

FROM Phrase

The FROM phrase is an optional component of the RELEASE, REWRITE, and WRITE statements, all of which specify a record-name as their primary operand. The identifier in the FROM phrase and the record-name must not reference the same storage area or overlapping storage areas.

The result of executing a statement that specifies the FROM phrase is equivalent to the execution of these statements in this order:

1. “MOVE identifier TO record-name” executed in accordance with the rules specified for the MOVE statement
2. The same RELEASE, REWRITE, or WRITE statement without the FROM phrase

After execution of the actual statement is complete, the information in the storage area specified by the identifier is available, even though the information in the area specified by record-name is not available (except as specified by the SAME AREA clause).

INTO Phrase

The INTO phrase is an optional component of the READ and RETURN statements, both of which specify a file name as their primary operand. This phrase can be specified in a READ or RETURN statement only if either:

- The specified file has only one associated record description entry.
- The data description entry for the data item specified by the identifier in the INTO phrase and all record description entries associated with the file describe data structures or elementary alphanumeric items.

The storage area specified by the identifier in the INTO phrase must not be the same as or overlap the record area associated with the file identified by the file name.

The result of executing a statement that specifies the INTO phrase is equivalent to:

1. Execution of the same READ or RETURN statement without the INTO phrase (which obtains a logical record of the file)
2. After successful execution of the READ or RETURN statement, assignment of the current record in the file record area to the data item specified by the identifier in accordance with the rules for the MOVE statement; therefore (except for a destination described with the JUSTIFIED clause), long records are truncated at the right, and short records are padded at the right with spaces according to the standard alignment rules.

The size of the current record is the size of the record when it was written (see RECORD CONTAINS Clause (page 175)).

The record is then available in both the record area of the file and the storage area associated with the data item specified by the identifier. The move operation does not occur unless execution of the READ or RETURN statement was successful.

Input-Output

These topics are common to several input-output statements:

- I-O Status Code
- Diagnosing Input-Output Errors
- Recovering from Input-Output Errors
- Timed Input and Output

I-O Status Code

Each file-control entry can include a FILE STATUS clause that designates a 2-character file-status data item declared elsewhere in the program as the receptacle for that file’s I-O status code. The leftmost character position of the file-status data item is Status Key 1. The rightmost character position of the file-status data item is Status Key 2.

Whenever a program executes a CLOSE, DELETE, LOCKFILE, OPEN, READ, REWRITE, START, UNLOCKFILE, UNLOCKRECORD, or WRITE statement for a file that has a FILE STATUS clause, the run-time routines record the I-O status code in the specified file-status data item. The storage operation occurs prior to the execution of any applicable USE procedure or any applicable imperative statement associated with the input-output statement (in AT END, NOT AT END, INVALID KEY, or NOT INVALID KEY phrases).

The I-O status code indicates the success or failure of the input-output statement, and (if failure), the reason for the failure. Table 9-6 through Table 9-11 show the possible values of the file-status data item.

Topics:

- Status Key 1
- Status Key 2

Status Key 1

The leftmost character position of the file-status data item is known as Status Key 1 and is set for these conditions.

Table 9-5 Status Key 1 Values

Value	Condition	Explanation
0	Successful completion	The input-output operation was completed without error.
1	At-end condition	A sequential READ statement was unsuccessfully executed as a result of one of: • The physical end of the file was reached. • An optional file was not present.
2	Invalid-key condition	The input-output statement was unsuccessfully executed as a result of one of: • Sequence error • Duplicate key • No record found • Boundary violation
3	Permanent error	The input-output statement was unsuccessfully executed as a result of a boundary violation for a sequential file or as a result of an input-output error, such as data check, parity error, or transmission error.
4	Logic error	The input-output statement was unsuccessfully executed because the program attempted an improper sequence of input-output operations or because a user-defined limit was violated.
9	Implementor-defined error	Implementor-defined error means the input-output statement was unsuccessfully executed as a result of one of: • Program logic error • File description inconsistent with associated file

Status Key 2

The rightmost character position of the file-status data item is known as Status Key 2 and further describes the results of an input-output operation.

Table 9-6 Status Key 2 Values: Successful Completion

I-O Status Code	File Exception Condition
"00"	The input-output statement executed successfully, and no further information concerning the input-output operation is available.
"02"	The input-output statement executed successfully, but a duplicate key is detected in one of these ways: • For a READ statement, the key value for the current key of reference is equal to the value of that same key in the next record within the current key of reference • For a REWRITE or WRITE statement, the record just written created a duplicate key value for at least one alternate record key for which duplicates are allowed.
"04"	A READ statement executed successfully, but the length of the record being processed does not conform to the fixed file attributes for that file.

Table 9-6 Status Key 2 Values: Successful Completion *(continued)*

I-O Status Code	File Exception Condition
"05"	The input-output statement executed successfully; however, for an OPEN statement, the referenced optional file is not present at the time the OPEN statement is executed (if the open mode is I-O or EXTEND, the file has been created).
"07"	The input-output statement executed successfully, but a requested option could not be done for of one of these reasons: • For a CLOSE statement with the NO REWIND, REEL or UNIT REMOVAL phrase or for an OPEN statement with the NO REWIND phrase, the referenced file does not reside on a reel/unit medium. • For an OPEN statement that references a file that qualifies for HP COBOL Fast I-O, sufficient memory for HP COBOL Fast I-O was not available.
"97"	The input-output statement is successfully executed, but the circumstances were not entirely as expected. For an OPEN statement, either the referenced file has labels and LABEL RECORDS OMITTED was specified, or it does not have labels and LABEL RECORDS STANDARD was specified. For a READ statement, the retrieved record is currently locked, whether through some other file name of the run unit or by some other process. For a START statement, the start operation performed a read operation to validate a position, and the record that was read was locked.

Table 9-7 Status Key 2 Values: Unsuccessful Completion—At-End Condition

I-O Status Code	File Exception Condition
"10"	A sequential READ statement is attempted and no next logical record exists in the file because the logical end of the file has been reached.
"14"	A sequential READ statement is attempted for a relative file, but the relative record number cannot be assigned to the relative key data item described for the file without a loss of significance.

Table 9-8 Status Key 2 Values: Unsuccessful Completion—Invalid-Key Condition

I-O Status Code	File Exception Condition
"21"	A sequence error exists for a sequentially-accessed Indexed file. The prime record key value has been changed by the program between the successful execution of a READ statement and the execution of the next REWRITE statement for that file, or the ascending sequence requirements for successive record key values are violated.
"22"	An attempt has been made to write or rewrite a record that would create a duplicate relative key, a duplicate prime record key, or a duplicate alternate record key for which uniqueness is required.
"23"	An attempt has been made to access a record identified by a key, and that record does not exist in the file.
"24"	An attempt has been made to write beyond the externally-defined boundaries of a relative or indexed file, or a sequential WRITE statement is attempted for a relative file but the relative record number cannot be assigned to the relative key data item described for the file without a loss of significance.

Table 9-9 Status Key 2 Values: Unsuccessful Completion—Permanent Error Condition

I-O Status Code	File Exception Condition
“30”	A permanent error exists and no further information is available concerning the input-output operation.
“34”	A permanent error exists because of a boundary violation. This condition indicates that an attempt has been made to write beyond the externally-defined boundaries of a sequential file.
“35”	A permanent error exists because an OPEN statement with the INPUT, I-O, or EXTEND phrase is attempted on a required file that is not present.
“37”	A permanent error exists because an OPEN statement is attempted on a file that is required to be a disk file but is supported on some other medium.
“38”	A permanent error exists because an OPEN statement is attempted on a file previously closed with lock.
“39”	The OPEN statement is unsuccessful because a conflict has been detected between the fixed file attributes and the attributes specified for that file in the program.

Table 9-10 Status Key 2 Values: Unsuccessful Completion—Logic Error Condition

I-O Status Code	File Exception Condition
“41”	An OPEN statement is attempted for a file already in the open mode, or for one in a set of files that resides on a multiple file reel set when some other file in that set is already in the open mode.
“42”	A CLOSE, LOCKFILE, or UNLOCKFILE statement is attempted for a file not in the open mode.
“43”	In the sequential access mode, the last input-output statement executed for the file prior to the execution of a DELETE or REWRITE statement was not a successfully executed READ statement.
“44”	A boundary violation exists for one of these reasons: • An attempt is made to write or rewrite a record that is larger than the largest record description entry specified for the associated file name or smaller than the smallest record allowed by the RECORD clause of the associated file name. • An attempt is made to rewrite a record to a Sequential file and the record is not the same size as the record being replaced.
“46”	A sequential READ statement is attempted on a file open in the input or I-O mode and no valid next record has been established for one of these reasons: • The preceding START statement was unsuccessful. • The preceding READ statement was unsuccessful but did not end at an at-end condition.
“47”	The execution of a READ or START statement is attempted on a file not open in the Input or I-O mode.
“48”	The execution of a WRITE statement is attempted on a file not open in the I-O, Output, or Extend mode.
“49”	The execution of a DELETE or REWRITE statement is attempted on a file not open in the I-O mode.

Table 9-11 Status Key 2 Values: Unsuccessful Completion—Conditions Defined by HP

I-O Status Code	File Exception Condition
“90”	A Logic Error has occurred; it is not one of those defined under the “4*x*” codes, and no recovery is possible.
“91”	An I-O Error from which recovery might be possible has occurred.

Diagnosing Input-Output Errors

The I-O status code returned to the program indicates what type of input-output error occurred. The run-time routines send diagnostic messages for permanent error conditions and logic error conditions to the home terminal (or designated execution-log file), regardless of whether an error is recoverable. You must examine the I-O status code and to determine whether an unsuccessful input-output operation must cause the program to terminate abnormally or allow it to continue processing. To facilitate program execution and error recovery, the COBOL run-time input-output errors are classified as either:

- Program Logic Errors
- External Errors

Program Logic Errors

A program logic error is considered nonrecoverable and causes the program to terminate abnormally after execution of any applicable declarative section. This type of error causes I-O status code "4*x*" where *x* is a decimal digit, or I-O status code "90" (for errors defined by HP).

External Errors

An external error is recoverable if the program includes an associated declarative section; otherwise, an external error causes the program to terminate abnormally. The types of recoverable input-output errors are:

- An external error that has an associated operating system input-output error causes I-O status code "30". The run-time routine then calls the FILEERROR procedure to check whether an operation can be retried. If the operation is not retried, the operating system reports an error, giving the file name and the file-system error number, which the run-time routine includes in an error message that it delivers to the process's log file (usually the home terminal).
- An external input-output failure that is not related to an operating system error causes the return of I-O status code "91".

The format of the run-time error message depends on the environment. For details, see Chapter 49: Run-Time Diagnostic Messages (page 1193).

The run-time routines can also generate warnings, indicating that a minor error condition exists. A warning message is issued, but no declaratives or error actions are executed, and the program continues.

Each COBOL run unit contains a special register called GUARDIAN-ERR, a COMPUTATIONAL data item defined as

```
PIC 9(4) COMPUTATIONAL.
```

The process updates this register each time it executes an input-output statement. The value of GUARDIAN-ERR is delivered to the run-time routines by the operating system upon its completion of the requested input-output statement. Do not use the library routine COBOLFILEINFO or the Guardian routine FILEINFO to retrieve the error, because they can return incorrect values.

NOTE: GUARDIAN-ERR augments, but does not replace, the I-O status code.

The value of GUARDIAN-ERR derives only from the operating system. The run-time routines merely make the value available for the process to examine after the process has examined the appropriate I-O status code.

Table 9-12 is provided for your guidance only. The final authority on the meaning of the values of GUARDIAN-ERR is the current documentation of the operating system. Table 9-12 is not exhaustive; values other than those shown for GUARDIAN-ERR might be returned.

Table 9-12 I-O Status Codes Augmented by GUARDIAN-ERR

Status Code	GUARDIAN-ERR Value	COBOL Statement(s)	Cause of Status Code
“00”	0	All	Operation was successful
	6	READ sequential	Read from $RECEIVE and system message was read. Operation was still successful
“04”	0	READ	Record on file shorter than expected
“05”	11	OPEN	Open for INPUT, I-O, or EXTEND—OPTIONAL specified and file does not exist
	14	OPEN	Open for INPUT, I-O, or EXTEND—OPTIONAL specified and device does not exist for a temporary file
“07”	0	OPEN	Insufficient memory for HP COBOL Fast I-O
	0	CLOSE NO REWIND, OPEN NO REWIND	NO REWIND on a device that is not a tape
	0	CLOSE REEL	REEL/UNIT on a device that is not a tape
“10”	1	READ sequential	End of file
“21”	0	WRITE (indexed file)	Sequence violation
	11	REWRITE with sequential access	Record does not exist
	23	REWRITE with sequential access	Relative key out of bounds
	46	REWRITE with sequential access	Invalid key specified
“22”	10	REWRITE	Alternate key without duplicates already exists
	10	WRITE	Record already exists
“23”	0	READ random (relative file)	The position not as expected
	0	START	On an optional file that is missing or record at that position not desired one
	1	START	Record not found before EOF
	1	READ random (relative or indexed file)	Record does not exist or relative file has reached end
	11	DELETE, READ random, REWRITE	Record does not exist
	23	DELETE, READ random, REWRITE, START	Relative key out of bounds
	46	DELETE, READ random, REWRITE, START	Invalid key specified
“24”	23	WRITE	Relative key out of bounds

Table 9-12 I-O Status Codes Augmented by GUARDIAN-ERR *(continued)*

Status Code	GUARDIAN-ERR Value	COBOL Statement(s)	Cause of Status Code
	45	WRITE	File is full
	46	WRITE	Invalid (duplicated) key specified)
“30”	xxx	CLOSE	Fatal error on WRITE to write final block, or to write held spacing or record for print files, or attempt to write tape marks on tape failed
	xxx	CLOSE REEL	Reel swap failed
	xxx	DELETE	Attempt to position failed, or attempt to delete record failed, or attempt to reposition after deletion failed
	xxx	LOCK, UNLOCK	Attempt to lock/unlock file or record failed
	xxx	LOCK, UNLOCK	Wait on preread of file failed
	xxx	OPEN	System open failed; position for EXTEND failed; purge for existing file failed; create for nonexistent file failed; or other system call failed
	xxx	READ random	System error on read
	xxx	READ sequential	System error on read; reel swap failed
	xxx	REWRITE	System error on position request; replace failed
	xxx	START	System error on position request
	xxx	WRITE	System error on various operations
	11	OPEN	Opening for OUTPUT, I-O or EXTEND on file with alternate keys when no FUP CREATE has been done to create file
	18	OPEN	The node that file is on is not up
	40	LOCKFILE, READ, START	Operation timed out
	73	LOCKFILE, READ, START	Record in file locked
“34”	45	CLOSE, WRITE	File is full
“35”	11	OPEN	Opening for INPUT, I-O, or EXTEND on nonexistent file: OPTIONAL not specified

Table 9-12 I-O Status Codes Augmented by GUARDIAN-ERR *(continued)*

Status Code	GUARDIAN-ERR Value	COBOL Statement(s)	Cause of Status Code
	14	OPEN	Opening for INPUT, I-O, or EXTEND on temporary file: device does not exist, and OPTIONAL not specified
“37”	0	OPEN	Opening for INPUT or OUTPUT: device won’t support it
“38”	0	OPEN	File is locked
“39”	0	OPEN	File is not disk or is an unstructured disk file and organization is not sequential or file has alternate keys LINAGE specified on file that is not a printer or process Multiple files specified on a device that is not a tape OUTPUT specified on an EDIT file Record too large for file buffer size Fixed length records not specified for unstructured file Organization does not match file type Key specs for file do not match program key specs
“41”	0	OPEN	File is not closed
“42”	0	CLOSE	Attempt to close an unopened file
	0	LOCKFILE, UNLOCKFILE	Attempt to access an unopened file
“43”	0	DELETE, REWRITE (sequential access)	No successful READ
“44”	0	REWRITE	New record has different size than replaced one
“46”	0	READ sequential	Current position undefined
“47”	0	READ, START	File is not opened for INPUT or I-O
“48”	0	WRITE (sequential access)	Indexed or relative file not open for OUTPUT and not open EXTEND sequential access
	0	WRITE (random or dynamic access)	Indexed or relative file not open for I-O
	0	WRITE (sequential file)	File was not opened for OUTPUT, EXTEND, or I-O
“49”	0	DELETE, REWRITE	File was not opened for I-O

Table 9-12 I-O Status Codes Augmented by GUARDIAN-ERR *(continued)*

Status Code	GUARDIAN-ERR Value	COBOL Statement(s)	Cause of Status Code
"90"	0	LOCKFILE, READ, START	File was not opened for nowait I-O, and TIME LIMIT was specified
	0	WRITE	Wrong file type for ADVANCING operations
"91"	xxx	OPEN	Unable to initialize EDIT file
	0	OPEN	No buffer space available in user data space
	xxx	READ sequential	Unable to read EDIT file
	0	READ sequential	Read with lock and preread on
"97"	9	READ, START	A locked record was read or nominated in a START statement
	0	OPEN	The program described a file as having standard labels, and no standard label was found

xxx = Whatever file-system error is returned

Recovering from Input-Output Errors

When a process executes an I-O statement against a given file, either the I-O operation is successful or it is not. In either case, if there is an I-O status data item associated with the file, the run-time routines store an I-O status code there. (See I-O Status Code.)

After the run-time routines store the I-O status code, the behavior of the process depends on the nature of the exception and on the presence of any exception handling phrases, or if no exception handling phrases are present, then on the presence of an applicable declarative procedure.

Topics:

- At-End Condition or Invalid-Key Condition
- Other Error Conditions

At-End Condition or Invalid-Key Condition

If the error condition is the at-end condition or the invalid-key condition, the file is not affected (that is, if the statement was an output statement, the contents of the file remain unchanged). The run-time routines then perform these actions in this order:

1. If the statement that caused the condition includes any positive exception handling phrase (AT END or INVALID KEY), transfer control to that phrase's imperative statement. Ignore any USE AFTER EXCEPTION procedure and any negative exception handling phrases (NOT AT END, NOT INVALID KEY).

   If control reaches the end of the AT END or INVALID KEY phrase, transfer control to the statement immediately following the terminating period or scope terminator of the I-O statement that caused the exception condition.

2. If the statement that caused the condition does not include a positive exception handling phrase, but does include a negative exception handling phrase (NOT AT END or NOT INVALID KEY), ignore any USE AFTER EXCEPTION procedure and any negative exception handling phrases. Transfer control to the statement immediately following the terminating period or scope terminator of the I-O statement that caused the exception condition.

3. If all exception handling phases (AT END, NOT AT END, INVALID KEY, or NOT INVALID KEY) are absent but a declarative procedure of the form USE AFTER EXCEPTION is associated with the file, execute that procedure.
4. If control reaches the end of declarative procedure, transfer control to the statement immediately following the terminating period or scope terminator of the I-O statement that caused the exception condition.
5. If neither exception handling phrases nor declarative procedures apply, transfer control to the statement immediately following the terminating period or scope terminator of the I-O statement that caused the exception condition.

Other Error Conditions

If the execution is unsuccessful for a reason other than an at-end or invalid-key condition, the behavior of the program depends on the presence or absence of a declarative procedure for the file in question. If there is such a declarative procedure, the behavior of the program also depends on the presence or absence of any negative exception handling phrase (NOT AT END or NOT INVALID KEY).

When the run-time routines have recognized the exception condition, generated a run-time diagnostic message, and stored the I-O status code, they take these actions in this order:

1. If a declarative procedure of the form USE AFTER EXCEPTION is associated with the file, execute the declarative procedure.

   Then, if recovery from the exception is not possible (the I-O status code is “90” or Status Key 1 is “4”), terminate the process.

   If recovery from the exception is possible (either when Status Key 1 is “3,” or when it is “9” and Status Key 2 is not “0”), and the declarative procedure has not voluntarily terminated execution of the process, continue the execution:

   a. If the statement that caused the condition includes any negative exception handling phrase (NOT AT END or NOT INVALID KEY), transfer control to that phrase’s imperative statement. Ignore any positive exception handling phrases (AT END, INVALID KEY).
   b. If the statement that caused the condition does not include any negative exception handling phrase, transfer control to the statement immediately following the terminating period or scope terminator of the I-O statement that caused the exception condition.
2. If no applicable declarative procedure exists, terminate the process.

Timed Input and Output

Timed input-output prevents the deadlock that can occur when processes require concurrent access to shared data files. Timed I-O is enabled for a file when the file is opened with an OPEN statement containing a TIME LIMIT phrase. When timed I-O is enabled for a file, you can use a TIME LIMIT phrase in a LOCKFILE, READ, or START statement that applies to the file. The time limit is the number of seconds within which the operation must finish. A negative time limit means “wait indefinitely” (until the contention is resolved or the waiting process is terminated by an external agency).

Topics:

- Expired Time Limit
- Overhead
- Fatal Error
- $RECEIVE Timeout

Expired Time Limit

An expired time limit indicates a potential deadlock. The programmed recovery action (declarative procedure) must release any locks held by the program and restart the execution of the current request.

If no declarative procedure applies to the file when the operation terminates, the program terminates abnormally.

If a declarative procedure does apply to the file and the time limit expires, the declarative procedure is performed and program execution continues with the statement following the one terminated.

The value of the file position indicator becomes undefined when an operation exceeds its time limit. Because you cannot determine where in the operation the time limit was exceeded, you cannot necessarily try the operation again immediately at the current record.

Overhead

When a file is opened with timed I-O enabled, each I-O statement incurs more overhead than a file opened without a time limit. Avoid using timed I-O unnecessarily.

When a file is being read with APPROXIMATE positioning, the value used for time limit must take into account that a READ can take somewhat longer than expected. This can occur when a nonexistent record is sought, because the operating system searches through the file looking for the next defined record before reporting the absence of the record sought.

Fatal Error

If a file is opened without the TIME LIMIT phrase, and the TIME LIMIT phrase is specified in a LOCKFILE, READ, or START statement with a nonnegative value for the time limit, a run-time error is reported to the process’s home terminal, and the process terminates abnormally with the I-O status code “90”.

$RECEIVE Timeout

A process that must avoid unnecessary suspension when checking for messages on $RECEIVE can use timed I-O. If the READ statement specifies the time limit 0 and no message is present, the request times out immediately.

Arithmetic Operations

Many different statements tell the compiler to perform arithmetic operations, either because they are arithmetic statements or because they include arithmetic expressions.

The arithmetic statements are:

- ADD (page 295)
- COMPUTE (page 318)
- DIVIDE (page 325)
- MULTIPLY (page 381)
- SUBTRACT (page 473)

Topics:

- Common Features of Arithmetic Statements
- Arithmetic Expressions
- Arithmetic Precision

Common Features of Arithmetic Statements

This section describes the common features of the arithmetic statements, which are:

- Data Conversion and Alignment
- Composite of Operands
- Intermediate Data Items
- Multiple Results
- Incompatible Data

Data Conversion and Alignment

The data descriptions of the operands need not be the same; any necessary conversion and decimal point alignment is supplied by the COBOL compiler throughout the calculation.

Composite of Operands

The maximum size of each arithmetic operand is 18 decimal digits, independent of any decimal point. When the computer performs arithmetic, it must handle operands of different data descriptions. To discuss the restrictions on arithmetic operations, the COBOL community coined the term "composite of operands."

The composite of operands for a given operation is a fictitious data item. It has as many positions in its integer portion (to the left of the decimal point) as the operand that has the largest number of integer positions, and as many positions in its fraction portion (to the right of the decimal point) as the operand that has the largest number of fraction positions.

For example, in the data descriptions

```
01  A PIC S9(8)V9(4)
01  B PIC S9(2)V9(7)
01  C PIC S9(4)V9(9)
```

the composite of operands for an arithmetic operation involving only A, B, and C would have a data description of S9(8)V9(9), or 17 digits.

The composite of operands for an arithmetic statement other than COMPUTE must not exceed a size of 18 decimal digits. For arithmetic expressions or COMPUTE statements, the composite of operands does not apply.

Intermediate Data Items

For each arithmetic operation, an intermediate data item holds the result value until that value is either used as an operand in another operation or assigned to a receiving data item. The size of the intermediate data item depends on the operations and data items used in the operation and varies from 16 bits (about four digits) to 128 bits (about 39 digits). If the algebraic size of the result exceeds the capacity of this intermediate data item, a binary, floating-point, intermediate data item is used and the compiler issues warning 85. Because floating-point arithmetic has a maximum precision of 16 digits and often is not exact, the result might be incorrect in the rightmost digit or digits.

Multiple Results

The ADD, COMPUTE, DIVIDE, MULTIPLY, and SUBTRACT statements can have multiple results from performing arithmetic necessary to arrive at the final result to be stored in the receiving item. During execution, multiple results are the same as the results produced by a sequence of statements that either combine the value of an intermediate data item with a single result or transfer the value to a receiving item. These statements are in the same left-to-right order as that of the multiple results in the actual statement.

For example, the results of the statement

```
ADD  A B C  TO  C D (C) E
```

are equivalent to those of the statements

```
ADD A B C GIVING TEMP
ADD TEMP TO C
ADD TEMP TO D (C)
ADD TEMP TO E
```

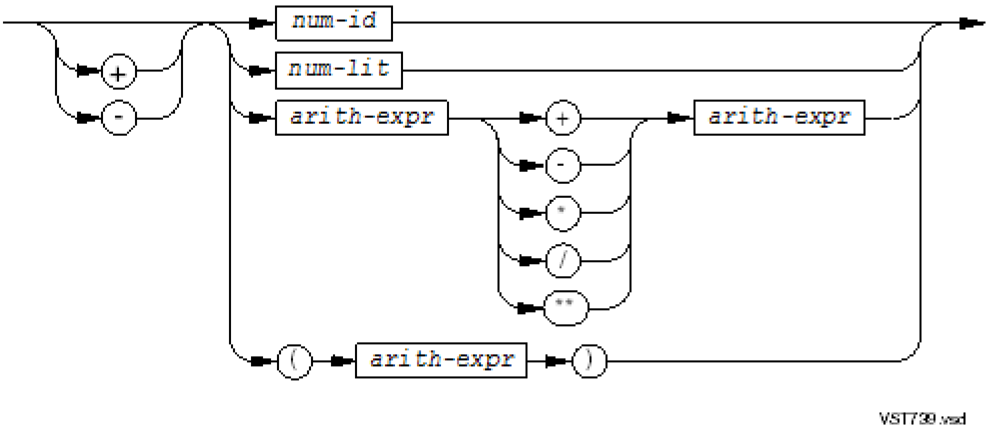
TEMP is the intermediate data item. Any subscripts specified in a reference to a receiving item are evaluated just prior to the assignment operation for that item.

Incompatible Data

Normally, whenever the contents of a data item are used in the Procedure Division and the current contents of that data item are not compatible with the class and size specified by its PICTURE clause, the result of the reference is undefined; that is, the semantic rules of the COBOL language apply only when operands have values corresponding to their descriptions. The single exception to this rule is the class condition, which exists specifically to permit testing whether or not an operand's value corresponds to its description. The presence of incompatible data can cause execution of the run unit to terminate abnormally.

Some computations can generate a negative zero (for example, when the value of an expression is -0.05, and this value is stored in a variable that is described as PICTURE S99V9). When the program uses the value negative zero, it ignores the sign.

Arithmetic Expressions



num-id

is the identifier of a numeric data item.

num-lit

is a numeric literal.

arith-expr

is an arithmetic expression.

The value of an arithmetic expression is a numeric value.

Arithmetic expressions can appear in:

- ENTER statements
- EVALUATE statements
- Reference modifiers
- Sign conditions
- Relation conditions
- Function arguments

The values computed when arithmetic expressions are evaluated are transitory and are not stored for later use by the program.

Topics:

- Operands
- Arithmetic Operators
- Formation and Evaluation Rules

Operands

An operand is a numeric literal, the identifier of a numeric data item, or any arithmetic expression enclosed within balanced left and right parentheses. The identifiers and literals appearing in an arithmetic expression must represent numeric data items and numeric literals upon which arithmetic can be performed.

Arithmetic Operators

Each operator in Table 9-13 must be preceded and followed by a separator (usually spaces or parentheses).

Table 9-13 Arithmetic Operators

Operator		
Symbol	Kind	Meaning
+	Unary	Multiplication by +1
	Binary	Addition
-	Unary	Multiplication by -1
	Binary	Subtraction
*	Binary	Multiplication
/	Binary	Division
**	Binary	Exponentiation

The minus sign (-) and the plus sign (+) must be preceded by a space or a left parenthesis and followed by a space. The exponentiation sign (**) cannot contain an embedded space, but it can be split across two lines through the use of the hyphen continuation character.

A plus (+) or minus (-) appearing as the first character of a numeric literal is the sign character of that literal, not an arithmetic operator. For example, +2 is not an arithmetic expression. In contrast, a plus (+) or minus (-) followed by a separator is interpreted as a binary operator when preceded by an operand or as a unary operator when not preceded by an operand. For example, X + 2 represents a simple arithmetic expression. The presence of a sign character in a numeric literal does not affect its use as an operand; therefore, both X + +2 and X + 2 are valid and, in this case, equivalent expressions.

Formation and Evaluation Rules

Parentheses can be used to specify the order in which the elements of an expression are to be evaluated. Expressions within parentheses are evaluated first. Within nested parentheses, evaluation proceeds from the least inclusive set to the most inclusive set. When parentheses are not used, or parenthesized expressions are at the same level of inclusiveness, the hierarchical order of execution is shown in Table 9-14.

Table 9-14 Hierarchy of Operators

Hierarchy	Operators
1st	Unary plus and minus
2nd	Exponentiation

Table 9-14 Hierarchy of Operators *(continued)*

Hierarchy	Operators
3rd	Multiplication and division
4th	Addition and subtraction

When no parentheses are present to specify otherwise, the order of execution of consecutive operations of the same hierarchical level is from left to right.

Table 9-15 shows some expressions that appear to be ambiguous and the COBOL interpretation of them.

Table 9-15 Precedence in Arithmetic Expressions

Ambiguous	Interpretation
A / B * C	(A / B) * C
A / B / C	(A / B) / C
A ** B ** C	(A ** B) ** C
A + B / C + D ** E * F - G	((A + (B / C)) + ((D ** E) * F)) - G

You can use arithmetic parentheses to:

- Override the normal hierarchical sequence of execution in expressions whose evaluation must not follow the normal precedence, for example:

  ```
  A / ( B * C )
  ```

- Clarify the hierarchical sequence of execution for the benefit of the reader, for example:

  ```
  PRIN + ( INT * PERIOD )
  ```

It is recommended that division be the last operation performed in an expression, if possible; otherwise, execution can be slow and precision can be lost.

Table 9-16 Operator-Operand Combinations

	Successor Element				
First Element	Variable	Binary Operator + - * / **	Unary Operator + or -	(	)
Identifier or literal	No	Yes	No	No	Yes
Binary Operator + - * / **	Yes	No	Yes	Yes	No
Unary Operator + or -	Yes	No	No	Yes	No
(	Yes	No	Yes	Yes	No
)	No	Yes	No	No	Yes

An arithmetic expression must begin with an operand (which can be a parenthetical expression) or a unary operator followed by an operand. Parentheses must always appear in balanced pairs such that each left parenthesis precedes its corresponding right parenthesis within the expression. Any operand can be preceded by a unary operator.

These rules apply to evaluation of the exponentiation operator in an arithmetic expression:

- If the value of the base (left-hand) operand is 0, the power (right-hand) operand must have a value greater than 0; otherwise, the size error condition exists.
- If the evaluation yields both a positive and a negative real number, the value returned as the result is the positive number.
- If the result of the evaluation is not a real number or is not representable by the computer system on which the operation is evaluated, the size error condition exists.
- COBOL accepts noninteger as well as integer exponents.

Arithmetic Precision

The precision of ADD, SUBTRACT, MULTIPLY, and DIVIDE statements can be fairly easily stated. The precision of arithmetic expressions is substantially more complex, because the compiler must create intermediate data items as it evaluates the expression. Arithmetic expressions occur only in the COMPUTE, ENTER, and EVALUATE statements, the relation and sign conditions, function arguments, and in reference modification.

In discussing precision, we ignore the presence of a scale factor or of any decimal point; the arithmetic processing records and handles these elements separately. After the computation is performed with integral values, the decimal point and scale factor (if any) are then provided for the result.

The largest value HP COBOL can store is the unsigned COMPUTATIONAL-5 value 18,446,744,073,709,551,615.

The largest value HP COBOL can store associated with a DISPLAY or COMPUTATIONAL data item (independent of any decimal point or scale factor) is the 20-digit number which is the highest value for an unsigned 64-bit number.

In evaluating arithmetic expressions, COBOL can manipulate intermediate values having up to 128 bits (about 39 digits).

These topics explain arithmetic precision as it applies to:

- SIZE ERROR
- ADD and SUBTRACT Statements
- MULTIPLY Statement
- DIVIDE Statement
- Arithmetic Expressions

SIZE ERROR

As explained in SIZE ERROR Phrase, the size error condition indicates a problem in computation. It arises when an arithmetic overflow occurs or when a program attempts to divide by zero, raise zero to the zero power, or raise a negative number to a power that produces other than a real number value. If no SIZE ERROR clause is present in the statement where the condition arises, the process can terminate abnormally with an arithmetic overflow or it can store invalid values into result items (mainly in items of USAGE COMPUTATIONAL).

ADD and SUBTRACT Statements

Each data item in an ADD or SUBTRACT statement, except the one after GIVING, is called an addend. In the statement

```
ADD P Q R S T   GIVING   W
```

the addends are P, Q, R, S, and T. In the statement

```
ADD P Q R S T   TO   X Y
```

the addends are P, Q, R, S, T, X, and Y.

The number of accurate fraction digits maintained during evaluation is determined by the addend having the greatest number of fraction digits. All remaining intermediate result representation space is used for nonfraction digits. The computed result is always completely accurate unless an internal overflow occurs, raising the size error condition; therefore, all digits in the value assigned to a receiving data item are accurate unless the size error condition occurs for that data item.

MULTIPLY Statement

The number of fraction digits in the product is the sum of the number of fraction digits in the two operands. The number of accurate nonfraction digits in the product is the sum of the nonfraction digits in the two operands. The computational method used verifies that an internal overflow never occurs for a MULTIPLY statement; therefore, all digits in the value assigned to a receiving data item are accurate unless the size error condition occurs for that data item.

DIVIDE Statement

Because division is a rather complicated mathematical operation, and because the computer is performing scaled integer arithmetic instead of floating-point arithmetic, the rules stating the precision of HP COBOL division are somewhat complicated.

- Effect of GIVING

  When the GIVING phrase is present, a single quotient is computed. The appropriate number of fraction digits in that quotient is determined from the receiving operand having the greatest number of fraction digits.

  When no GIVING phrase is present, a separate quotient is computed for each receiving data item. Such a DIVIDE statement with multiple receiving data items is exactly equivalent to a sequence of DIVIDE statements, each having a single receiving data item and all having the same divisor. The appropriate number of fraction digits in each quotient is determined from the corresponding receiving data item.

- Effect of Decimal Point Placement

  Under the mathematical rules for division, fraction digits in the divisor cause the significant digit positions of the quotient to appear shifted to the left with respect to the significant digit positions of the dividend. This corresponds to moving the decimal point of the dividend to the right the same number of positions as would be necessary to make the divisor an integer:

  ```
  1.00/0.3 = 10.0/03 = 03.3
  ```

  Put another way, each divisor fraction digit cancels a trailing digit position in the dividend, which then reappears as a leading digit position in the quotient. This causes a problem when the actual dividend has fewer fraction digits than the sum of the number of digits in the divisor's fraction and the number of digits in the appropriate quotient's fraction.

  Suppose you want to divide two data items described as:

  ```
  DIVIDEND   PICTURE 9(10)v9(5) USAGE DISPLAY
  DIVISOR    PICTURE 9(4)v9(7) USAGE DISPLAY
  ```

  If the computer aligns them by scaling each up by 7 decimal places, it has to extend the dividend's fraction with 2 additional digit positions filled with zeros. It would be dividing a 17-digit integer by an 11-digit integer.

  Because the sum of the number of actual dividend digits and the number of appended zero digits cannot exceed 36, there are very few combinations of operands in a DIVIDE statement for which it is impossible to append enough zeros.

- Size Error

  All digits in the value assigned to the receiving data item are accurate unless either the size error condition occurs for that data item or at least one trailing digit is fictitious (that is, set to 0 because it was not generated by the division operation).

Arithmetic Expressions

HP COBOL maintains intermediate results during the evaluation of arithmetic expressions. Arithmetic expressions occur only in the COMPUTE statement, the EVALUATE statement, the relation condition, the sign condition, the reference modifier, the parameters of the ENTER statement, and function arguments.

NOTE: The precision of COBOL arithmetic computation has limitations. The COMPUTE statement is particularly sensitive. Use of an appropriate sequence of the ADD, SUBTRACT, MULTIPLY, and DIVIDE statements permits you to attain the precision you desire, provided that the PICTURE clauses describing the operands do not imply potential values of exaggerated significance. There is no guarantee that the result of an arithmetic expression will be the same in different implementations.

The evaluation of an arithmetic expression is the result of the evaluation of a sequence of intermediate results. The maximum number of digits held for an intermediate result is 36. If this number is exceeded, the compiler might use binary floating-point arithmetic (and issue warning 85). The size error condition applies to both final results and intermediate results.

These abbreviations are used to explain intermediate operations:

Abbreviation	Description
IR	Number of integer places carried for an intermediate result.
DR	Number of decimal places carried for an intermediate result.
OP1	First operand in an arithmetic expression, which has the form 9(I1)V9(D1), where I1 is the number of integer places carried and D1 is the number of decimal places carried for the first operand.
OP2	Second operand in an arithmetic expression, which has the form 9(I2)V9(D2), where I2 is the number of integer places carried and D2 is the number of decimal places carried for the second operand.
OPR	Desired result, which has the form 9(IR)V9(DR), where IR is the number of places carried for the integer result, and DR is the number of places carried for the decimal result.

The number of decimal places in the intermediate result (DR) is chosen first, then the number of integer places in the intermediate result is determined from that and the characteristics of the two operands.

HP COBOL guarantees that the mathematical significance of results will be at least as good as application of these rules implies.

	Decimal Places (Worst-Case Precision)	
Operation	DR is the greater/greatest of ...	IR is the lesser of ...
OP1 + OP2 OP1 - OP2	D1 and D2	(The greater of I1 and I2) + 1) and (36 - DR)
OP1 * OP2	D1 and D2	(I1 + I2) and (36 - DR)
OP1 / OP2	D1 - D2, the D of the composite of all the result fields, and 1	(I1 + D2) and (36 - DR)
OP1 ** OP2	D1 or D2	(The greater of I1 or I2) and (36 - DR)

NOTE: If (I1+D2+DR) > 36, the compiler uses floating-point arithmetic (and issues warning 85).

When an arithmetic expression involves a division operation, the intermediate results are evaluated in these steps:

1. The actual division
2. The adjustment of that result for use in further computations

Therefore, in each of these instances of the arithmetic expression "A1/A2+A3*A4,"

```
COMPUTE AX = A1/A2 + A3 * A4
IF A1/A2 + A3 * A4 LESS THAN AX GO TO ...
```

the division is performed before further evaluation of either of the preceding statements. If the division operation is not the last operation in the expression and the divisor is not 2, 4, 5, 8, or a power of 10, the division is carried out to 36 digits. As many fractional digits as possible are kept. This number can be truncated depending on subsequent operations. In some cases adjustment is not possible, so the compiler uses binary floating-point arithmetic (and issues warning 85). Since the precision of floating-point arithmetic is 16 digits and numbers cannot always be represented exactly, the result might be slightly larger or smaller than the exact number. You might have to use individual operations or revise the expression to guarantee accurate results.

To obtain the maximum accuracy in an arithmetic expression that involves a division, use parentheses or revise the expression as necessary to assure that the division is the last operation performed. For example, you can rewrite the expression "a / b * c" as "(a * c) / b".

When a conditional expression compares a variable and an expression, the number of decimal places carried for the variable is used for the number of decimal places carried for the expression.

When a conditional expression compares two expressions, the compiler determines the smallest number of decimal places suitable for each of the expressions and then uses the larger of those numbers of decimal places.

In a COMPUTE statement, the number of decimal places of the composite of operands of the receiving fields is the number of decimal places of the expression.

If ROUNDED is specified on any operand, one additional decimal position is used in the computation, then the rounding is applied to the result.

Conditional Expressions

Many different statements include conditional expressions. A conditional expression is a syntactically correct combination of simple conditions, logical operators, and parentheses that can be evaluated to a truth value. The truth value determines which of two paths of control the object program takes.

The simplest form of a conditional expression is a simple condition. Complex conditions are combinations of simple conditions and any of the logical operators NOT, AND, and OR. COBOL allows you to abbreviate sequences of complex relation conditions. You can use balanced sets of parentheses to control or clarify the order of evaluation within a conditional expression.

Topics:

- Simple Conditions
- Complex Conditions
- Abbreviated Combined Relation Conditions
- Condition Evaluation Rules

Simple Conditions

A simple condition has a truth value of TRUE or FALSE. Enclosing a simple condition within parentheses does not affect its truth value. The simple conditions are:

- Relation Conditions in General
- Relation Conditions With Nonpointer Operands
- Relation Conditions With Pointer Operands
- Class Conditions
- Condition-Name Conditions (Conditional Variables)
- Switch-Status Conditions
- Sign Conditions

Relation Conditions in General

A relation condition causes a comparison of two operands. The relation condition has a truth value of TRUE if the relation exists between the operands; otherwise, it has a truth value of FALSE.

Usage Considerations:

- Terminology

  In the preceding syntax diagrams, the left-hand operand is called the subject of the condition; the right-hand operand is called the object of the condition.

- Where Pointer Relations Are Allowed

  A relation with pointer operands is allowed in EVALUATE, IF, PERFORM, and SEARCH VARYING statements. It is not allowed in SEARCH ALL statements, because pointer data items have no meaningful order.

- Cannot Compare Literal With Literal

  At least one nonpointer operand must be an index-name, identifier, or arithmetic expression containing at least one reference to a data item; that is, the relation must include at least one nonliteral element.

- Operators

  The relational operators specify the type of comparison to be made in the relation condition, as this table shows. NOT and the component following it are considered to be a single relational operator. For example, NOT EQUAL is a truth test for an unequal comparison. In this table, optional words are in brackets.

Relational Operator		
Words	**Symbol**	**Meaning**
GREATER [THAN]	>	Greater than
NOT GREATER [THAN]	NOT >	Not greater than
LESS [THAN]	<	Less than
NOT LESS [THAN]	NOT <	Not less than
EQUAL [TO]	=	Equal to
NOT EQUAL [TO]	NOT =	Not equal to
GREATER [THAN] OR EQUAL [TO]	>=	Greater than or equal to
LESS [THAN] OR EQUAL [TO]	<=	Less than or equal to

- Numeric Comparisons

  Numeric comparisons are made with respect to the algebraic values of the operands. The number of digits present in the representation of an operand is not significant. Comparison of numeric operands is permitted regardless of the manner in which their usage is described. Unsigned numeric operands are considered positive for purposes of comparison. Zero is considered a unique value regardless of its sign.

- Nonnumeric Comparisons

  For nonnumeric operands, or one numeric and one nonnumeric operand, a comparison is made with respect to the program collating sequence (see OBJECT-COMPUTER Paragraph (page 114)).

  - One operand is numeric:

    - The numeric operand must be a numeric literal or a reference to a numeric data item; therefore, it cannot be an arithmetic expression. COBOL requires the operand to be an integer and, if a data item, to have DISPLAY usage. HP COBOL removes both of these restrictions.
    - When the nonnumeric operand is either an elementary data item or a nonnumeric literal, the numeric operand is handled as though it were moved to an elementary alphanumeric data item, and the value of this item were then compared to the nonnumeric operand. The size of this conceptual data item is the same as the number of digit positions in the numeric operand. The numeric operand is handled as if it were an unsigned integer; therefore any sign or assumed decimal point is deleted by the conceptual move, and no COMPUTATIONAL items are converted to usage DISPLAY.
    - When the nonnumeric operand is a data structure, the numeric operand is handled as though it were moved to a data structure, and the value of this item were then compared to the nonnumeric operand. The size of this conceptual data item is the same as the number of character positions occupied by the numeric operand. Because the operand is handled as if it were alphanumeric, the sign of the numeric operand (if any) is not deleted by the conceptual move, and no COMPUTATIONAL usage items are converted to DISPLAY usage.

  - Operand size

    The size of an operand is the total number of characters it contains. Operands can be of equal or unequal size.

    - Operands of equal size

      If the operands are of equal size, the process compares characters in corresponding character positions, starting with the leftmost character position and continuing until it either encounters a pair of unequal characters or exhausts the operands. The process determines that the operands are equal if all character pairs in the operands are equal; otherwise, the process compares the first pair of unequal characters it encounters to determine their relative position in the collating sequence active for the comparison. The operand that contains the character that has a higher position in the collating sequence is considered to be the greater operand.

    - Operands of unequal size

      If the operands are of unequal size, comparison proceeds as though the shorter operand were extended on the right by sufficient spaces to make the operands of equal size.

- Index-Name and Index Data Item Comparisons

  Relation tests can be made between these operands:

  - Two index-names

    The result is the same as if the corresponding occurrence numbers were compared.

  - An index-name and a numeric data item or numeric literal

    The occurrence number that corresponds to the value of the index-name is compared to the value of the data item or literal.

  - An index data item and an index-name or another index data item

    The actual values are compared without conversion.

  Neither an index-name nor an index data item can be compared with any operand other than those mentioned in the preceding rules.

- National Data Items and National Literals

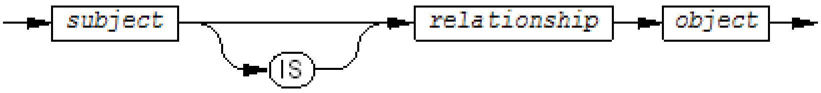
  A national data item or a national literal can be compared only to another national data item or a national literal.

The first three simple relation conditions in Example 9-11 are equivalent.

Example 9-11 Simple Relation Conditions

```
I + 1 > HIGH-I
(I + 1) > HIGH-I
HIGH-I < (I + 1)
TAXABLE-INCOME GREATER THAN ZERO
LAW-NAME = "MURPHY"
"POOL" NOT EQUAL TROUBLE-SOURCE
A-INDEX NOT > 23
```

Relation Conditions With Nonpointer Operands

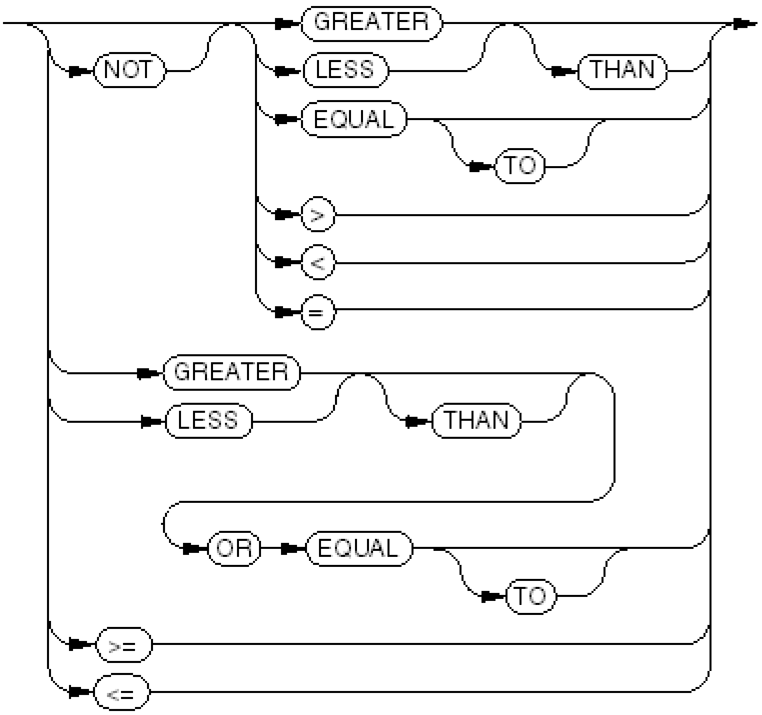


VST118.vsd

subject, *object*

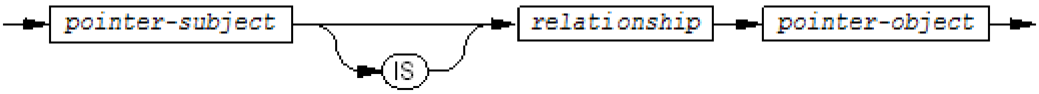
is an identifier, a literal, an arithmetic expression, an index-name.

relationship



VST119.vsd

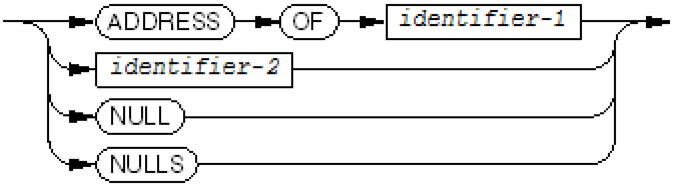
Relation Conditions With Pointer Operands



VST602.vsd

pointer-subject

pointer-object



VST603.vsd

identifier-1

is a level-01 or level-77 data item defined in the Linkage Section or Data Division.

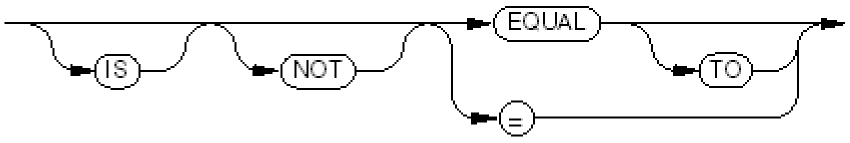
identifier-2

is a data item with USAGE POINTER.

NULL, NULLS

is a null address (all 1s) that causes an address fault if a pointer with that value is referenced. NULL or NULLS can be used for *pointer-subject* or *pointer-object*, but not both.

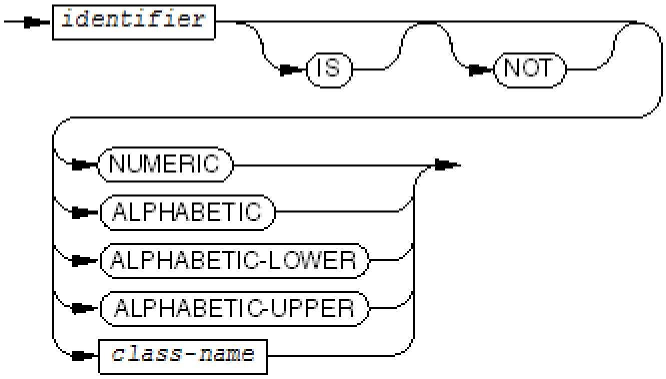
relationship



VST604.vsd

Class Conditions

The class condition determines whether the operand is numeric, alphabetic, lowercase alphabetic, uppercase alphabetic, or contains only characters in the set defined by a CLASS clause in the SPECIAL-NAMES paragraph of the Environment Division.



VST120.vsd

identifier

is the identifier of a data item for which either:

- It is described as USAGE DISPLAY.
- It is of the numeric class, and its usage is not DISPLAY.

class-name

is the name of a class described in the SPECIAL-NAMES paragraph.

Usage Considerations:

- NOT Modifier

  When NOT appears, the compiler considers it and the next keyword to form a single class condition. For example, NOT NUMERIC is a truth test that determines if the operand is nonnumeric.

- NUMERIC and NOT NUMERIC

  You cannot use the NUMERIC and NOT NUMERIC tests with either:

  — An alphabetic data item

  — A data structure that contains any elementary item not described as USAGE DISPLAY or whose description indicates the presence of an operational sign

  If the description of the tested item does not indicate the presence of an operational sign, the item belongs to the numeric class only if:

  — The content of the item is numeric (consists entirely of the digit characters `0` through `9`).

  — No operational sign is present.

If the description of the tested item indicates the presence of an operational sign, the item belongs to the numeric class only if:

- The content of the item is numeric (consists entirely of the digit characters `0` through `9`).
- The item is not described as USAGE DISPLAY, and the content of the item consists entirely of a valid representation for the usage. If a PICTURE clause is specified, its numeric value is within the range of values implied by the PICTURE clause.

- ALPHABETIC and NOT ALPHABETIC

  You cannot use the ALPHABETIC and NOT ALPHABETIC tests with a numeric data item. Normally, the tested item belongs to the alphabetic class only if its content consists entirely of some combination of the alphabetic characters `A` through `Z`, `a` through `z`, and space. When the CHARACTER-SET clause specifies a program character set other than USASCII or UK, the set of characters that constitute the alphabetic class is extended as appropriate (see OBJECT-COMPUTER Paragraph (page 114)).

- ALPHABETIC-LOWER and NOT ALPHABETIC-LOWER

  You cannot use the ALPHABETIC-LOWER and NOT ALPHABETIC-LOWER tests with a numeric data item. The tested item belongs to the alphabetic-lower class only if its content consists entirely of the lowercase alphabetic characters `a` through `z` and space. When the CHARACTER-SET clause specifies a program character set other than USASCII or UK, the set of characters that constitute the alphabetic class is extended as appropriate (see OBJECT-COMPUTER Paragraph (page 114)).

- ALPHABETIC-UPPER and NOT ALPHABETIC-UPPER

  You cannot use the ALPHABETIC-UPPER and NOT ALPHABETIC-UPPER tests with a numeric data item. The tested item belongs to the alphabetic-upper class only if its content consists entirely of the uppercase alphabetic characters `A` through `Z` and space. When the CHARACTER-SET clause specifies a program character set other than USASCII or UK, the set of characters that constitute the alphabetic class is extended as appropriate (see OBJECT-COMPUTER Paragraph (page 114)).

- Class-name and NOT Class-name

  You cannot use the class-name and NOT class-name tests with a numeric data item. The result of the test is TRUE if the content of the data item consists entirely of the characters listed in the definition of the class-name in the SPECIAL-NAMES paragraph; otherwise the result is FALSE.

  Class conditions are useful for field validation:

  ```
  PART-NUMBER IS NUMERIC
  REPLY-FIELD IS NOT ALPHABETIC
  DIAGNOSTIC-MESSAGE IS NOT ALPHABETIC-UPPER
  ```

  If the SPECIAL-NAMES paragraph includes the description

  ```
  CLASS VOWEL IS "A" "E" "I" "O" "U" "a" "e" "i" "o" "u".
  ```

  the procedure division can use this condition for validation:

  ```
  SOME-FIELD IS VOWEL
  ```

Condition-Name Conditions (Conditional Variables)

In a condition-name condition, a conditional variable is tested to determine whether or not its value is equal to one of the values or within one of the ranges of values associated with a condition-name.

Usage Considerations:

- Condition-Name Conditions

  The rules for comparing a conditional variable with the literal values corresponding to a condition-name are the same as those specified for relation conditions.

  The result of the test is TRUE if either of these relations is satisfied:

  — The value of the conditional variable equals one of the single values corresponding to the condition-name.

  — The value of the conditional variable falls within one of the ranges of values, including the end values, corresponding to the condition-name.

Example 9-12 Conditional Variables

```
WORKING-STORAGE SECTION.
01  ZIP-CODE.
    03 ZIP-FIRST-3 PICTURE 999.
       ...
       88 NEW-YORK     VALUE IS 090 THRU 098,
                                100 THRU 149.
       88 PENNSYLVANIA VALUE IS 150 THRU 196.
...
PROCEDURE DIVISION.
...
    ELSE IF NEW-YORK     MOVE "NY" TO STATE-FROM-ZIP
    ELSE IF PENNSYLVANIA MOVE "PA" TO STATE-FROM-ZIP
...
    IF STATE-INPUT NOT = STATE-FROM-ZIP
       PERFORM QUERY-ZIP-OR-STATE-CORRECTION.
```

Switch-Status Conditions

A switch-status condition determines whether an external switch is on or off. The particular switch and the ON or OFF value associated with the condition must be named in the SPECIAL-NAMES paragraph of the Environment Division.

Usage Considerations:

- TRUE Value

  The result of the test is TRUE if the switch is set to the position corresponding to the one identified by the condition-name; otherwise, the result of the test is FALSE.

- Setting External Switches

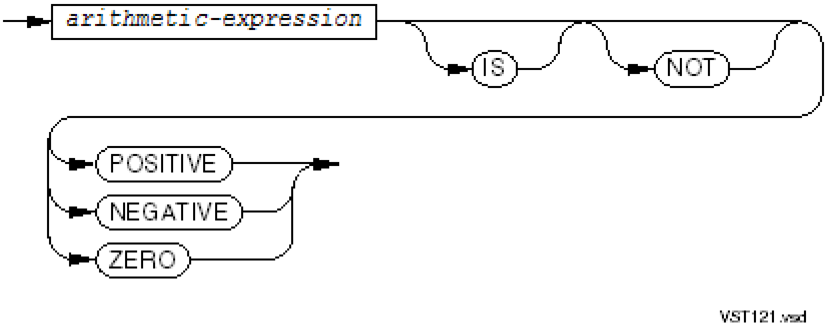
  Set external switch value by using the PARAM command of the command interpreter (see SPECIAL-NAMES Paragraph (page 118)) or by the SET statement (see SET (page 444)).

Example 9-13 External Switches

```
SPECIAL-NAMES.
   SWITCH-1 IS IN-SWITCH  ON  STATUS IS TAPE-INPUT
                          OFF STATUS IS DISK-INPUT
   SWITCH-2 IS OUT-SWITCH ON  STATUS IS TAPE-OUTPUT
                          OFF STATUS IS DISK-OUTPUT.
...
PROCEDURE DIVISION.
...
   IF TAPE-INPUT   OPEN INPUT TAPE-SOURCE
   ELSE            OPEN INPUT DISK-SOURCE.
   IF TAPE-OUTPUT  OPEN OUTPUT TAPE-SINK
   ELSE            OPEN OUTPUT DISK-SINK.
...
```

Sign Conditions

The sign condition determines whether the algebraic value of an arithmetic expression is less than, greater than, or equal to zero.



arithmetic-expression

is an arithmetic expression, as described in Arithmetic Expressions.

Usage Considerations:

- NOT Modifier

  When NOT appears, it and the next keyword are considered to form a single sign condition. For example, NOT ZERO is a truth test that determines if the operand value is nonzero (is positive or negative).

- Definition

  An operand is positive if its value is greater than zero, negative if its value is less than zero, and zero if its value is equal to zero.

- Expression Cannot Be Composed Entirely of Literals

  The arithmetic expression must contain at least one variable operand (at least one operand that is not a literal).

Example 9-14 Sign Conditions

```
( A - 10 ) IS POSITIVE
B NOT ZERO
```

Complex Conditions

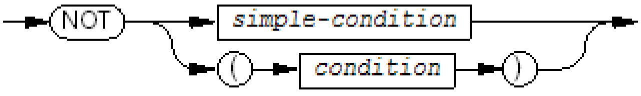
A complex condition is a combination of simple conditions and any of the logical operators NOT, AND and OR. Its truth value is the one that results from the interaction of the stated logical operators on the individual truth values of the simple conditions or conditions enclosed within parentheses.

Table 9-17 Logical Operators

Operator		
Symbol	**Kind**	**Meaning**
NOT	Unary	Logical negation (reversal of truth value): The truth value is TRUE if the condition is FALSE, FALSE if the condition is TRUE
AND	Binary	Logical conjunction: The truth value is TRUE if both conjoined conditions are TRUE, FALSE if one or both conjoined conditions are FALSE
OR	Binary	Logical inclusive OR: The truth value is TRUE if one or both included conditions are TRUE, FALSE if both included conditions are FALSE

A complex condition formed by applying the unary operator NOT to a simple or parenthetical condition is called a negated condition. A complex condition formed by applying the binary operator AND or OR to two conditions (either of which can be simple or parenthetical) is called a combined condition.

Negated Conditions



VST122.vsd

simple-condition

is described in Simple Conditions.

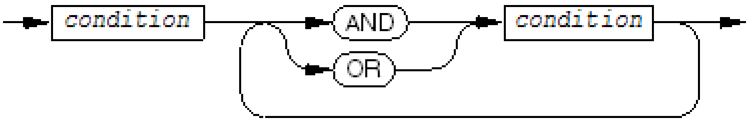
condition

is a simple or complex condition.

The truth value of a negated condition is the opposite of the truth value of the operand condition.

You do not need to use parentheses when you use either AND or OR exclusively in a combined condition. When you use a mixture of AND, OR, and NOT, you can use parentheses to effect a final truth value.

Combined Conditions



VST123.vsd

condition

is a simple or complex condition.

Table 9-18 Conditions, Logical Operators, and Parentheses

	Location in Conditional Expression		In a left-to-right sequence of elements	
Element	**First**	**Last**	**When not first, element can be immediately preceded only by**	**When not last, element can be immediately followed only by**
simple-condition	Yes	Yes	AND, NOT, OR, (	AND, OR,)
AND or OR	No	No	*simple-condition*,)	NOT, (, *simple-condition*

Table 9-18 Conditions, Logical Operators, and Parentheses *(continued)*

	Location in Conditional Expression		In a left-to-right sequence of elements	
Element	**First**	**Last**	**When not first, element can be immediately preceded only by**	**When not last, element can be immediately followed only by**
NOT	Yes	No	AND, OR, (	(, `simple-condition`
(	Yes	No	AND, NOT, OR, (	NOT, (, `simple-condition`
)	No	Yes	`simple-condition`,)	AND, OR,)

As Table 9-18 shows, you can use the element pair OR NOT, but you cannot use the pair NOT OR. Also, you can use NOT, but you cannot use NOT NOT. Within the combined condition, parentheses must always be in balanced pairs, so that each left parenthesis precedes its corresponding right parenthesis.

These combined conditions are valid:

LARRY AND MOE AND (CURLY OR SHEMP)	All four are condition-names—level-number 88.
I < E AND (E NOT > C)	All three symbols are alphanumeric data items.
NOT ((A IS POSITIVE) OR (B IS POSITIVE))	Both A and B are numeric data items.

Abbreviated Combined Relation Conditions

COBOL enables you to abbreviate a sequence of complex relation conditions. For example, you can use

```
A NOT EQUAL B OR C
```

instead of having to use

```
(A NOT EQUAL B) OR (A NOT EQUAL C)
```

You can abbreviate any condition in a sequence except the first one by omitting either of these:

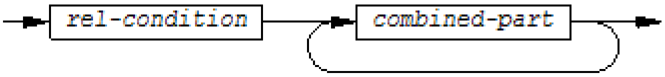
- The subject, for example:

  ```
  A NOT EQUAL B OR NOT EQUAL C
  ```

- The relational operator and the subject, for example:

  ```
  A NOT EQUAL B OR C
  ```

The subject is the term to the left of the operator. For more information, see Usage Considerations.

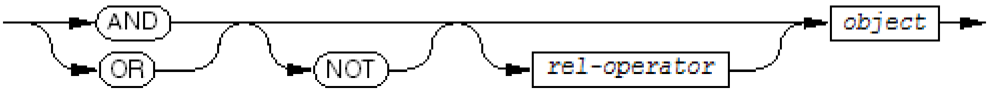


VST124.vsd

`rel-condition`

is a relational condition.

`combined-part`



VST125.vsd

`rel-operator`

is a relational operator.

object

is an identifier, a literal, an arithmetic expression, or an index-name.

This abbreviation technique is available when a group of the characteristics shown in Table 9-19 is present.

Table 9-19 Abbreviated Combined Relation Conditions

Characteristics	Example
Simple relation conditions	A = C
or	
negated simple relation conditions	NOT A = B
are combined using AND and OR operators	NOT A = B OR A = C
in which a relation condition subject or subject and relational operator is repeated and there are no parentheses (except those delimiting subscripts or reference modifiers) within the sequence.	"A =" is repeated
When these conditions are met, any relation condition except the first one in the series can be abbreviated in one of these ways:	The first "A ="
• Omit the subject.	NOT A = B OR = C
• Omit the subject and the relational operator.	NOT A = B OR C

When either abbreviated form is used, the omitted subject is considered to be the same as the last explicitly stated subject, and the omitted operator is considered to be the same as the last explicitly stated operator.

If any portion of such an abbreviated condition is enclosed in parentheses, all the subjects and operators required for the evaluation of that portion must be included in the same set of parentheses.

Usage Considerations:

- Abbreviation in a Sequence of Relation Conditions

  Within a sequence of relation conditions, both of the preceding forms of abbreviation can be used. The effect of using such abbreviations is as if the last preceding stated subject were inserted in place of the omitted subject, and the last stated relational operator were inserted in place of the omitted relational operator. This insertion of an omitted subject or relational operator terminates once a complete simple condition is encountered within a complex condition.

- NOT as Modifier or Operator

  The reserved word NOT in an abbreviated combined relation condition is interpreted as a part of the relational operator if the word immediately following NOT is GREATER, >, LESS, <, EQUAL, or =; otherwise, NOT is interpreted as a logical operator, and the implied insertion of a subject or relational operator results in a negated relation condition.

Some examples of abbreviated combined and negated relation conditions and expanded equivalents follow:

Abbreviated Combined Relation Condition	Expanded Equivalent
A > B AND NOT < C OR D	((A > B) AND (A NOT < C)) OR (A NOT < D)
A NOT EQUAL B OR C	(A NOT EQUAL B) OR (A NOT EQUAL C)
NOT (A GREATER B OR < C)	NOT ((A GREATER B) OR (A < C))

Abbreviated Combined Relation Condition	Expanded Equivalent
NOT (A NOT > B AND C AND NOT D)	NOT ((((A NOT > B) AND (A NOT > C)) AND (NOT (A NOT > D))))
(A + B - C) > D AND NOT < E OR F	(A + B - C) > D AND (A + B - C) NOT < E OR (A + B - C) NOT < F

Condition Evaluation Rules

Parentheses can be used to specify the order in which individual conditions are to be evaluated when it is necessary to depart from the implied evaluation precedence. Conditions within parentheses are evaluated first. Within nested parentheses, evaluation proceeds from the least inclusive condition to the most inclusive condition. When parentheses are not used, or parenthetical conditions are at the same level of inclusiveness, this hierarchical order of logical evaluation is implied until the final truth value is determined:

- Values are established for arithmetic expressions.
- Truth values for simple conditions are established in this order:
  - Relation
  - Class
  - Condition-name
  - Switch-status
  - Sign
- Truth values for negated simple conditions are established.
- Truth values for combined conditions are established, first by applying the AND logical operators, then by applying the OR logical operators.
- Truth values for negated combined conditions are established.
- When the sequence of evaluation is not completely specified by parentheses, the order of evaluation of consecutive operations of the same hierarchical level is from left to right.

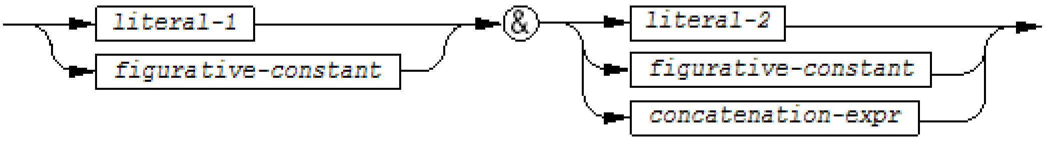
Using this order of evaluation, once a truth value for an entire complex condition is established, evaluation of the condition ceases. For example, in the complex condition

```
A NOT < 1 AND A NOT > 10 AND B (A) > 5
```

no attempt is made to fetch B(A) if A is not a valid subscript, because determination that the first condition is FALSE guarantees that the complete complex condition is FALSE.

Concatenation Expressions

The value of a concatenation expression is the concatenation of the value of its operands. It is the equivalent of a literal of the same class and value, and can be used anywhere that a literal of that class can be used, except in a COPY or REPLACE statement.



VST801.vsd

literal-1

is either a simple nonnumeric literal, a hexadecimal nonnumeric literal, or a national literal. For information about these literals, see:

- Simple Nonnumeric Literals (page 81)
- Hexadecimal Nonnumeric Literals (page 82)
- National Literals (page 83)

literal-2

is either a simple nonnumeric literal, a hexadecimal nonnumeric literal, or a national literal of the same class (alphanumeric or national) as *literal-1*.

figurative-constant

is a figurative constant that does not include the word ALL. For information about figurative constants, see Figurative Constants (page 84).

concatenation-expr

is a concatenation expression of the same class (alphanumeric or national) as *literal-1*.

Table 9-20 Maximum Length of Result of Concatenation Expression

Class of Operands	Length of Result
Alphanumeric	160 alphanumeric character positions
National	80 national character positions
A figurative constant occupies one character position.	

Table 9-21 Class of Result of Concatenation Expression

Number of Operands That Are Figurative Constants	Class of Result
0	Same class as the operands (neither of which is a figurative constant)
1	Same class as the other operand (which is not a figurative constant)
2	Alphanumeric

Example 9-15 Concatenation Expressions

```
"A zero-terminated string" & X"0"
"Another way of getting a zero-terminated string" & LOW-VALUE
"A string terminated by the zero digit" & ZERO
BLANK & BLANK & "Two leading blanks in the result"
"The symbolic constant" & COPYRIGHT & "is defined in SPECIAL-NAMES paragraph"
```

10 Procedure Division Verbs

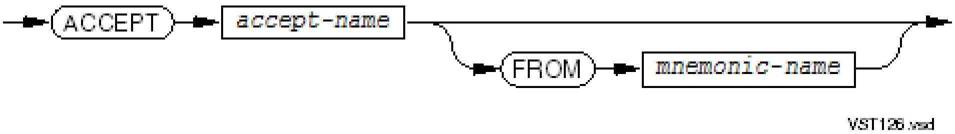
This section describes the COBOL verbs that you can use in the Procedure Division, in alphabetic order. The descriptions of some verbs, such as ADD and INSPECT, show more than one syntax format.

For descriptions of the COPY and REPLACE verbs, which you can use in any divisions, see COPY Statement (page 507) and REPLACE Statement (page 516).

ACCEPT

ACCEPT With Mnemonic-Name

ACCEPT with a `mnemonic-name` delivers small amounts of data to a process from a terminal or another process.



`accept-name`

is the identifier of an elementary item or data structure (not an index-name, index data item, special register, or national data item) in which the process can store accepted data.

`mnemonic-name`

identifies a terminal or another process from which the process can accept data. You must associate *`mnemonic-name`* with a terminal or file name in the Environment Division (see SPECIAL-NAMES Paragraph (page 118)).

NOTE: If *`mnemonic-name`* is associated with CONSOLE in the SPECIAL-NAMES paragraph, you cannot use it in an ACCEPT statement. CONSOLE maps to $0, an output-only device.

Usage Considerations:

- Differences in the OSS and Guardian Environments

  In the OSS environment but not in the Guardian environment:

  — No prompt is given for an ACCEPT statement.

  — If an ACCEPT statement includes *`mnemonic-name`*, it must be the OSS pathname of a Guardian process or terminal. If *`mnemonic-name`* is an OSS device, the compiler issues a warning and the default input device (#IN) is used instead.

- If *`mnemonic-name`* Is Not Specified

  If you omit the phrase FROM *`mnemonic-name`*, the process accepts data from its default input device or default input-output device. (See Specifying Default Input and Output Devices (page 841) or Specifying a Default Input-Output Device (page 837).) The default input device or default input-output device is usually the home terminal, except in a Pathway environment where the COBOL run unit is functioning as a server. In that case, the server class command SET SERVER determines the default input device or default input-output device (see the *TS/MP System Management Manual*).

- Specifying the Default Input-Output Device as mnemonic-name

  If you want the process to accept data from its default input-output device, omit the FROM phrases from the ACCEPT statements and specify a default input-output device before executing the program. (See Specifying Default Input and Output Devices (page 841) or

Specifying a Default Input-Output Device (page 837).) Do not specify the default input-output device as the *mnemonic-name*, because this causes the process to terminate abnormally.

- If *mnemonic-name* Is Not Open

  If *mnemonic-name* specifies a process that has not been opened by an earlier ACCEPT or DISPLAY statement, or specifies a terminal, then the accepting process opens the process or terminal that *mnemonic-name* specifies as if it were a file.

- If Process Cannot Open or Accept Data From mnemonic-name

  If the process cannot open or accept data from the terminal or process that *mnemonic-name* specifies, the process tries to accept data from its home terminal. If the home terminal is also unavailable, the process assigns the value 0 to each numeric or numeric edited receiving item and fills any other category of receiving item with spaces.

- Cursor Placement When *mnemonic-name* Is a Terminal

  If *mnemonic-name* specifies a terminal that was the object of a DISPLAY NO ADVANCING statement, then the ACCEPT statement sets the cursor at the location following that where the display operation ended; otherwise, the ACCEPT statement prompts with a question mark (?) and awaits input.

- Whether Process Closes *mnemonic-name* or Leaves It Open

  If *mnemonic-name* specifies a terminal that was not the object of a DISPLAY NO ADVANCING statement, then the process closes the terminal after accepting data from it. If *mnemonic-name* specifies another process or a terminal that was the object of a DISPLAY NO ADVANCING statement, then the process leaves the other process or terminal open.

- Case Sensitivity

  The ACCEPT statement does not change the letters in the accepted data from lowercase to uppercase or from uppercase to lowercase when it stores them in *accept-name*. To perform such a conversion, use the INSPECT CONVERTING statement.

- Alphabetic, Alphanumeric, and Alphanumeric Edited Data

  If *accept-name* is an alphabetic, alphanumeric, or alphanumeric edited data item, the process collects data in an intermediate data item using one or more read operations. Each read operation collects zero or more characters from the process or terminal that *mnemonic-name* specifies, up to the maximum physical record size for that process or terminal (a typical maximum physical record size for a terminal is 80 characters).

  If a read operation gets a null response (that is, if the user enters only a carriage return), the ACCEPT statement appends space characters to the intermediate data item until its length is that of *accept-name*.

  If a response is not null but contains fewer characters than the maximum physical record size of the process or terminal that *mnemonic-name* specifies, the ACCEPT statement appends space characters to the response until its length is that of *accept-name*.

  The ACCEPT statement constructs the intermediate item by appending responses in the order in which it receives them, and continues to execute read operations until it has filled *accept-name* or received a null response.

  Finally, the ACCEPT statement moves the value of the intermediate item to *accept-name*, following the rules for the MOVE statement.

- Numeric and Numeric Edited Data

  If *accept-name* is a numeric or a numeric edited data item, the process collects data with one read operation. The response to the read operation must be in numeric literal format (see Decimal Numeric Literals (page 80) and Hexadecimal Numeric Literals (page 81)). A null response or a response containing only spaces is invalid for a numeric or a numeric edited *accept-name*.

If the response to the read operation is not in numeric literal format, the ACCEPT statement reprompts with:

```
** Improper numeric value.  Resupply input **
```

If the response is in numeric literal format, the ACCEPT statement converts it to a numeric literal in an intermediate data item and then moves the value of the intermediate data item to *accept-name*. If *accept-name* has no fractional part, any necessary truncation of the intermediate data item value deletes the rightmost digits of the value; otherwise, the transfer of the data from the intermediate item to *accept-name* follows the rules for the MOVE statement.

Example 10-1 ACCEPT Statement Reading From a Terminal

```
WORKING-STORAGE SECTION.
01   COMMAND-IN     PICTURE X(7)   VALUE SPACES.
    88  ADD-C                      VALUE "ADD".
    88  UPDATE-C                   VALUE "UPDATE".
    88  DELETE-C                   VALUE "DELETE".
    88  EXIT-C                     VALUE "EXIT".
    ...
PROCEDURE DIVISION.
    ...
    DISPLAY "ENTER COMMAND"
    ACCEPT COMMAND-IN
    IF ADD-C ...
```

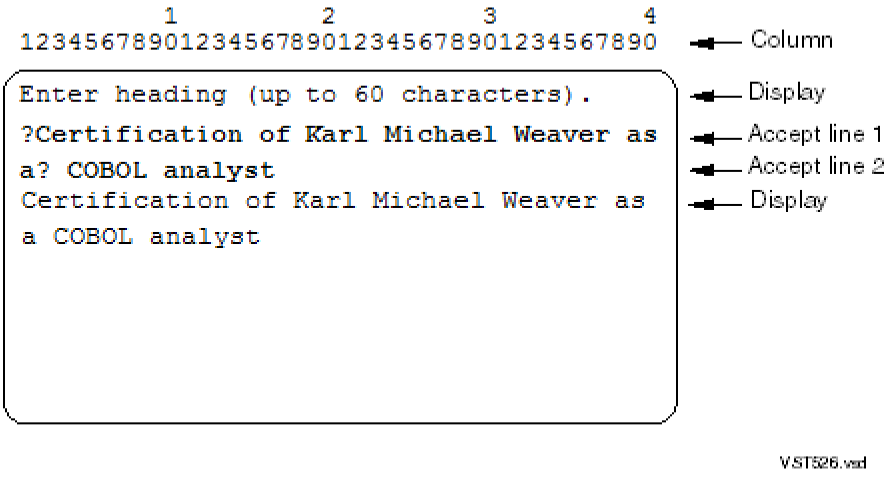
In Example 10-2, the ACCEPT statement reads an alphanumeric value into an *accept-name* whose length (60 characters) exceeds the record size of the terminal (40 characters).

Example 10-2 ACCEPT Statement Reading Alphanumeric Data

```
WORKING-STORAGE SECTION.
01  HEADING-IN     PICTURE X(60) VALUE SPACES.
    ...
PROCEDURE DIVISION.
    ...
    DISPLAY "Enter heading (up to 60 characters)"
    ACCEPT HEADING-IN
    DISPLAY HEADING-IN
    MOVE HEADING-IN TO HEADING-OUT
    ...
```

Figure 10-1 shows the run-time interactions at the terminal. The user’s responses to the ACCEPT statement’s prompts are in bold font. When the user enters the fortieth character with no carriage return, the ACCEPT statement prompts again.

Figure 10-1 ACCEPT Statement Collecting Alphanumeric Data



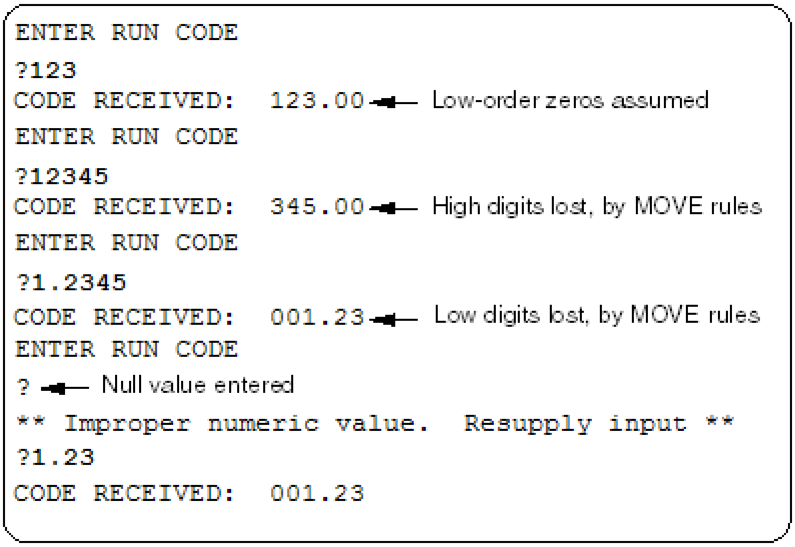
```
         1         2         3         4
1234567890123456789012345678901234567890
Enter heading (up to 60 characters).
?Certification of Karl Michael Weaver as
a? COBOL analyst
Certification of Karl Michael Weaver as
a COBOL analyst
```

Example 10-3 ACCEPT Statement Reading Numeric Data

```
ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
SOURCE-COMPUTER. HP System.
OBJECT-COMPUTER. HP System.
SPECIAL-NAMES.
   FILE "#TERM" IS USER-TERMINAL.
...
WORKING-STORAGE SECTION.
01  PROGRAM-CODE   PICTURE 999V99.
...
PROCEDURE DIVISION.
DEMO.
   PERFORM UNTIL PROGRAM-CODE = 0
      DISPLAY "ENTER RUN CODE" UPON USER-TERMINAL
      ACCEPT PROGRAM-CODE FROM USER-TERMINAL
      DISPLAY "CODE RECEIVED: "
              PROGRAM-CODE UPON USER-TERMINAL
   END-PERFORM
...
```

Figure 10-2 shows run-time interactions at the home terminal (#TERM). The user’s responses to the ACCEPT statement’s prompts are in bold font.

Figure 10-2 ACCEPT Statement Collecting Numeric Data



```
ENTER RUN CODE
?123
CODE RECEIVED:   123.00
ENTER RUN CODE
?12345
CODE RECEIVED:   345.00
ENTER RUN CODE
?1.2345
CODE RECEIVED:   001.23
ENTER RUN CODE
?
** Improper numeric value.  Resupply input **
?1.23
CODE RECEIVED:   001.23
```

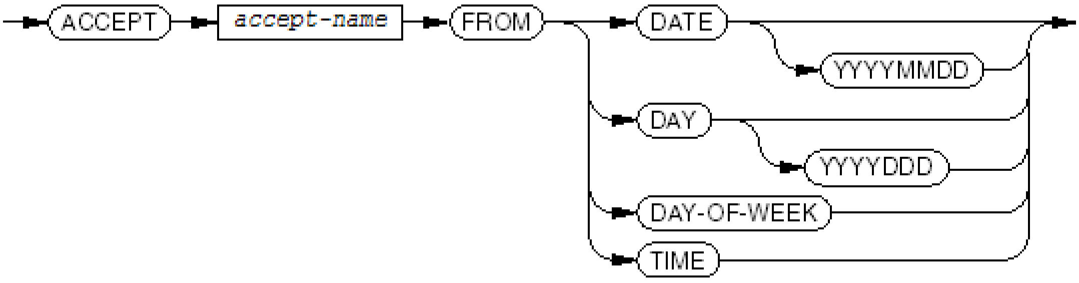
Low-order zeros assumed
High digits lost, by MOVE rules
Low digits lost, by MOVE rules
Null value entered

VST527.vsd

ACCEPT With DATE, DAY, DAY-OF-WEEK, or TIME Phrase

ACCEPT with a DATE, DAY, DAY-OF-WEEK, or TIME phrase retrieves the numeric representation of the date, day, day of the week, or time from the operating system.

NOTE: A simpler alternative is the CURRENT-DATE Function (page 672).



VST127.vsd

accept-name

is the identifier of the data item to which the representation of the day, date, day of the week, or time is moved (following the rules for the MOVE statement). Typical definitions for the data items are:

Data Item		Typical Definitions
DATE	Without YYYYMMDD	PICTURE 9(6) or PICTURE 99B99B99
	With YYYYMMDD	PICTURE 9(8) or PICTURE 9999B99B99
DAY	Without YYYYDDD	PICTURE 9(5) or PICTURE 99B999
	With YYYYDDD	PICTURE 9(7) or PICTURE 9999B999

Data Item	Typical Definitions
`DAY-OF-WEEK`	`PICTURE 9`

`TIME`	`PICTURE 9(8) or`
	`PICTURE 99B99B99B99`

DATE

delivers the current date.

`YYYYMMDD`

Without `YYYYMMDD`, `DATE` behaves as if it were a data item described with the PICTURE character-string 9(6) organized as *yymmdd* where *yy* is the year of the century; for example, July 10, 1992, is represented as 920710.

With `YYYYMMDD`, `DATE` behaves as if it were a data item described with the PICTURE character-string 9(8) organized as *yyyymmdd*; for example, July 10, 1992, is represented as 19920710.

DAY

delivers the current year and serial day number.

YYYYDDD

Without `YYYYDDD`, `DAY` behaves as if it were a data item described with the PICTURE character-string 9(5) organized as *yyddd*; for example, January 23, 1992, is represented as 92023.

With `YYYYDDD`, `DAY` behaves as if it were a data item described with the PICTURE character-string 9(7) organized as *yyyyddd*; for example, January 23, 1992, is represented as 1992023.

DAY-OF-WEEK

delivers the value of the current day of the week expressed as an integer between 1 and 7, inclusive. The value 1 represents Monday, 2 represents Tuesday, and so on.

TIME

delivers the current time of day. TIME behaves like a data item described with the PICTURE character-string 9(8) organized as four pairs of digits—*hhmmsscc*, where *hh* is the hour (based on a 24-hour clock), *mm* is the minutes, *ss* is the seconds, and *cc* is the hundredths of seconds. For example, 2:41 p.m. is expressed as 14410000. The minimum value of TIME is 00000000; the maximum value is 23595999.

In Example 10-4, ACCEPT statements tell the executing process to store the current day of the week (integer) in DAY-SUB, the current date (*yymmdd*) in TODAYS-DATE, the current time (*hhmmsscc*) in TIME-RIGHT-NOW, and 40 characters from the home terminal in USER-REPLY.

Example 10-4 ACCEPT Statements Reading Current Data and Time

```
WORKING-STORAGE SECTION.
01  DATE-AND-TIME-FIELDS.
    05  DAY-SUB         PIC 9       VALUE ZERO.
    05  TODAYS-DATE     PIC 9(6)    VALUE ZERO.
    05  TIME-RIGHT-NOW  PIC 9(8)    VALUE ZERO.
77  USER-REPLY          PIC X(40)   VALUE SPACES.
   ...
PROCEDURE DIVISION.
   ...
   ACCEPT DAY-SUB        FROM DAY-OF-WEEK
   ACCEPT TODAYS-DATE    FROM DATE
   ACCEPT TIME-RIGHT-NOW FROM TIME
   ACCEPT USER-REPLY
```

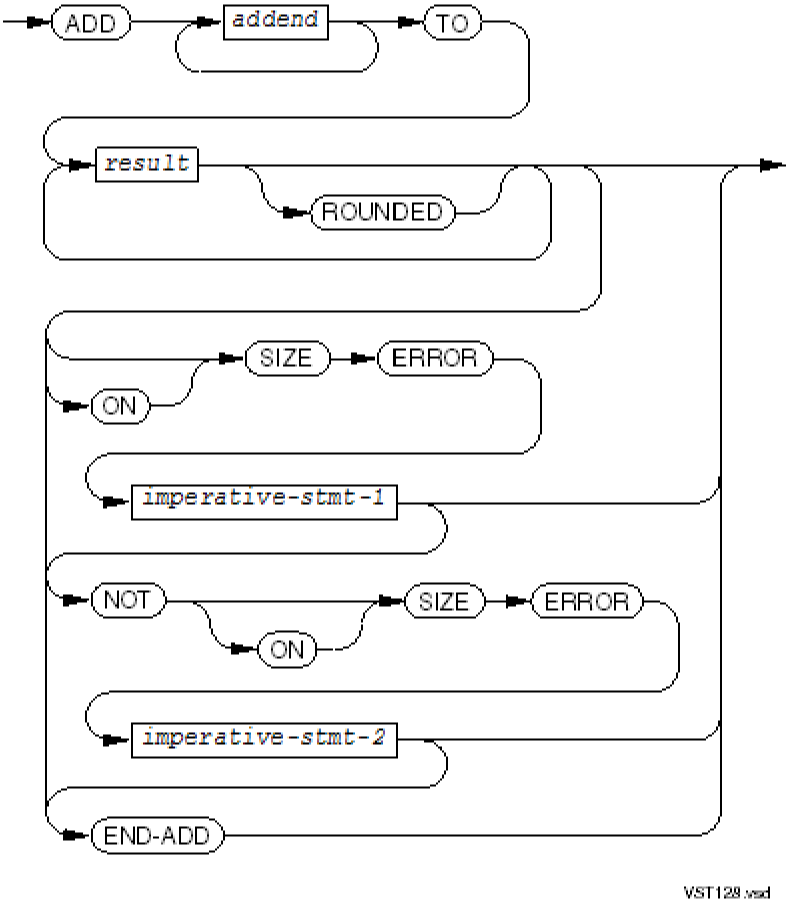
ADD

ADD TO

ADD TO adds the sum of one or more numeric values to one or more result data items; for example,

```
ADD A B C TO D E
```

stores A+B+C+D in D and stores A+B+C+E in E.



addend

is a numeric literal or the identifier of an elementary numeric data item.

result

is the identifier of an elementary numeric data item to which *addend* or the sum of the addends is added.

ROUNDED

specifies that `result` is to be rounded before being stored.

imperative-stmt-1

is an imperative statement to be executed if a size error occurs during the addition of addends or the storing of a result.

imperative-stmt-2

is an imperative statement to be executed if no size error occurs during the addition of addends or the storing of a result.

END-ADD

ends the scope of the ADD statement and makes it a delimited-scope statement. If you omit END-ADD but include the SIZE ERROR or NOT SIZE ERROR phrase, the ADD statement is a conditional statement and ends at the next period separator.

Usage Considerations:

- Specifying the Same Data Item for More Than One Result

  For each `result`, the sum of the addends is added to the current value of `result`, and that new sum becomes the new value of `result` (after rounding if rounding was specified). If more than one `result` specifies the same data item, the final value of that item reflects multiple additions of the intermediate sum. For example, if the initial value of A is 2, then the statement

  ```
  ADD 5 TO A A A
  ```

  changes the value of A to 7, and then 12, and finally 19.

- Operand Identification Order

  For each `result`, operand identification occurs just prior to the add-and-store operation; therefore, in the statement

  ```
  ADD A B C TO I I X(I)
  ```

  the subscript is not evaluated until A+B+C is added to I twice.

- Arithmetic Operations

  See Arithmetic Operations (page 267) for information on data conversion and alignment, intermediate results, multiple results (and subscript evaluation), and incompatible data.

- Precision

  See ADD and SUBTRACT Statements (page 272) for information on precision of addition.

- ROUNDED, SIZE ERROR, and NOT SIZE ERROR Phrases

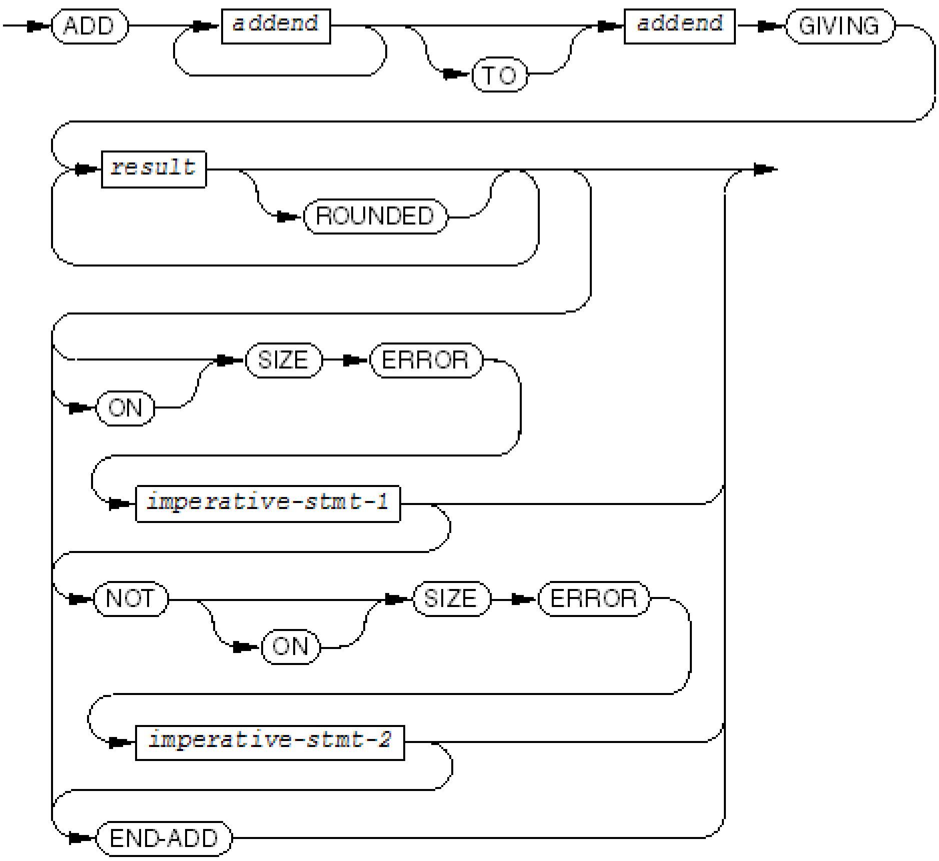
  See ROUNDED Phrase (page 254) and SIZE ERROR Phrase (page 254) for information on these phrases.

ADD GIVING

ADD GIVING adds two or more numeric values and replaces the current value of one or more result data items with that sum; for example,

```
ADD A B C GIVING D E
```

stores A+B+C in both D and E.



VST129.vsd

`addend`

is a numeric literal or the identifier of an elementary numeric data item.

`result`

is the identifier of an elementary numeric data item to which `addend` or the sum of the addends is added.

ROUNDED

specifies that `result` is to be rounded before being stored.

`imperative-stmt-1`

is an imperative statement to be executed if a size error occurs during the addition of addends or the storing of a result.

`imperative-stmt-2`

is an imperative statement to be executed if no size error occurs during the addition of addends or the storing of a result.

END-ADD

ends the scope of the ADD statement and makes it a delimited-scope statement. If you omit END-ADD but include the SIZE ERROR or NOT SIZE ERROR phrase, the ADD statement is a conditional statement and ends at the next period separator.

Usage Considerations:

- Changing Addend Values

  The ADD GIVING statement does not change the value of an `addend` unless you also specify that `addend` for `result`.

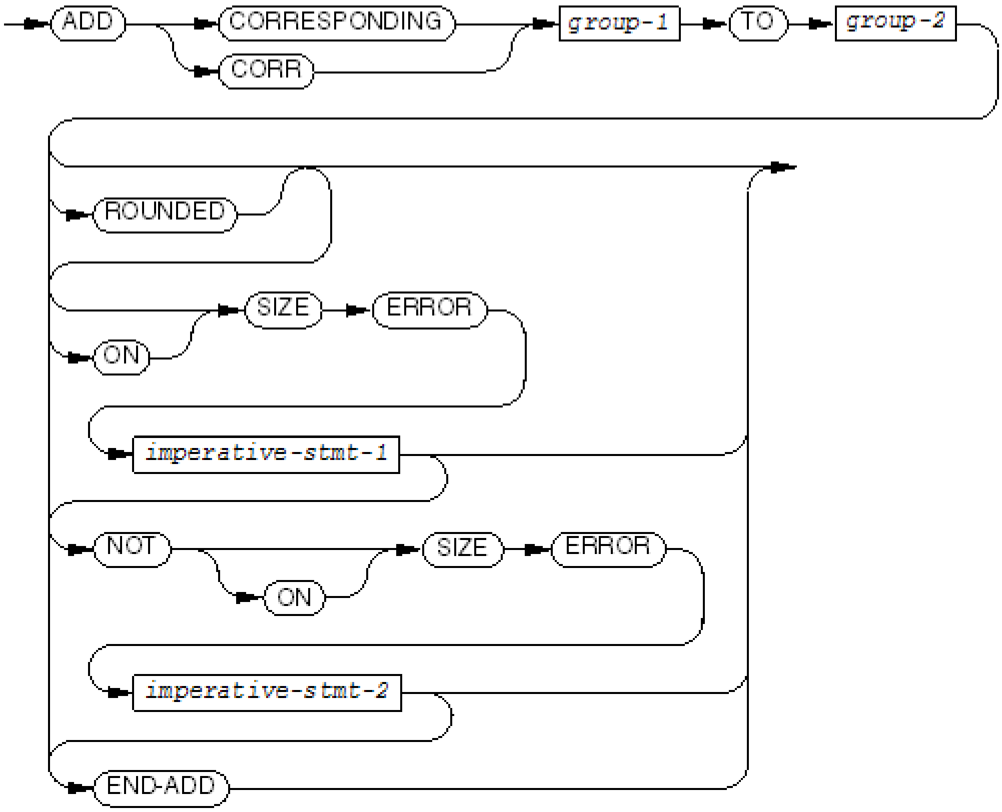
- See these usage considerations in ADD TO:
  - Operand Identification Order
  - Arithmetic Operations

— Precision
— ROUNDED, SIZE ERROR, and NOT SIZE ERROR Phrases

ADD CORRESPONDING

ADD CORRESPONDING adds numeric elements of one data structure to corresponding numeric elements of another data structure.

△ **CAUTION:** ADD CORRESPONDING is not recommended, because minor changes to one data structure can change the correspondence between its elements and those of the other data structure, and this is difficult to detect.



VST130.vsd

CORRESPONDING, CORR

are equivalent and specify that the values of corresponding elementary items of two data structures are to be added.

`group-1`

is the identifier of a data structure in which some or all of the elementary items are numeric.

`group-2`

is the identifier of a data structure in which some or all of the elementary items are numeric. The value of each elementary numeric item in `group-2` that corresponds to an elementary `numeric` item in `group-1` is replaced by the sum of the two corresponding items, as in the ADD TO statement.

ROUNDED

specifies that `result` is to be rounded before being stored.

`imperative-stmt-1`

is an imperative statement to be executed if a size error occurs during the addition of addends or the storing of a result.

imperative-stmt-2

is an imperative statement to be executed if no size error occurs during the addition of addends or the storing of a result.

END-ADD

ends the scope of the ADD statement and makes it a delimited-scope statement. If you omit END-ADD but include the SIZE ERROR or NOT SIZE ERROR phrase, the ADD statement is a conditional statement and ends at the next period separator.

Usage Considerations:

- Definition of Correspondence

  See CORRESPONDING Phrase (page 253).

- ADD CORRESPONDING Statement Generates ADD TO Statements

  Each pair of corresponding elementary data items, *group-1-item* and *group-2-item*, generates an ADD TO statement equivalent to this:

  ```
  ADD group-1-item TO group-2-item [ROUNDED]
  ```

- Problem Adding Names to Group Data Items

  When adding names to group data items that appear in ADD CORRESPONDING statements, verify that the new name will not be included in unintended addition operations.

- See these usage considerations in ADD TO:
  - Operand Identification Order
  - Arithmetic Operations
  - Precision
  - ROUNDED, SIZE ERROR, and NOT SIZE ERROR Phrases

Correspondence depends on the names of the elementary items, not on the physical order of the elementary items. In Example 10-5, all data item names correspond except STAPLES and PAPER. Add operations skip STAPLES and PAPER.

Example 10-5 ADD CORRESPONDING Statement

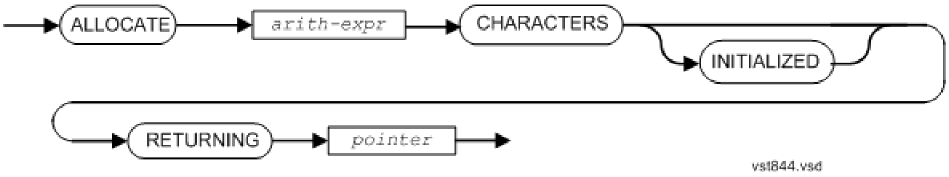
```
WORKING-STORAGE SECTION.
01  CABINET-SUPPLIES.
    05  PAPER-CLIPS       PIC 99.
    05  WRITING-TOOLS.
        10  PENCILS       PIC 99.
        10  PENS          PIC 99.
        10  ERASERS       PIC 99.
        10  PAPER         PIC 99.
    05  STAPLES           PIC 99.
01  STOCKROOM-SUPPLIES.
    05  WRITING-TOOLS.
        10  PENCILS       PIC 99.
        10  ERASERS       PIC 99.
        10  PENS          PIC 99.
    05  PAPER-CLIPS       PIC 99.
    05  PAPER             PIC 99.
  ...
PROCEDURE DIVISION.
  ...
   ADD CORRESPONDING CABINET-SUPPLIES TO STOCKROOM-SUPPLIES.
```

ALLOCATE

Form	Description
ALLOCATE bytes	Obtains a specified number of bytes of dynamic memory
ALLOCATE BASED item	Obtains dynamic memory for a BASED item

ALLOCATE Bytes

ALLOCATE bytes obtains dynamic memory for a specified number of bytes.



arith-expr

is an arithmetic expression specifying the number of bytes of memory to obtain. If *arith-expr* does not evaluate to an integer, the result is rounded up to the next whole number. If *arith-expr* evaluates to zero or a negative value, no memory is obtained and the value NULL is assigned to *pointer*.

pointer

is a data item described as USAGE POINTER. *pointer* is set to the address of the allocated memory, or to NULL if *arith-expr* evaluates to zero or a negative value.

Usage Considerations:

- Requested memory not available

  If the requested amount of memory is not available, *pointer* is set to NULL, and execution continues.

- INITIALIZED specified

  If the optional INITIALIZED phrase is specified, the allocated memory is initialized to binary zeros. If INITIALIZED is omitted, the content of the allocated memory is undefined.

- Compatible with `malloc()` function

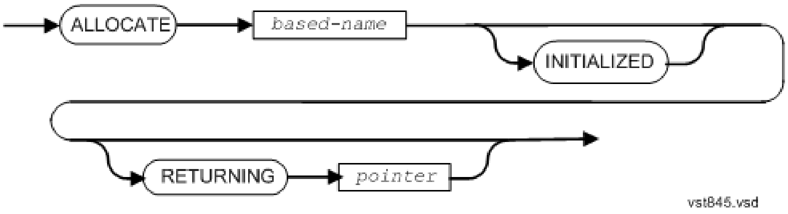
  The ALLOCATE statement is compatible with the C Run-Time Library function `malloc()` and related functions. To release the memory, a non-NULL pointer returned by ALLOCATE can be passed to a C-language routine that calls `free()`.

- Cannot be checkpointed

  Dynamic memory cannot be checkpointed to a backup process.

ALLOCATE Memory for a BASED Item

ALLOCATE with a BASED item allocates dynamic memory for the BASED item. The size of the memory allocated is derived from the size of the item.



`based-name`

is a level-01 or level-77 BASED data item. The number of bytes of memory to allocate is the size of the data item described by *`based-name`*. If the item is a record containing an OCCURS DEPENDING ON clause, the maximum size is used. The implicit pointer associated with *`based-name`* is set to the address of the allocated memory.

`pointer`

is a data item described as USAGE POINTER. *`pointer`* is set to the address of the allocated memory.

Usage Considerations:

- With *`pointer`* specified

  If you specify *`pointer`*, the pointer item is set to the address of the allocated memory, and then the implicit pointer associated with *`based-name`* is set to the same address.

- Sufficient memory is not available

  If sufficient memory is not available, *`pointer`* (if specified) and the implicit pointer associated with *`based-name`* are set to NULL, and execution continues.

- INITIALIZED specified

  The optional INITIALIZED phrase causes the allocated memory to be initialized. If *`based-name`* describes an elementary data item, it is initialized according to one of the following rules. If *`based-name`* describes a record, each elementary subordinate item is initialized according to one of the following rules. The first rule that applies is the rule that is used.

  — If the item has a VALUE clause, it is initialized to the specified value.

  — If the item is described as USAGE POINTER, it is initialized to NULL.

  — If neither of the preceding apply, the item is initialized according to the rules for the INITIALIZE statement with no REPLACING phrase, except that FILLER items are also initialized.

- Compatible with `malloc()` function

  The ALLOCATE statement is compatible with the C Run-Time Library function `malloc()` and related functions. To release the memory, a non-NULL pointer returned by ALLOCATE can be passed to a C-language routine that calls `free()`.

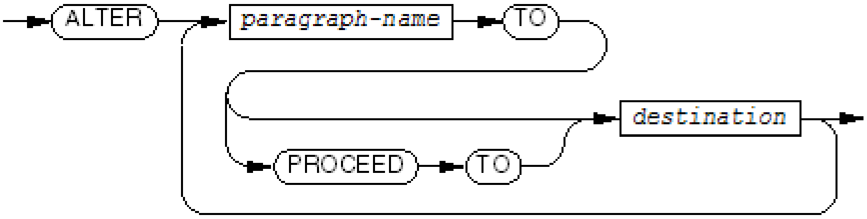
- Dynamic memory cannot be checkpointed

  Dynamic memory cannot be checkpointed to a backup process.

ALTER

NOTE: The 1985 COBOL standard classifies ALTER as **obsolete**, so you are advised not to use it. Instead, use a flag, a conditional GO TO statement, and a MOVE statement.

ALTER changes the destination of a GO TO statement, which can cause maintenance problems.



VST131.vsd

paragraph-name

  is the name of a paragraph that contains only an unconditional GO TO statement.

destination

  is the name of a paragraph or section to which the GO TO statement in *paragraph-name* transfers control.

Usage Considerations:

- Maintenance Problems

  ALTER can cause maintenance problems because it enables a GO TO statement to transfer control anywhere in the program. When you read the source code, the only clue you have that the destination was altered is that the GO TO statement is alone in a paragraph.

- Using a Flag, Conditional GO TO, and MOVE Instead of ALTER

  Suppose that you want to initialize a routine the first time it is called, but not each time it is called. You could either use an ALTER statement as Example 10-6 does (not recommended) or a flag, a conditional GO TO statement, and a MOVE statement as Example 10-7 does (recommended and no less efficient).

Example 10-6 ALTER Statement

```
PROCEDURE DIVISION.
ROUTINE-1.
   GO TO INITIALIZATION-ROUTINE.
INITIALIZATION-ROUTINE.
      ...
         ALTER ROUTINE-1 TO PROCEED TO ROUTINE-2.
   GO TO ROUTINE-2.
ROUTINE-2.
   ...
```

Example 10-7 Alternative to ALTER Statement

```
DATA DIVISION.
   WORKING-STORAGE SECTION.
      77 FLAG   VALUE IS 1.
PROCEDURE DIVISION.
   ROUTINE-1.
      GO TO INITIALIZATION-ROUTINE
            ROUTINE-2
            DEPENDING ON FLAG.
   INITIALIZATION-ROUTINE.
      ...
      MOVE 2 TO FLAG.
      GO TO ROUTINE-2.
   ROUTINE-2.
   ...
```

- `paragraph-name` in an Independent Segment

  If `paragraph-name` is in an independent segment (a section whose segment-number is greater than 49), these restrictions apply:

  - Every ALTER statement that references the `paragraph-name` must be in a section that has the same segment-number as the section containing `paragraph-name`.
  - If the `destination` is in a section that does not have the same segment-number as the section containing `paragraph-name`, these restrictions apply:
    - An ALTER statement in the Declaratives Portion cannot refer to a `paragraph-name` or `destination` in the other portion of the Procedure Division.
    - An ALTER statement cannot be used to establish a potential transfer of control between the Declaratives Portion and the other portion of the Procedure Division.
    - An ALTER statement cannot refer to a `paragraph-name` or `destination` in any debugging declarative procedure unless the ALTER statement itself is in a debugging declarative procedure.
    - An ALTER statement cannot be used to establish a potential transfer of control between a debugging declarative procedure and a nondebugging declarative procedure.

- Canceling the Effects of ALTER Statements

  The effect of an ALTER statement persists until the execution of either:

  - Another ALTER statement with the same `destination`
  - A CANCEL statement in which `program-name` is the program containing the paragraph in which `destination` appears.

CALL

CALL's behavior is determined by the PORT directive. If the program is not compiled with the PORT directive, CALL transfers control from one COBOL program to another COBOL program.

The called program can be in the current compilation unit or can be extracted from an object file, provided that the entry point called is in a code block created by a COBOL compiler on an HP system. To transfer control from a COBOL program to a program written in another language in the absence of the PORT directive, use ENTER (see ENTER).

If the program is compiled with the PORT directive, CALL behaves like the X/Open CALL statement (which is not an element of COBOL). The X/Open CALL statement transfers control from an HP COBOL program to a program written in another language.

NOTE: If you use the X/Open CALL statement in the OSS environment, see Mixed-Language Programs (page 722).

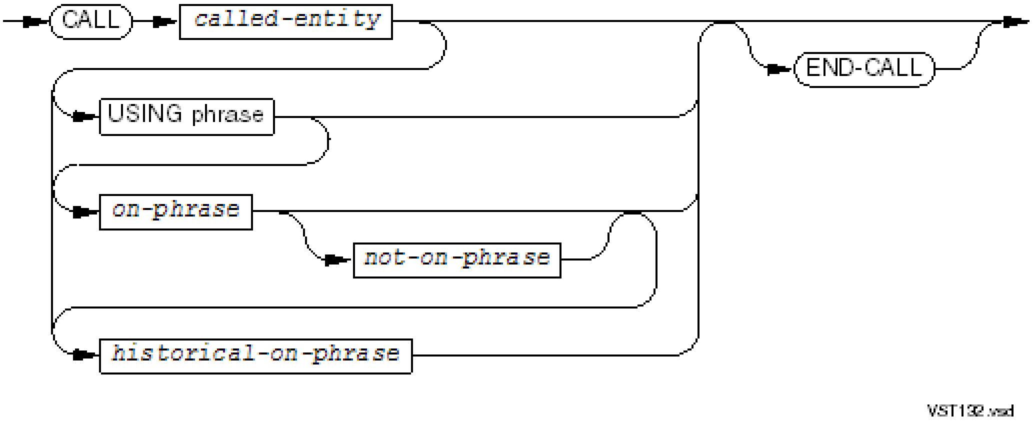
If the called program is not a function, the X/Open CALL statement does not change the value of RETURN-CODE.

If the value of the function is greater than 99,999, arithmetic overflow occurs.

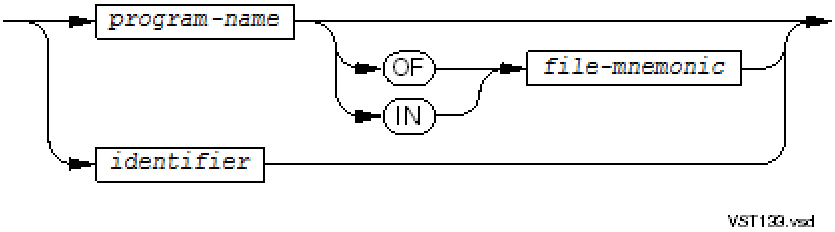
If the called program is not a function, the called program cannot change the value of RETURN-CODE.

If a called program changes the value of RETURN-CODE, the calling program can access that value when control returns to the calling program.

If the program was not compiled with the PORT directive, RETURN-CODE is inaccessible.



`called-entity`



`program-name`

is the `program-name` in the called program's PROGRAM-ID paragraph. It can be expressed as a nonnumeric literal (enclosed in quotation marks) or the actual name (not enclosed in quotation marks). The form enclosed in quotation marks is recommended.

If the program is not compiled with the PORT directive, the called program must be a COBOL program.

If the program is compiled with the PORT directive, the called program can be compiled by any of these TNS/E compilers:

- C
- C++

- ECOBOL
- EpTAL

file-mnemonic

is the alias for the object file to be searched for the called program. The association between *file-mnemonic* and the object file is established by the File-Mnemonic clause of the SPECIAL-NAMES paragraph. The *file-mnemonic* is not part of *program-name* and is not a qualifier that can make *program-name* unique.

file-mnemonic must specify either:

- A linkfile
- An import library
- A DLL
- An archive file

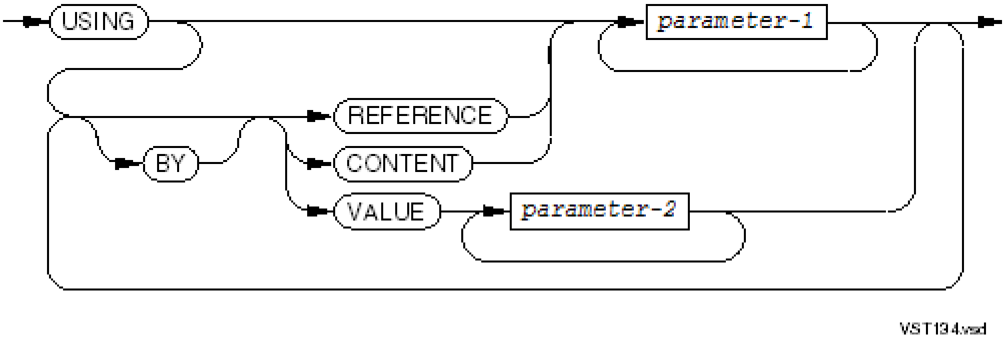
identifier

can be used only if you are calling a COBOL program. It is an alphanumeric identifier whose value, the *program-name* in the called program's PROGRAM-ID paragraph, is not known until run time. The compiler ignores leading and trailing spaces in *program-name* and converts it to uppercase.

You must include the called program in the run unit, usually with a separate bind or link step.

All parameters are passed as extended addresses; therefore, formal parameters in the called program must not specify ACCESS MODE STANDARD.

USING phrase



USING

marks the beginning of the parameter list. The Procedure Division header of the called program must contain a corresponding USING phrase.

REFERENCE

specifies that the called program uses the actual data item of the calling program. If the called program changes the value of the parameter, it changes the value of the data item in the calling program. REFERENCE applies to all the parameters that follow it until a CONTENT or VALUE phrase appears. This is the default.

CONTENT

specifies that the calling program makes a copy of the data item and passes the address of the copy to the called program. If the called program changes the value of the parameter, it does not change the value of the data item in the calling program. CONTENT applies to all the parameters that follow it until a REFERENCE or VALUE phrase appears.

VALUE

specifies that the calling program passes the value of the data item to the called program. Nothing that the called program does with the value affects the data item in the calling program. VALUE applies to all the parameters that follow it until a REFERENCE or CONTENT phrase appears.

If you specify VALUE, the called program must be not be written in COBOL and you must compile the calling program with the PORT directive and either the ENV COMMON or ENV LIBRARY directive.

VALUE is not an element of COBOL. If you also compile the calling program with a FIPS directive with NONSTANDARDEXT in the *flag-option-list*, the compiler issues a warning message if it finds the keyword VALUE.

parameter-1

is a data item defined in the Linkage, File, Working-Storage, or Extended-Storage Section. It must be one of:

- An elementary data item
- A level-01 data item
- A level-77 data item
- A data item whose level is other than 01 or 77 is aligned on a 2-byte boundary

If *parameter-1* is subscripted, its first occurrence must be on a 2-byte boundary and the number of occurrences must be even.

The number of *parameter-1* s must be the same as the number of parameters defined in the USING phrase of the called program. The size of each parameter must be the same as the size of the corresponding parameter in the called program. See Linkage Section (page 191).

If the called program is not written in COBOL, *parameter-1* must be exactly the data type that the called program expects.

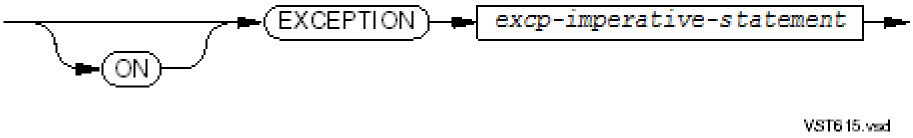
parameter-2

is a data item defined in the Linkage, File, Working-Storage, or Extended-Storage Section. It must be one of these data types:

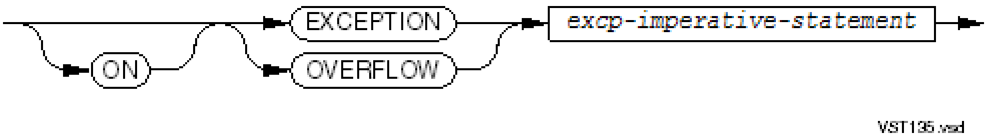
parameter-2	Corresponding Formal Parameter
Numeric data item described as COMP-5 with PICTURE S9(4) or NATIVE-2	16-bit data item
Numeric data item described as COMP-5 with PICTURE S9(9) or NATIVE-4	32-bit data item (1 word)
Numeric data item described as COMP-5 with PICTURE S9(18) or NATIVE-8	64-bit data item (2 words)
1-character alphanumeric data item	Single character

If you are concerned about portability, use COMP-5 rather than NATIVE-2, NATIVE-4, or NATIVE-8.

on-phrase



historical-on-phrase

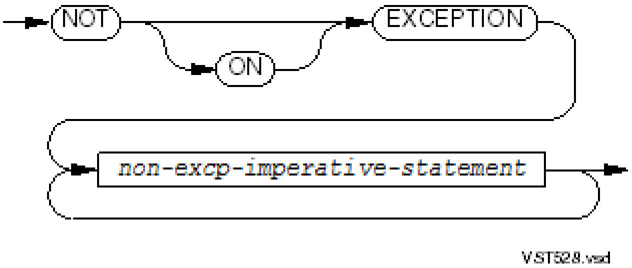


excp-imperative-statement

is an imperative statement to be executed when an exception prevents calling the specified program.

NOTE: Although you can specify *excp-imperative-statement* in any CALL statement, it only works when the called program is a COBOL program. For programs written in languages other than COBOL, it is ignored.

not-on-phrase



non-excp-imperative-statement

is an imperative statement to be executed when the specified program completes its execution and returns control to the program that called it.

END-CALL

ends the scope of the CALL statement and makes it a delimited-scope statement. If you omit END-CALL but include the EXCEPTION or NOT EXCEPTION phrase, the CALL statement is a conditional statement and ends at the next period separator.

Usage Considerations:

- Currently Active Program

  The currently active programThe currently active program, the one containing the CALL statement, is the calling program; the program identified by *program-name* or by the value of *identifier* is the called program.

- Effect of CALL Statement Without PORT Directive

  If the object form of the called program is included in the run unit that contains the calling program, or if the called program is in the user library, execution of the CALL statement transfers control from the calling program to the called program. When the called program executes an EXIT PROGRAM statement or allows control to pass beyond the end of its Procedure Division, control returns to the calling program. If a NOT ON EXCEPTION phrase (*not-on-phrase*) is specified, control passes to *non-excp-imperative-statement* and execution continues according to the rules for that statement. Unless that statement transfers control elsewhere, control passes from that statement to the end of the CALL statement.

  If the object form of the called program is not included in the run unit, the action of the CALL statement depends on the presence or absence of an EXCEPTION phrase (*on-phrase* or *historical-on-phrase*). If an EXCEPTION phrase is specified, control passes to *excp-imperative-statement*. If an EXCEPTION phrase is not specified, the run unit terminates abnormally with a failure message.

- Determining Which Program a CALL *program-name* Statement Calls

  The steps for determining which program the CALL statement calls follow.

  1. The value of *program-name*—the program name—is stripped of leading and trailing spaces and is converted to uppercase. (All COBOL program names are converted to uppercase for the purpose of matching.)
  2. If the program name from Step 1 is that of a COBOL program that was compiled in this compilation unit, then that program is the called program.
  3. If the program name from Step 1 is not that of a program that was compiled in this compilation unit, then the program searches for a program by that name in these places, in this order:
     a. The file associated with *file-mnemonic* (if *file-mnemonic* is specified)
     b. The files on the primary search list (established by the SEARCH directive)
     c. The files on the tertiary search list (established by the CONSULT directive)
     d. For a TNS program, the TNS user library (established by the LIBRARY directive)
     e. The files associated with SEARCH DEFINEs
  4. If the program name from Step 1 is that of a program in one of the preceding files, that program is the called program.
  5. The program name from Step 1 is restored to its original case.
  6. Step 3 is repeated.
  7. The program name from Step 4 is converted back to uppercase and compilation continues. When compilation ends (at the end of the source file), if a COBOL program with the desired name has been compiled, then it is the called program.
  8. If no program is found using the above steps, then the calling program looks for a VALUE phrase. If the VALUE phrase is absent, the calling program assumes that the CALL statement is a COBOL CALL statement. It passes the uppercase name to the `eld` utility and passes all parameters by reference as extended addresses.
  9. If the VALUE phrase is present, the calling program assumes that the CALL statement is an X/Open CALL statement. It passes the name as it was specified in `program-name` to the `eld` utility and passes all parameters as extended addresses—the VALUE parameters by value and all others by reference.
  10. If you are calling a program written in a language other than COBOL, and it has no VALUE parameters, you must make the program stub of the called program available to the compiler using one of the files listed in Step 3.
  11. If the program that *program-name* specifies is not available at execution time, an exception exists.
  12. If you want `rld` to run your program with unresolved externals, you must link your program with either of these `eld` options:
      — `-set rld_unresolved IGNORE`
      — `-set rld_unresolved WARN`

- Two COBOL Programs With the Same Name

  Two or more COBOL programs in a run unit can have the same program name. If this happens, the compilation distinguishes between them according to these scope rules:

  — If program A lacks the COMMON attribute and is directly contained in program B, then only CALL statements in B can call A.

  — If program A has the COMMON attribute and is directly contained in program B, then only CALL statements in B and CALL statements in other programs contained in B (except for A and those contained in A) can call A.

- Linking a Program Called With a CALL `identifier` Statement

  Binding or linking a program called with a CALL `identifier` statement into a run unit is not automatic. Verify that all such programs are bound or linked into the run unit by doing one of:

  — Compile all the program units in a single source file.

  — Use the `eld` utility to link them into the run unit (see the *eld Manual*).

  — Have each required program bound or linked into the run unit by referring to the program explicitly in the source program with a dummy CALL statement and then calling the program by an `identifier`.

  — Use the SEARCH directive to add the program to the primary search list.

- Initial State of Called Programs

  A program that has the INITIAL attribute is in its initial state every time it is called. Programs directly or indirectly contained within a program that has the INITIAL attribute also have the INITIAL attribute.

  A program that does not have the INITIAL attribute is in its initial state the first time it is called, but each successive time it is called, it is usually in the state in which its previous execution left it. This state includes the final value of each internal data item (except those described in the Linkage Section), the final status of each internal file connector, and the final status of each GO TO statement whose destination procedure was specified by an ALTER statement.

  A program that does not have the INITIAL attribute is not in the state in which its previous execution left it if its calling program cancelled it with a CANCEL statement. In this case, the called program is in its initial state the next time it is called. All PERFORM statements in the program are restored to their initial states. All files are closed.

- Storage Allocation

  For programs with the INITIAL attribute, HP COBOL uses dynamic storage allocation, allocating space for data items in the Working-Storage Section each time the program is called and releasing it each time the program is exited. This can cause stack overflow, which can cause the run unit to terminate abnormally.

  For programs without the INITIAL attribute, HP COBOL allocates space at the start of execution of the run unit for all program data except internal data, which it allocates when the program is called. It is unlikely that allocating space for internal data (usually a small amount) will cause stack overflow.

- Passing Parameters

  Parameters passed by the USING phrase are data items from a calling program that the called program can reference.

- Correspondence of Formal and Actual Parameters

  The formal parameters in the USING phrase of the Procedure Division header and the actual parameters in the USING phrase of the CALL statement correspond by position, not by name.

- Declaration Locations and Access Modes of Formal and Actual Parameters

  You must declare formal parameters in the Linkage Section. Every parameter has a 32-bit address. If you specify STANDARD access mode, the ECOBOL compiler ignores it and issues a warning.

- Parameter Validation

  The ECOBOL compiler reports errors if the number of actual parameters (in the CALL statement) differs from the number of formal parameters (in the Procedure Division heading).

The HP COBOL compilers report errors under these conditions:

- The type (alphanumeric, numeric, and so on) of an actual parameter differs from that of its corresponding formal parameter.
- The size of an actual parameter differs from that of its corresponding formal parameter.

- Passing Index Values

  You cannot pass an index-name (defined by the INDEXED phrase of an OCCURS clause) as a parameter because it is not a level 01, level 77, or elementary data item. You can pass an index value in an index data item (defined by a USAGE INDEX clause) as a parameter, but there is no correspondence between the index-name used with the table in the calling program and the index-name used with the table in the called program. Index-names in the called and calling program always refer to separate data areas.

- Recursion

  Although called programs can contain CALL statements, a called program cannot call itself explicitly or implicitly.

- Calling and Called Program Using the Same File

  A called program does not inherit access to an internal file opened by its caller. A calling program and a called program can operate on records of the same file in these ways:

  - One of the programs does all the reading and writing and passes the data items as parameters to the other.
  - Each program opens the file as an internal file with a separate file connector.
  - The file name has the GLOBAL attribute.
  - The file is declared in any program and is given the EXTERNAL attribute. Then the file belongs to the run unit, and any program in the run unit that declares the same file with the EXTERNAL attribute can share in its manipulation.

- Difference Between X/Open CALL Statement and ENTER Statement

  The X/Open CALL statement reports an error if the types of the actual and formal parameters do not match; the ENTER statement attempts to convert the actual parameters into the types of the formal parameters.

Example 10-8 Called Program That Calls Another Program

```
Main Program (source $DATA.MYSUBVOL.MAINSRC, object $DATA.MYSUBVOL.MAINOBJ)
IDENTIFICATION DIVISION.
PROGRAM-ID.  COBOLMAIN.
ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
SOURCE-COMPUTER.  HP System.
OBJECT-COMPUTER.  HP System.
SPECIAL-NAMES.
   FILE "$DATA.MYSUBVOL.SUB1OBJ" IS SUB1.

DATA DIVISION.
WORKING-STORAGE SECTION.
01  DATA-OUT   PIC X(80)  VALUE "I'M THE MAIN AND I'M ALIVE".
77  PARM1      PIC 99     VALUE 10.
77  PARM2      PIC XX    VALUE "AB".
01  MAIN-TABLE.
    05  TABLE-DATA  PIC X
                     OCCURS 10 TIMES
                     INDEXED BY TABLE-INDEX.
01  SAVE-TABLE-INDEX PIC 999.

PROCEDURE DIVISION.
START-PROGRAM.
   DISPLAY DATA-OUT
   SET TABLE-INDEX TO 5
   SET SAVE-TABLE-INDEX TO TABLE-INDEX
   DISPLAY "SAVE-TABLE-INDEX = " SAVE-TABLE-INDEX
   CALL COBSUB1 OF SUB1 USING PARM1
                              PARM2
                              MAIN-TABLE
                              SAVE-TABLE-INDEX
   END-CALL
   DISPLAY "PROGRAM END"
   STOP RUN.

Level 1 Subprogram (source $DATA.MYSUBVOL.SUB1SRC, object $DATA.MYSUBVOL.SUB1OBJ)
IDENTIFICATION DIVISION.
PROGRAM-ID.  COBSUB1.
ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
SOURCE-COMPUTER.  HP System.
OBJECT-COMPUTER.  HP System.
SPECIAL-NAMES.
   FILE "$DATA.MYSUBVOL.SUB2OBJ" IS SUB2.

DATA DIVISION.
WORKING-STORAGE SECTION.
01  DATA-OUT  PIC X(80)  VALUE "I'M COBSUB1 ALIVE AND WELL".
LINKAGE SECTION.
01  PARM1       PIC 99.
01  PARM2       PIC XX.
01  SUBPROGRAM-TABLE.
    05  SUB-TABLE-ENTRY, PIC X
        OCCURS 10 TIMES INDEXED BY SUB-INDEX.
01  SAVE-SUB-INDEX  PIC 999.

PROCEDURE DIVISION USING PARM1,
                         PARM2,
                         SUBPROGRAM-TABLE,
                         SAVE-SUB-INDEX.
START-PROGRAM.
  SET SUB-INDEX TO SAVE-SUB-INDEX
  DISPLAY DATA-OUT
  DISPLAY "SAVE-SUB-INDEX = " SAVE-SUB-INDEX
  DISPLAY PARM1
  DISPLAY PARM2
  CALL "COBSUB2" OF SUB2 USING PARM1, PARM2
  DISPLAY "I'M COBSUB1 AND I'M RETURNING TO MAIN NOW".

Level 2 Subprogram (source $DATA.MYSUBVOL.SUB2SRC, object $DATA.MYSUBVOL.SUB2OBJ)
```

```
IDENTIFICATION DIVISION.
PROGRAM-ID.  COBSUB2.
ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
SOURCE-COMPUTER.  HP System.
OBJECT-COMPUTER.  HP System.

DATA DIVISION.
WORKING-STORAGE SECTION.
77  DATA-MESSAGE PIC X(80) VALUE "I'M COBSUB2 ALIVE AND WELL".
LINKAGE SECTION.
01  PARM1    PIC 99.
01  PARM2    PIC XX.
PROCEDURE DIVISION USING PARM1, PARM2.
START-PROGRAM.
   DISPLAY DATA-MESSAGE
   DISPLAY PARM1
   DISPLAY PARM2
   DISPLAY "I WILL NOW RETURN TO THE MAIN PGM VIA COBSUB1".
```

For the programs in Example 10-8 to work properly, they must be compiled in this order:

```
>COBOL85 /IN SUB2SRC, / SUB2OBJ
>COBOL85 /IN SUB1SRC, / SUB1OBJ
>COBOL85 /IN MAINSRC, / MAINOBJ
```

If you want to eliminate the SPECIAL-NAMES paragraphs from the programs in Example 10-8, compile them, in the same order, with these additional directives:

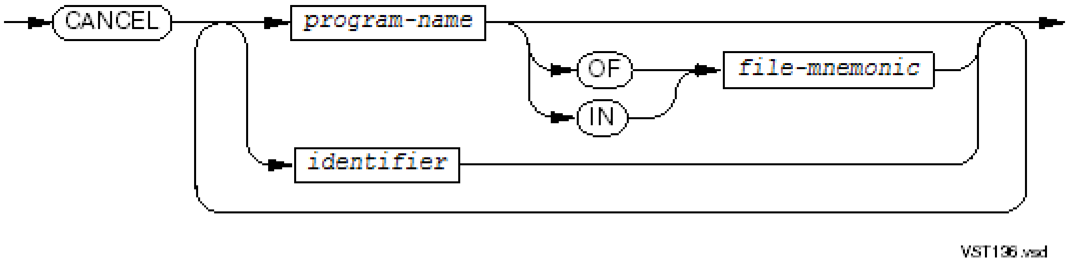
```
In COBSUB1:    ?SEARCH SUBOBJ2
In COBOLMAIN:  ?SEARCH SUBOBJ1
```

Output from running the programs in Example 10-8 as a single run unit looks like this:

```
I'M THE MAIN AND I'M ALIVE
SAVE-TABLE-INDEX = 005
I'M SUBCOB1 ALIVE AND WELL
SAVE-SUB-INDEX = 005
10
AB
I'M COBSUB2 ALIVE AND WELL
10
AB
I WILL NOW RETURN TO THE MAIN PGM VIA COBSUB1
I'M COBSUB1 AND I'M RETURNING TO MAIN NOW
PROGRAM END
```

CANCEL

CANCEL signals that you are done with a program that you called. On some computer systems, CANCEL releases memory used by a program that is no longer needed. On NonStop systems, CANCEL restores each specified program to its initial state (see Initial State (page 604)). Nothing happens if a CANCEL statement specifies an initial program, a program that was not called, or a program that has already been cancelled.



program-name

is a nonnumeric literal whose value is the *program-name* in the PROGRAM-ID paragraph of the COBOL program to be cancelled.

file-mnemonic

specifies the object file containing the program to be canceled. Associate *file-mnemonic* with the operating system file name in the SPECIAL-NAMES paragraph of the Environment Division (see SPECIAL-NAMES Paragraph (page 118)).

identifier

is an alphanumeric, nonnational data item whose value (which is not known until run time) is the *program-name* in the PROGRAM-ID paragraph of the COBOL program to be cancelled.

Usage Considerations:

- Internal Data Items Resume Their Initial Values

  Each internal data item (except those in the Linkage Section) is reset to the value that it had the first time the program was called. (CANCEL does not affect external data items.)

- Internal File Connectors Are Implicitly Closed

  Each internal file connector whose open mode is neither closed nor locked is implicitly closed. The implicit close operation proceeds as if a CLOSE statement with no optional phrases were executed for each of the internal files. (CANCEL does not affect external file connectors.)

  If a called program that does not have the INITIAL attribute opens an internal file, the file remains open until either:

  — The called program closes the file.

  — The calling program cancels the called program.

  — The process terminates.

- PERFORM Statements Resume Their Initial State (Inactive)
- GO TO Statements Modified by ALTER Statements Resume Their Initial Forms

  Each GO TO statement that was modified by an ALTER statement is restored to the form specified in the source program.

- When Not to Use CANCEL

  — On programs compiled with the NOCANCEL directive

  The NOCANCEL directive prevents the compiler from generating code that initializes the program the first time the program is called after being canceled by a CANCEL statement.

  — On programs that are still executing, including the main program

  If you cancel the main program of the run unit or a called program that has not returned control to its calling program, the run unit terminates abnormally with a failure message.

  — On routines written in languages other than COBOL and called with X/Open CALL statement

  Applying a CANCEL statement to a routine written in a language other than COBOL that was called with an X/Open CALL statement terminates the run unit.

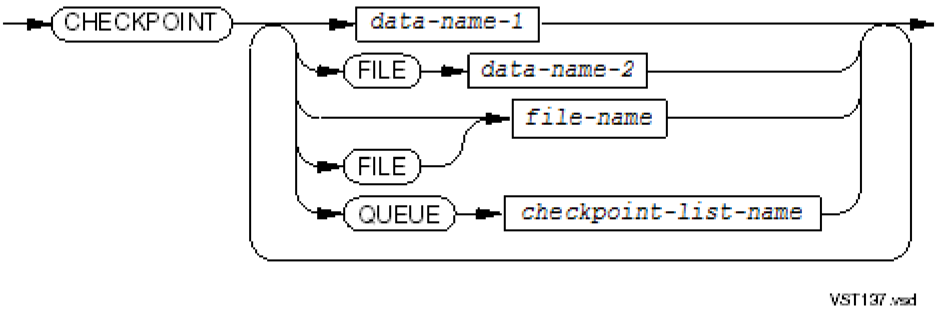
  — Within programs that are to run as process pairs

  If the primary process of a process pair executes a CANCEL statement, the backup process can become invalid.

CHECKPOINT

NOTE: Do not use this directive in the OSS environment.

CHECKPOINT defines a restart point for the backup process of a process pair and transfers the information that the backup requires to restart.



data-name-1

is the name of a data item to checkpoint, typically an item used for an I-O transfer of data that is relevant to the current state of the program and is needed if the backup process is to continue the operation in the event of a failure. *data-name-1* cannot be a BASED item.

FILE

indicates that the next data item is the name or number of a file.

data-name-2

is an integer, the operating system file number of the file whose sync block is to be checkpointed. Use *data-name-2* only when files are managed entirely with ENTER statements that call TAL routines (rather than COBOL I-O statements).

file-name

is the name of a file (usually a disk file) whose sync block is to be checkpointed.

QUEUE

indicates that the next data item is the name of a checkpoint list.

checkpoint-list-name

is the name of a checkpoint list containing information that must be transferred to the backup process. The checkpoint list is a COBOL record with a specific structure in which the message changes are recorded. The information is returned by these Saved Message Utility (SMU) routines:

ALTERPARAMTEXT	DELETESTARTUP	PUTPARAMTEXT
DELETEASSIGN	PUTASSIGNTEXT	PUTSTARTUPTEXT
DELETEPARAM	PUTASSIGNVALUE	

For information on SMU routines and a description of the checkpoint list record, see Saved Message Utility (SMU) Overview (page 619).

Usage Considerations:

- Reason to Checkpoint

  If a backup process takes over, the contents of items that are not checkpointed are undefined.

- Conditions Under Which Data Items Are Checkpointed

  The compiler checkpoints only those data items that one or more CHECKPOINT statements specify explicitly. These data items can be in either the Working-Storage Section or the Extended-Storage Section.

- Sending Values to a Backup Process

  When CHECKPOINT executes, the values of the data items and file sync blocks are sent to the fault-tolerant facility in the backup process. If the primary process fails before another checkpoint is made, the backup begins processing from the current checkpoint. The set of values specified in any checkpoint must be sufficient for the backup process to continue processing correctly with only that information.

  After the CHECKPOINT statement transfers the information in the checkpoint list to the backup process, it resets the checkpoint list to empty and its storage space is available to record further message changes.

- Multiple Checkpoints for a Transaction

  Use multiple checkpoints for a transaction if the number of WRITE statements executed for a file exceeds the value of the sync depth with which the file was opened. If a file was opened with sync depth *n*, use at least one CHECKPOINT for every *n* WRITE statements executed for a file.

- CHECKPOINT for Process Pairs

  When a requester that is running as a process pair sends a message, and the backup process takes over before the requester receives a reply, the server can resend the reply automatically—if a CHECKPOINT statement executes after the server reads the requester's message but before the server writes a reply (or generates one automatically with another READ). When this happens, the server does not see the duplicate message. The CHECKPOINT statement can execute even if the server is not running as a process pair.

- A Process Pair Cannot Update an SQL/MP or SQL/MX Database

  An HP COBOL program that runs as a process pair cannot update an SQL/MP or SQL/MX database.

NOTE: A COBOL program must not modify the contents of a checkpoint list directly. The execution logic of the routines and the CHECKPOINT statement maintain checkpoint lists without the need for any other program action.

CLOSE

CLOSE terminates processing of one or more open files or reels of tape. An optional LOCK phrase prevents the program from reopening the file (unless the file is dynamically assignable—see #DYNAMIC (page 852)).

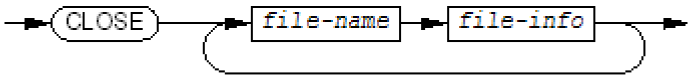
If a file is closed but not locked, the only I-O operation the process can perform on that file is an open operation. If the file is closed and locked, the process cannot perform any I-O operation on the file.

CLOSE has these formats:

- CLOSE for Sequential and Line Sequential Files
- CLOSE for Relative, Indexed, and Queue Files

CLOSE for Sequential and Line Sequential Files

Throughout this topic, "sequential file" means both sequential and line sequential files unless otherwise noted.



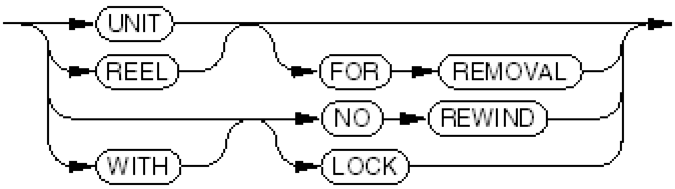
VST138.vsd

file-name

is the name of the sequential file to be closed.

When more than one *file-name* appears, the files can have different organization and access modes and the optional phrases following one *file-name* are independent of those following any other *file-name*.

file-info



VST139.vsd

UNIT

REEL

specify that the current reel is to be closed and rewound, and a new reel is to be mounted.

REMOVAL

specifies that the reel of a multiple-reel tape file be rewound and unloaded, and a new reel is to be mounted.

NO REWIND

specifies that the sequential file is to be left in its current position, not rewound.

LOCK

means the file associated with *file-name* cannot be opened again during the current run. If the file is dynamically assignable, the LOCK phrase has no effect (see #DYNAMIC (page 852)).

Usage Considerations:

- Effects of CLOSE

  For each *file-name* in the CLOSE statement, the run-time routines perform an appropriate close operation. These close operations occur as if separate CLOSE statements were executed for each *file-name* in the order listed. Subsequent usage considerations describe actions taken during the close operation for one file.

- File-Status Data Item

  If the file being closed has an associated file-status data item, the CLOSE statement assigns it an appropriate I-O status code. The possible I-O status codes and their meanings are:

I-O Status Code	Meaning
“00”	The close operation completed successfully.
“07”	The close operation completed successfully, but the CLOSE statement included a NO REWIND, REEL, UNIT, or REMOVAL phrase and the designated file does not reside on a reel or unit medium.

I-O Status Code	Meaning
“30”	The close operation failed due to causes outside of COBOL. The file is not closed. If the operation was a reel close, the reel might not have been closed properly.
“42”	The program attempted to close a file that was not open.

Failure of a close operation can terminate the run unit. When a run unit terminates for any reason (normal or abnormal), the run-time routines implicitly close any open files as if they were specified in a CLOSE statement without optional phrases.

- Closing Single-Tape and Multiple-Tape Files

  See Table 10-1.

- Closing a File Open Under More Than One Name

  If a program has one operating system file open under more than one file name, closing one file name does not affect the availability of the operating system file through any other file name, except when the LOCK phrase is present. The LOCK phrase applies to all file names that the program has assigned to the operating system file except those assigned dynamically with the COBOLASSIGN, COBOL_ASSIGN_, or COBOL_ASSIGN_OSS_ routine.

Table 10-1 CLOSE Statements for Sequential Tape File

CLOSE Statement	Single Reel	Multiple Reel
CLOSE	See note 5	See notes 1 & 5
CLOSE NO REWIND	See note 2	See notes 1 & 2
CLOSE LOCK	See note 3	See notes 1 & 3
CLOSE REEL	See note *	See notes 4 & 5
CLOSE REEL FOR REMOVAL	See note *	See notes 3, 4, & 5
CLOSE REEL NO REWIND	See note *	See notes 2 & 4
1. OTHER REELS UNAFFECTED	If more reels remain in the file, they are not processed.	
2. NO REWIND	The reel is left in its current position.	
3. REEL REMOVAL	The current reel is rewound and unloaded.	
4. REEL SWAP	When proceeding to the next reel in a multiple-reel set, a message is written to the home terminal and a read is done. If the reply is a carriage return, the new reel is assumed to be on the same device; otherwise, the reply must be a device name that has the next reel in the set.	
5. REEL REWOUND	The reel is rewound.	
* If the program issues a CLOSE REEL of any form for a file that is open for INPUT, and the operator limits the file to the current reel by responding “NO” to the reel swap prompt, the at-end condition occurs for that file. When an operator responds “NO” for a file that is open for other than INPUT, the run-time routine rejects the “NO” answer.		

- Closing Blocked Files

  The close operation RVUs an incomplete block to the file if all of these conditions are true:

  — The file is declared with a BLOCK CONTAINS clause specifying more than one record for each block.

  — The current block contains at least one record but is not yet filled.

  — The file’s open mode is OUTPUT or EXTEND.

If this RVU causes a boundary violation exception condition, the results are:

- The block is not transmitted to the file.
- The close operation terminates immediately, setting any file-status variable associated with the file to 34 (for a sequential file) or 24 (for a relative or indexed file).
- Any applicable USE procedure executes.
- The file is not closed.

- COBOL and FUP Closing Procedures Are Incompatible (Multiple-Tape File)

  When the run-time routine closes a nonfinal reel of a multiple-tape file, it writes an end-of-file mark, a trash record, and an end-of-file mark. When the File Utility Program (FUP) closes a multiple-reel tape file, it writes only the two consecutive end-of-file marks. Because these closing procedures differ, you must not use FUP to copy tapes written by and for COBOL programs. Instead, write a simple COBOL program to copy such tapes.

- Process Pairs

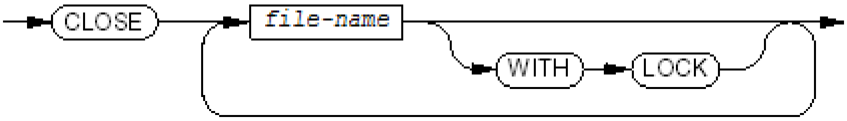
  When the run unit is executing as a process pair, the execution of a CLOSE *file-name* also executes an implied statement of this form:

  ```
  CHECKPOINT FILE file-name
  ```

- Repositioning a Sequential File to Its Beginning

  Although you can reposition a sequential file to its beginning by closing it and reopening it, you can do it faster with the routine COBOL_REWIND_SEQUENTIAL_. For information on these routines, see COBOL_REWIND_SEQUENTIAL_ (page 628).

CLOSE for Relative, Indexed, and Queue Files



VST140.vsd

file-name

is the name of the relative,indexed, or queue file to be closed.

When more than one *file-name* appears, the files can have different organization and access modes and the optional phrases following one *file-name* are independent of those following any other *file-name*.

LOCK

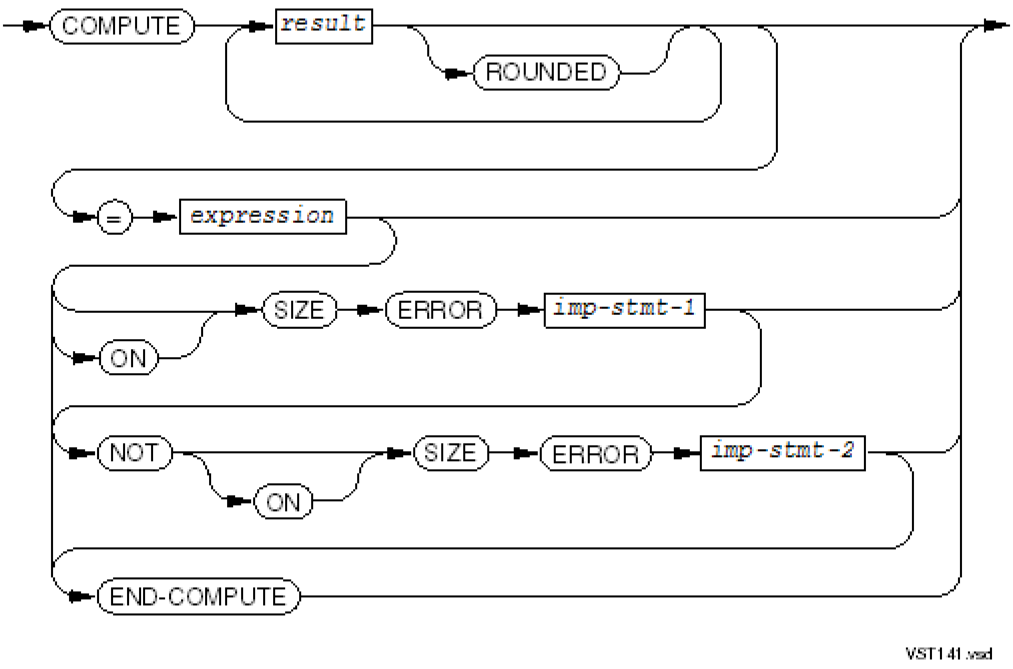
means the file associated with *file-name* cannot be opened again during the current run. If the file is dynamically assignable, the LOCK phrase has no effect (see #DYNAMIC (page 852)).

See these usage considerations in CLOSE for Sequential and Line Sequential Files:

- Effects of CLOSE
- File-Status Data Item
- Closing a File Open Under More Than One Name
- Closing Blocked Files
- Process Pairs

COMPUTE

COMPUTE evaluates an arithmetic expression and stores the result in the specified data item or items.



result

is the identifier of a numeric elementary item or numeric edited elementary item where the result of the computation is to be stored.

ROUNDED

specifies that the value is rounded before being stored.

expression

is an arithmetic expression.

imp-stmt-1

is an imperative statement to be executed when a size error has been detected in the computation or in storing the result.

imp-stmt-2

is an imperative statement to be executed when no size error is detected in the computation or in storing the result.

END-COMPUTE

ends the scope of the COMPUTE statement, causing the COMPUTE to be a delimited-scope statement. If the COMPUTE statement does not end with an END-COMPUTE phrase, the presence of the SIZE ERROR or the NOT SIZE ERROR phrase causes the COMPUTE statement to be a conditional statement, which ends at the next period separator.

Usage Considerations:

- See Arithmetic Operations (page 267) for information on data conversion and alignment, intermediate results, multiple results, and incompatible data.
- See Arithmetic Precision (page 272) for information on precision of addition.
- See ROUNDED Phrase (page 254) and SIZE ERROR Phrase (page 254) for information on these phrases.

For Example 10-9 and Example 10-10, these descriptions are in the Working-Storage Section of a program:

Example 10-9 COMPUTE Statement

```
WORKING-STORAGE SECTION.
01  COMPUTE-RESULT      PIC 999     VALUE ZEROS.
01  DIAGNOSTIC-FIELD    PIC X(35).
01  WS-RESULT           PIC S9(9)   VALUE ZEROS.
01  WS-99               PIC S99     VALUE 99.
01  WS-FIVE-ONES        PIC S9(5)   VALUE 11111.
01  EXPONENT            PIC 9(5)    VALUE ZERO  COMP.
01  A                   PIC 9(4)V99.
01  B                   PIC 9(4)V99 VALUE 8.
01  C                   PIC 9(4)V99 VALUE 5.
01  U                   PIC 9(4)V99 VALUE 7.
01  X                   PIC 9V99.
```

This statement specifies that the result be stored in COMPUTE-RESULT without being rounded:

```
COMPUTE COMPUTE-RESULT =
   (((24.0 + 1) * (60 - 10)) / 125) ** 2
END-COMPUTE
```

(The result is 100.)

This COMPUTE statement specifies that the result be rounded and then stored in WS-RESULT:

```
MOVE 2 TO EXPONENT
COMPUTE WS-RESULT ROUNDED =
  WS-99 / 10 * WS-99 ** EXPONENT + WS-FIVE-ONES
END-COMPUTE
```

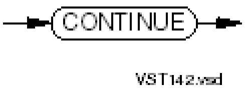
(The result is 108,141.)

Example 10-10 Combination of IF and COMPUTE Statements

```
IF A > 0
   COMPUTE X = B ** 2 - ( 4 * A * C ) / ( 2 * A)
   ON SIZE ERROR
      MOVE "DIVISION ERROR" TO DIAGNOSTIC-FIELD
   END-COMPUTE
ELSE
   MOVE "DIVISION BY ZERO" TO DIAGNOSTIC-FIELD
END-IF.
```

CONTINUE

CONTINUE is a no-operation statement.



Example 10-11 CONTINUE Statement

```
IF SALARIED
   IF ANNUAL-SAL > 50000
      IF BELOW-QUOTA
         CONTINUE
      ELSE
         PERFORM ADD-BONUS-TO-OVERPAID-SALESPERSON
      END-IF
   ELSE
      IF BELOW-QUOTA
         CONTINUE
      ELSE
         PERFORM ADD-BONUS-TO-FLUNKY
      END-IF
ELSE
   IF OVERTIME
   ...
```

Usage Considerations:

- Effect of CONTINUE

  A CONTINUE statement has no effect on the execution of the program.

- Where CONTINUE Can Appear

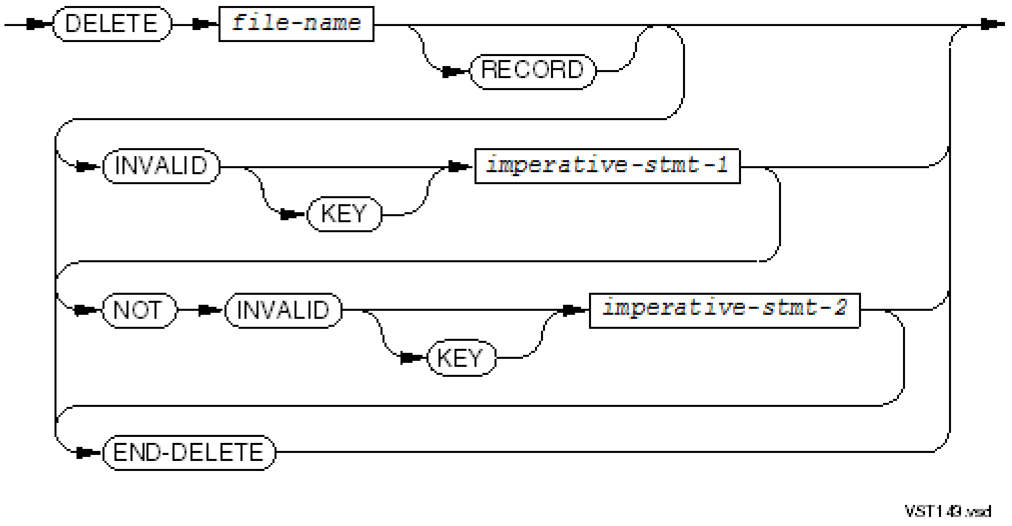
  The CONTINUE statement can appear anywhere a conditional statement or an imperative statement can appear. The normal use is as an unused branch of an IF statement.

COPY

COPY summons source text from a file set up as a COPY library. You can use COPY in any division. For more information, see COPY Statement (page 507).

DELETE

DELETE removes a record from a relative or indexed file that is open in I-O mode.



file-name

is the name of a relative or indexed file that is open in I-O mode.

imperative-stmt-1

is an imperative statement to be executed when the invalid-key condition arises during the delete operation. It is required for a file whose access mode is random or dynamic (but prohibited for a file whose access mode is sequential) when there is no USE procedure that applies to the file.

`imperative-stmt-2`

is an imperative statement to be executed (after a declarative procedure) when no exception or an exception other than the invalid-key condition arises during the delete operation.

END-DELETE

ends the scope of the DELETE statement and makes it a delimited-scope statement. If you omit END-DELETE but include the INVALID KEY or NOT INVALID KEY phrase, the DELETE statement is a conditional statement and ends at the next period separator.

Usage Considerations:

- I-O Status Codes

  If the file being closed has an associated file-status data item, the DELETE statement assigns it an appropriate I-O status code. The possible I-O status codes and their meanings are:

I-O Status Code	Meaning
“00”	The delete operation was successful.
“23”	The designated record does not exist.
“30”	The delete operation failed due to causes outside of COBOL. The specified record might or might not have been deleted. The file position indicator might be undefined.
“43”	The delete operation failed because the last input-output statement executed for the file (which is in the sequential access mode) was not a successfully executed READ statement.
“49”	The file was not open in the I-O mode or was not opened by an HP COBOL program.

  When the deletion is successful or the invalid-key condition occurs, these items are not affected:

  — Record area

  — Data item specified in the DEPENDING phrase of the RECORD clause, specifying the length of the record

  — Key of reference

  — File position indicator

- Record Deleted When File Access Mode is Sequential

  If the file’s access mode is sequential and the last operation performed on the file was a successful READ statement, then DELETE logically removes from the file the record that the READ statement retrieved. If the last operation performed on the file was not a successful READ statement, DELETE terminates with I-O status code “43.”

  In a Pathway environment under the NonStop Transaction Management Facility (TMF), the last operation on the file must have been a successful execution of a READ LOCK statement (rather than a READ statement). For information on coding servers, see the *Pathway/TS SCREEN COBOL Reference Manual* for your system.

  The record retrieved by the READ statement is deleted.

- Record Deleted When File Access Mode is Random or Dynamic

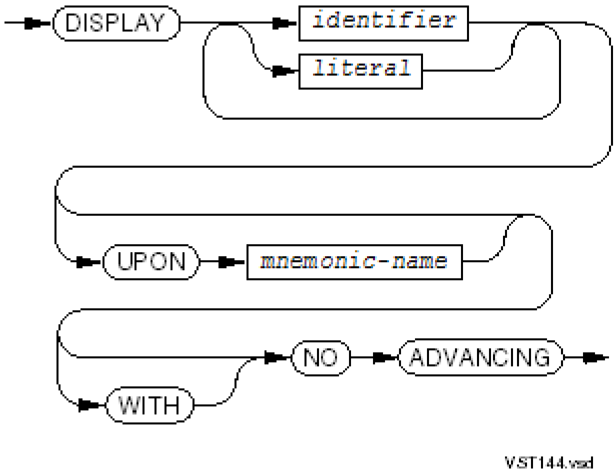
  If the file’s access mode is random or dynamic, the record to be deleted is defined by the record key, or, for relative files, the relative key (indexed or prime key). If the designated record does not exist, an invalid-key condition occurs with I-O status code “23.”

Example 10-12 DELETE Statement

```
INPUT-OUTPUT SECTION.
FILE-CONTROL.
   SELECT EMPLOYEE-MASTER ASSIGN TO "EMPMAST"
      ORGANIZATION IS INDEXED
      ACCESS MODE IS DYNAMIC
      RECORD KEY IS EMPLOYEE-NUMBER
      FILE STATUS IS FILE-STAT.
      ...
FD EMPLOYEE-MASTER
    LABEL RECORDS ARE OMITTED
      ...
    05  EMPLOYEE-NUMBER  PIC X(7).
      ...
PROCEDURE DIVISION.
      ...
DELETE-RECORD.
    MOVE KEY-TO-DELETE TO EMPLOYEE-NUMBER
    DELETE EMPLOYEE-MASTER RECORD
       INVALID KEY PERFORM KEY-ERROR
END-DELETE
```

DISPLAY

DISPLAY delivers a small amount of data (such as an error message) to a terminal, a printer, or another process.



identifier

is the identifier of any data item except an index data item.

literal

is a literal or figurative constant name.

mnemonic-name

is the device on which the process displays the data. You must define *mnemonic-name* in the SPECIAL-NAMES paragraph of the Environment Division. When you omit *mnemonic-name*, the process delivers the data to its output file, typically its home terminal.

NO ADVANCING

specifies that the device to which the display is directed is not reset to the next line after the data is displayed.

The DISPLAY statement in Example 10-3 delivers an error message to the output file of a process

Example 10-13 DISPLAY Statement

```
IF IO-STATUS = "23"
   DISPLAY "I-O ERROR " IO-STATUS
      " - NO RECORD FOR KEY = " INVOICE-NUMBER
END-IF
```

If INVOICE-NUMBER is 00246, the output from the code in Example 10-3 looks like this:

```
I-O ERROR 23 - NO RECORD FOR KEY = 00246
```

Usage Considerations:

- Differences in the OSS and Guardian Environments

  In the OSS environment (but not in the Guardian environment), if a DISPLAY statement includes *mnemonic-name*, it must be either the OSS pathname of a Guardian file or the name of an OSS text file.

- Devices

  The devices to which the DISPLAY statement can transmit data are terminals (including the operator's console), printers, processes (including spooler collectors), and entry-sequences files. Once the DISPLAY device is assigned, the program cannot change it. The WRITE statement is recommended for all but the smallest amounts of data.

- Specifying the Wrong Device

  If you specify a device that cannot be used for DISPLAY output, the process displays a run-time error message and the output on its home terminal. The error message includes this line:

  ```
  Device assigned to ACCEPT or DISPLAY not a legal device
  ```

- Opening and Closing Devices

  The display operation opens and closes the terminal or printer (but not a process) for each DISPLAY statement. This prevents a COBOL program from locking out other users between displays. Because a process is not prevented from accepting requests from other run units, COBOL does not close a process after each display operation; instead, it waits until execution of the run unit terminates.

- How Different Types of Values Are Displayed

  — Numeric values

    A numeric value is displayed as a numeric literal, preceded by a minus sign if the value is negative, and with a decimal point before any fractional digits. The number of digits displayed is the number of digits defined for the sending item.

    A function that returns a floating-point value (such as NUMVAL or NUMVAL-C) is displayed as if the value were described as PICTURE S9(18), with no decimal positions. For this reason, do not reference such functions in DISPLAY statements. Instead, move the values that such functions return to temporary variables that have decimal points in the appropriate positions and display the temporary variables.

  — Nonnumeric values

    A nonnumeric value is displayed as a string of characters.

  — Figurative constant values

    A figurative constant is displayed as a single occurrence of the constant value (even if the figurative constant includes the prefix ALL).

  — Multiple values

    When you specify more than one value in a DISPLAY statement, DISPLAY combines the values (in the order you specify them) into a single character-string and displays the character-string. The size of the displayed item is the sum of the sizes of all the values.

If the character-string exceeds the line capacity of the device, DISPLAY displays the data as a sequence of lines. Every line except (perhaps) the last one contains its maximum number of characters. The last line contains whatever characters are left, up to the maximum.

- Displaying Records and Large Amounts of Data

  Use the WRITE statement to display records and character-strings longer than several lines. The DISPLAY statement is not designed to deliver logical records or large amounts of data to a device.

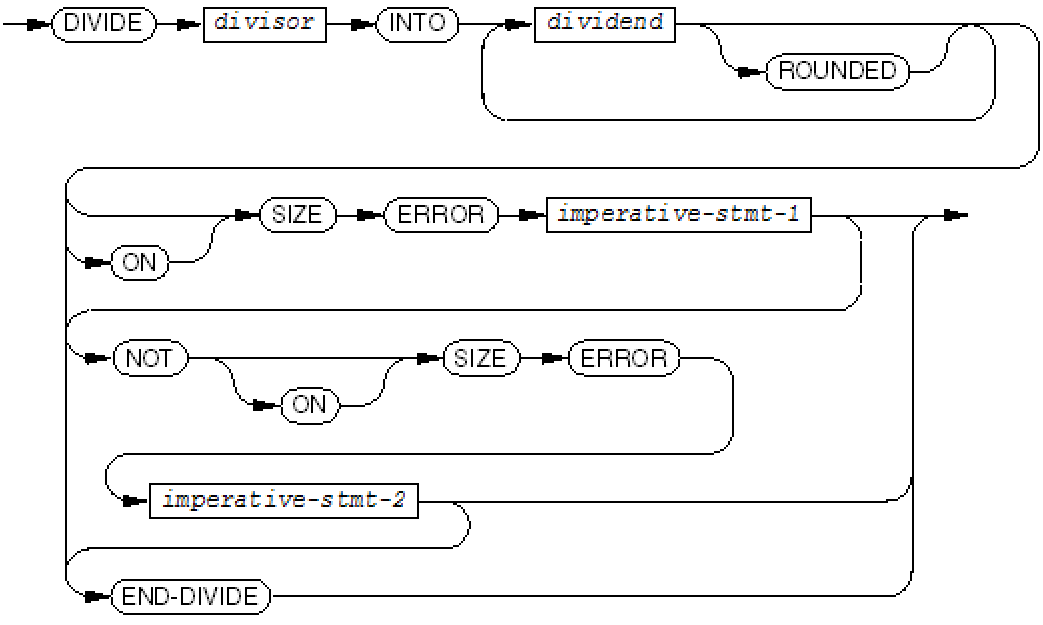
- NO ADVANCING Phrase

  If you include the NO ADVANCING phrase, the device is not changed after it displays the final operand (for example, the device is not repositioned to the next line). If the device is a printer or a spooler process, this causes the next line displayed to overprint the current line (useful for underlining). If the device is a terminal, the cursor appears after the last character displayed (useful for issuing a prompt and accepting the response on the same line).

DIVIDE

DIVIDE INTO

DIVIDE INTO divides one data item into one or more other data items and stores the quotient(s) in the respective dividend data item(s).



VST145.vsd

divisor

is the identifier of the elementary numeric data item or numeric literal that is the divisor.

dividend

is the identifier of an elementary numeric data item that is both the dividend and the receiver of the quotient.

ROUNDED

specifies that quotient is to be rounded before being stored.

`imperative-stmt-1`

is an imperative statement for the process to execute if it detects a size error in the division or in storing the result.

`imperative-stmt-2`

is an imperative statement for the process to execute if it does not detect a size error in the division or in storing the result.

END-DIVIDE

ends the scope of the DIVIDE statement and makes it a delimited-scope statement. If you omit END-DIVIDE but include the SIZE ERROR or NOT SIZE ERROR phrase, the DIVIDE statement is a conditional statement and ends at the next period separator.

Usage Considerations:

- Mathematics

  The statement

  ```
  DIVIDE A INTO B
  ```

  means store B /A in B.

- Precision

  For information on the precision of HP COBOL division, see Arithmetic Precision (page 272).

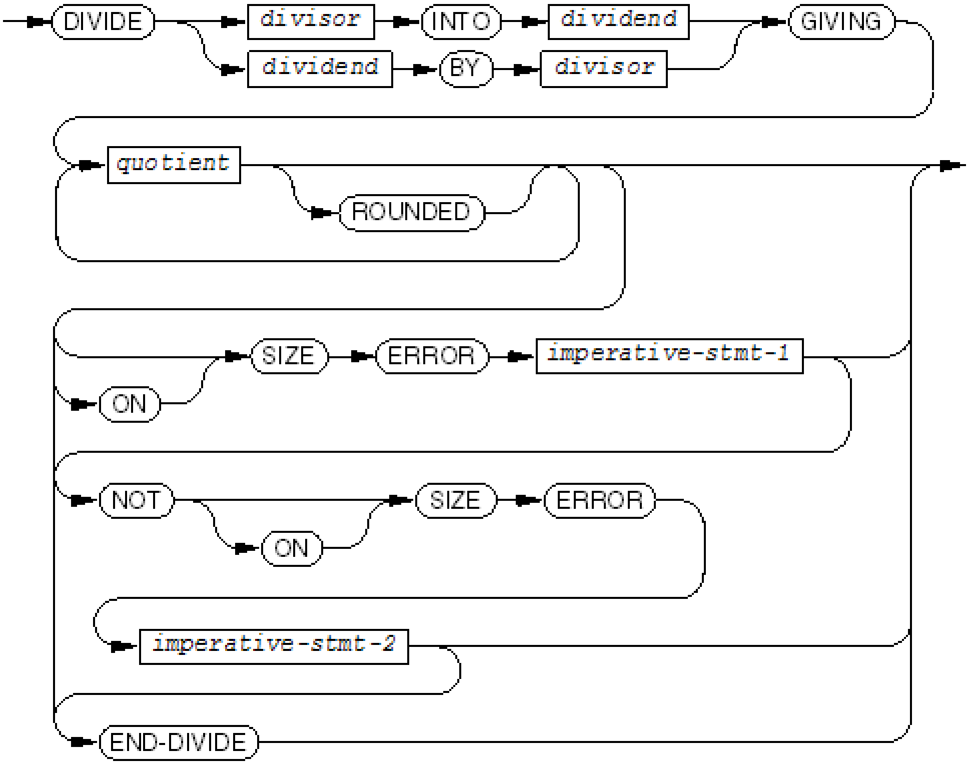
- Order of Evaluation

  The value of *divisor* is copied into a temporary data item for computation. The value of each *dividend* is divided by the value of the temporary data item and the value of the result is stored (after rounding, if specified) into *dividend*. If the same data name occurs more than once in the list of dividends, the final value of the item reflects multiple divisions by the value of *divisor*.

  Each time the result is stored in a *dividend*, the process of operand identification (including subscript evaluation and reference modification) occurs just before the storage operation.

DIVIDE GIVING

DIVIDE GIVING divides one data item into another data item and stores the quotient in one or more data items.



divisor

is the identifier of the elementary numeric data item or numeric literal that is the divisor.

dividend

is the identifier of an elementary numeric data item that is the dividend.

quotient

is the identifier of the elementary numeric item or elementary numeric edited data item in which the quotient is to be stored.

ROUNDED

specifies that quotient is to be rounded before being stored.

imperative-stmt-1

is an imperative statement for the process to execute if it detects a size error in the division or in storing the result.

imperative-stmt-2

is an imperative statement for the process to execute if it does not detect a size error in the division or in storing the result.

END-DIVIDE

ends the scope of the DIVIDE statement and makes it a delimited-scope statement. If you omit END-DIVIDE but include the SIZE ERROR or NOT SIZE ERROR phrase, the DIVIDE statement is a conditional statement and ends at the next period separator.

Usage Considerations:

- Mathematics

  The statement

  ```
  DIVIDE A INTO B GIVING C
  ```

  means store B /A in C.

  The statement

  ```
  DIVIDE A BY B GIVING C
  ```

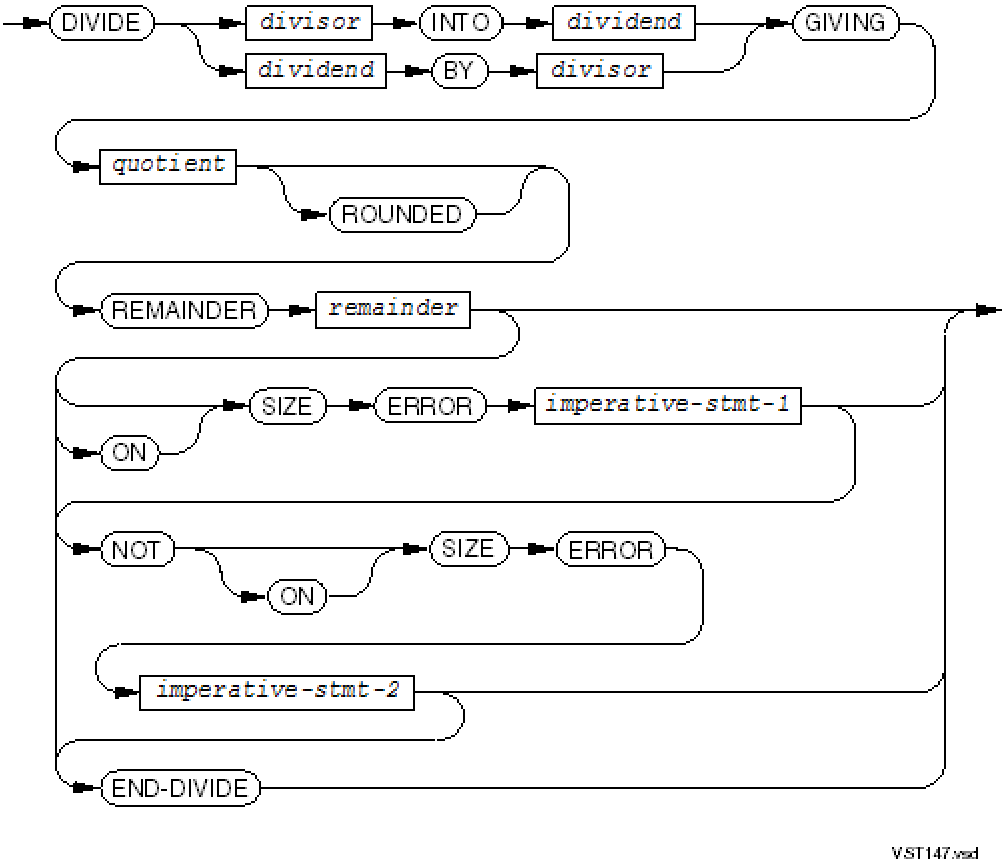
means store A /B into C.

- Precision

  For information on the precision of HP COBOL division, see Arithmetic Precision (page 272).

DIVIDE GIVING REMAINDER

DIVIDE GIVING REMAINDER divides one data item into another data item and stores the quotient and remainder in specified data items.



`divisor`
: is the identifier of the elementary numeric data item or numeric literal that is the divisor.

`dividend`
: is the identifier of an elementary numeric data item that is the dividend.

`quotient`
: is the identifier of the elementary numeric item or elementary numeric edited data item in which the quotient is to be stored.

ROUNDED
: specifies that quotient is to be rounded before being stored.

`remainder`
: is the identifier of an elementary numeric or elementary numeric edited data item where the remainder is stored.

`imperative-stmt-1`
: is an imperative statement for the process to execute if it detects a size error in the division or in storing the result.

imperative-stmt-2

is an imperative statement for the process to execute if it does not detect a size error in the division or in storing the result.

END-DIVIDE

ends the scope of the DIVIDE statement and makes it a delimited-scope statement. If you omit END-DIVIDE but include the SIZE ERROR or NOT SIZE ERROR phrase, the DIVIDE statement is a conditional statement and ends at the next period separator.

Usage Considerations:

- Mathematics

  The statement

  ```
  DIVIDE A INTO B GIVING C REMAINDER D
  ```

  means store the quotient portion of the value of B /A in C and store the remainder in D.

  The statement

  ```
  DIVIDE A BY B GIVING C REMAINDER D
  ```

  means store the quotient portion of the value of A /B in C, and store the remainder in D.

- Precision

  For a discussion of the precision of HP COBOL division, see Arithmetic Precision (page 272).

- Remainder

  HP COBOL computes the remainder by performing the division with a signed quotient having the same number of decimal places as the quotient. For example, dividing 2 into -5 yields a remainder of -1. This differs from COBOL, in which the intermediate value has the same sign as the quotient.

Example 10-14 DIVIDE INTO Statement With GIVING and REMAINDER Phrases

```
WORKING-STORAGE SECTION.
01  ARITHMETIC-WORK-SPACE.
    03  LEAP-YEAR       PIC 9   VALUE ZERO.
    03  DIVIDE-RESULT   PIC 99  VALUE ZERO.
    ...
01  INVOICE-DATE.
    05  INV-MONTH    PIC 99.
    05  INV-DAY      PIC 99.
    05  INV-YEAR     PIC 9999.
    ...
PROCEDURE DIVISION.
     ...
    DIVIDE 4 INTO INV-YEAR GIVING DIVIDE-RESULT
                           REMAINDER LEAP-YEAR
    END-DIVIDE
    ...
```

Example 10-15 DIVIDE BY Statement With GIVING and REMAINDER Phrases

```
WORKING-STORAGE SECTION.
01  ARITHMETIC-WORK-SPACE.
    03  LEAP-YEAR       PIC 9   VALUE ZERO.
    03  DIVIDE-RESULT   PIC 99  VALUE ZERO.
    ...
01  INVOICE-DATE.
    05  INV-MONTH    PIC 99.
    05  INV-DAY      PIC 99.
    05  INV-YEAR     PIC 9999.
    ...
 PROCEDURE DIVISION.
    ...
    DIVIDE INV-YEAR BY 4 GIVING DIVIDE-RESULT
                         REMAINDER LEAP-YEAR
    END-DIVIDE
    ...
```

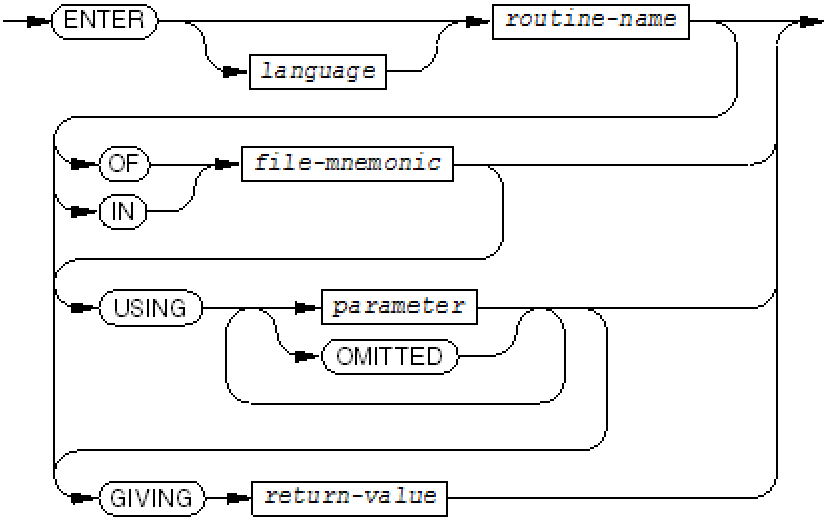
ENTER

ENTER calls a routine written in a language other than COBOL. (To call a COBOL routine, use CALL.)

NOTE: If you use the ENTER statement in the OSS environment, see Mixed-Language Programs (page 722).

In some implementations of COBOL, ENTER marks the beginning of an embedded routine in some other language.

In HP COBOL, ENTER is analogous to CALL but is used to call a non-COBOL routine. An HP COBOL program's mode and operating environment determine what types of non-COBOL routines it can call. A called routine can have an ordinary, VARIABLE, or EXTENSIBLE parameter list.



VST148.vsd

language

is unnecessary, because the compiler can determine the language of the called program, but if specified, it must be C or TAL. If *language* is TAL, the compiler expects a pTAL program. The compiler does not accept TAL programs.

routine-name

is either the actual name of the called routine or it is a nonnumeric literal whose value is the name of the called routine.

If *routine-name* is a nonnumeric literal or a COBOL reserved word, or if it contains a caret (^) or underscore (_), it must be enclosed in quotation marks.

If *routine-name* is not enclosed in quotation marks, the compiler handles it as the actual name of the routine. Quotation marks are recommended.

If *routine-name* specifies a C or C++ function whose name includes lowercase letters, the call fails, because the compiler automatically converts all names to uppercase.

file-mnemonic

specifies the object file containing the called routine's code and data blocks, enabling the `eld` utility to find the routine during preparation of the run unit.

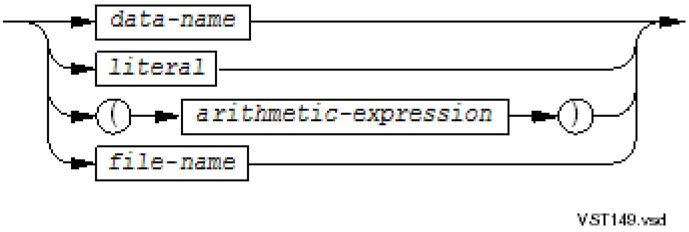
file-mnemonic must specify either:

- A linkfile
- An import library
- A DLL
- An archive file

Establish the association between *file-mnemonic* and the operating system file name in the SPECIAL-NAMES paragraph of the Environment Division.

The *file-mnemonic* phrase is not part of the *routine-name*. You cannot use the phrase to make the *routine-name* unique.

parameter



is a value to be passed to the called routine.

data-name

can be qualified, subscripted, and include a reference modifier. If *data-name* is a TAL or pTAL *string*:*length* parameter, you can set its length with a reference modifier (see Restrictions on Calling pTAL Routines.).

literal

is a numeric literal whose value corresponds to a value parameter in the routine being called.

arithmetic-expression

is an arithmetic expression whose value corresponds to a value parameter in the routine being called.

file-name

is the file description name of a file.

OMITTED

must be specified in place of any omitted parameter surrounded by other parameters (because parameters are recognized by their order). OMITTED is only permitted when the called routine has the VARIABLE or the EXTENSIBLE attribute.

return-value

is a numeric or numeric edited elementary data item where the return value is stored when the called routine is a function. The *return-value* can be a 2-byte, 4-byte, or 8-byte integer or a 4-byte or 8-byte floating-point value (in TAL terms: INT, INT(32), FIXED, REAL, or REAL(64)).

Usage Considerations:

- *language*

  If you call a routine with a particular value of *language* (or with *language* omitted), you must use the same value of *language* each time you call that routine in the same compilation unit.

- *file-mnemonic*

  If you specify *file-mnemonic* but the compiler does not find the called routine in the associated object file, the compiler reports an error. For information on resolving external references, see Finding the Entered Program (page 529).

- USING Phrase

  The number and type of the parameters in the USING phrase must be compatible with the expectations of the called routine. The called routine might interpret the parameters slightly differently than the calling program does (see Appendix B: Data Type Correspondence (page 1239)).

- GIVING Phrase

  When calling a function (a routine that returns a value), include a GIVING phrase in the ENTER statement. When control returns to the COBOL program, the value that the function returns is assigned to *return-value*.

Scaling of the function's value (if needed) is performed before the value is assigned to `return-value`. If the function's value is larger than the maximum value allowed for a COBOL identifier, the COBOL program terminates with an arithmetic overflow condition during the conversion process. The assignment operation follows MOVE conventions (see MOVE TO (page 378)).

- Passing Parameters by Value

  When a formal parameter (in the called routine) is to be passed by value, the actual parameter (in the ENTER statement) must be a numeric literal, a numeric data item, a special register, an arithmetic expression, or the word OMITTED. If the parameter is not OMITTED, it is evaluated, scaled, and converted to the storage size and type of the formal parameter. The resulting value is passed to the called routine. This conversion might cause an arithmetic overflow.

- Passing Parameters by Reference

  When a formal parameter (in the called routine) is to be passed by reference, the actual parameter (in the ENTER statement) must be a data item or, in some routines supplied by HP, a file name.

  If the actual parameter is a data item, the compiler generates code to pass the address of the data item's storage space to the called routine. The calling program and the called routine must interpret the value of the data item the same way.

  If the actual parameter is a file name, the compiler generates code to pass the address of the COBOL file control block to the called routine.

- Addressing Parameters

  All instructions use 32-bit addressing. Many COBOL data items are byte-addressed. Some data items in other languages are 2-byte-addressed. When you pass a byte-addressed parameter to a routine that expects a 2-byte-addressed parameter, the data item must be aligned on a 2-byte boundary; otherwise, problems can arise, because the called routine is expecting an aligned parameter. The compiler does not issue a warning in this case.

  — pTAL

  In pTAL modules, each data item declaration (including parameter declarations) specifies either 1-byte or 2-byte addressing. All declarations use 32-bit addressing.

  — C and C++

  In HP C and HP C++ modules, all data item declarations (including parameter declarations) use 32-bit addressing.

- Restrictions on Calling HP C and HP C++ Functions

  An HP COBOL program cannot call an HP C or HP C++ function for which any of these conditions are true:

  — The function name includes lowercase letters.

  — The function prototype specifies a variable number of parameters.

  — The function returns a structured value.

  — The function specifies a formal parameter whose type has no corresponding COBOL type (see Appendix B: Data Type Correspondence (page 1239)).

  Do not include the GIVING phrase in an ENTER statement that accesses an HP C or HP C++ function whose type is a pointer.

  Because the names of HP C++ routines are often modified to reflect their classes and argument types, it is recommended that HP COBOL programs call only regular HP

C++ functions, not constructs that are peculiar to HP C++ (such as member functions and templates).

- Restrictions on Calling pTAL Routines

  If your HP COBOL program calls a TAL or pTAL routine that has a *string*:*length* parameter, you only need to give the name of the corresponding actual parameter, because the compiler can determine its length.

  If a TAL or pTAL routine that has a string:length parameter does not ignore trailing spaces and you want to pass it an actual parameter that is shorter than the one defined for the corresponding formal parameter, use reference modification.

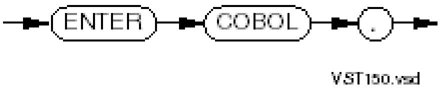
  The HP COBOL code in Example 10-16 calls the ppTAL routine FILENAME_COMPARE_, which does not ignore trailing spaces and has two formal parameters, *filename1*:*length1* and *filename2*:*length2*.

Example 10-16 Calling a pTAL Routine That Does Not Ignore Trailing Spaces

```
01  file-name-1   PIC X(255).
01  file-name-2   PIC X(255).
01  len-1         PIC 999 COMP.
01  len-2         PIC 999 COMP.
...
MOVE 0 TO len-1, len-2
INSPECT file-name-1 TALLYING len-1
   FOR CHARACTERS BEFORE SPACE
INSPECT file-name-2 TALLYING len-2
   FOR CHARACTERS BEFORE SPACE
ENTER "FILENAME_COMPARE_" USING file-name-1 (1: len-1)
                                file-name-2 (1: len-2)
                          GIVING ...
```

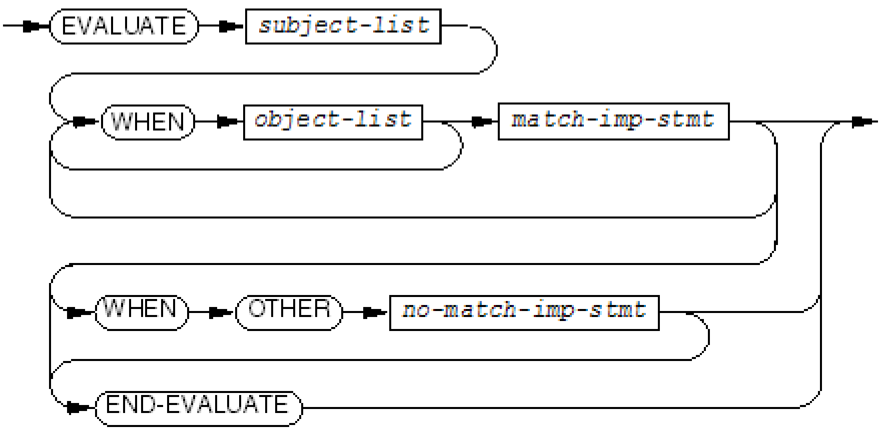
ENTER COBOL

HP COBOL compilers treat ENTER COBOL as a comment. In implementations that use ENTER to mark the beginning of an embedded routine in a language other than COBOL, the ENTER COBOL statement ends the embedded routine and the resumption of COBOL source code.



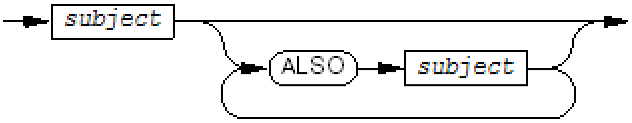
EVALUATE

EVALUATE defines a multiple-branch structure, a decision table. EVALUATE executes a different group of statements when one or more data items or expressions have certain sets of values.



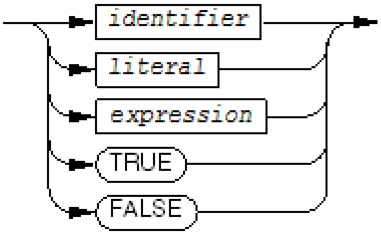
VST151.vsd

subject-list



VST616.vsd

subject



VST152.vsd

identifier

is any identifier (it can be qualified, subscripted, or reference-modified).

literal

is any literal.

expression

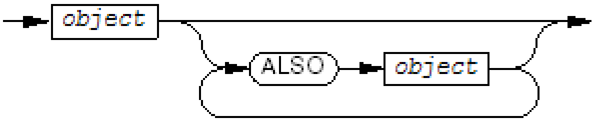
is an arithmetic or conditional expression.

TRUE

FALSE

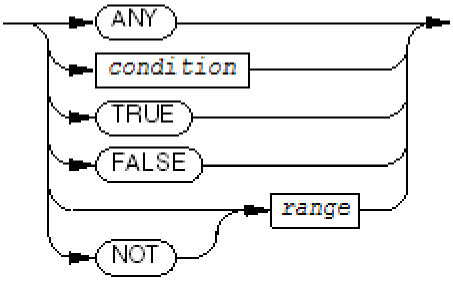
are logical values.

object-list



VST153.vsd

object



VST154.vsd

ANY

matches any corresponding selection subject.

condition

is a relation condition, class condition, condition-name condition, switch-status condition, sign condition, or complex condition.

TRUE

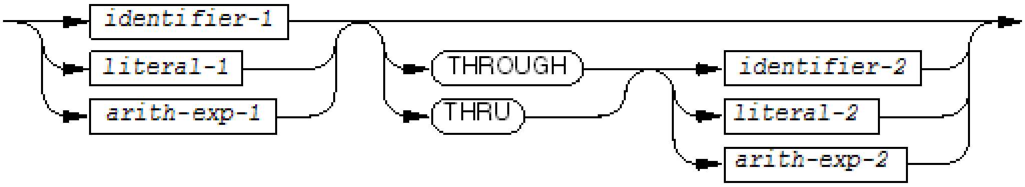
FALSE

are truth values.

NOT

inverts the comparison. A match occurs when the value of a *subject* differs from the value of the corresponding *object* or lies outside the range specified for the *object*.

range



VST155.vsd

identifier-1, *identifier-2*

are any identifiers (they can be qualified, subscripted, or reference-modified). They can be compared to any corresponding selection subjects being compared in a conditional expression.

literal-1, *literal-2*

can be any literals. They can be compared to any corresponding selection subject being compared in a conditional expression.

arith-exp-1, *arith-exp-2*

are arithmetic expressions. They can be compared to any selection subject being in a conditional expression.

THROUGH, THRU

indicate that a range of values is to be compared. Two operands combined by THROUGH or THRU must be of the same class (alphabetic, alphanumeric, or numeric). The two operands thus connected constitute a single selection object.

`match-imp-stmt`

is an imperative statement to be executed when the values of the objects in the associated `object-list` match the values of the corresponding subjects in the `subject-list`.

`no-match-imp-stmt`

is an imperative statement to be executed when no `match-imp-stmt` applies. After `no-match-imp-stmt` executes, control passes to the end of the EVALUATE statement.

END-EVALUATE

ends the scope of the EVALUATE statement and makes it a delimited-scope statement. If you omit END-EVALUATE, the EVALUATE statement is a conditional statement and ends at the next period separator.

Usage Considerations:

- EVALUATE in Comparison to "Case" Statement

  The EVALUATE statement is a generalization of what some languages call the "case" statement, a statement that enables you to execute one of several groups of statements depending on the value of a single data item.

Example 10-17 EVALUATE Statement as "Case" Statement

```
EVALUATE TRAN-CODE
  WHEN 1 PERFORM TRAN-1
  WHEN 2 PERFORM TRAN-2
  WHEN 3 PERFORM TRAN-3
  WHEN OTHER PERFORM TRAN-BAD
END-EVALUATE
```

Example 10-18 EVALUATE Statement With Both Ranges and Discrete Values

```
EVALUATE SALARY OF PAYROLL-REC
  WHEN 15000 THRU 29999.99 PERFORM LOW-BRACKET
  WHEN 30000 THRU 49999.99 PERFORM MIDL-BRACKET
  WHEN 50000 THRU 99999999 PERFORM HIGH-BRACKET
  WHEN OTHER               PERFORM NO-BRACKET
END-EVALUATE
```

The EVALUATE statement extends the "case" statement concept by allowing several selection subjects and matching objects—the equivalent of a decision table. The EVALUATE statement in Example 10-19 uses the values of two distinct subjects to make its selection.

Example 10-19 EVALUATE Statement With Two Distinct Subjects

```
EVALUATE DEDUCTIONS ALSO SALARY
  WHEN 0         ALSO  0     THRU 14999.99 PERFORM XXA
  WHEN 0         ALSO  15000 THRU 29999.99 PERFORM YYA
  WHEN 0         ALSO  30000 THRU 49999.99 PERFORM ZZA
  WHEN 1 THRU 2  ALSO  0     THRU 19999.99 PERFORM XXB
  WHEN 1 THRU 2  ALSO  20000 THRU 69999.99 PERFORM YYB
  WHEN 1 THRU 2  ALSO  70000 THRU 99999999 PERFORM ZZB
  WHEN OTHER PERFORM FURTHER-ANALYSIS
END-EVALUATE
```

The selection objects (the "0" and "1 THRU 2" of Example 10-19) are not restricted to being literals or ranges of literals; they can also be (for example) condition names, identifiers, and arithmetic expressions.

The WHEN OTHER phrase is commonly used as the mechanism for handling errors, enabling the program to detect invalid values for subjects and improper specification of objects.

Example 10-20 EVALUATE Statement as a Decision Table

```
EVALUATE XXX        ALSO "GONE" ALSO ( A + B ) / C ALSO X = Y
    WHEN  5         ALSO ANY    ALSO      25       ALSO TRUE
      PERFORM PROC-A
    WHEN 10 THRU 30 ALSO YYY    ALSO       2       ALSO FALSE
      PERFORM PROC-B
    WHEN OTHER        ADD 1 TO VACUOUS-COUNT
END-EVALUATE
```

- Subject-Object Correspondence

  Every `object-list` must have an `object` for every `subject` in the `subject-list`. Each `object` in an `object-list` must correspond to the `subject` that has the same ordinal position in the `subject-list`. The rules that determine whether a `subject` and `object` correspond are:

  — The `object` value ANY corresponds to any `subject` value.

  — An `object` that is a condition or the value TRUE or FALSE corresponds to a `subject` that is a conditional expression or the value TRUE or FALSE.

  — An `object` composed of identifiers, literals, or arithmetic expressions corresponds to a `subject` if the value of the `object` is a valid operand for comparison to the `subject`.

- Subject-Object Comparisons

  EVALUATE operates as if each corresponding `subject` and `object` were evaluated and assigned one of:

  — A numeric or nonnumeric value

  — A range of numeric or nonnumeric values

  — TRUE

  — FALSE

  The `subject` is determined:

subject	EVALUATE uses ...
identifier	The value and class of the data item referenced by identifier
literal	The value and class of literal
arithmetic expression	The numeric value determined by the rules for evaluating arithmetic expressions
conditional expression	The truth value determined by the rules for evaluating conditional expressions
TRUE	TRUE
FALSE	FALSE

  The `object` is determined:

object	EVALUATE uses ...
identifier (without NOT or THRU)	The value and class of the data item referenced by identifier
literal (without NOT or THRU)	The value and class of literal
ZERO or ZEROS or ZEROES (without NOT or THRU)	The value 0 and the class of the corresponding selection subject
arithmetic expression (without NOT or THRU)	The numeric value determined by the rules for evaluating arithmetic expressions

object	**EVALUATE uses …**
condition	The truth value determined by the rules for evaluating conditional expressions
ANY	No value—ANY matches any selection subject
A range of items specified with the keyword THRU but without the keyword NOT	The range of all permissible subject values greater than or equal to the first operand and less than or equal to the second operand (if the value of the first operand is greater than the value of the second operand, there are no values in the range)
The keyword NOT followed by an item or a range of items	All permissible subject values not equal to the value, or not included in the range of values, that would have been used if the keyword NOT had been omitted
TRUE	TRUE
FALSE	FALSE

- Execution of the EVALUATE Statement

  Execution of the EVALUATE statement begins with the comparison phase illustrated in Figure 10-3. A comparison is satisfied if one of these conditions is true:

  — The *object* is ANY (and the value of the *subject* is irrelevant).

  — The corresponding *subject* and *object* were both assigned TRUE or both assigned FALSE.

  — The corresponding *subject* and *object* were assigned the same type of values (numeric or nonnumeric) and the value assigned to the *subject* is equal to the value (or within the range of values) assigned to the *object*, according to the rules for comparison defined for a relation condition (see Relation Conditions in General (page 276)).

  If each comparison in an *object-list* is satisfied, the *object-list* “qualifies.” After the comparison phase ends, execution of the EVALUATE statement proceeds as shown in Figure 10-4.

Figure 10-3 Comparison Phase of EVALUATE Statement

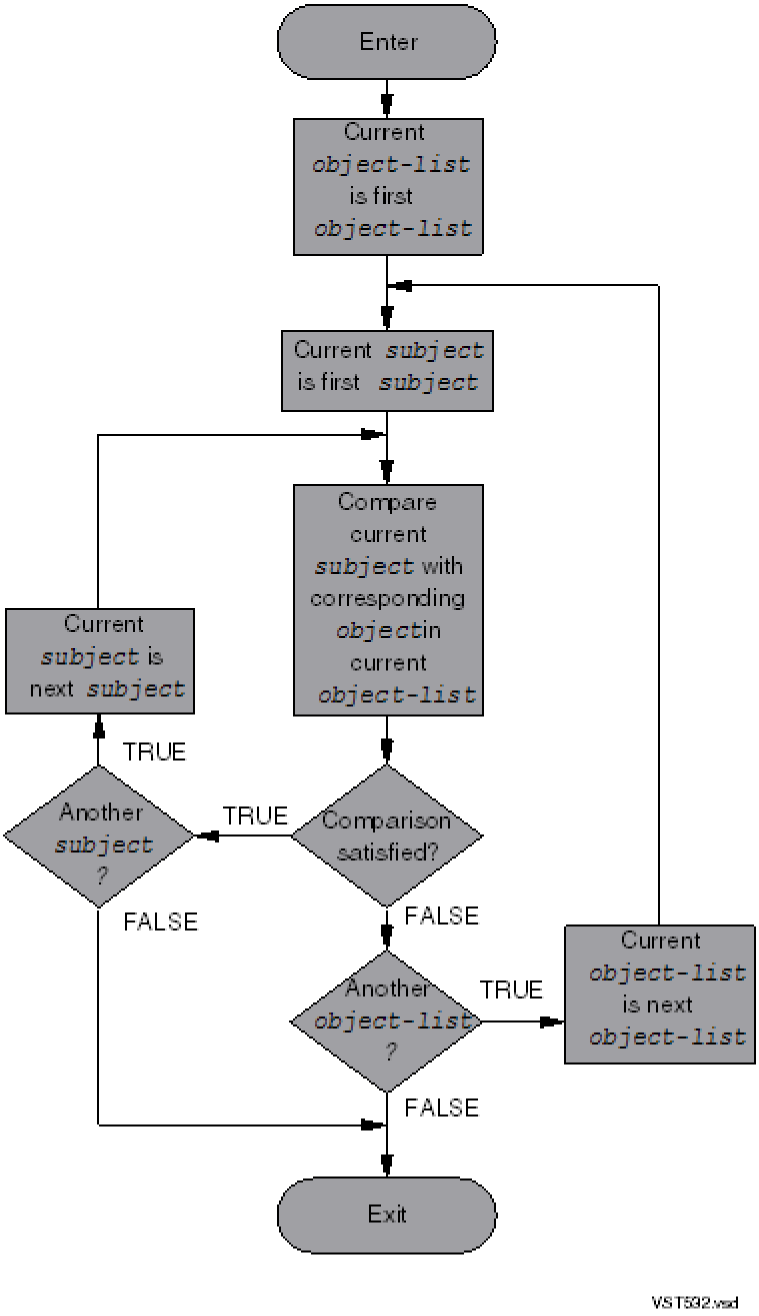
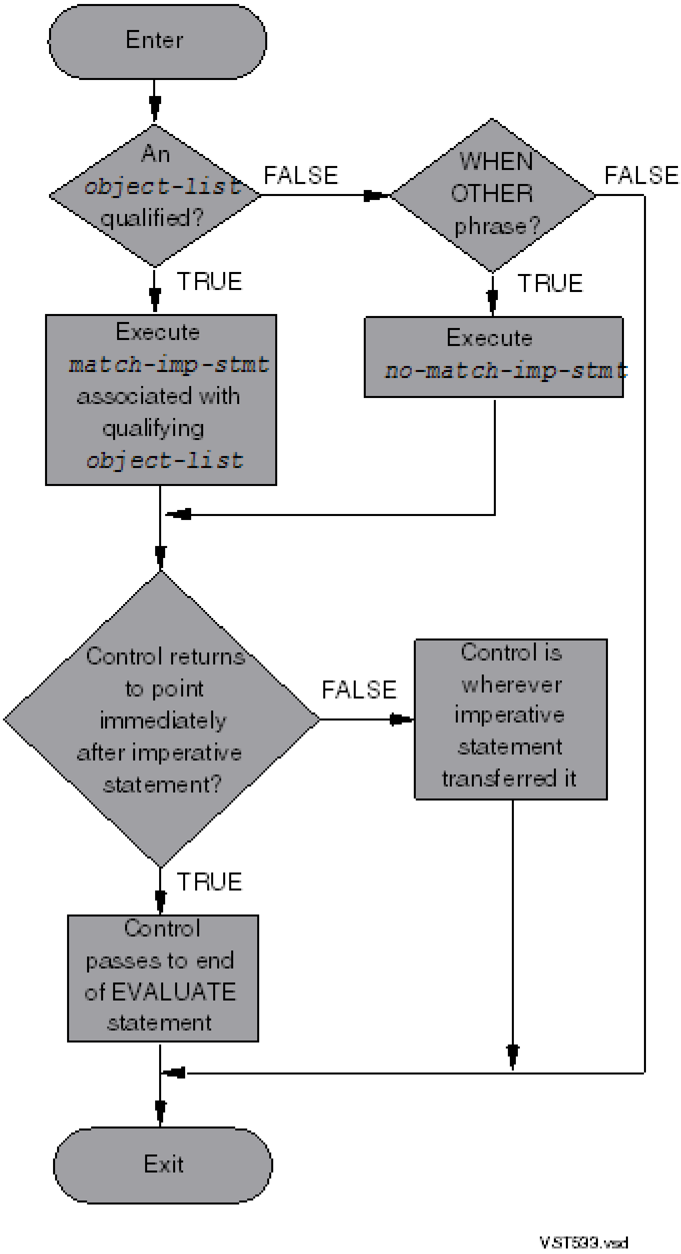

Enter
Current object-list is first object-list
Current subject is first subject
Compare current subject with corresponding objectin current object-list
Comparison satisfied?
TRUE
Another subject ?
TRUE
Current subject is next subject
FALSE
FALSE
Another object-list ?
TRUE
Current object-list is next object-list
FALSE
Exit
VST532.vsd

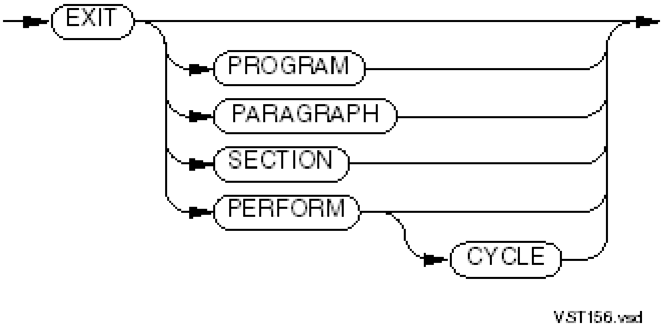
Figure 10-4 Execution Phase of the EVALUATE Statement


Enter
An *object-list* qualified?
FALSE
WHEN OTHER phrase?
FALSE
TRUE
TRUE
Execute *match-imp-stmt* associated with qualifying *object-list*
Execute *no-match-imp-stmt*
Control returns to point immediately after imperative statement?
FALSE
Control is wherever imperative statement transferred it
TRUE
Control passes to end of EVALUATE statement
Exit
VST533.vsd

EXIT

EXIT provides:

- A common end point for a series of procedures
- A marker for the logical end of a called program
- A way to return to the test in an in-line PERFORM statement
- A way to go to the end of the last statement in a paragraph or section



PROGRAM, PARAGRAPH, SECTION, PERFORM, CYCLE

are explained in Table 10-2.

Table 10-2 EXIT Statement Restrictions and Effects

Statement	Restrictions	Effect
EXIT	Must appear in a sentence by itself, and that sentence must be the only sentence in the paragraph.	Provides a common end point for a group of procedures or indicates the logical end of a called program. Does not affect program compilation or execution.
EXIT PROGRAM	If it appears in a consecutive sequence of imperative statements within a sentence, it must be the last statement in the sequence.Not allowed while executing a GLOBAL declarative procedure, except within a program that was called while the declarative procedure was executing.	Depends on the program that executes it—see Usage Considerations.
EXIT PARAGRAPH	Must be contained in a paragraph.	Transfers control to an implicit CONTINUE statement immediately preceding the next procedure declaration. If there is no next procedure declaration, it transfers control to an implicit CONTINUE statement at the end of the program.
EXIT SECTION	Must be contained in a section.	Transfers control to an implicit CONTINUE statement immediately preceding the next section declaration. If there is no next section declaration, it transfers control to an implicit CONTINUE statement at the end of the program.
EXIT PERFORM	Must be contained in an in-line PERFORM statement.	Immediately transfers control to an implicit CONTINUE statement immediately following the END-PERFORM statement associated with the in-line PERFORM statement (see Example 10-21).
EXIT PERFORM CYCLE	Must be contained in an in-line PERFORM statement.	Immediately transfers control to an implicit CONTINUE statement immediately preceding the END-PERFORM statement associated with the in-line PERFORM statement (see Example 10-22).

Example 10-21 EXIT PERFORM Statement

```
0001-TEST-EXIT-PERF.
   INITIALIZE WS-ANS.
   INITIALIZE WS-NUMBER.
   INITIALIZE WS-NUMBER-PERF.
   INITIALIZE WS-TEST-RESULTS.
   DISPLAY "TEST EXITS-1 FOR EXIT PERFORM BEGINS".
PERFORM
```

```
  UNTIL WS-NUMBER-PERF = 99
    ADD 3 TO WS-NUMBER-PERF
    IF WS-NUMBER-PERF = 96
      MOVE 1 TO WS-NUMBER-PERF
      EXIT PERFORM
    END-IF
    IF WS-NUMBER-PERF = 1
      DISPLAY "DID NOT EXIT PERFORM SUCCESSFULLY"
      MOVE 99 TO WS-NUMBER-PERF
    END-IF
END-PERFORM.

IF WS-NUMBER-PERF = 1
  DISPLAY "TEST EXITS-1 FOR EXIT PERFORM SUCCESSFUL."
ELSE
  DISPLAY "TEST EXITS-1 FOR EXIT PERFORM FAILED .".
0001-TEST-EXIT-PERF-EXIT.
EXIT.
```

Example 10-22 EXIT PERFORM CYCLE Statement

```
?SYMBOLS
 IDENTIFICATION DIVISION.
 PROGRAM-ID. EXIT-PERFORM.
 DATA DIVISION.
 WORKING-STORAGE SECTION.
 77 X PIC 999.
 77 X PIC 999.
 77 X PIC 999.
 PROCEDURE DIVISION.
 SUNRISE.
    DISPLAY "EXIT PERFORM TEST".
    MOVE 0 TO X
    MOVE 0 TO Y
    MOVE 0 TO Z
* The loop appears to execute 200 times ...
    PERFORM UNTIL X=200
       ADD 1 TO X
* however, we exit after only 100 times.
    IF X = 100
       EXIT PERFORM
       END-IF
* Y counts to 4, so that every 4th time, exit the perform
* cycle without incrementing Z ...
    ADD 1 TO Y
    IF Y = 4
        MOVE 0 TO Y
        EXIT PERFORM CYCLE
        END-IF
* so Z is incremented only 75 times, not 100.
     ADD 1 TO Z
     END-PERFORM
      .
    DISPLAY "X: " X.
    DISPLAY "Y: " Y.
    DISPLAY "Z: " Z.
```

Usage Considerations:

- EXIT PROGRAM Statement in a Program That Was Not Called

  If a program that was not called by another program executes an EXIT PROGRAM statement, program execution continues with the next executable statement.

- EXIT PROGRAM Statement in a Called Initial Program

  If an initial program that was called by another program executes an EXIT PROGRAM statement, the called program is cancelled (see CANCEL).

- EXIT PROGRAM Statement in a Called Program That is Not Initial

  If a program that was called by another program and is not initial executes an EXIT PROGRAM statement, program execution continues with the executable statement following the CALL statement in the calling program.

  The program state of the calling program is the same as it was when it executed the CALL statement, except for possible changes in the contents of data items and files that the calling and called programs shared.

  The only change in the program state of the called program is that the ends of the ranges of all PERFORM statements that it executed are considered to have been reached.

- EXIT PERFORM in Nested In-Line PERFORM Statements

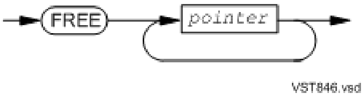
  An EXIT PERFORM statement in a nested in-line PERFORM statement causes the innermost PERFORM statement to be exited.

Example 10-23 EXIT Statement in Nested In-Line PERFORM Statement

```
    ...
    PERFORM REPORT-EXPLOSION THROUGH REPORT-EXPLOSION-END.
    ...
REPORT-EXPLOSION.
    READ MASTER-EXP RECORD
      AT END GO TO REPORT-EXPLOSION-END
    END-READ
    GO TO SUB-ASSY-1
          SUB-ASSY-2
          ...
          SUB-ASSY-23 DEPENDING ON SUB-ASSY OF MASTER-EXP
    PERFORM REPORT-BAD-SUB-ASSEMBLY
    GO TO REPORT-EXPLOSION-END.
SUB-ASSY-1.
    ...
SUB-ASSY-2.
    ...
REPORT-EXPLOSION-END.
    EXIT.
```

FREE

FREE releases dynamic memory previously allocated by the ALLOCATE statement.



pointer

is a data item described as USAGE POINTER. Its value must be either the address of memory previously obtained by the ALLOCATE statement, or NULL. If the value of *pointer* is not NULL, the memory is released and *pointer* is set to NULL

If the value of *pointer* is not NULL and is not the address of memory obtained by ALLOCATE, execution terminates with an error.

Usage Considerations:

- Freeing memory addressed by a BASED item

  The ALLOCATE statement can assign an address to either the implicit pointer of a BASED item or a USAGE POINTER item. The FREE statement can free memory only from a USAGE POINTER data item. To free memory addressed only by a BASED item, you must first transfer the address to a USAGE POINTER item using the SET statement.

- Compatible with the `malloc()` function

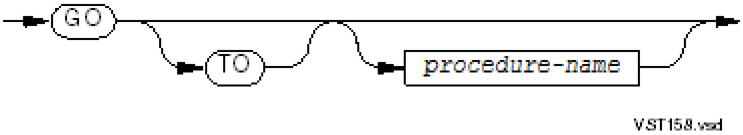
  The FREE statement is compatible with the C Run-Time Library function `malloc()` and related functions. A pointer to memory allocated by a C-language routine calling `malloc()` can be passed to the FREE statement to release the memory.

GO TO

One of several procedures, depending on the value of a variable data item

Unconditional GO TO

Unconditional GO TO passes control to the beginning of a paragraph or section in the current program.



procedure-name

is the name of the procedure (paragraph or section) to which the process transfers control. If no *procedure-name* is present, the process must execute an ALTER statement naming the procedure before it executes the GO TO statement to set the destination *procedure-name*.

NOTE: The 1985 COBOL standard classifies ALTER as **obsolete**, so you are advised not to use it, and therefore, not to use unconditional GO TO without *procedure-name*.

Usage Considerations:

- Declarative and Nondeclarative Procedures

  A GO TO statement in a declarative procedure (a procedure in the Declaratives Portion of the Procedure Division) cannot refer to a nondeclarative procedure (a procedure in the other portion of the Procedure Division). A GO TO statement in a nondeclarative procedure cannot refer to a declarative procedure.

- Debugging and Nondebugging Declarative Procedures

  A GO TO statement in a debugging declarative procedure (a declarative procedure introduced by a USE DEBUGGING statement) cannot refer to a nondebugging declarative procedure. A GO TO statement in a nondebugging declarative procedure cannot refer to a debugging declarative procedure.

- ALTER Statement

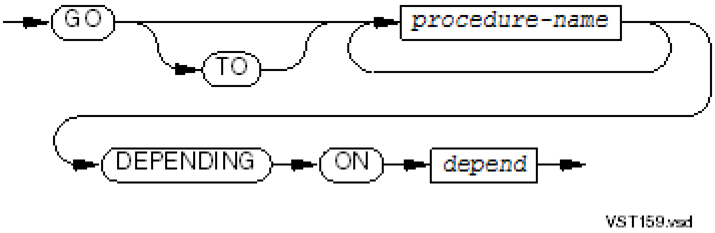
  Because the 1985 COBOL standard classifies ALTER as an obsolete element, and because the ALTER statement can cause maintenance problems, so you are advised not to use it. For more information, see ALTER.

- Consecutive Imperative Statements in a Sentence

  When a GO TO statement includes *procedure-name* and the paragraph containing the GO TO statement is not referenced by an ALTER statement, the GO TO statement can be the last of a sequence of consecutive imperative statements within a sentence.

Conditional GO TO

Conditional GO TO passes control to one of several procedures, depending on the value of a variable data item.



procedure-name

is the name of the procedure (paragraph or section) to which the process transfers control, depending on the value of *depend*. The GO TO statement can have as many as 255 different procedure names.

depend

is the identifier of an elementary integer data item. Its value determines which procedure will receive control. If the value of *depend* is less than 1 or exceeds the number of procedures, control passes to the next statement.

Usage Considerations:

- Declarative and Nondeclarative Procedures

  A GO TO statement in a declarative procedure (a procedure in the Declaratives Portion of the Procedure Division) cannot refer to a nondeclarative procedure (a procedure in the other portion of the Procedure Division). A GO TO statement in a nondeclarative procedure cannot refer to a declarative procedure.

- Debugging Declarative Procedures

  A GO TO statement in a debugging declarative procedure cannot refer to a nondebugging declarative procedure. A GO TO statement in a nondebugging declarative procedure cannot refer to a debugging declarative procedure.

  In Example 10-24, if BRANCH-FLAG equals 1, control passes to PROC-X. If BRANCH-FLAG equals 2, control passes to PROC-Y. If BRANCH-FLAG equals 3, control passes to PROC-Z. If BRANCH-FLAG is less than 1 or greater than 3, control passes to the MOVE statement immediately following the GO TO statement.

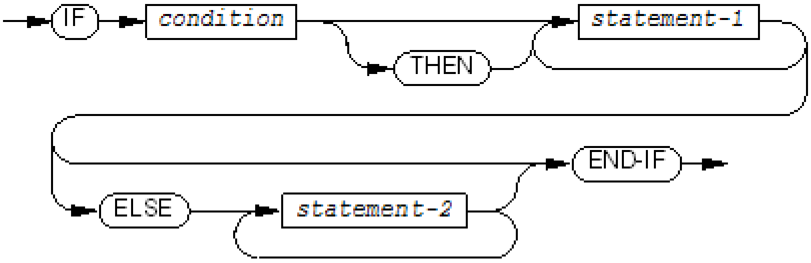
Example 10-24 Three-Way Conditional GO TO Statement

```
PROCEDURE-BRANCH.
  GO TO PROC-X
    PROC-Y
    PROC-Z DEPENDING ON BRANCH-FLAG
  MOVE 0 TO BRANCH-FLAG
```

IF

IF transfers control if the value of a condition is TRUE or FALSE.

Delimited-Scope Form



VST160.vsd

condition

is any conditional expression (see Conditional Expressions (page 275)).

statement-1

is an imperative or conditional statement to be executed if the value of *condition* is TRUE. It can contain other IF statements.

ELSE

ends *statement-1*.

statement-2

is an imperative or conditional statement to be executed if the value of *condition* is FALSE. It can contain other IF statements.

END-IF

ends the scope of the IF statement, making it a delimited-scope statement. Without END-IF, the IF statement is a conditional statement that ends at the next period separator.

Usage Considerations:

- Delimited-Scope Statements Are Recommended Over Conditional Statements

  For clarity and convenience, delimited-scope statements are recommended over conditional statements.

- Period Separator in Delimited-Scope IF Statement

  A delimited-scope IF statement does not require a period separator. If a period separator occurs within a delimited-scope IF statement, it terminates the statement and the compiler diagnoses the END-IF as unmatched.

- Nested Delimited-Scope IF Statements

  Delimited-scope IF statements can be “nested” (included) within other delimited-scope IF statements. Nested delimited-scope IF statements are interpreted by pairing each IF phrase with an ELSE phrase, ELSE … END-IF pair, or isolated END-IF phrase, proceeding from left to right within a sentence. Each ELSE, ELSE … END-IF, or isolated END-IF that the compiler encounters is considered to correspond to the nearest previous IF phrase that has not already been paired with an ELSE, ELSE … END-IF, or isolated END-IF.

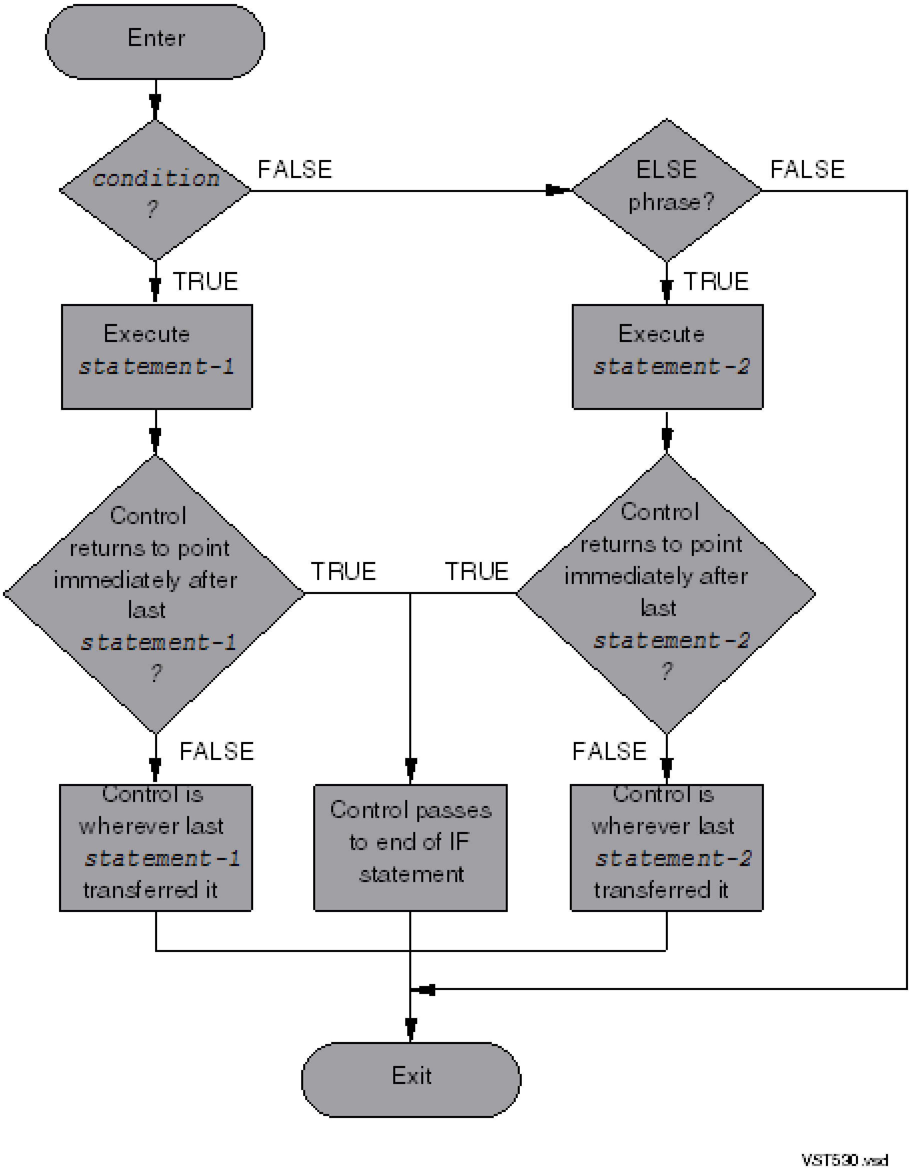
- How the Delimited-Scope IF Statement Works

  The `condition` is evaluated. If its value is TRUE, `statement-1` is executed. If control reaches the point immediately following `statement-1` (that is, `statement-1` completes without executing a GO TO statement or the equivalent), control passes to the end of the IF statement. The ELSE phrase, if present, is ignored.

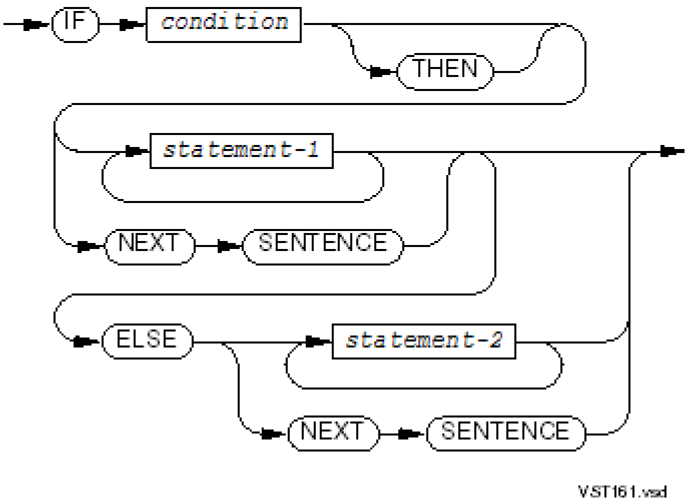
  If the value of `condition` is FALSE and an ELSE phrase is present, `statement-2` is executed. If control reaches the point immediately following `statement-2` (that is, `statement-2` completes without executing a GO TO statement or the equivalent), control passes to the end of the IF statement.

  If the value of `condition` is FALSE and there is no ELSE phrase, control passes to the end of the IF statement.

Figure 10-5 How the Delimited-Scope IF Statement Works


Enter
condition ?
FALSE
ELSE phrase?
FALSE
TRUE
TRUE
Execute statement-1
Execute statement-2
Control returns to point immediately after last statement-1 ?
TRUE
TRUE
Control returns to point immediately after last statement-2 ?
FALSE
FALSE
Control is wherever last statement-1 transferred it
Control passes to end of IF statement
Control is wherever last statement-2 transferred it
Exit
VST530.vsd

Conditional Form



condition

is any conditional expression (see Conditional Expressions (page 275)).

statement-1

is an imperative or conditional statement to be executed if the value of *condition* is TRUE. It can contain other IF statements.

NEXT SENTENCE

specifies that control be passed directly to the end of the sentence containing the IF statement. It is not recommended (see "Conditional Form").

ELSE

ends *statement-1*.

statement-2

is an imperative or conditional statement to be executed if the value of *condition* is FALSE. It can contain other IF statements.

Usage Considerations:

- CONTINUE and END-IF Are Recommended Over NEXT SENTENCE

  NEXT SENTENCE transfers control to the next period (.) while CONTINUE transfers control to END-IF. Either or both *statement-1* and *statement-2* can be CONTINUE statements.

- IF Sentences

  A conditional IF statement followed by a period separator is called an "IF sentence." An IF sentence can contain IF statements (of either form) only if such IF statements do not end with period separators.

- Nested Conditional IF Statements

  Conditional IF statements can be "nested" (included) within other conditional IF statements. When conditional IF statements are nested, each optional ELSE phrase is considered to be the next phrase of the nearest preceding unterminated conditional IF statement with which that phrase is permitted to be associated according to the syntax of the conditional IF statement, but with which no such phrase has already been associated. An unterminated statement is one that has not been previously terminated either explicitly or implicitly. The separator period that terminates the sentence also terminates all nested statements.

- How the Conditional IF Statement Works

  The *condition* is evaluated. If its value is TRUE and NEXT SENTENCE is specified (as opposed to *statement-1*), control passes to the next executable sentence. The ELSE phrase, if present, is ignored.

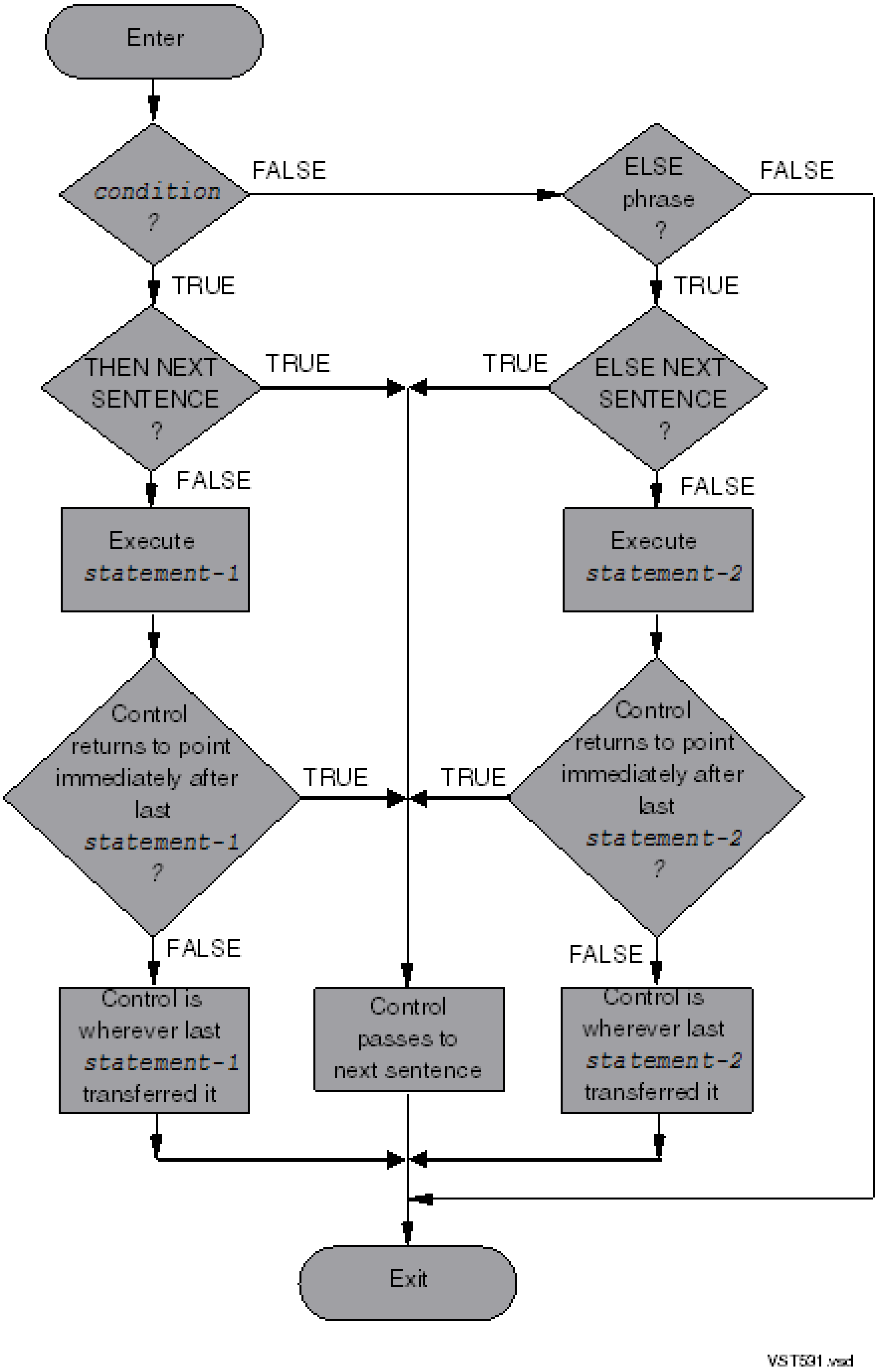
  If the value of *condition* is TRUE and *statement-1* is specified (as opposed to NEXT SENTENCE), *statement-1* is executed. If control reaches the point immediately following *statement-1* (that is, *statement-1* completes without executing a GO TO statement or the equivalent), control passes to the end of the IF statement. The ELSE phrase, if present, is ignored.

  If the value of *condition* is FALSE and an ELSE phrase specifies NEXT SENTENCE, control passes to the next executable sentence.

  If the value of *condition* is FALSE and an ELSE phrase specifies *statement-2*, *statement-2* is executed. If control reaches the point immediately following *statement-2* (that is, *statement-2* completes without executing a GO TO statement or the equivalent), control passes to the end of the IF statement.

  If the value of *condition* is FALSE and there is no ELSE phrase, control passes to the end of the IF statement.

Figure 10-6 How the Conditional IF Statement Works


Enter
condition ?
FALSE
ELSE phrase ?
FALSE
TRUE
TRUE
THEN NEXT SENTENCE ?
TRUE
TRUE
ELSE NEXT SENTENCE ?
FALSE
FALSE
Execute statement-1
Execute statement-2
Control returns to point immediately after last statement-1 ?
TRUE
TRUE
Control returns to point immediately after last statement-2 ?
FALSE
FALSE
Control is wherever last statement-1 transferred it
Control passes to next sentence
Control is wherever last statement-2 transferred it
Exit
VST591.vsd

Example 10-25 and Example 10-26 are equivalent.

Example 10-25 Simple Conditional IF Statement

```
IF JULIAN-DAYS IS GREATER THAN 59,
   ADD LEAP-YEAR TO JULIAN-DAYS.
```

Example 10-26 Delimited-Scope IF Statement

```
IF JULIAN-DAYS IS GREATER THAN 59
   ADD LEAP-YEAR TO JULIAN-DAYS
END-IF
```

Example 10-27 and Example 10-28 are equivalent.

Example 10-27 Simple Conditional IF ELSE Statement

```
IF TALLY GREATER THAN 0 MOVE 0 TO TALLY
                        MOVE 3 TO MSG-INDEX
                        PERFORM PRINT-ERROR-ROUTINE
ELSE                    MOVE 1 TO FLAG.
```

Example 10-28 Delimited-Scope IF ELSE Statement

```
IF TALLY GREATER THAN 0 MOVE 0 TO TALLY
                        MOVE 3 TO MSG-INDEX
                        PERFORM PRINT-ERROR-ROUTINE
ELSE                    MOVE 1 TO FLAG
END-IF
```

Example 10-29 and Example 10-30 are equivalent.

Example 10-29 IF ELSE Statement Nested Within PERFORM Statement

```
PERFORM DIV-IT VARYING I FROM 1 BY 1 UNTIL I > N
...
DIV-IT.
   MOVE INF TO Q (I)
   IF D (I) > 0 DIVIDE X (I) BY D (I) GIVING Q (I).
```

Example 10-30 Delimited-Scope IF ELSE Statement Nested Within PERFORM Statement

```
PERFORM VARYING I FROM 1 BY 1 UNTIL I > N
     IF D (I) > 0 THEN DIVIDE X (I) BY D (I) GIVING Q(I)
                  ELSE MOVE INF TO Q (I)
     END-IF
END-PERFORM
```

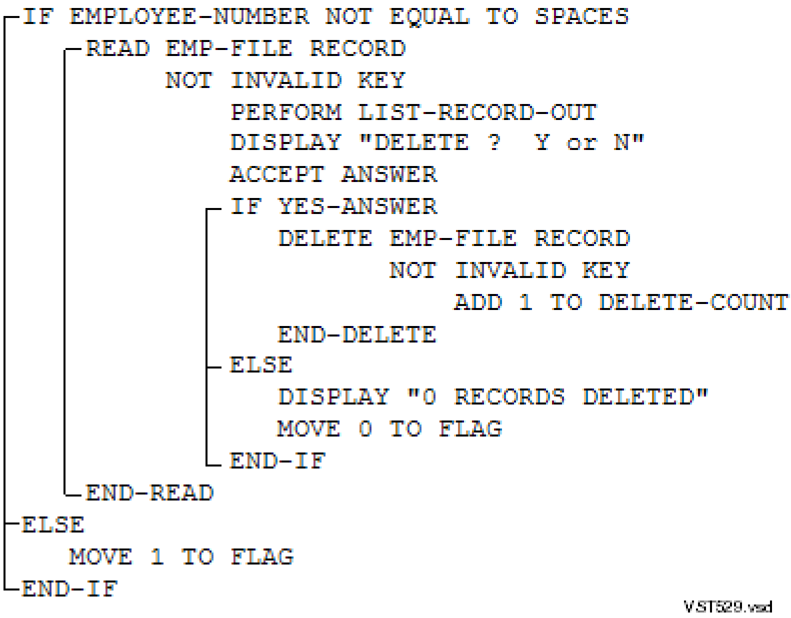
Example 10-31 Nested Conditional IF Statements

```
 IF EMPLOYEE-NUMBER NOT EQUAL TO SPACES
     PERFORM READ-ROUTINE
*            Must perform out of line
*            to do INVALID KEY test
     IF NO-ERROR
        PERFORM LIST-RECORD-OUT
        DISPLAY "DELETE ?  Y or N"
        ACCEPT ANSWER
        IF YES-ANSWER
           PERFORM DELETE-MASTER
*            Must perform out of line
*            to do INVALID KEY test
           IF NO-DELETE-ERROR
              ADD 1 TO DELETE-COUNT
```

```
          ELSE
             NEXT SENTENCE
       ELSE
          DISPLAY "0 RECORDS DELETED"
          MOVE 0 TO FLAG
   ELSE
       NEXT SENTENCE
ELSE
   MOVE 1 TO FLAG.
```

Example 10-32 shows the equivalent delimited-scope IF statement with delimited-scope statements marked.

Example 10-32 Delimited-Scope IF Statement

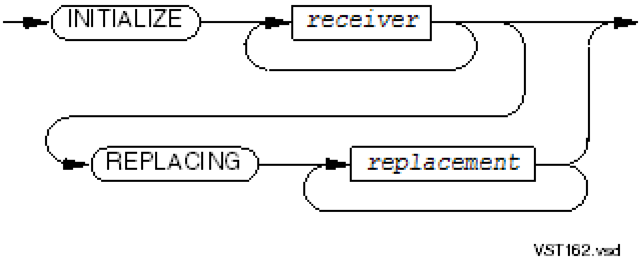


```
IF EMPLOYEE-NUMBER NOT EQUAL TO SPACES
    READ EMP-FILE RECORD
         NOT INVALID KEY
             PERFORM LIST-RECORD-OUT
             DISPLAY "DELETE ?  Y or N"
             ACCEPT ANSWER
             IF YES-ANSWER
                DELETE EMP-FILE RECORD
                       NOT INVALID KEY
                           ADD 1 TO DELETE-COUNT
                END-DELETE
             ELSE
                DISPLAY "0 RECORDS DELETED"
                MOVE 0 TO FLAG
             END-IF
    END-READ
ELSE
   MOVE 1 TO FLAG
END-IF
```

VST529.vsd

INITIALIZE

INITIALIZE sets selected types of data items to predetermined values; for example, numeric data to zeros or alphanumeric data to spaces.



receiver

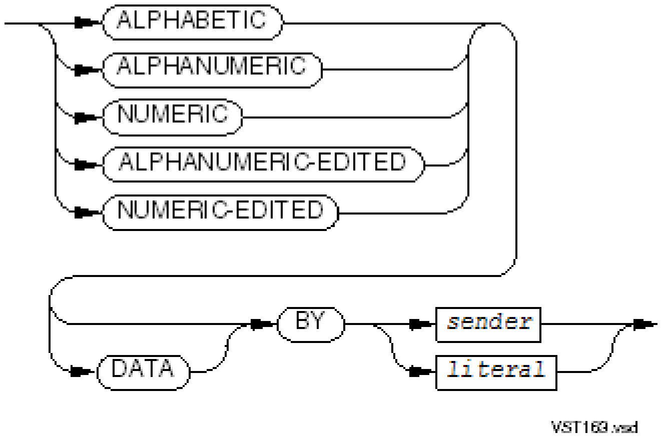
is the identifier of an elementary item or data structure. These restrictions apply to the data item:

- It cannot be an index data item or a pointer data item.
- It cannot be a special register.

- Its description cannot include a RENAMES clause.
- Its description (and the descriptions of its subordinate data items, if any) cannot include an OCCURS clause with a DEPENDING phrase.

The maximum number of receivers in an INITIALIZE statement is 127.

replacement



specifies the category of elementary item in *receiver* to which the INITIALIZE operation is to assign a value. The default *replacement* is ZEROS for each NUMERIC or NUMERIC-EDITED *receiver* and SPACES for each ALPHABETIC, ALPHANUMERIC, or ALPHANUMERIC-EDITED *receiver*. The *receiver* is in one of these categories whether you specify *replacement* or not (see PICTURE Clause (page 202)).

ALPHABETIC, ALPHANUMERIC, NUMERIC, ALPHANUMERIC-EDITED, NUMERIC-EDITED

are categories defined in PICTURE Clause (page 202). The category you specify must be permitted for a receiving operand in a MOVE statement where the corresponding sending operand is *sender* or *literal*. You cannot specify a category more than once in a single INITIALIZE statement.

sender

is the identifier of a data item from which the INITIALIZE statement obtains a value to store into the elementary items specified by *receiver*. The *sender* cannot be a national or index data item.

literal

is the representation of a value that the INITIALIZE statement stores into the elementary item specified by *receiver*. *literal* cannot be a national literal.

The INITIALIZE statements in Example 10-33 assign initial values to data structures composed of counters and pointers that a program uses for UNSTRING and INSPECT statements. In the first statement, the REPLACING phrase can be omitted if UNSTRING-COUNTERS and INSPECT-COUNTERS are numeric.

Example 10-33 INITIALIZE Statements

```
INITIALIZE UNSTRING-COUNTERS
           INSPECT-COUNTERS
           REPLACING NUMERIC DATA BY ZERO
INITIALIZE UNSTRING-POINTERS
           INSPECT-POINTERS
           REPLACING NUMERIC DATA BY 1
```

The INITIALIZE statement in Example 10-34 assigns a value of zero to only the numeric-edited items in a table composed of a variety of categories.

Example 10-34 INITIALIZE Statement

```
01 STOCK.
   03 STOCK-ITEM OCCURS 500 TIMES.
      05 S-NAME            PIC X(30).
      05 S-WHOLESALE-PRICE PIC $$$,$$$.99.
      05 S-MARKUP-PCT      PIC P999.
...
INITIALIZE STOCK
           REPLACING NUMERIC-EDITED DATA BY ZERO
```

Usage Considerations:

- Execution of the INITIALIZE Statement

  The INITIALIZE statement is equivalent to a series of MOVE statements, each of which has an elementary item as its receiving operand.

- Determining the Receiving Operands

  If *receiver* includes reference modification, the data item to which it refers is handled as an elementary item.

  If receiver references an elementary data item, that data item is a receiving operand unless the INITIALIZE statement includes a REPLACING phrase and the category of the elementary data item is not specified in any of the phrases following the REPLACING phrase. If receiver references an elementary data item that is not a receiving operand, that data item is not initialized.

  If *receiver* references a data structure, each elementary item subordinate to the data structure (including each element in a subordinate table) is a receiving operand; however, the initialize operation ignores:

  — Index or pointer data items

  — Elementary FILLER data items

  — Data items whose descriptions contain REDEFINES clauses

  — Data items subordinate to data items whose descriptions contain REDEFINES clauses

  — Data items that are not in the same categories as the initial values specified for them; for example, an ALPHABETIC *receiver* is not initialized to a NUMERIC *replacement* value even if the INITIALIZE statement specifies this

- Determining the Sending Operands

  The initialize operation determines the sending operand in each implicit MOVE statement:

  — If you include `replacement`, the sending operand is the `literal` or `sender` associated with the category of the receiving operand.

  — If you omit `replacement`, the sending operand is the implied figurative constant SPACES or ZEROS, as this table shows:

Category of Receiving Operand	Sending Operand
ALPHABETIC	SPACES
ALPHANUMERIC	SPACES
ALPHANUMERIC-EDITED	SPACES
NUMERIC	ZEROS
NUMERIC-EDITED	ZEROS

  The value of `sender` is established before the implicit MOVE statement executes.

- Execution of the Implicit MOVE Statements

  For each receiving operand, the initialization operation executes an implicit MOVE statement of the form:

```
MOVE SENDING-OPERAND TO RECEIVING-OPERAND
```

  The implicit MOVE statements are executed in the order that the `receiver`s associated with their receiving operands appear in the INITIALIZE statement (reading from left to right). When a `receiver` references a data structure, the affected elementary items are initialized in the order that they are defined within the data structure. Tables within data structures are initialized element by element.

- Operand Overlap

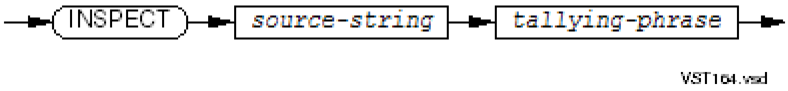
  The storage area referenced by a `sender` cannot be the same as or overlap the storage area referenced by a `receiver`. If this rule is violated, the operation is undefined.

INSPECT

INSPECT scans a data item and counts and/or replaces occurrences of a single character or groups of characters

INSPECT TALLYING

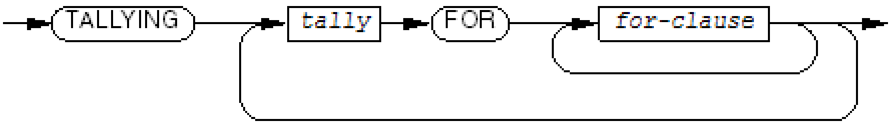
INSPECT TALLYING counts occurrences of a sequence of one or more characters in a source string.



`source-string`

is the identifier of an elementary item or data structure with USAGE DISPLAY. Inspection proceeds from left to right within `source-string`.

`tallying-phrase`

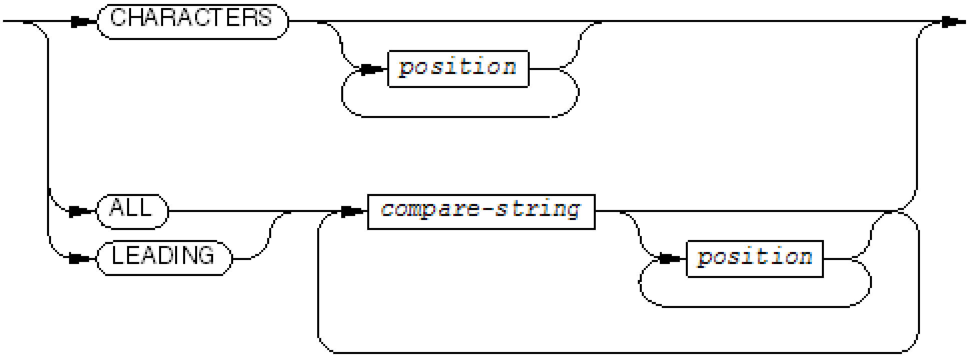


VST165.vsd

`tally`

is the identifier of an elementary numeric item where the number of occurrences is to be stored. The INSPECT statement does not initialize `tally` but adds to its current value. If you want `tally` to begin at a given number, set it before the INSPECT statement executes.

`for-clause`

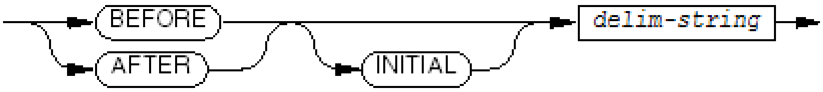


VST166.vsd

CHARACTERS

specifies that any character in `source-string` that has not satisfied a previous `compare-string` relation is counted, regardless of its value.

`position`



VST167.vsd

marks a beginning or ending point for an INSPECT scan cycle. Each CHARACTERS phrase, ALL phrase, or LEADING phrase can have a single BEFORE phrase, a single AFTER phrase, or one BEFORE phrase and one AFTER phrase in either order.

BEFORE

specifies that scanning is up to but not including the character or characters matching `delim-string`. If INSPECT does not find `delim-string` in `source-string`, it scans all of `source-string`.

AFTER

specifies that scanning starts at the position immediately following the character or characters matching `delim-string`. If INSPECT does not find `delim-string` in `source-string`, it does not scan `source-string`.

delim-string

is the identifier of an elementary item with USAGE DISPLAY or a nonnumeric literal. It can be a figurative constant that does not include the keyword ALL, in which case it represents one character.

ALL

specifies that all occurrences of *compare-string* are tallied.

LEADING

specifies scanning for consecutive occurrences of *compare-string* beginning at the current position within *source-string*.

compare-string

is the identifier of an elementary data item with USAGE DISPLAY or a nonnumeric literal. It can be a figurative constant that does not include the keyword ALL, in which case it represents one character. The *compare-string* is the character-string being sought.

Usage Considerations:

- Definition of "Leading"

  Characters are "leading" when they begin in the leftmost position of *source-string* and have not satisfied another *compare-string* condition. For example, two leading As are found in "AARDVARK" when

  ```
  INSPECT ... FOR LEADING "A"
  ```

  is coded, but no leading As are found in "AARDVARK" when

  ```
  INSPECT ... FOR ALL "A" ... FOR LEADING "A".
  ```

  is coded. The ALL phrase finds all three As, so none are left for the LEADING phrase to find. (For a complete explanation of the logic of the comparison, see comparison operation.)

  An example of LEADING ZERO is 00012.

- National Data Items and National Literals

  National data items and national literals cannot be used in an INSPECT statement.

- How Parameter Categories Affect INSPECT

  1. If *source-string*, *compare-string*, or *delim-string* is alphanumeric, the INSPECT statement handles the contents of each of them as a character-string.
  2. If *source-string*, *compare-string*, or *delim-string* is alphanumeric edited, numeric edited, or unsigned numeric, the *source-string* is inspected as though it had been redefined as alphanumeric and the INSPECT statement had been written to refer to the redefined item.
  3. If *source-string*, *compare-string*, or *delim-string* is described as a signed numeric, *source-string* is inspected as though it had been moved to an unsigned numeric data item of the same length (excluding any separate sign), and then item 2 applies.

- Comparison Operation

  The comparison operation to determine the occurrences of *compare-string* works like this:

  1. Each *tally* is considered in the order specified in the INSPECT statement (reading from left to right). The first *compare-string* is compared to an equal number of consecutive characters of *source-string*, starting with *source-string*'s leftmost character. If the *compare-string* and the substring of *source-string* are equal, character for character, then they match.
  2. If no match occurs in the comparison of the first *compare-string*, the comparison is repeated with each successive *compare-string* until either a match is made or there

is no next `compare-string`. When there is no next `compare-string`, the character position in `source-string` immediately to the right of the leftmost character position considered in the last comparison cycle is considered to be the leftmost character position, and the comparison cycle begins again with the first `compare-string`.

3. Whenever a match occurs, tallying occurs as described in tallying. The character position in `source-string` immediately to the right of the rightmost character position that participated in the match is now considered to be the leftmost character position of `source-string`, and the comparison cycle resumes with the first `compare-string`.
4. The comparison operation continues until the rightmost character position of `source-string` has participated in a match or has been considered to be the leftmost character position. When this occurs, inspection terminates.
5. If the CHARACTERS phrase appears, an implied 1-character operand participates in the comparison cycle, except that no comparison to the contents of `source-string` occurs. This implied character is considered to match the leftmost character of `source-string` participating in the current comparison cycle.

- BEFORE and AFTER Phrases

  The BEFORE and AFTER phrases affect the comparison operation like this:

  1. If no BEFORE or AFTER phrase is present, `compare-string` or the operand implied by CHARACTERS participates in the comparison operation.
  2. If the BEFORE phrase appears, the associated `compare-string` or the operand implied by CHARACTERS participates only in comparison cycles that involve the portion of `source-string` from its leftmost character position up to, but not including, the first occurrence of `delim-string`. The position of this first occurrence is determined before the first cycle of the comparison operation begins.

     If, on any comparison cycle, `compare-string` or the operand implied by CHARACTERS is not eligible to participate, it is not considered to match the contents of `source-string`. If `delim-string` does not occur within `source-string`, its associated `compare-string` or the operand implied by CHARACTERS participates in the comparison operation as though the BEFORE phrase was not specified.

  3. If the AFTER phrase appears, the associated `compare-string` or the operand implied by CHARACTERS can participates only in comparison cycles that involve the portion of `source-string` following the first occurrence of `delim-string`. The position of the first occurrence of `delim-string`. is determined before the first cycle of the comparison operation begins. The portion of `source-string` following the first occurrence of `delim-string` begins with the character position immediately to the right of the rightmost character of the first occurrence of `delim-string` and ends with the rightmost character of `source-string`.

     If, on any comparison cycle, `compare-string` or the operand implied by CHARACTERS is not eligible to participate, it is not considered to match the contents of `source-string`. If `delim-string` does not occur within `source-string`, its associated `compare-string` or the operand implied by CHARACTERS is not eligible to participate in the comparison operation.

- Tallying

  The INSPECT statement does not initialize `tally`. During the inspection of `source-string`, each properly matched occurrence of `compare-string` is tallied, using these rules:

  — If the ALL phrase appears, `tally` is incremented by one for each occurrence of `compare-string` matched within `source-string`.

  — If the LEADING phrase appears, `tally` is incremented by one for each contiguous occurrence of `compare-string` matched within `source-string`, provided that the leftmost such occurrence is at the point where comparison began in the first comparison cycle in which *compare-string* was eligible to participate.

— If CHARACTERS appears, *tally* is incremented by one for each character matched within *source-string*.

In the Example 10-35, INSPECT TALLYING checks for spaces in a data item. If spaces are present (J-TALLY is greater than 0), the code reports an error.

Example 10-35 INSPECT TALLYING With One Compare-String

```
MOVE ZERO TO J-TALLY
INSPECT JOB-CLASS TALLYING J-TALLY FOR ALL SPACES
IF J-TALLY GREATER THAN 0
   MOVE 0 TO J-TALLY
   MOVE 3 TO MESSAGE-INDEX
   PERFORM PRINT-ERROR
ELSE
   MOVE 1 TO FLAG
END-IF
```

In Example 10-36, an INSPECT TALLYING statement uses multiple *compare-string*s. Some characters look as if they qualify as matches, yet are not counted, because a character can be counted only once, no matter how many comparisons it can satisfy.

Example 10-36 INSPECT TALLYING With Multiple Compare-Strings

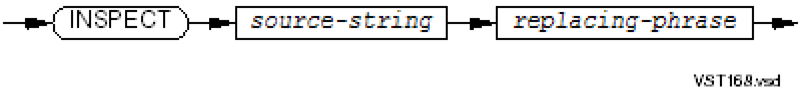
```
WORKING-STORAGE SECTION.
01  INSPECT-COUNTERS.
    03  COUNTER-1     PIC 99      VALUE 0.
    03  COUNTER-2     PIC 99      VALUE 0.
    03  COUNTER-3     PIC 99      VALUE 0.
    03  COUNTER-4     PIC 99      VALUE 0.
    03  COUNTER-5     PIC 99      VALUE 0.
77  ITEM-A        PIC X(15)   VALUE "  00001,003,200".
PROCEDURE DIVISION.
      ...
  INSPECT ITEM-A TALLYING
                 COUNTER-1 FOR ALL "0",
                 COUNTER-2 FOR CHARACTERS BEFORE INITIAL ","
                 COUNTER-3 FOR LEADING " ",
                 COUNTER-4 FOR ALL "0" BEFORE INITIAL ",",
                 COUNTER-5 FOR ALL "0" AFTER INITIAL ","
```

After execution of the INSPECT statement in Example 10-36, the counters have these values:

Counter	Value
COUNTER-1	8
COUNTER-2	3
COUNTER-3	0
COUNTER-4	0
COUNTER-5	0

INSPECT REPLACING

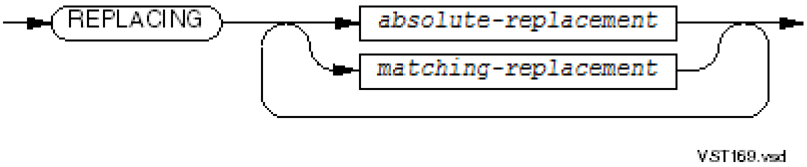
INSPECT REPLACING replaces occurrences of a sequence of one or more characters in a data item with a specified value.



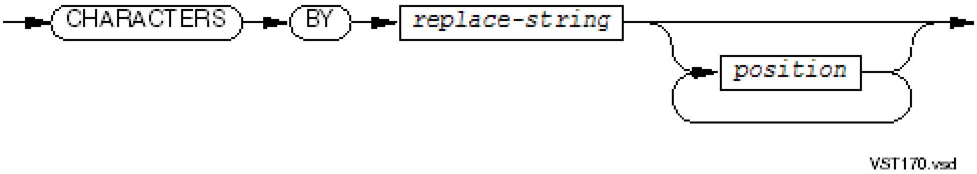
source-string

is the identifier of an elementary item or data structure with USAGE DISPLAY. Inspection proceeds from left to right within *source-string*.

replacing-phrase



absolute-replacement



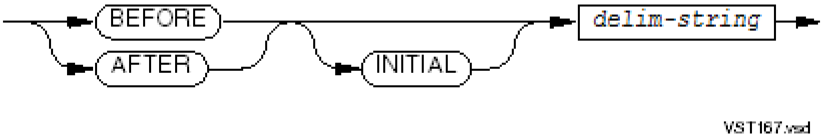
CHARACTERS

specifies that any character in *source-string* can be replaced, regardless of its value. If you use CHARACTERS, *replace-string* and *delim-string* must each be one character in length.

replace-string

is the identifier of an elementary item with USAGE DISPLAY or a nonnumeric literal. It can be a figurative constant that does not include the keyword ALL, in which case it represents one character.

position



marks a starting and ending place for a replacement cycle. It follows the same item rules as *replace-string*. A CHARACTERS phrase, ALL phrase, LEADING phrase, or FIRST phrase can have a single BEFORE phrase, a single AFTER phrase, or one BEFORE phrase and one AFTER phrase in either order.

BEFORE

specifies that scanning is up to but not including the character or characters matching *delim-string*. If INSPECT does not find *delim-string* in *source-string*, it scans all of *source-string*.

AFTER

specifies that scanning starts at the position immediately following the character or characters matching `delim-string`. If INSPECT does not find `delim-string` in `source-string`, it does not scan `source-string`.

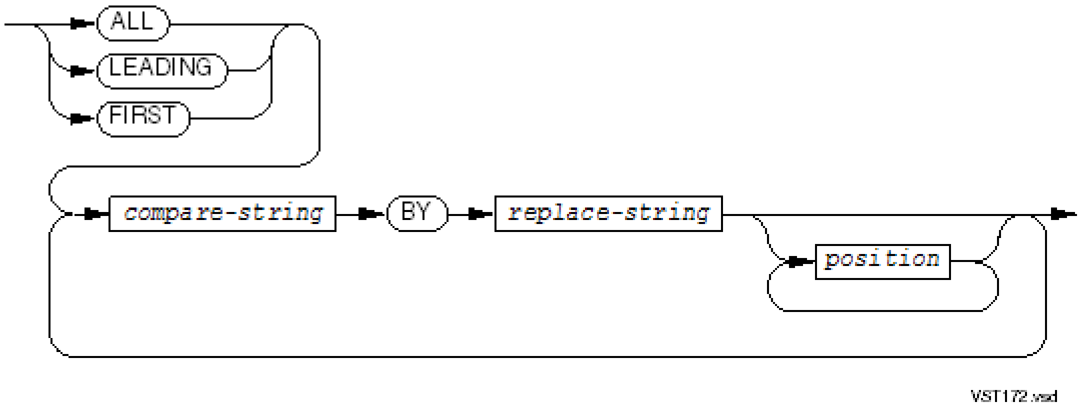
INITIAL

is for documentation only and has no effect.

`delim-string`

is the identifier of an elementary item with USAGE DISPLAY or a nonnumeric literal. It can be a figurative constant that does not include the keyword ALL, in which case it represents one character.

`matching-replacement`



defines the character-string being scanned for.

ALL

specifies that all occurrences of `compare-string` are replaced.

LEADING

specifies scanning for consecutive occurrences of `compare-string` beginning at the current position within `source-string`. For example, REPLACING LEADING “0” replaces only the first three zeros in “000120020.”

FIRST

specifies that only the first occurrence of `compare-string` is replaced.

`compare-string`

is the identifier of an elementary data item with USAGE DISPLAY or a nonnumeric literal. It can be a figurative constant that does not include the keyword ALL, in which case it represents one character. The `compare-string` is the character-string being sought.

`replace-string`

is an identifier of an elementary data item with USAGE DISPLAY or a nonnumeric literal. It can be a figurative constant that does not include the keyword ALL, in which case it represents one character. When `replace-string` and `compare-string` are used together, they must be the same size.

`position`

is as described earlier for `absolute-replacement`.

Usage Considerations:

- Rules for Replacement
  - CHARACTERS causes each character in *source-string* to be replaced by *replace-string*.
  - ALL causes each occurrence of *compare-string* in *source-string* to be replaced by *replace-string*.
  - LEADING causes each contiguous occurrence of *compare-string* in *source-string* to be replaced by *replace-string*, provided that the leftmost occurrence is at the point where comparison began in the first comparison cycle in which compare-string was eligible to participate.
  - FIRST causes the leftmost occurrence of *compare-string* in *source-string* to be replaced by *replace-string*.

    ALL, LEADING, and FIRST apply to each succeeding BY phrase until the next ALL, LEADING, or FIRST appears.
- See these usage considerations in INSPECT TALLYING (page 356):
  - How Parameter Categories Affect INSPECT
  - Comparison Operation
  - BEFORE and AFTER Phrases
- Leading Zeros

  HP COBOL does not treat leading spaces in a numeric data item as if they were leading zeros, as some implementations of COBOL do. To convert leading spaces to zeros, use the INSPECT statement as Example 10-37 does.

Example 10-37 INSPECT Statement Converting Leading Spaces to Zeros

```
WORKING-STORAGE SECTION.
03  A-VALUE PICTURE X(10) VALUE "        23".
03  N-VALUE PICTURE 9(10) REDEFINES A-VALUE.
    ...
    INSPECT A-VALUE REPLACING LEADING " " BY "0".
```

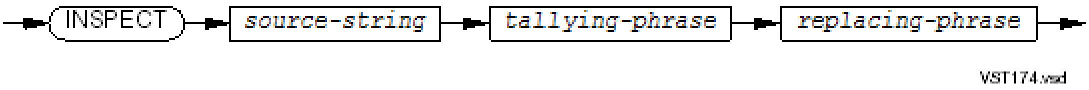
In Example 10-38, several replacements occur during the execution of one INSPECT statement. When the INSPECT statement finishes executing, ITEM-C equals “$$9,XXX.00.”

Example 10-38 One INSPECT Statement, Several Replacements

```
WORKING-STORAGE SECTION.
77  ITEM-C      PIC X(10)   VALUE "009,999.  ".
PROCEDURE DIVISION.
   INSPECT ITEM-C REPLACING ALL " " BY "0",
                  LEADING "0" BY "$",
                  ALL "9" BY "X" AFTER INITIAL ","
```

INSPECT TALLYING REPLACING

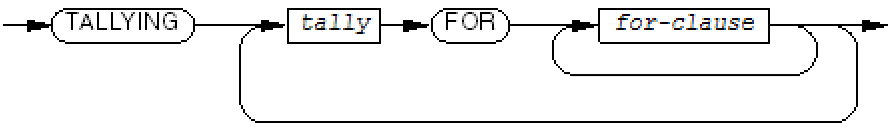
INSPECT TALLYING REPLACING counts occurrences of a sequence of one or more characters in a data item and replaces each occurrence with a specified value.



source-string

is the identifier of an elementary item or data structure with USAGE DISPLAY. Inspection proceeds from left to right within *source-string*.

tallying-phrase

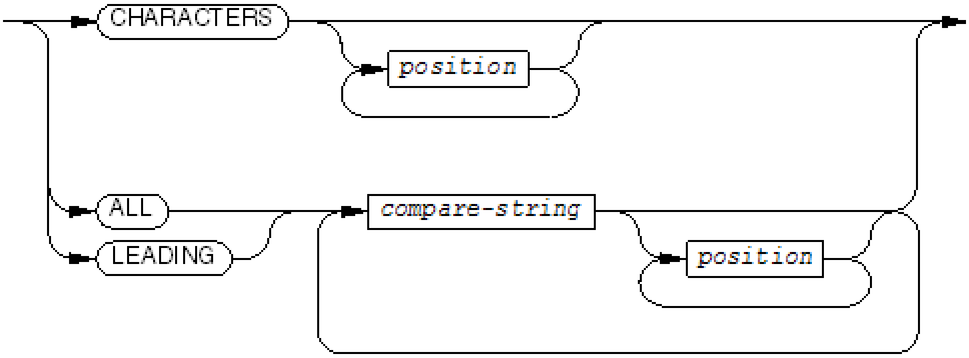


VST165.vsd

tally

is the identifier of an elementary numeric item where the number of occurrences is to be stored. The INSPECT statement does not initialize *tally* but adds to its current value. If you want *tally* to begin at a given number, set it before the INSPECT statement executes.

for-clause

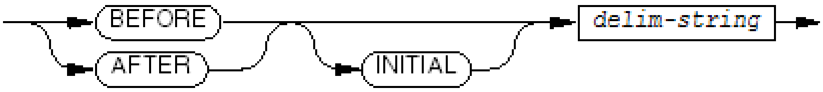


VST166.vsd

CHARACTERS

specifies that any character in *source-string* that has not satisfied a previous *compare-string* relation is counted, regardless of its value.

position



VST167.vsd

marks a beginning or ending point for an INSPECT scan cycle. A CHARACTERS phrase, ALL phrase, or LEADING phrase can have a single BEFORE phrase, a single AFTER phrase, or one BEFORE phrase and one AFTER phrase in either order.

BEFORE

specifies that scanning is up to but not including the character or characters matching *delim-string*. If INSPECT does not find *delim-string* in *source-string*, it scans all of *source-string*.

AFTER

specifies that scanning starts at the position immediately following the character or characters matching *delim-string*. If INSPECT does not find *delim-string* in *source-string*, it does not scan *source-string* .

INITIAL

is for documentation only and has no effect.

delim-string

is the identifier of any type of elementary item with USAGE DISPLAY or a nonnumeric literal. It can be a figurative constant that does not include the keyword ALL, in which case it represents one character.

ALL

specifies that all occurrences of *compare-string* are tallied.

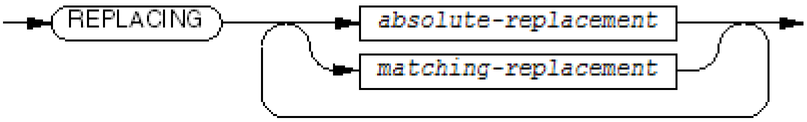
LEADING

specifies scanning for consecutive occurrences of *compare-string* beginning at the current position within *source-string*.

compare-string

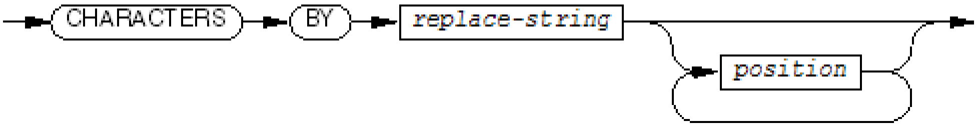
is the identifier of an elementary data item with USAGE DISPLAY or a nonnumeric literal. It can be a figurative constant (except one beginning with ALL), in which case it represents one character. The *compare-string* is the character-string being sought.

replacing-phrase



VST169.vsd

absolute-replacement



VST170.vsd

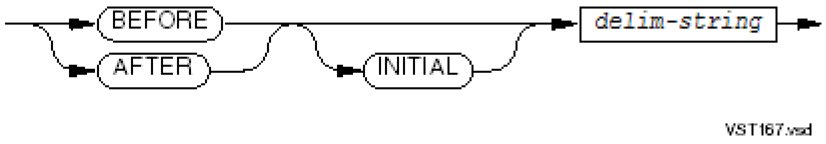
CHARACTERS

specifies that any character in *source-string* can be replaced, regardless of its value. If you use CHARACTERS, *replace-string* and *delim-string* must each be one character in length.

replace-string

is the identifier of an elementary item with USAGE DISPLAY or a nonnumeric literal. It can be a figurative constant that does not include the keyword ALL, in which case it represents one character.

position



marks a starting and ending place for a replacement cycle. It follows the same item rules as *replace-string*. A CHARACTERS phrase, ALL phrase, LEADING phrase, or FIRST phrase can have a single BEFORE phrase, a single AFTER phrase, or one BEFORE phrase and one AFTER phrase in either order.

BEFORE

specifies that scanning is up to but not including the character or characters matching *delim-string*. If INSPECT does not find *delim-string* in *source-string*, it scans all of *source-string*.

AFTER

specifies that scanning starts at the position immediately following the character or characters matching *delim-string*. If INSPECT does not find *delim-string* in *source-string*, it does not scan *source-string* .

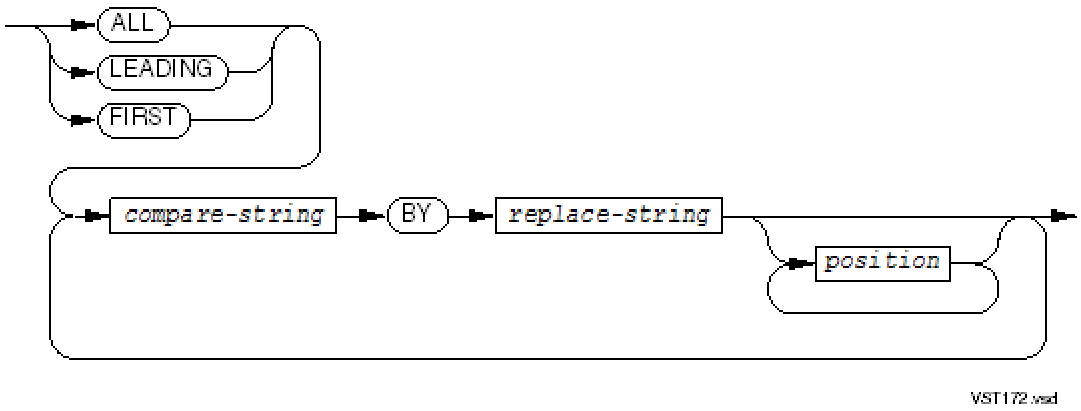
INITIAL

is for documentation only and has no effect.

delim-string

is the identifier of an elementary item with USAGE DISPLAY or a nonnumeric literal. It can be a figurative constant that does not include the keyword ALL, in which case it represents one character.

matching-replacement



defines the character-string being scanned for.

ALL

specifies that all occurrences of *compare-string* are replaced.

LEADING

specifies scanning for consecutive occurrences of *compare-string* beginning at the current position within *source-string*. For example, REPLACING LEADING “0” replaces only the first three zeros in “000120020.”

FIRST

specifies that only the first occurrence of `compare-string` is replaced.

compare-string

is the identifier of any type of elementary item with USAGE DISPLAY or nonnumeric literal. It can be a figurative constant that does not include the keyword ALL, in which case it represents one character.

replace-string

is an identifier of any type of elementary item with USAGE DISPLAY or a nonnumeric literal. It can be a figurative constant that does not include the keyword ALL, in which case it represents one character. When `replace-string` and `compare-string` are used together, they must be the same size.

position

is as described earlier for `absolute-replacement`.

Usage Considerations:

- Action of the INSPECT TALLYING REPLACING Statement

  The INSPECT TALLYING REPLACING statement is a combination of the INSPECT TALLYING and the INSPECT REPLACING statements. See INSPECT TALLYING (page 356) Usage Considerations: and INSPECT REPLACING (page 360) Usage Considerations.

- Difference Between INSPECT TALLYING REPLACING and INSPECT TALLYING Followed by INSPECT REPLACING

  The only difference between an INSPECT TALLYING REPLACING statement and an INSPECT TALLYING statement followed immediately by an INSPECT REPLACING statement with the same `source-string`s, `compare-string`s, and `delim-string` is the side effects.

  Each form of the INSPECT statement performs all operand identification at the beginning of execution. The INSPECT TALLYING REPLACING statement performs only one operand identification operation, but the INSPECT TALLYING statement followed by the INSPECT REPLACING statement performs operand identification before the INSPECT TALLYING statement and again before the INSPECT REPLACING statement. If subscripts or reference modifiers in any operands of the REPLACING clause use values that the TALLYING phrase changes, the INSPECT TALLYING REPLACING statement works differently than the INSPECT TALLYING statement followed immediately by an INSPECT REPLACING statement.

  For example, suppose X has the value “AAABBBABAB” and C is an array containing the values 0, 1, 2, 3, 4, 5, 6, 7, 8, and 9.

  Executing the three statements

  ```
  MOVE 1 TO Y
  INSPECT X TALLYING Y FOR LEADING "A"
  INSPECT X REPLACING LEADING "A" BY C (Y)
  ```

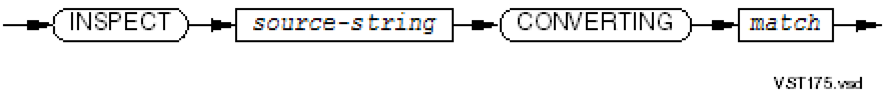
  changes the value of X to “333BBBABAB,” because the subscript Y in the REPLACING clause has the value 4 when the INSPECT REPLACING statement begins execution; however, executing the two statements

  ```
  MOVE 1 TO Y
  INSPECT X TALLYING Y FOR LEADING "A"
          X REPLACING LEADING "A" BY C (Y)
  ```

  changes the value of X to “000BBBABAB,” because the subscript Y in the REPLACING clause has the value 1 when the INSPECT TALLYING REPLACING statement begins executing.

INSPECT CONVERTING

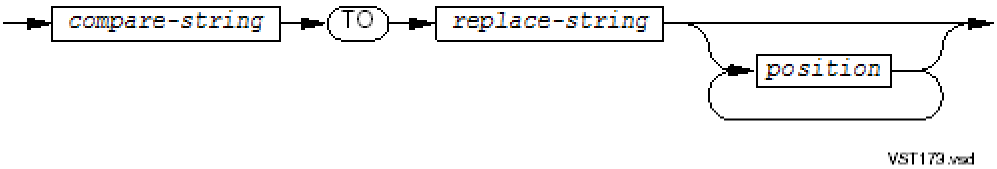
INSPECT CONVERTING performs a character-for-character replacement. It is easier to express such a replacement with the CONVERTING phrase than it is with a series of REPLACING ALL … BY … phrases.



source-string

is the identifier of an elementary item or data structure with USAGE DISPLAY. Inspection proceeds from left to right within *source-string*.

match



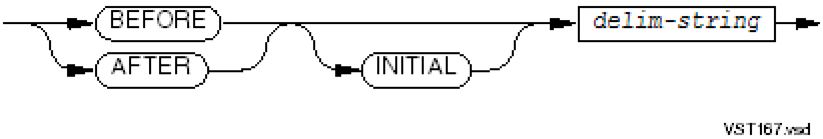
compare-string

is the identifier of any type of elementary item with USAGE DISPLAY or a nonnumeric literal. It can be a figurative constant that does not include the keyword ALL, in which case it represents one character. A character must not appear more than once in *compare-string*.

replace-string

is the identifier of any type of elementary item with USAGE DISPLAY or a nonnumeric literal, that defines what to replace the *source-string* characters with. It can be a figurative constant that does not include the keyword ALL, in which case it represents one character. The size of *replace-string* and *compare-string* must be the same.

position



marks a starting and ending place for a replacement cycle. It follows the same item rules as *replace-string*. A CONVERTING phrase can have a single BEFORE phrase, a single AFTER phrase, or one BEFORE phrase and one AFTER phrase in either order.

BEFORE

specifies that scanning is up to but not including the character or characters matching *delim-string*. If INSPECT does not find *delim-string* in *source-string*, it scans all of *source-string*.

AFTER

specifies that scanning starts at the position immediately following the character or characters matching *delim-string*. If INSPECT does not find *delim-string* in *source-string*, it does not scan *source-string*.

INITIAL

is for documentation only and has no effect.

delim-string

is the identifier of any type of elementary item with USAGE DISPLAY or a nonnumeric literal. It can be a figurative constant that does not include the keyword ALL, in which case it represents one character.

Example 10-39 shifts any lowercase letters found in IN-BUFFER to uppercase letters.

Example 10-39 INSPECT CONVERTING Statement

```
INSPECT IN-BUFFER
        CONVERTING "abcdefghijklmnopqrstuvwxyz"
                TO "ABCDEFGHIJKLMNOPQRSTUVWXYZ".
```

Usage Considerations:

- Action of the INSPECT CONVERTING Statement

  The compiler treats a statement of the form

  ```
  INSPECT source-string CONVERTING compare-string
  TO replace-string
  ```

  as if it were a statement of the form

  ```
  INSPECT source-string REPLACING
  ALL comp-char-1 BY repl-char-1
  ALL comp-char-2 BY repl-char-2 ...
  ```

  In the INSPECT CONVERTING statement, the *n* th character of *compare-string* corresponds to the *n* th character of *replace-string*.

- Restrictions on Overlapping Operands

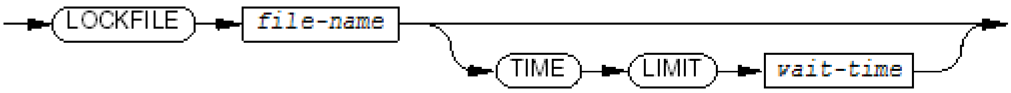
  The storage area referenced by *compare-string*, *replace-string*, or *delim-string* cannot be the same as the storage area referred to by *source-string*, and must not overlap the storage area referred to by *source-string*. Violation of this rule produces unpredictable results.

- See these usage considerations in INSPECT TALLYING (page 356):
  — How Parameter Categories Affect INSPECT
  — Comparison Operation
  — BEFORE and AFTER Phrases

LOCKFILE

LOCKFILE prevents other processes from accessing the records in a file. All records remain locked until either:

- You use an UNLOCKFILE statement.
- You close the file.
- The run unit terminates.
- You are executing a transaction with TMF and the transaction ends.



VST176.vsd

file-name

is the file description name of the file to lock. For LOCKFILE to work, *file-name* must specify a disk file.

wait-time

is a numeric literal or the name of a numeric data item. Its value must have no more than seven digits preceding any decimal point. Any fractional portion is rounded to two decimal places. For example:

```
03 WAIT-TIME   PIC 9(7)V9(2) COMPUTATIONAL.
```

A nonnegative *wait-time* is the number of seconds within which the LOCKFILE operation must finish. If the LOCKFILE operation does not finish within that time, it terminates with no error message. The I-O status code has the value "30" and the GUARDIAN-ERR register has the value 40.

If *wait-time* is negative or no *wait-time* is specified, the LOCKFILE operation has no time limit. The program can wait indefinitely.

If *file-name* was not opened with a TIME LIMITS phrase, including *wait-time* in the LOCKFILE statement causes a run-time error.

Usage Considerations:

- Action of the LOCKFILE Statement

  The LOCKFILE statement has no effect if the process has already locked the file or if the file is not a disk file. Whether effective or not, the LOCKFILE statement does not affect the key of reference, the file position indicator, or the contents of the record area associated with the file.

- I-O Status Code

  If the file has an associated file-status data item, the LOCKFILE statement assigns it an appropriate I-O status code. The possible I-O status codes that result from LOCKFILE operations are:

I-O Status Code	Meaning
"00"	LOCKFILE executed successfully.
"30"	Either the specified (nonnegative) time limit elapsed before the LOCKFILE completed (in which case, GUARDIAN-ERR is also set to 40), or the LOCKFILE operation failed due to non-COBOL causes. The file might or might not have been locked.
"42"	The file was not open.
"90"	The `wait-time` is nonnegative but the file was not opened with time limits enabled. An error message is delivered to the process's home terminal.

- One File Open Under Multiple Names

  If a file-system file is opened twice or more times, using two or more distinct file descriptions, only one of the file descriptions can have the file locked at a time.

- Declarative Procedures

  If a declarative procedure applies to the file and the time interval expires, the declarative procedure is performed, and program execution continues with the statement following the one terminated (see Expired Time Limit (page 267)). If error message 42 or 90 is returned after the declarative procedure executes, the run unit terminates abnormally.

- Interaction of LOCKFILE and READ LOCK Statements

  If your process executes a READ LOCK statement on a file that it or any other process has locked with a LOCKFILE statement, or your process executes a LOCKFILE statement against a file that has an outstanding READ LOCK, the TIME LIMIT phrase determines what happens.

If the second statement attempting to lock the file has a TIME LIMIT phrase, it keeps trying to lock the file until the time limit expires. Either it fails and then times out or it succeeds in locking the file.

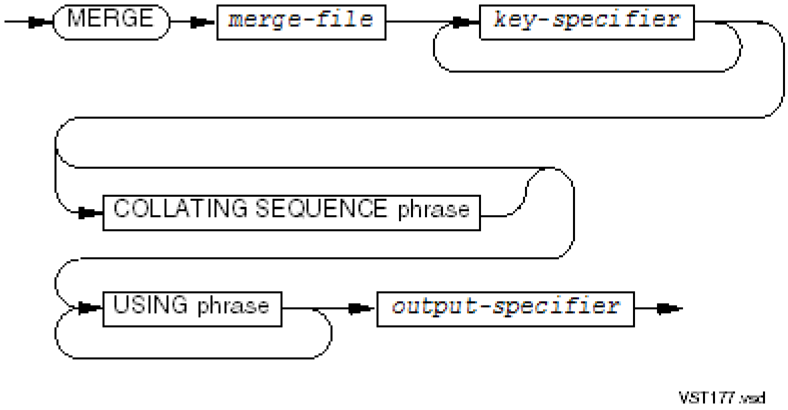
If the second statement attempting to lock the file has no TIME LIMIT phrase, it suspends execution until the statement succeeds because the contending lock is removed or until the program is terminated by an external agency (such as the TACL command STOP).

Example 10-40 LOCKFILE Statement With TIME LIMIT Phrase

```
SELECT IN-MASTER-FILE
       ASSIGN ...
       ...
       FILE STATUS IS IN-MASTER-STATUS.
...
FD IN-MASTER-FILE.
   ...
PROCEDURE DIVISION.
DECLARATIVES.
DECL SECTION.
   USE AFTER ERROR PROCEDURE ON IN-MASTER-FILE.
DECL-ROUTINE.
   IF GUARDIAN-ERR NOT = 40
      STOP RUN
   END-IF
END DECLARATIVES.
      ...
   OPEN INPUT IN-MASTER-FILE WITH TIME LIMITS SHARED
   LOCKFILE IN-MASTER-FILE
            TIME LIMIT WAIT-TIME
   IF IN-MASTER-STATUS NOT = "00"
      IF GUARDIAN-ERR = 40
         PERFORM RECOVER-MASTER-LOCK
      END-IF
   END-IF
   PERFORM UP-DATE-MASTER UNTIL DONE
   MOVE 0 TO M-FLAG
   UNLOCKFILE IN-MASTER-FILE
   ...
```

MERGE

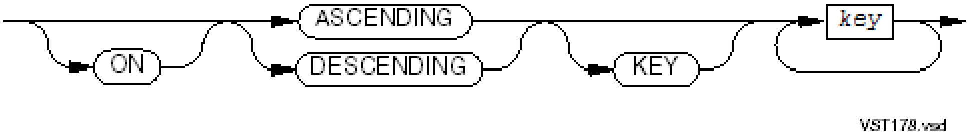
MERGE combines two or more files into another file, ordered by the same key. MERGE is performed by the FastSort utility, using files that are not open to the COBOL program. MERGE opens, reads, writes, and closes these files. MERGE can return records to an output procedure that can then write them to some file that is open to the COBOL program.



merge-file

is a sort-merge file description (SD) name. *merge-file* can have variable-length or fixed-length records. Its record description entry defines the data item or items used as the key or keys.

key-specifier



ASCENDING

specifies ascending merge order.

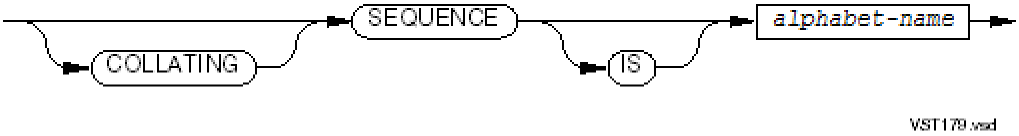
DESCENDING

specifies descending merge order.

key

is a data item to be used as a merge key. If *merge-file* has variable-length records, *key* must refer to data within the first *x* character positions of the record, where *x* is the minimum record size for *merge-file*.

COLLATING SEQUENCE phrase

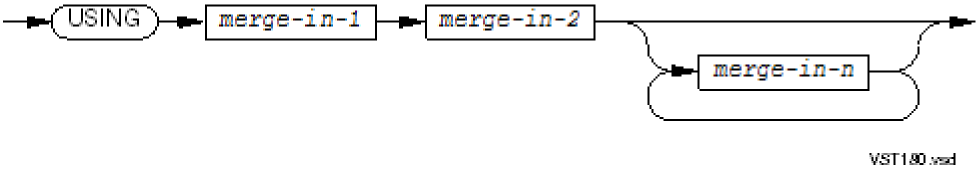


specifies a collating sequence for sorting.

alphabet-name

must be associated with a collating sequence in the SPECIAL-NAMES paragraph of the Environment Division (see SPECIAL-NAMES Paragraph).

USING phrase



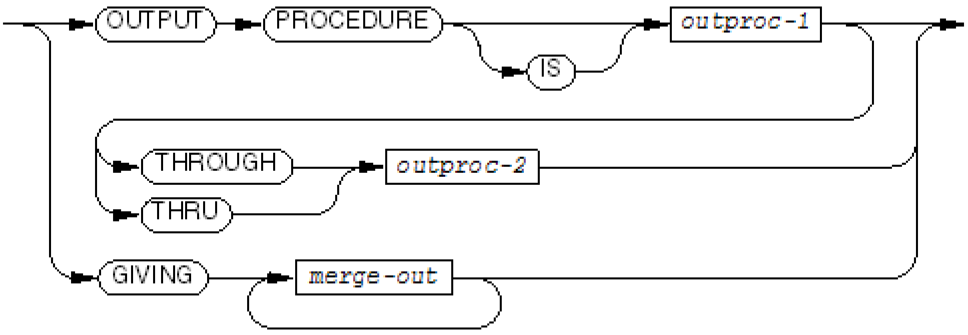
merge-in-1, *merge-in-2*, *merge-in-n*

are file description (FD) names. Multiple-reel tape files are permitted. The files can have variable-length or fixed-length records.

If *merge-file* has variable-length records, then the records of *merge-in-1*, *merge-in-2*, and *merge-in-n* must not be shorter than the shortest record that *merge-file* can have or longer than the longest record that *merge-file* can have.

If *merge-file* has fixed-length records, then the records of *merge-in-1*, *merge-in-2*, and *merge-in-n* must not be greater than that fixed length. If a record of *merge-in-1*, *merge-in-2*, or *merge-in-n* is shorter than the fixed length of each *merge-file* record, then the shorter record is space-padded on the right when it is released to *merge-file*.

output-specifier



VST181.vsd

specifies the procedure to which the merge operation is to deliver the records of the merged files, in order.

outproc-1

is the paragraph-name or section-name of the first (and maybe only) paragraph or section of the procedure to which the merge operation delivers the records of the merged files.

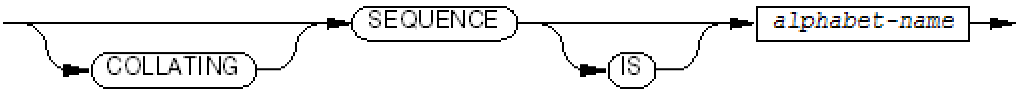
outproc-2

is the paragraph-name or section-name of the last paragraph or section of the procedure in the group of procedures to which the merge operation delivers the records of the merged files.

merge-out

is a file description (FD) name, the name of a file that results from the merge operation. The *merge-out* file can have variable-length or fixed-length records. If *merge-out* has fixed-length records, any record in *merge-file* that is shorter than that fixed length is space-padded on the right when that record is returned to *merge-out*.

COLLATING SEQUENCE phrase



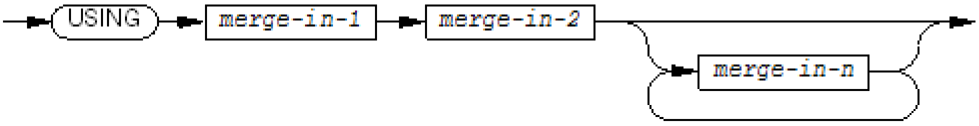
VST179.vsd

specifies a collating sequence for sorting.

alphabet-name

must be associated with a collating sequence in the SPECIAL-NAMES paragraph of the Environment Division (see SPECIAL-NAMES Paragraph).

USING phrase



VST180.vsd

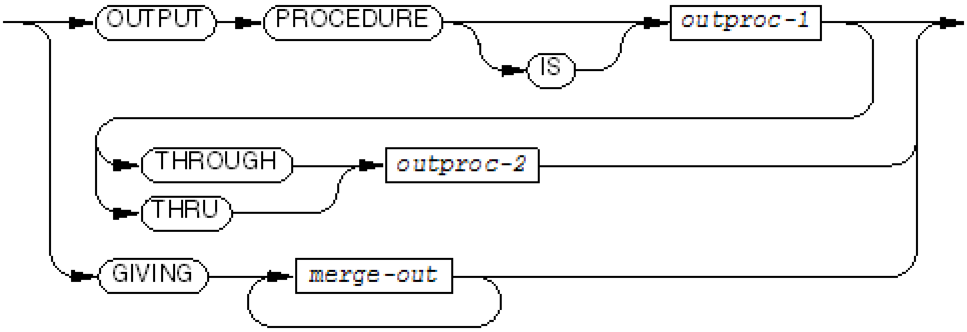
merge-in-1, *merge-in-2*, *merge-in-n*

are file description (FD) names. Multiple-reel tape files are permitted. The files can have variable-length or fixed-length records.

If *merge-file* has variable-length records, then the records of *merge-in-1*, *merge-in-2*, and *merge-in-n* must not be shorter than the shortest record that *merge-file* can have or longer than the longest record that *merge-file* can have.

If *merge-file* has fixed-length records, then the records of *merge-in-1*, *merge-in-2*, and *merge-in-n* must not be greater than that fixed length. If a record of *merge-in-1*, *merge-in-2*, or *merge-in-n* is shorter than the fixed length of each *merge-file* record, then the shorter record is space-padded on the right when it is released to *merge-file*.

output-specifier



VST181.vsd

specifies the procedure to which the merge operation is to deliver the records of the merged files, in order.

outproc-1

is the paragraph-name or section-name of the first (and maybe only) paragraph or section of the procedure to which the merge operation delivers the records of the merged files.

outproc-2

is the paragraph-name or section-name of the last paragraph or section of the procedure in the group of procedures to which the merge operation delivers the records of the merged files.

merge-out

is a file description (FD) name, the name of a file that results from the merge operation. The *merge-out* file can have variable-length or fixed-length records. If *merge-out* has fixed-length records, any record in *merge-file* that is shorter than that fixed length is space-padded on the right when that record is returned to *merge-out*.

Usage Considerations:

- Placement of MERGE Statements

  A MERGE statement cannot appear in the Declaratives Portion of the Procedure Division.

- Files Specified in the MERGE Statement

  You can merge to and from these types of files:

  — Disk files
  — Blocked tape files
  — Multiple-reel tape files
  — Tape files on a multiple-file reel

The files specified in the MERGE statement are subject to these restrictions:

- Every file record must contain all of the `key` fields.
- A file name cannot appear more than once in the same MERGE statement.
- Two files of a multiple-file tape reel cannot appear in the same MERGE statement.
- Files that appear in the same MERGE statement cannot appear in a SAME AREA or SAME SORT-MERGE AREA clause.
- Except for *`merge-out`* files, files that appear in the same MERGE statement cannot appear in the SAME RECORD AREA clause.
- A *`merge-in-1`*, *`merge-in-2`*, or *`merge-in-n`* file whose SELECT clause includes the OPTIONAL phrase must be present at execution time.

- Merge Keys

  Each merge key is subject to these restrictions:

  - The data item specified by *`key`* must be described within a record associated with *`merge-file`*. When *`merge-file`* has more than one record description entry, a merge key data item can be defined within any one of those entries.
  - No *`key`* can have an OCCURS clause or be subordinate to an item that has an OCCURS clause.
  - No *`key`* can have a variable size (that is, have an OCCURS DEPENDING clause in the description of a subordinate item).

  Keys are listed from left to right within the MERGE statement in order of decreasing significance (that is, the first *`key`* is the most significant and the last *`key`* is the least significant). ASCENDING and DESCENDING phrases do not affect keys' significance.

  When a *`key`* is in an ASCENDING phrase, the merged sequence is from the record with the lowest value in the *`key`* to the record with the highest value. When a *`key`* is in a DESCENDING phrase, the merged sequence is from the record with the highest value in the *`key`* to the record with the lowest value. In both cases, the rules for comparison of operands in a relation condition determine which value is higher.

  The results of the merge operation are predictable only when the records in each of the input files are ordered as described in the *`key-specifier`* phrases.

  The PROGRAM COLLATING SEQUENCE clause affects merge operations.

- Output Procedure

  The output procedure is *`outproc-1`* or *`outproc-1`* through *`outproc-2`*. The output procedure must have a RETURN statement. An output procedure can be any procedure needed to select, modify, or copy the records that the RETURN statement makes available to *`merge-file`* one at a time in merged order.

  The range of an output procedure includes statements that execute due to transfer of control by CALL, EXIT, GO TO, and PERFORM statements in the range of the output procedure and statements in declarative procedures that execute as a result of execution of statements in the range of the output procedure. The range of an output procedure must not cause the execution of any of these statements:

  - A MERGE, RELEASE, or SORT statement
  - A RETURN statement that specifies a sort-merge file other than *`merge-file`*
  - A statement that manipulates *`merge-in-1`*, *`merge-in-2`*, *`merge-in-n`*, or *`merge-out`* or uses the record area associated with one of these files

  If the MERGE statement is in a section that is not in an independent segment (a segment whose segment-number is from 00 through 49), then any output procedures must either be totally within dependent segments or they must be wholly contained in a single independent segment.

If the MERGE statement is in an independent segment (a segment whose segment-number is from 50 through 99), then any output procedures must either be totally within dependent segments or they must be wholly contained in the same independent segment that contains the MERGE statement.

- Execution Phases

  The execution of a MERGE statement consists of an input-and-merge phase and an output phase.

- Input-and-Merge Phase

  The input-and-merge phase transfers records from the input files *merge-in-1*, *merge-in-2*, and *merge-in-n* to *merge-file* in the order specified by the merge keys and the collating sequence *alphabet-name*.

  When the MERGE statement begins execution, *merge-file* and the input files must be closed but not locked. The input-and-merge phase performs an implicit OPEN statement on *merge-file* and each of the input files. The input-and-merge phase opens each input file in INPUT mode, performs implicit READ NEXT statements to retrieve their records, and performs implicit RELEASE statements to release the records to the *merge-file*.

  The MERGE statement determines the merging logic by retrieving the first record from each input file to form a set of candidate records. After selecting the proper candidate to transfer to *merge-file*, the merging logic retrieves the next record from that input file, if any, to replace it. This process continues until all input files are exhausted. The candidate records are first ranked in accordance with their values for the most significant *key*. When two or more records have equal values for the current *key*, that subset is then ranked in accordance with the record values for the next most significant *key*, and so on.

  When all records have been merged, the input-and-merge phase performs an implicit CLOSE statement on each input file. If the MERGE statement has an OUTPUT PROCEDURE phrase, the close operations occur after the specified procedure completes execution and returns control to the MERGE statement.

- Output Phase

  If the MERGE statement has an OUTPUT PROCEDURE phrase, the output phase transfers the merged records to the output procedure using an implicit PERFORM statement. The implicit PERFORM retrieves the *merge-file* records by executing implicit RETURN statements. When the implicit PERFORM statement finishes executing, control returns to the merge operation.

  If the MERGE statement has a GIVING phrase, the output phase transfers the merged records to one or more *merge-out* files. When the MERGE statement begins execution, each *merge-out* file must be closed but not locked. The output phase uses implicit OPEN statements to open each *merge-out* file in OUTPUT mode. Then the output phase executes implicit RETURN statements to retrieve records from the merge file and implicit WRITE statements to release them to the *merge-out* files. Finally, the output phase uses implicit CLOSE statements to close the *merge-out* files.

  Regardless of whether the MERGE statement has an OUTPUT PROCEDURE phrase or a GIVING phrase, the merge operation uses an implicit CLOSE statement to close *merge-file* after the output phase ends. Then the MERGE statement terminates execution.

  The implicit OPEN and CLOSE statements are equivalent to OPEN and CLOSE statements without optional phrases.

- How the Scratch File Is Determined

NOTE: If a scratch file is specified but its value is all spaces, assume that no scratch file was specified.

- — If COBOL_SET_SORT_PARAM_TEXT_ specifies a SCRATCH-FILE, then that file is the scratch file.
- — If the SELECT statement associated with `merge-file` specifies the =_SORT_DEFAULTS DEFINE as the `define-name-literal`, then:
- — If the =_SORT_DEFAULTS DEFINE exists and specifies a scratch file, then that file is the scratch file.
- — If the =_SORT_DEFAULTS DEFINE exists but does not specify a scratch file, then a temporary file on the volume $SYSTEM is the scratch file.
- — If no =_SORT_DEFAULTS DEFINE exists, then a temporary file on the volume $SYSTEM is the scratch file.
- — For more information on the =_SORT_DEFAULTS DEFINE, see Establishing Parameters With =_SORT_DEFAULTS DEFINE (page 618).
- — If the SELECT statement associated with `merge-file` does not specify the =_SORT_DEFAULTS DEFINE as the `define-name-literal`, then the file that the SELECT statement specifies is the scratch file.

- How the Volume of the Swap File Is Determined

NOTE: If a swap file is specified but its value is all spaces, assume that no swap file was specified.

The operating system assigns a swap file to swap pages in and out of memory while the compiler is running. The swap file mirrors all of the data areas that the compiler uses. The ideal swap file is a fast device that is neither busy nor mirrored. To redirect the swap file, give `define-name-literal` the value =_SORT_DEFAULTS.

The swap file is a temporary file with a volume but no subvolume. If you specify a swap file, including a volume, the volume is used but the `file-id` is not. If you specify only a `file-id`, the default volume is used.

- — If COBOL_SET_SORT_PARAM_TEXT_ specifies a SWAP-FILE, then the swap file is created on that file’s volume.
- — If the SELECT statement associated with `merge-file` specifies the =_SORT_DEFAULTS DEFINE as the `define-name-literal`, then:
  - ◦ If the =_SORT_DEFAULTS DEFINE exists and specifies a swap file, then the swap file is created on that file’s volume.
  - ◦ If the =_SORT_DEFAULTS DEFINE exists but does not specify a swap file, then the swap file is created on the volume used for the scratch file.
  - ◦ If no =_SORT_DEFAULTS DEFINE exists, then the swap file is created on the volume used for the scratch file.

  For more information on the =_SORT_DEFAULTS DEFINE, see Establishing Parameters With =_SORT_DEFAULTS DEFINE (page 618).
- — If the SELECT statement associated with `merge-file` does not specify the =_SORT_DEFAULTS DEFINE as the `define-name-literal`, then the swap file is created on the volume of the file that the SELECT statement specifies.

In Example 10-41, a MERGE statement merges two files to produce a third.

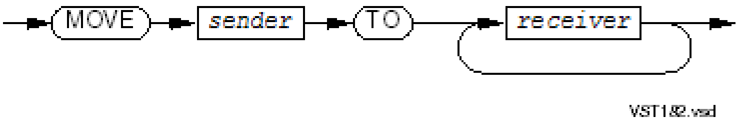
Example 10-41 MERGE Statement

```
INPUT-OUTPUT SECTION.
FILE-CONTROL.
  SELECT PARTS-ON-HAND ASSIGN TO "H215432.ONHAND"
     ORGANIZATION IS SEQUENTIAL
     ACCESS MODE IS SEQUENTIAL.
  SELECT PARTS-RECEIVED ASSIGN TO "H215432.RECD"
     ORGANIZATION IS SEQUENTIAL
     ACCESS MODE IS SEQUENTIAL.
  SELECT PARTS-TOGETHER ASSIGN TO "H215432.TGTHR"
     ORGANIZATION IS SEQUENTIAL
     ACCESS MODE IS SEQUENTIAL.
  SELECT MERGER ASSIGN TO "$HISPD.#TEMP".
DATA DIVISION.
  ...
FILE SECTION.
  FD PARTS-ON-HAND
     RECORD CONTAINS 120 CHARACTERS
     LABEL RECORDS ARE OMITTED.
  01 P-O-H.
     05 PARTNO    PIC 9(8).
     ...
  FD PARTS-RECEIVED
     RECORD CONTAINS 120 CHARACTERS
     LABEL RECORDS ARE OMITTED.
  01 P-R.
     05 PARTNO    PIC 9(8).
     ...
  FD PARTS-TOGETHER
     RECORD CONTAINS 120 CHARACTERS
     LABEL RECORDS ARE OMITTED.
  01 P-T.
     05 PARTNO    PIC 9(8).
     ...
  SD MERGER
     RECORD CONTAINS 120 CHARACTERS.
  01 PARTNO       PIC 9(8).
  ...
PROCEDURE DIVISION.
  ...
  MERGE MERGER ON ASCENDING KEY PARTNO OF MERGER
        USING PARTS-ON-HAND
              PARTS-RECEIVED
        GIVING PARTS-TOGETHER
  ...
```

MOVE

MOVE TO

MOVE TO copies data from a data item and stores it in one or more other data items.



sender

is the literal or the identifier of the data item from which data is copied.

receiver

is the identifier of a data item to which the data is to be copied.

Usage Considerations:

- Evaluation of Subscripts and Indexes

  Subscripts or indexes in *sender* are evaluated only once, immediately before data is moved to the first *receiver*. Subscripts or indexes in receivers are evaluated in the order in which the receivers are specified. For example,

  ```
  MOVE A(B) TO B C(B)
  ```

  is equivalent to

  ```
  MOVE A(B) TO temp
  MOVE temp TO B
  MOVE temp TO C(B)
  ```

- MOVE Statements That Are Not Recommended

  Do not attempt these moves:

  — SPACE, alphanumeric edited, or alphabetic data item to numeric or numeric edited data item

  — ZERO, numeric literal, numeric, or numeric edited data item to alphabetic data item

- MOVE Conventions

  Table 10-3 summarizes the legal MOVE TO statements by category of sending and receiving data items. Data is converted and stored according to the category of the receiving data item.

Table 10-3 Summary of Legal Moves

		Category of Receiving Data Item			
Category of Sending Data Item		**Alphabetic**	**Alphanumeric**	**Numeric**	**National**
Alphabetic		Yes	Yes	No	No
Alphanumeric	Edited	Yes	Yes*	No	No
	Non-Edited	Yes	Yes	Yes	No
Numeric	Integer	No	Yes	Yes	No
	Noninteger	No	Yes	Yes	No
	Edited	No	Yes	Yes	No
National		No	No	No	Yes

* Although it is possible to move an alphanumeric item to a numeric item, the presence of a nonnumeric character in the data moved makes any operation on the numeric item unpredictable.

- Alphabetic, Alphanumeric, Alphanumeric Edited, or National Receiving Data Items

  Data is stored beginning at the leftmost position of *receiver*.If *sender* is shorter than *receiver*, spaces are filled according to standard alignment rules. If *sender* is longer than *receiver*, the value of *sender* is truncated on the right to the length of *receiver*. Any editing required by *receiver* is performed unless *sender* is a group data item.

  If *sender* is a signed numeric data item, the operational sign is not moved to *receiver*. If sender is described with the SEPARATE phrase in the SIGN clause, the size of *sender* is considered to be one character less than the actual size of *sender*.

  If *sender* is a numeric item with the scaling factor (`P`) in its rightmost digits, positions containing `P` are considered to contain zeros.

  If *sender* is a numeric item with decimal digits (that is, *sender* has `V` in its PICTURE clause), the decimal point is ignored and *sender* is treated as if it were an alphanumeric item whose size is equal to the number of *9*s in the PICTURE clause. If its USAGE is not DISPLAY, it is converted to DISPLAY for the move operation. For example:

  ```
  PIC 999V99 VALUE 123.45
  ```

is treated as if it were PIC X(5); that is, as the value 12345.

If the number of digits in *sender* is greater than 18 (for example, when a NUMVAL function is used), the leftmost 18 digits are moved and any to the right of this are discarded.

The COBOL standard requires that the numeric item have no decimal positions. HP allows this as an extension.

If an elementary *receiver* is described as JUSTIFIED, see JUSTIFIED Clause (page 228).

- Numeric or Numeric Edited Receiving Data Items

  Data is aligned by decimal point and filled with zeros as necessary.

  If *receiver* is signed, the sign of *sender* is moved to *receiver*. If *sender* has no sign, the new sign of *receiver* is positive.

  If *receiver* is not signed, the absolute value of *sender* becomes the new value of *receiver*.

  If *sender* is alphanumeric, its value is treated as an unsigned numeric integer.

  If *sender* is numeric-edited, the operand's unedited numeric value is determined (it can be signed). The unedited numeric value of *sender* is moved to *receiver*.

- Group MOVE Statements

  Group MOVE statements (MOVE statements in which *sender*, *receiver*, or both are data structures) are treated as alphanumeric-to-alphanumeric moves, with no data conversion. Receiving items are filled without regard to individual or subordinate items in either *sender* or *receiver*s. If a *receiver* is an elementary item defined with a JUSTIFIED clause, justification does occur. If *sender* is a national data item, its size is assumed to be twice the number of national characters. Verify the receiving data structure can hold all of the national characters.

In Example 10-42, several MOVE TO statements set the values of two records in working storage.

Example 10-42 MOVE TO Statements

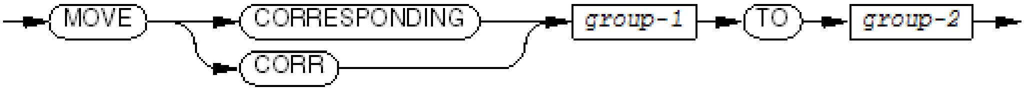
```
WORKING-STORAGE SECTION.
 01   RECORD-IN.
     03  ITEM-A     PIC X(5)    VALUE "AAAAA".
     03  ITEM-B     PIC 99V99   VALUE "2.38".
        03  ITEM-C     PIC ZZ,ZZ9.99.
        ...
 01   TEMPS.
     03  TEMP1      PIC X(4).
     03  TEMP2      PIC X(8).
     03  TEMP3      PIC 9(5)V999.
     03  TEMP4      PIC 9V9.
     03  TEMP5.
         05 T       PIC 99 OCCURS 12 TIMES.
 PROCEDURE DIVISION.
 BEGIN-PROCESSING.
    MOVE ITEM-A TO TEMP1
*                        TEMP1 is set to "AAAA"
    MOVE ITEM-A TO TEMP2
*                        TEMP2 is set to "AAAAA  "
    MOVE ITEM-B TO TEMP3
*                        TEMP3 is set to 00002v380
*                        where "v" marks the decimal location
    MOVE ITEM-B TO TEMP4
*                        TEMP4 is set to 2v3
*                        where "v" marks the decimal location
    MOVE SPACES TO RECORD-IN
*                        RECORD-IN is set to "         "
*                                             (9 spaces)
```

```
    MOVE ZEROS TO ITEM-B
*                         ITEM-B is set to 00v00
*                         where "v" marks the decimal location
    MOVE 5280 TO ITEM-C.
*                         ITEM-C is set to " 5,280.00"
    MOVE ITEM-C TO TEMP-3.
*                         TEMP-3 is set to 05280v000
*                         where "v" marks the decimal location
    MOVE ZEROS TO TEMP-5.
*                         Each 2-digit T in TEMP-5 is set to
*                         the 2-character value "00".  Do not
*                         move ZEROS to any group containing
*                         COMP or NATIVE-n items, because the
*                         operation delivers characters.
*                         Do not move LOW-VALUES to a group
*                         of numeric items -- instead use
*                         INITIALIZE ...
*                         REPLACING NUMERIC DATA BY ZERO
```

MOVE CORRESPONDING

MOVE CORRESPONDING copies elements of one group to corresponding elements of another group. Groups of data correspond if they have the same names and qualifier names, beyond the group-names in the MOVE statement, and if they meet restraints explained under CORRESPONDING Phrase (page 253).

CAUTION: MOVE CORRESPONDING is not recommended, because minor changes to one group can change the correspondence between its elements and those of the other group, and this is difficult to detect. If someone adds a new name to a group, the new name might be included in unintended move operations.



VST183.vsd

group-1

is the group-name of the data to be copied.

group-2

is the group-name to which the data is to be copied.

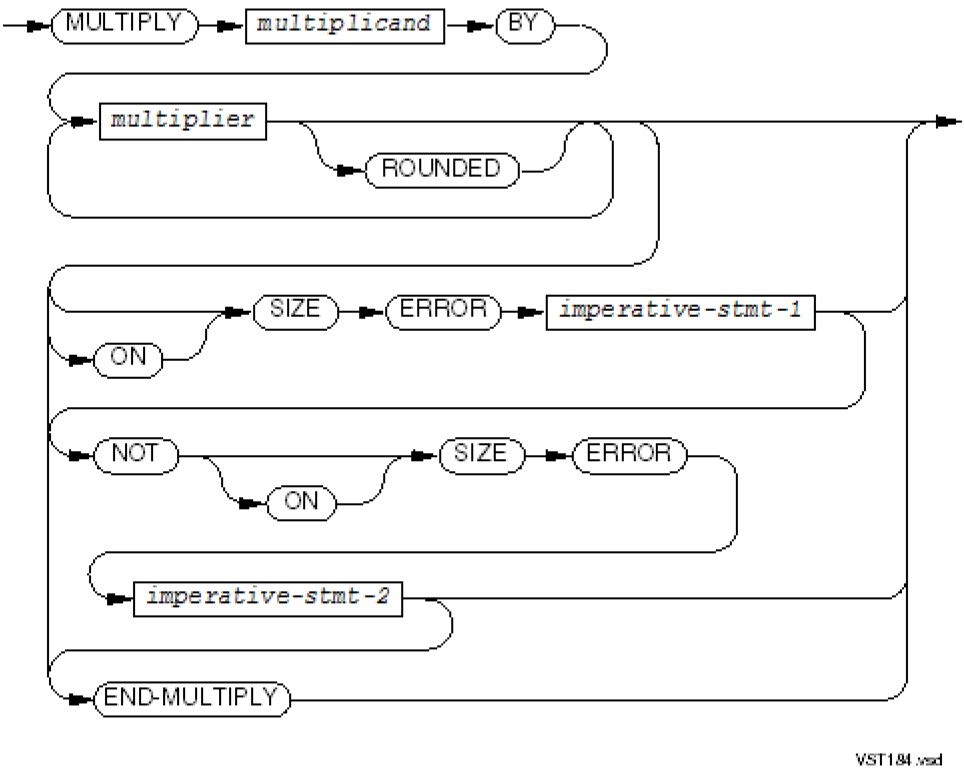
MULTIPLY

MULTIPLY BY

MULTIPLY BY multiplies one data item by one or more other data items and stores the result(s) in the other data item(s); for example, the statement

```
MULTIPLY A BY B C D
```

means store A x B in B, A x C in C, and A x D in D.



multiplicand

is a numeric literal or the identifier of an elementary numeric data item.

multiplier

is the identifier of an elementary numeric data item.

ROUNDED

specifies that the product is to be rounded before being stored as the new value of *multiplier*.

imperative-stmt-1

is an imperative statement to be executed when a size error is detected in the multiplication or in storing the product in *multiplier*.

imperative-stmt-2

is an imperative statement to be executed when no size error is detected in the multiplication or in storing the product in *multiplier*.

END-MULTIPLY

ends the scope of the MULTIPLY statement and makes it a delimited-scope statement. If you omit END-MULTIPLY but include the SIZE ERROR or NOT SIZE ERROR phrase, the MULTIPLY statement is a conditional statement and ends at the next period separator.

Usage Considerations:

- Mathematics

  The statement

  ```
  MULTIPLY A BY B C D
  ```

  means store A x B into B, A x C into C, and A x D into D.

- Repeating a Multiplier

  If more than one *multiplier* specifies the same data item, the final value of that data item reflects more than one multiplication by *multiplicand*. For example,

  MULTIPLY 4 BY A B B

results in A being replaced by 4 x A and B being replaced by 16 x B.

- Operand Identification

  For each *multiplier*, operand identification occurs just prior to the multiply-and store operation. For example, in the statement

  ```
  MULTIPLY 4 BY A B C(B)
  ```

  the subscript is not evaluated until B has been multiplied by 4. See Operand Identification (page 252).

- Arithmetic Operations

  See Arithmetic Operations (page 267) for information on data conversion and alignment, intermediate results, multiple results (and subscript evaluation), and incompatible data.

- Precision

  If you omit the SIZE ERROR phrase, arithmetic overflow can cause the run unit to terminate abnormally. For information on precision of multiplication, see Arithmetic Precision (page 272).

- ROUNDED and SIZE ERROR Phrases

  See ROUNDED Phrase (page 254) and SIZE ERROR Phrase (page 254) for information on these.

Example 10-43 converts a length in feet to a length in inches.

Example 10-43 MULTIPLY BY Statement

```
05 LENGTH        PICTURE S9(6)V9(6).
   ...
   MULTIPLY 12 BY LENGTH
```

Example 10-44 converts several dimensions from centimeters to inches, with rounding.

Example 10-44 MULTIPLY BY Statement With ROUNDED Phrase

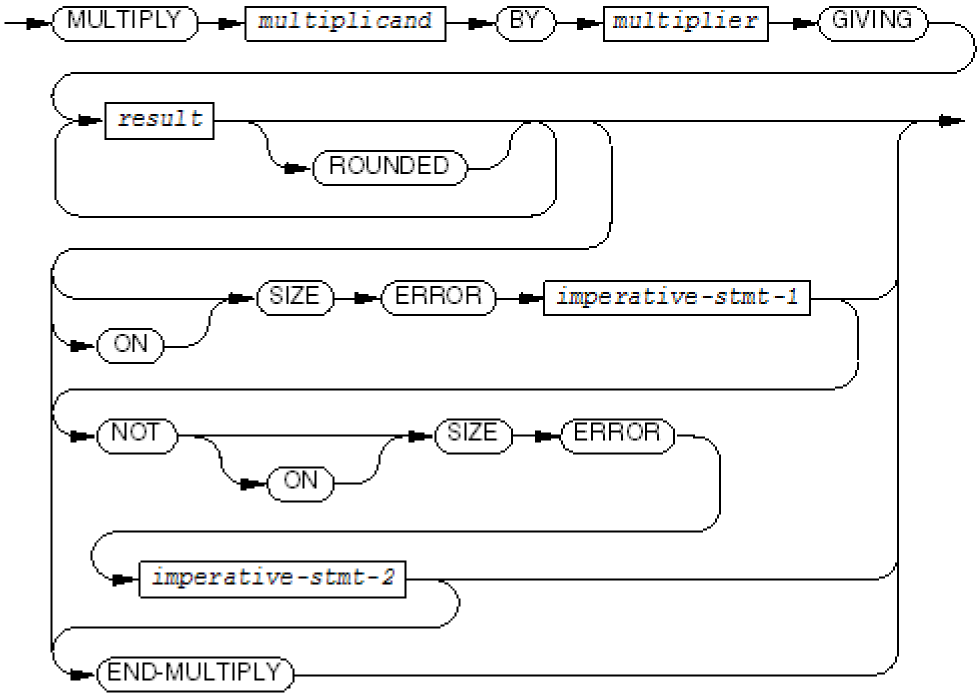
```
03 CM-TO-INCHES          PICTURE S9V99
                         VALUE 0.39.
     ...
03 LENGTH                PICTURE S9(3)V9(2).
03 WIDTH                 PICTURE S9(3)V9(2).
03 DEPTH                 PICTURE S9(3)V9(2).
     ...
   MULTIPLY CM-TO-INCHES BY LENGTH ROUNDED
                            WIDTH ROUNDED
                            DEPTH ROUNDED
```

MULTIPLY GIVING

MULTIPLY GIVING multiplies two data items and stores the product in one or more other data item(s); for example, the statement

```
MULTIPLY A BY B GIVING C D E
```

means store A x B in C, D, and E.



VST185.vsd

multiplicand

is a numeric literal or the identifier of an elementary numeric data item.

multiplier

is a numeric literal or the identifier of an elementary numeric data item.

result

is the identifier of an elementary numeric or numeric edited data item that is to receive the product.

ROUNDED

specifies that the product is to be rounded before being stored in *result*.

imperative-stmt-1

is an imperative statement to be executed when a size error is detected in the multiplication or in storing the product in *result*.

imperative-stmt-2

is an imperative statement to be executed when no size error is detected in the multiplication or in storing the product in *result*.

END-MULTIPLY

ends the scope of the MULTIPLY statement and makes it a delimited-scope statement. If you omit END-MULTIPLY but include the SIZE ERROR or NOT SIZE ERROR phrase, the MULTIPLY statement is a conditional statement and ends at the next period separator.

Usage Considerations:

- Mathematics

  The statement

  ```
  MULTIPLY A BY B GIVING C D E
  ```

means store A x B into C, D, and E. The values of A and B do not change.

- Operand Identification

  For each *result*, operand identification occurs just prior to the storage operation. For example, in the statement

  ```
  MULTIPLY A BY B GIVING C D(C)
  ```

  the subscript is not evaluated until C has been set to A x B. See Operand Identification (page 252).

- Arithmetic Operations

  See Arithmetic Operations (page 267) for information on data conversion and alignment, intermediate results, multiple results (and subscript evaluation), and incompatible data.

- Precision

  If you omit the SIZE ERROR phrase, arithmetic overflow can cause the run unit to terminate abnormally. For information on precision of multiplication, see Arithmetic Precision (page 272).

- ROUNDED and SIZE ERROR Phrases

  See ROUNDED Phrase (page 254) and SIZE ERROR Phrase (page 254) for information on these.

- Edited Result

  The MULTIPLY GIVING statement can produce an edited result, as Example 10-45 shows.

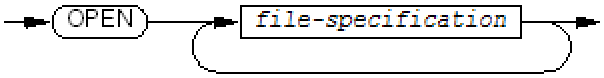
Example 10-45 MULTIPLY GIVING Statement

```
03 UNIT-PRICE      PICTURE S9(5)V999  COMPUTATIONAL.
03 ORDERED         PICTURE S9(5)      COMPUTATIONAL.
03 NET-PRICE       PICTURE S9(10)V999 COMPUTATIONAL.
03 NET-PRICE-DSP   PICTURE $$,$$$,$$$,$$$.99.
   ...
   MULTIPLY UNIT-PRICE BY ORDERED
            GIVING NET-PRICE-DSP ROUNDED
                   NET-PRICE
```

OPEN

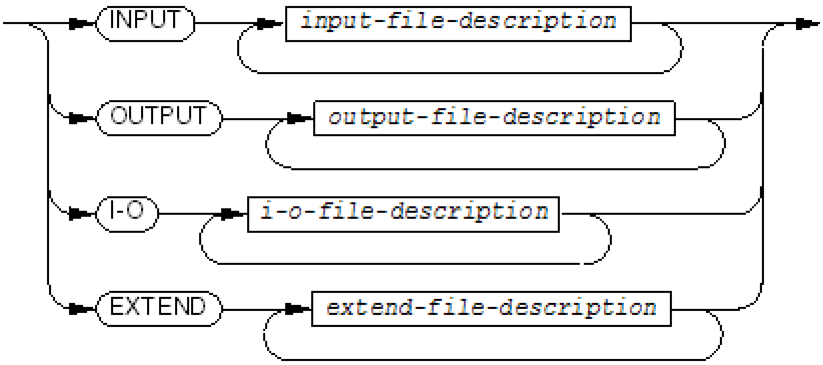
OPEN makes a file accessible for input, output, or both. OPEN associates a COBOL file name within the program with a file name known to the file system.

The NonStop operating system treats processes as files, so an HP COBOL program can open a terminal or another process as a file. The mode in which such files are opened determines how they are treated (see Table 5-4: File Open Modes (page 93)).



VST186.vsd

file-specification



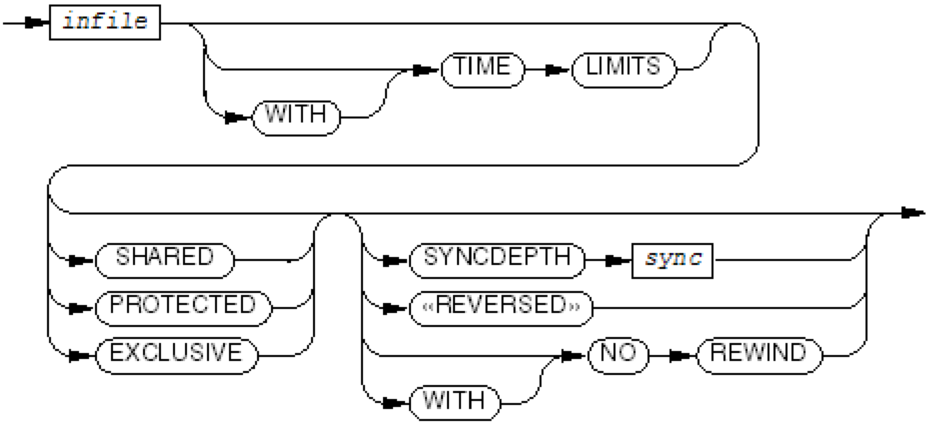
VST187.vsd

INPUT

specifies that the file or files in *input-file-description* are being opened for reading only.

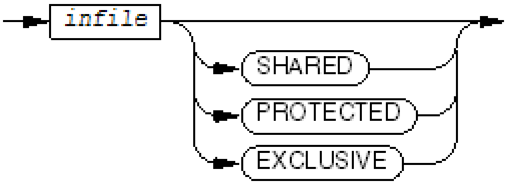
input-file-description

for a sequential, relative, indexed, or queue file:



VST188.vsd

for a line sequential file:



VST627.vsd

infile

is the file description file name of a file to open in INPUT mode, for read operations only.

TIME LIMITS

allows you to use the TIME LIMIT phrase in LOCKFILE, READ, and START statements that apply to *infile*.

SHARED

allows other processes to read or write the file while this process is open. SHARED is the default for terminals.

PROTECTED

allows other processes to read but not write the file while this process is open. PROTECTED is the default for input files that are not terminals.

EXCLUSIVE

prevents other processes from reading or writing the file while this process is open. EXCLUSIVE is the default for all other files.

sync

is a numeric literal that:

- Specifies the number of write requests that cannot be tried again whose I-O status code is to be recorded by the file system.
- Determines the number of write operations that the primary process of a process pair can perform on the file without doing a checkpoint to its backup process.
- If *sync* is 0, no checkpointing or automatic disk recovery is performed. To allow recovery from the failure of a processor that is controlling the disk during a write operation, the value of *sync* must be 1 or greater. The default value of *sync* is 1.

REVERSED

NOTE: The 1985 COBOL standard classifies REVERSED as **obsolete**, so you are advised not to use it.

is ignored with a warning.

NO REWIND

specifies that a tape is not to be rewound. This phrase applies to single-reel, single-file tapes; single-reel, multiple-file tapes; and multiple-reel, single-file tapes. If NO REWIND is specified for other types of files, it is ignored. If NO REWIND is not specified for a tape file, the tape is rewound.

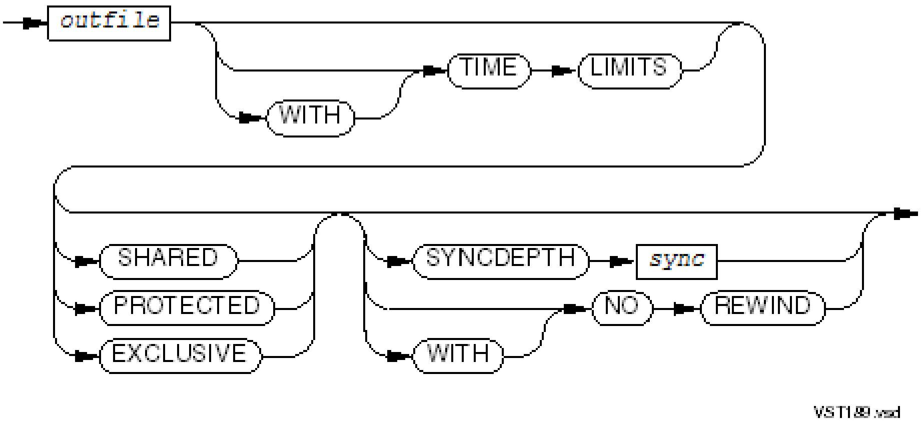
The tape must be positioned at the beginning of the file prior to the execution of the OPEN statement. No file repositioning is done.

OUTPUT

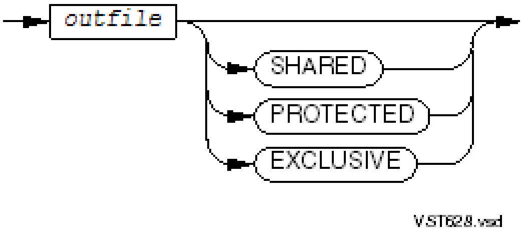
specifies that the file or files in *output-file-description* are being opened for writing only.

output-file-description

for a sequential, relative, indexed, or queue file:



for a line sequential file:



outfile

is the file description file name of a file to open in OUTPUT mode, for write operations only.

TIME LIMITS, SHARED, PROTECTED, EXCLUSIVE, *sync*, NO REWIND

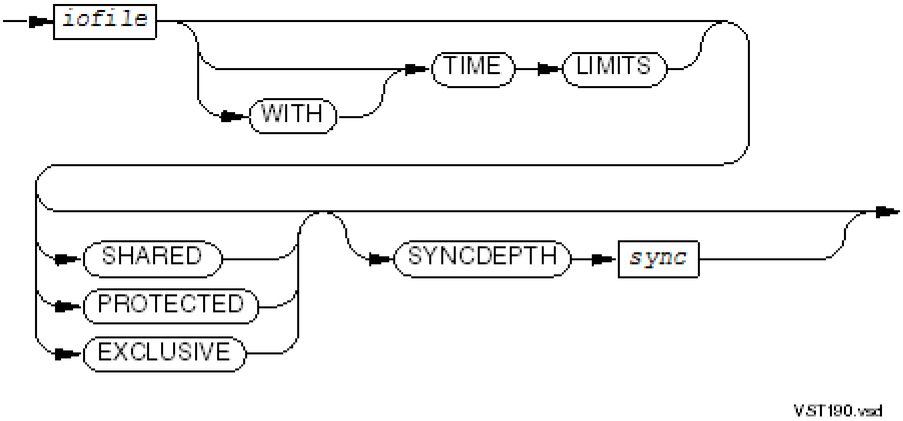
are the same as described earlier for *infile*.

I-O

specifies that the file or files in *i-o-file-description* are being opened for reading or writing.

i-o-file-description

for a sequential, relative, indexed, or queue file:



I-O mode is not supported for line sequential files.

iofile

is the file description file name of a file to open in I-O mode, for both read and write operations.

TIME LIMITS, SHARED, PROTECTED, EXCLUSIVE, *sync*

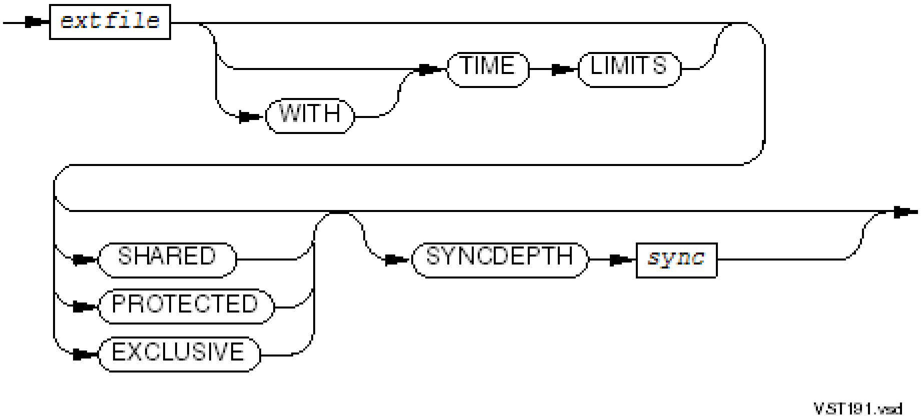
are the same as described earlier for *infile*.

EXTEND

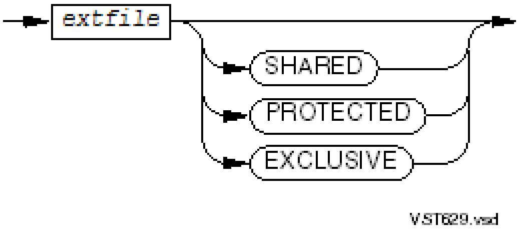
specifies that the file or files in *extend-file-description* are being opened for writing additional data following any existing data.

extend-file-description

for a sequential, relative, indexed, or queue file:



for a line sequential file:



extfile

is the file description file name of a sequential file to open in EXTEND mode, for write operations that append records to the file. The file is positioned after the last logical record when opened. All operations on the file must be write operations, as if the file had been opened in OUTPUT mode. If the file is an Enscribe unstructured file, its size must be a multiple of the record size.

TIME LIMITS, SHARED, PROTECTED, EXCLUSIVE, *sync*

are the same as described earlier for `infile`.

Usage Considerations:

- Associating COBOL File Names With System File Names

  Before you can open a file, you must associate its COBOL file name with the name of a system file name, using one of:

  — The ASSIGN clause of the file-control entry for the COBOL file (see FILE-CONTROL Paragraph (page 127))

  — The TACL command ASSIGN (see ASSIGN Command (page 590))

- — A DEFINE of the class MAP, SPOOL, or TAPE (see DEFINEs (page 601))
  - — The COBOL_ASSIGN_ routine (see COBOL_ASSIGN_ (page 654))
- Devices That You Can Open

  The OPEN statement can open devices that accept normal read and write operations (as opposed to read and write operations that require special control information). This table shows which file organizations are compatible with which devices:

File Organization	Devices Allowed	Device Type Numbers
Relative, indexed, or queue	Disk file	3
Sequential without LINAGE	Process $RECEIVE Disk file Tape Printer Page mode terminal Envoy line Card reade X25 treated as a process COMM system manager	0 2 3 4 5 6 7 8 9 50
Sequential with LINAGE	Process Printer	0 5

- Successful and Unsuccessful Open Operation

  Execution of an OPEN statement sets an I-O status code for each file specified in the statement. Successful opening of a file sets the I-O status code of that file to “00,” “05,” “07,” or “97.” Unsuccessful opening of a file sets the I-O status code to one of the values in Table 10-4 and causes execution of any applicable USE AFTER EXCEPTION CONDITION procedure.

  If you declared a file-status data item for a file (see FILE-CONTROL Paragraph (page 127)), the process stores the I-O status code in that data item. For an extensive discussion of I-O status codes, see I-O Status Code (page 257).

Table 10-4 I-O Status Codes for Unsuccessful Open Operations

I-O Status Code	Unsuccessful Open Operation
“30”	One of: — The file was to be opened for OUTPUT and an existing disk file with unsuitable attributes cannot be purged. — The file was to be opened for OUTPUT and the data in an existing disk file cannot be purged. — The file was to be opened for EXTEND and the file cannot be positioned at its end. — The open operation failed due to some cause outside COBOL.
“35”	The file was to be opened for INPUT, I-O, or EXTEND; the file is not optional; and the file does not exist.
“37”	One of: — The file was to be opened for INPUT, but the device is not suitable for INPUT. — The file was to be opened for OUTPUT, but the device is not suitable for OUTPUT. — The file was to be opened for I-O, but the device is not suitable for I-O (both input and output). Only disks and terminals can be opened for I-O.
“38”	The file is locked.

Table 10-4 I-O Status Codes for Unsuccessful Open Operations *(continued)*

I-O Status Code	Unsuccessful Open Operation
"39"	The attributes of the file under the file system do not correspond to the attributes specified for the file in the program, for example: — The file was to be opened for INPUT but it is assigned to a printer device. — The file description includes a LINAGE clause, but the file is assigned to a tape drive. — The file description includes a MULTIPLE FILE TAPE clause, but the file is not assigned to a tape drive. — The file is described as having keys, but the file is assigned to a nondisk file. — The prime record key attributes of a disk file are inconsistent with the description for the COBOL file. — The alternate record key attributes of a disk file are inconsistent with the description for the COBOL file.
"41"	The program already has that file open, or the file is one of several on a tape reel and another of the files on the reel is open.
"91"	The file is an EDIT format, unstructured disk file and the call to the operating system routine that reads EDIT files failed. The open operation cannot allocate a resource required to process the file.

- Labeled Files

  When the LABEL RECORDS clause in the file description entry indicates that label records are present, the beginning file or reel labels are processed in accordance with the conventions for the specified open mode. If the file is opened for INPUT or EXTEND, the label records are verified. If the file is opened for OUTPUT, the label records are created. If the device assigned is not a tape, the LABEL RECORDS clause is ignored.

- File Position Indicator

  The file position indicator specifies the next record to be accessed within the opened file during certain sequences of input-output operations. For files opened for INPUT or I-O, it is defined to be just before the first record in the file. For files opened for OUTPUT or EXTEND, it is undefined.

  When a file is opened for OUTPUT, run-time routines position the file at its beginning point. The first WRITE statement executed for the file creates the first logical record in the file.

  When a file is opened for EXTEND, run-time routines position the file to point immediately following the last existing logical record, which is defined:

File Organization	Last Existing Record
Sequential	Last record written to the file
Relative	Existing record with highest relative record number
Indexed or queue	Existing record with highest prime record key value

  If a file opened for EXTEND is empty, the first WRITE statement executed for the file creates the first logical record. If the file is not empty, the first WRITE statement executed for the file creates the successor record to the current last record.

  If a file opened for EXTEND is an Enscribe unstructured file, its size must be a multiple of the record size.

- Key of Reference

  For a relative file, the key of reference is the relative key. For an indexed or queue file, the key of reference is the prime record key.

- Locked System Files

  An HP COBOL program cannot open a system file that it closed and locked earlier in its current execution.

- Opening a File Multiple Times, Simultaneously

  An HP COBOL program can open one system file under more than one COBOL file name. This is particularly helpful when you are developing a server. You can open $RECEIVE twice—once for INPUT and once for OUTPUT—and use TACL ASSIGN commands to make one terminal (or obey file) your input simulator and another terminal your output monitor.

  If an HP COBOL program opens a file with a certain COBOL file name, it must close that COBOL file (using that COBOL file name) before it can open it again with that COBOL file name.

- Operations by Other Input-Output Statements

  Before any other input-output statement can operate upon a file, an OPEN statement must make the file accessible.

  Whether you can use a verb to operate on a file depends on the mode in which you open the file (INPUT, OUTPUT, I-O, or EXTEND), the file's organization and access mode, and the device with which the file is associated.

  The READ, WRITE, REWRITE, START, and DELETE statements can operate only on files opened with certain options. Table 10-5 summarizes the open modes that make files having different organizations and access modes accessible to these statements.

  The START and DELETE statements can operate only on files associated with disk devices.

  The REWRITE statement can operate only on files associated with disk devices.

  The CLOSE, LOCKFILE, UNLOCKFILE, and UNLOCKRECORD statements can operate on any open file.

- Direct Calls to NonStop Operating System I-O Procedures

  If you open a file with the OPEN statement, do not perform I-O operations on that file by making direct calls to NonStop operating system I-O procedures. If you do, the results are undefined.

Table 10-5 I-O Statements You Can Use in Different Open Modes

	Open Mode			
I-O Statement	**INPUT**	**OUTPUT**	**I-O**	**EXTEND**
READ	ALL		ALL	
WRITE		ALL	Org: R, I Acc: R, D	Org: S, R, I Acc: S
REWRITE			ALL	
START	Org: S*, R, I Acc: S, D		Org: S*, R, I Acc: S, D	
DELETE			Org: R, I	
Org is the file's organization:		I = Indexed or queue		
		R = Relative		
		S = Sequential		
		S* = Sequential (but only for START specifying an alternate key)		
Acc is the file's access mode:		S = Sequential		
		R = Random		

Table 10-5 I-O Statements You Can Use in Different Open Modes *(continued)*

	Open Mode			
I-O Statement	INPUT	OUTPUT	I-O	EXTEND
		D = Dynamic		
ALL means all organizations and access modes				

- Nonexistent Files

  When the INPUT, I-O, or EXTEND phrase applies and the SELECT clause includes the OPTIONAL phrase, the run-time routines determine whether or not the file is present.

  If the file is not present, one of these occurs:

  — If the I-O or EXTEND phrase applies, the open operation creates a file. This creation occurs as if these statements were executed in the order shown:

  ```
  OPEN OUTPUT file-name.
  CLOSE file-name.
  ```

  The OPEN statement as specified in the source program is then executed. If the file is defined to have alternate keys, the creation attempt fails because you cannot create such files with COBOL verbs. They must be created by an application outside the HP COBOL language (for example, by FUP, the File Utility Program, or by using ENTER to call Enscribe routines).

  If the creation attempt succeeds, then the file exists. Files Assigned to Disk Devices explains the rules for existing disk files.

  After a successful open operation that creates a file in the manner described, the I-O status code is “05,” rather than “00” (as it would have been if the operation had opened an existing file).

  Although no Guardian file is open, if the file might be created later by the current or another program, you must close the file before opening it in COBOL.

  — If the INPUT phrase applies, the file position indicator is set to indicate that an optional file is not present. In this case, the I-O status code is “05” and the other steps described later are omitted from the open operation.

  When the INPUT, I-O, or EXTEND phrase applies, the file is not described as OPTIONAL, and the file is not present at run-time, the open operation terminates immediately with I-O status code “35.”

- Files Assigned to Processes

  Any file assigned to a process must be described with sequential organization without alternate keys in its file-control entry. The mode in which you open the file determines its use.

Table 10-6 Open Modes for Files Assigned to Processes

Open Mode	Action
INPUT	The file is treated as if it were a terminal. No carriage-control messages are sent to the file, and it cannot have a LINAGE clause in its file description entry. READ, with or without the PROMPT phrase, is the only I-O statement that can be executed on the file; however, prompts are ignored in this open mode.
OUTPUT or EXTEND	If the process has a device subtype of 31, it is a spooler process, treated like a printer. If the file description entry for the file includes a LINAGE clause, or the file was opened by the routine COBOL85^SPECIAL^OPEN or COBOL_SPECIAL_OPEN_, it is assumed to be a printer or a spooler process. Carriage-control messages are sent to the file, and it can have a LINAGE clause in its file description entry. WRITE, with or without the ADVANCING phrase, is the only I-O statement that can be executed on the file. Lines of output are held until a subsequent write or close operation, and consecutive spacing operations due to ADVANCING clauses are consolidated to minimized output operations. If the process does not meet the criteria stated earlier, it is assumed to be an ordinary process. No consolidation of operations occurs, and each line is written to the process immediately. No control information or extra blank lines are written to the process.
I-O	The file is treated as if it were a terminal. No carriage-control messages are sent to the file, and it cannot have a LINAGE clause in its file description entry. READ and WRITE statements can be executed on the file, but DELETE, REWRITE, and START statements cannot.

- Files Assigned to Terminals

  Any file assigned to a terminal must be described with sequential organization without alternate keys in its file-control entry. The mode in which you open the file determines its use.

Table 10-7 Open Modes for Files Assigned to Terminals

Open Mode	Action
INPUT	No carriage-control messages are sent to the file, and it cannot have a LINAGE clause in its file description entry. READ, with or without the PROMPT phrase, is the only I-O statement that can be executed on the file; however, prompts are ignored in this open mode.
OUTPUT or EXTEND	No carriage-control messages are sent to the file, and it cannot have a LINAGE clause in its file description entry. WRITE, with or without the ADVANCING phrase, is the only I-O statement that can be executed on the file; however, “ADVANCING mnemonic-name,” if used, is ignored.
I-O	No carriage-control messages are sent to the file, and it cannot have a LINAGE clause in its file description entry. READ and WRITE statements can be executed on the file, but DELETE, REWRITE, and START statements cannot.

- Files Assigned to Disk Devices

  When the object of an OPEN statement is a disk file, the presence or absence of such a file on disk, as well as both the open mode and the file’s description, determines what happens.

  In all cases, a disk file cannot be described with a LINAGE clause in its file description entry, nor can a WRITE statement that operates on a disk file include an ADVANCING phrase.

- Unstructured Disk Files

  Unstructured files can be used by a COBOL program in only certain instances:

  — The program must specify sequential organization with no alternate keys.

  — The file contains zero or more complete records. For this to be true, one of these must be true:

    ◦ The COBOL program describes the records of the file as having an even number of characters, and the file size is some integer multiple of the record size.

    ◦ The COBOL program describes the records of the file as having an odd number of characters, the file system description of the file includes the "odd-unstructured" attribute, and the file size is some integer multiple of the record size.

    ◦ The COBOL program describes the records of the file as having an odd number of characters, the file system description of the file does not include the "odd-unstructured" attribute, and the file size is some integer multiple of (COBOL record size plus 1).

    ◦ If the file code is 101 (EDIT format) because either the file already exists with that code or an ASSIGN command with a CODE phrase has been processed for the file, the file can be opened in INPUT, OUTPUT, or EXTEND modes. Opening the file for OUTPUT deletes any existing records in the file and starts the line numbers at 1, and increments by 1. Opening the file for EXTEND retains existing records and starts the line numbers at 1 greater than the number of the last line in the file, and increments by 1. An EDIT-format file cannot be opened for OUTPUT or EXTEND if the NONSTOP directive has been specified.

    ◦ If the file code is 180 (line sequential, available only in the OSS environment).

    ◦ The file must be described as having fixed-length records (absence of the *rec-1* TO phrase and the VARYING phrase in the RECORD clause).

    ◦ If the file is opened in EXTEND mode, the file position indicator is advanced to the end of the file.

- Structured Disk Files Not Present at Run Time

  The open mode determines what happens when a structured disk file is not present at run time.

Table 10-8 Open Modes for Structured Disk Files Not Present at Run Time

Open Mode	Action
INPUT	When the SELECT clause for the file contains the word OPTIONAL, the OPEN statement completes successfully, and COBOL simulates the existence of an empty disk file. If the file is not described as OPTIONAL, the open operation fails.
OUTPUT or EXTEND or I-O	If a file is described with alternate-record keys, the open operation fails. A COBOL program cannot create files with alternate keys. When the disk file is otherwise described, COBOL creates the file (see Structured Disk Files Present at Run Time). If the device name is associated with a proper file name (such as $volumename.subvolname.filename), then the created file is permanent and continues to exist after the program has closed it. If the device name is not associated with a proper file name (that is, if its name has the form of just a volume name or is a special name like #TEMP), the created file is temporary and disappears when it is closed. The device name can be associated with a proper file name in any of these ways: — By the ASSIGN clause at compile time — By an ASSIGN command at run time — By a DEFINE of class MAP at run time

- Structured Disk Files Present at Run Time

  Certain requirements pertain to all files in this category:

  — The organization declared for the file must be consistent with the actual organization of the file (that is, if the file is declared ORGANIZATION SEQUENTIAL, the file must be entry-sequenced; if it is declared ORGANIZATION RELATIVE, the file must be relative; and if it is declared ORGANIZATION INDEXED, the file must be key-sequenced).

  — If the file is key sequenced or has alternate keys, the declaration of the keys in the COBOL program must agree with the declaration of the keys in the file system.

  — The file security must be appropriate for the open mode. You cannot open a file for INPUT or I-O if it is secured against your reading it. You cannot open a file for OUTPUT, I-O, or EXTEND if it is secured against your writing to it.

CAUTION: If you open a disk file for shared access, do not use sequential block buffering, because one process can read data from the file that is not up-to-date while another process is altering the file. See the explanation of the RESERVE clause in file-control entries under FILE-CONTROL Paragraph (page 127).

When the preceding requirements are met for a disk file present at run time, the open mode determines what happens during execution.

Table 10-9 Open Modes for Structured Disk Files Present at Run Time

Open Mode	Action
INPUT	If the record length of the physical file is shorter than the record length declared in the File Section, and the record length is not declared as variable (rec-1 TO rec-2 CHARACTERS or VARYING SIZE), the OPEN statement fails. If the open operation succeeds and the file organization is relative, the relative key is established as the key of reference. If the open operation succeeds and the file organization is indexed, the prime record key is established as the key of reference. The file position indicator is set so that the first execution of a READ statement retrieves either of these, depending on the file's organization: ◦ The first record in a sequential file ◦ The first record in the sequence defined by the key of reference in an indexed, queue, or relative file
OUTPUT	If the record length of the physical file is shorter than the record length declared in the File Section, the file is not suitable for use: ◦ If no alternate record keys exist, the file is purged and a new one is created. The new file has file code 0 unless the COBOL_ASSIGN_ routine or the TACL command ASSIGN provides another file code value. The new file has the standard HP COBOL defaults for file size and number of extents, rather than the size and extents of the original file. ◦ If alternate record keys exist, the file cannot be purged, and the open operation fails with I-O status code "30." If the record length of the physical file is not shorter than the record length declared in the File Section, and the file contains some data, the file is not purged, but its data is purged. In either case, no warning alerts you that a purge is taking place. After the file is purged, the file position indicator is set to either of these, depending on the file's organization: ◦ The first record in a sequential file ◦ An undefined value for an indexed, queue, or relative file, because the file contains no records at that point

Table 10-9 Open Modes for Structured Disk Files Present at Run Time *(continued)*

Open Mode	Action
EXTEND	If the record length of the physical file is shorter than the record length declared in the File Section, the file is not suitable for use. The open operation fails with I-O status code “30.” The file position indicator is set to the end of the file.
I-O	If the record length of the physical file is shorter than the record length declared in the File Section, the file is not suitable for use. The open operation fails with I-O status code “30.” The file position indicator is set to the first record in the file.

- HP COBOL Queues Printer File Records

  HP COBOL queues write operations to files that are associated with printers and to processes that behave like printers. The LINAGE clause of the file description entry and the ADVANCING phrase of the WRITE statement are valid only for such files.

  Records that HP COBOL writes to a process are queued only when the OPEN statement specifies an attribute of OUTPUT, EXTEND, or I-O, and at least one of these is true:

  — The file description entry includes a LINAGE clause

  — The OPEN statement discovers that the process with which it is associating the file has the device subtype attribute value of 31

  When the OPEN routine opens a printer-type file, it turns off automatic page ejection, computes any dynamic logical page attributes, and when appropriate, performs a page eject.

- Files Assigned to Tape Devices

  When the file device is a tape unit, the run-time routines request mounting of the first, last, or only reel of the file:

  — The first or only reel if the open mode is INPUT or OUTPUT

  — The last or only reel if the open mode is EXTEND

  See CLOSE (page 315) in this section for more information about the messages that the run-time routines issue about tape mounting.

  If the NO REWIND phrase appears, the tape is presumed to be already properly positioned; otherwise, the tape is rewound and, if it is a multiple-file tape, positioned to the file being opened.

  The NO REWIND phrase is ignored if the file device is not a tape unit. If the rest of the open operation actions complete successfully, the I-O status code is set to “07.”

- Process Pairs

  When the process is executing as a process pair, each open operation for a file also executes an implied statement of this form:

  ```
  CHECKPOINT FILE file-name
  ```

- Exclusion Modes

  The EXCLUSIVE, SHARED, or PROTECTED phrase specifies the appropriate file system exclusion mode for the file.

  If the OPEN statement does not have one of these phrases, the process determines the default exclusion mode: when a command interpreter ASSIGN command that applies to the file specifies an exclusion mode, the process uses that exclusion mode; otherwise, the process determines the exclusion mode from the type of file device:

Device	Exclusion Mode
Terminal (including the operator console)	SHARED
Disk file being opened for input	PROTECTED
Other file	EXCLUSIVE

A file that no process has open can be opened by any single process with any exclusion mode.

If some process already has a file open EXCLUSIVE, no other process can open the file.

A file intended for multi-user access must specify SHARED.

If some process already has a file open SHARED for any access mode except OUTPUT, another process can open the same file SHARED for any access mode.

If some process already has a file open SHARED for OUTPUT, no other process can open the file (because the OPEN would delete all the records in the file).

If some process already has a file open SHARED for INPUT access mode, another process can open the same file PROTECTED for any access mode.

If some process already has a file open PROTECTED for INPUT access mode, another process can also open the same file PROTECTED for INPUT access mode.

All other combinations of access mode and exclusion mode cause the open operation to fail.

- Record Prereading

  To save execution time by overlapping reading and processing, the run-time routines perform record prereading (starting the read for record *n*+1 when returning record *n* to the program) when all of these conditions are true:

  — The file access mode is SEQUENTIAL.

  — The file is open for INPUT.

  — The file is either a disk file opened for PROTECTED use or a disk, tape, terminal, or card reader opened for EXCLUSIVE use.

  — The process is not running as a process pair.

  — The file is not opened for timed I-O.

CAUTION: Verify the OPEN statement does not meet prereading requirements when the COBOL process does any of these, or the operation might begin with the wrong record:

- Enters a TAL routine to do input or output
- Executes the read operation as part of a TMF transaction (enters BEGINTRANSACTION, executes a READ statement, then enters ENDTRANSACTION)
- Executes a CLOSE NO REWIND statement on a tape file and subsequently executes an OPEN NO REWIND statement on the same file to continue from the previous position

Example 10-46 shows an OPEN statement with a TIME LIMITS and a SYNCDEPTH phrase and an OPEN statement without optional phrases.

Example 10-46 OPEN Statements

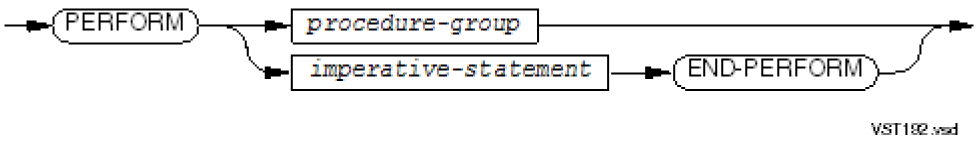
```
?NONSTOP
 IDENTIFICATION DIVISION.
   ...
 INPUT-OUTPUT SECTION.
 FILE-CONTROL.
   SELECT EMPLOYEE-MASTER ASSIGN TO "EMPMAST"
      ORGANIZATION IS INDEXED
      ACCESS MODE IS DYNAMIC.
     ...
   SELECT LISTING-FILE ASSIGN TO "LISTOUT"
      ORGANIZATION IS SEQUENTIAL
      ACCESS MODE IS SEQUENTIAL.
     ...
 PROCEDURE DIVISION.
     ...
     OPEN I-O EMPLOYEE-MASTER WITH TIME LIMITS SYNCDEPTH 1
     OPEN OUTPUT LISTING-FILE
```

PERFORM

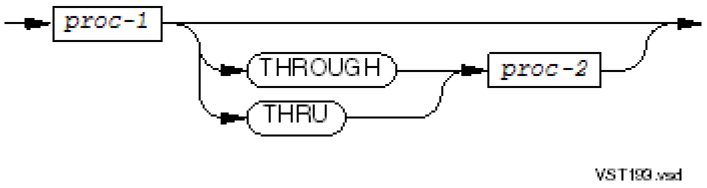
PERFORM executes one or more procedures in a program, simply or with looping. When a procedure-name is a section-name, PERFORM executes all the paragraphs in that section.

Unconditional PERFORM

Unconditional PERFORM executes a procedure, group of procedures, or imperative statement one time. When execution reaches the end of the procedure, group of procedures, or imperative statement, control returns to the statement following the PERFORM statement.



procedure-group



proc-1

is a paragraph-name or section-name. Without THROUGH or THRU, *proc-1* identifies the only procedure that is to be executed. With THROUGH or THRU, *proc-1* identifies the first procedure of a group.

THROUGH, THRU

indicate that a group of procedures is to be executed.

proc-2

identifies the last procedure in the group.

imperative-statement

is defined under Imperative Statement (page 240).

END-PERFORM

ends the scope of the PERFORM statement, causing the PERFORM to be a delimited-scope statement. If the PERFORM statement does not end with an END-PERFORM phrase, it is an out-of-line PERFORM. If the PERFORM statement ends with an END-PERFORM phrase, it is an in-line PERFORM.

Usage Considerations:

- Execution Cycle of a PERFORM Statement (PERFORM Cycle)

  Each execution of the range of a PERFORM statement is called a “PERFORM cycle.” It begins with the implicit transfer of control to the first statement of the range of the PERFORM and ends with an implicit transfer of control back to the internal decision logic of the PERFORM statement. The point at which the return occurs depends upon the `proc-1` THROUGH `proc-2` phrase:

  - When `proc-2` does not appear and `proc-1` is a paragraph-name, the return occurs after the execution of the last statement in the specified paragraph.
  - When `proc-2` does not appear and `proc-1` is a section-name, the return occurs after the execution of the last statement of the last paragraph of that section.
  - When `proc-2` does appear and is a paragraph-name, the return occurs after the execution of the last statement in that paragraph.
  - When `proc-2` does appear and is a section-name, the return occurs after the execution of the last statement of the last paragraph of that section.

  The preceding conditions for the return of control might be unsatisfied when the range includes a GO TO statement. If the flow of control does not pass through the last statement indicated above, the condition for return cannot be satisfied. Such a PERFORM cycle ends only when the program terminates, which it can do by executing one of these statements:

  - A STOP RUN statement
  - An EXIT PROGRAM statement (from within in a called program)
  - The last statement in the program

  Avoid including a GOTO statement in the range of a PERFORM statement. A GOTO statement can cause a run-time diagnostic indicating that the PERFORM stack is full.

- Procedure Relationships and the Ends of PERFORM Cycles

  There is no necessary relationship between `proc-1` and `proc-2`, except that a PERFORM cycle begun at the procedure named by `proc-1` ends when control reaches the return point following the last statement of the procedure named by `proc-2`. GO TO statements, PERFORM statements, CALL or ENTER statements, and so forth, can occur in the logical sequence of statements executed during a PERFORM cycle. If there are two or more logical points at which a cycle could end, then `proc-2` can name a paragraph consisting solely of the EXIT statement, and all execution paths can terminate cleanly by transferring control to that paragraph.

  The existence of a return point following the end of an execution range is a dynamic characteristic of an executing program. If control reaches the return point defined for a PERFORM statement that is not in the process of execution (that is, is not currently performing a cycle), then control passes through to the next paragraph in accordance with the normal rules for implicit transfer of control from one statement to the next.

- In-line and Out-of-Line PERFORM Statements

  The PERFORM `imperative-statement` END-PERFORM form of the statement is an “in-line PERFORM statement.” The PERFORM `procedure-group` form of the statement is an “out-of-line PERFORM statement.”

  The descriptions of the action of the PERFORM statement in this section are expressed in terms of the out-of-line PERFORM statement. The execution of an in-line PERFORM statement is exactly equivalent to that of an out-of-line PERFORM statement, with the exception that the statements contained in `imperative-statement` in the in-line PERFORM statement are executed in place of the statements within the range of `procedure-group`. Unless specially qualified by the term in-line or out-of-line, all the considerations that apply to the out-of-line PERFORM statement also apply to the in-line PERFORM statement.

- Execution and Transfers of Control

  Chapter 9: Procedure Division (page 237) explains the execution of sections and paragraphs as well as explicit and implicit transfers of control.

- Range of the PERFORM Statement

  The range of an out-of-line PERFORM statement is the statements contained within the range of `procedure-group`.

  The range of an in-line PERFORM statement is the statements contained within the PERFORM statement.

- Logical Range of the PERFORM Statement

  The logical range of any PERFORM statement is all statements that are executed as a result of the PERFORM statement, including the transfer of control to the statement following the PERFORM statement (or to the statement following the END-PERFORM). The logical range includes all statements executed as a result of a GO TO, PERFORM, or CALL statement in the range of the original PERFORM statement, as well as all statements in the Declaratives Portion that might be executed. There is no requirement for statements in the range of a PERFORM statement to appear consecutively.

- Nested PERFORM Statements

  The logical range of a PERFORM cycle can include another PERFORM statement (called a nested PERFORM statement), but these restrictions apply:

  — The logical range of the nested PERFORM statement must be either totally included in or totally excluded from the logical range of the outer PERFORM statement; therefore, an active PERFORM statement whose execution point begins within the range of another active PERFORM statement must not allow control to pass to the exit of the other active PERFORM statement.

  — The final paragraph in the range of one active PERFORM statement cannot be the same as the final paragraph in the range of any other active PERFORM statement, because the program terminates only one PERFORM cycle for each such final paragraph. The final paragraph would have to be executed twice to terminate both cycles.

  — The sequence of statements executed in any cycle of one PERFORM statement must not allow control to reach the return point for any other active PERFORM statement in which it is nested. If this does happen, neither PERFORM cycle terminates, and control falls through to succeeding statements.

  — The sequence of statements executed in any cycle of one PERFORM statement must not allow control to reach any active PERFORM statement, including itself.

  — As a consequence of the preceding rules, a cycle of an active PERFORM statement cannot end until after any PERFORM statements nested within it complete their execution.

The maximum number of PERFORM statements that can be nested is 50. Violation of the preceding rules often causes run-time diagnostic 148.

- Placement of Procedures

  As a general rule, both `proc-1` and `proc-2` must be in the same logically discrete area of the Procedure Division (in a specific declarative procedure, including any associated sections, or in the portion of the Procedure Division that does not include declaratives). Most violations of this rule cause the compiler to issue a warning but to accept the PERFORM statement;

however, if either *proc-1* or *proc-2* is in the Declaratives Portion, then both must be in the Declaratives Portion.

- Placement of PERFORM Statements

  As a general rule, a PERFORM statement must be in the same logically discrete area of the Procedure Division as *proc-1* and *proc-2*. Most violations of this rule cause the compiler to issue a warning; however, certain combinations are explicitly permitted, and others are totally prohibited:

  — For any PERFORM statement, *proc-1* and *proc-2* can be in any nondebugging declarative procedure; however, the compiler issues a warning if *proc-1* and *proc-2* are in two different declarative procedures.

  — When the PERFORM statement is in a debugging declarative procedure, its *proc-1* and *proc-2* can be in any declarative procedure; however, the compiler issues a warning if *proc-1* and *proc-2* are in two different declarative procedures.

  — When the PERFORM statement is not in a debugging declarative procedure, neither *proc-1* nor *proc-2* can be in any debugging declarative procedure.

  — When the PERFORM statement is in the Declaratives Portion, neither *proc-1* nor *proc-2* can be in the other portion of the Procedure Division.

Example 10-47 Unconditional PERFORM Statement With One Paragraph

```
IF REPORT-A
   PERFORM DO-REPORT-A
END-IF
```

Example 10-48 Unconditional PERFORM Statement With Several Paragraphs

```
IF REPORTS-TO-DO
   PERFORM DO-REPORTS THRU DO-REPORTS-EXIT.
IF MUST-EXIT
  ...
DO-REPORTS.
     ...
   (several paragraphs to create the reports)
      ...
DO-REPORTS-EXIT.
   EXIT.
```

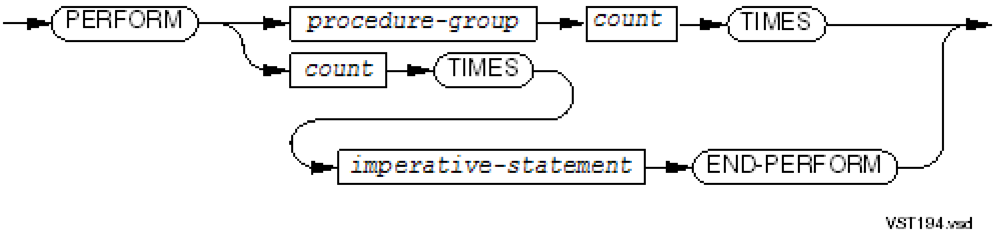
An in-line PERFORM statement can contain delimited-scope statements (such as a delimited-scope READ or IF statements) because they count as imperative statements.

Example 10-49 In-Line PERFORM Statement With Delimited-Scope Statements

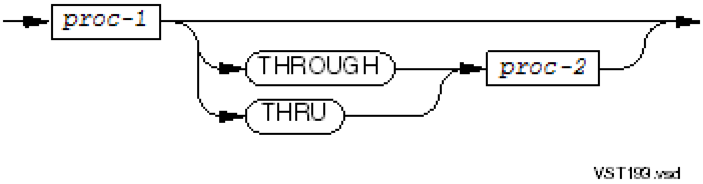
```
PERFORM UNTIL END-IX
   READ IX-FILE RECORD
        AT END
            SET END-IX TO TRUE
        NOT AT END
            IF IX-NUMBER > 0
              READ MASTR-FILE RECORD KEY IS IX
                   INVALID KEY DISPLAY "Bad IX value: " IX
                               STOP RUN
              END-READ
            END-IF
            MOVE MASTER-NAME TO CUST-NAME
            ...
   END-READ
END-PERFORM
```

PERFORM TIMES

PERFORM TIMES executes a procedure, a group of procedures, or an imperative statement a specified number of times.



procedure-group



proc-1

is a paragraph-name or section-name. Without THROUGH or THRU, *proc-1* identifies the only procedure that is to be executed. With THROUGH or THRU, *proc-1* identifies the first procedure of a group.

THROUGH, THRU

indicate that a group of procedures is to be executed.

proc-2

is the last procedure in the group.

count

is an integer numeric literal or the identifier of an integer data item that tells the process how many times to execute the statements in the range of the PERFORM statement.

imperative-statement

is defined in Imperative Statement (page 240).

END-PERFORM

ends the scope of the PERFORM statement, causing the PERFORM to be a delimited-scope statement. If the PERFORM statement does not end with an END-PERFORM phrase, it is an out-of-line PERFORM. If the PERFORM statement ends with an END-PERFORM phrase, it is an in-line PERFORM.

Example 10-50 specifies that the value of the identifier TRAN-COUNT controls the number of PERFORM cycles:

Example 10-50 PERFORM TIMES Statement

```
PERFORM LIST-TRANSACTIONS TRAN-COUNT TIMES.
```

In Example 10-51, an in-line PERFORM uses a TIMES phrase to initialize a table of squares.

Example 10-51 PERFORM TIMES Statement

```
MOVE 1 TO R
PERFORM 100 TIMES
        MULTIPLY R BY R GIVING R-SQ (R)
          ON SIZE ERROR MOVE "OVERFLOW" TO REASON
                        PERFORM REPORT-DEMISE
        END-MULTIPLY
        ADD 1 TO R
END-PERFORM
```

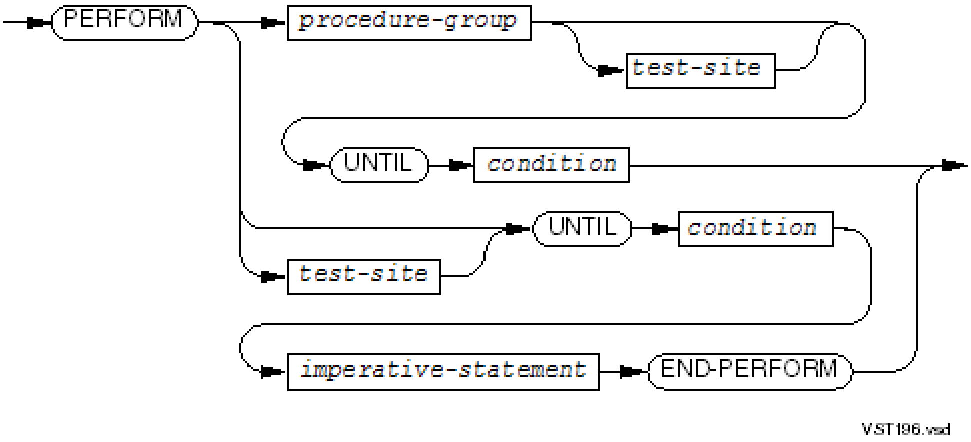
Usage Considerations:

- Negative or Zero Value in the TIMES Phrase

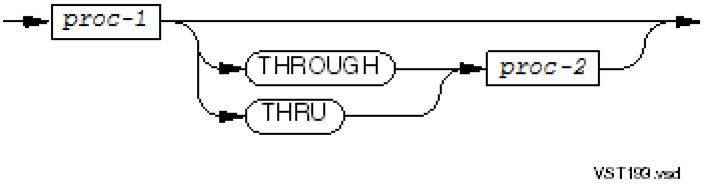
  The value of *count* can be a negative integer or 0, in which case the process does not perform the procedure or group.
- See the usage considerations in Unconditional PERFORM.

PERFORM UNTIL

PERFORM UNTIL executes a procedure, group of procedures, or imperative statement repeatedly until a condition is true. The condition is checked before or after each PERFORM cycle, and when the condition is met, the PERFORM ends.



procedure-group



proc-1

is a paragraph-name or section-name. Without THROUGH or THRU, *proc-1* identifies the only procedure that is to be executed. With THROUGH or THRU, *proc-1* identifies the first procedure of a group.

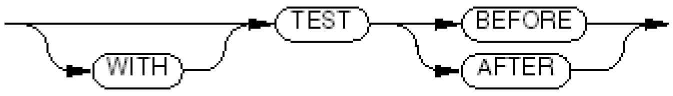
THROUGH, THRU

indicate that a group of procedures is to be executed.

proc-2

identifies the last procedure in the group.

test-site



VST197.vsd

specifies whether the condition is to be tested before or after the PERFORM range is executed. The default is TEST BEFORE.

condition

is any conditional expression.

imperative-statement

is defined in Imperative Statement (page 240).

END-PERFORM

ends the scope of the PERFORM statement, causing the PERFORM to be a delimited-scope statement. If the PERFORM statement does not end with an END-PERFORM phrase, it is an out-of-line PERFORM. If the PERFORM statement ends with an END-PERFORM phrase, it is an in-line PERFORM.

Usage Considerations:

- No Execution With TEST BEFORE if the Condition Value is TRUE

  If the TEST AFTER phrase is not present and the value of the condition is TRUE when control first arrives at the PERFORM statement, the PERFORM range is not executed.

- See Usage Considerations in Unconditional PERFORM.

The PERFORM UNTIL statement in Example 10-52 uses a condition-name condition. Once the add routine is successful, a SET ANY-ADDS-CV TO TRUE statement is executing (moving the value 1 to ANY-ADDS-CV); otherwise, ANY-ADDS-CV remains 0 when control returns to the PERFORM statement. As long as ANY-ADDS-CV is 0, the PERFORM cycle is re-executed.

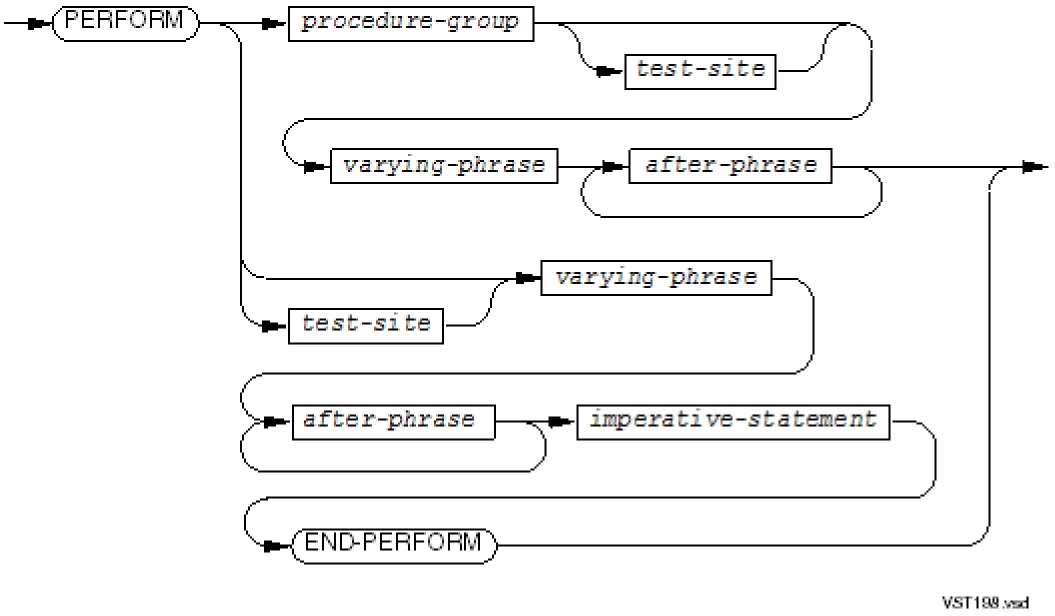
Example 10-52 PERFORM UNTIL Statement

```
WORKING-STORAGE SECTION.
01 ANY-ADDS-CV        PICTURE 9.
    88  SOME-MORE-ADDS       VALUE 0.
    88  NO-MORE-ADDS         VALUE 1.
        ...
PROCEDURE DIVISION.
        ...
   MOVE 0 TO ANY-ADDS-CV
   PERFORM ADD-ROUTINE UNTIL NO-MORE-ADDS
        ...
ADD-ROUTINE.
        ...
```

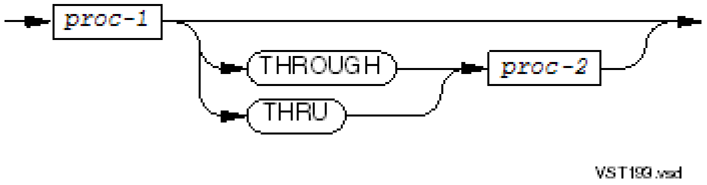
PERFORM VARYING

PERFORM VARYING executes a loop of procedures. PERFORM VARYING with TEST BEFORE is a "while loop;" with TEST AFTER, it is a "repeat loop."

A single PERFORM VARYING statement containing one or more AFTER phrases enables you to perform nested loops of procedures. The last AFTER phrase defines the innermost loop. The first set of parameters in the VARYING phrase defines the outermost loop.



procedure-group



proc-1

is a paragraph-name or section-name. Without THROUGH or THRU, *proc-1* is the only procedure that is to be executed. With THROUGH or THRU, *proc-1* is the first procedure of a group.

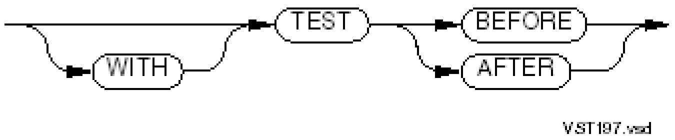
THROUGH, THRU

indicate that a group of procedures is to be executed.

proc-2

is the last procedure in the group.

test-site



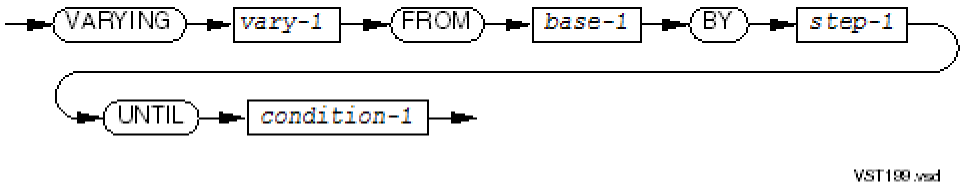
BEFORE

specifies that the condition is to be tested before the perform range is executed. This is the default.

AFTER

specifies that the condition is to be tested after the perform range is executed.

varying-phrase



specifies the outermost loop of the PERFORM statement control logic.

vary-1

is a numeric data item or an index. It is the iteration variable—the variable whose value is changed each time the code in the outermost loop is executed.

base-1

is a numeric literal, an index-name, or the identifier of a numeric data item. It is the initial value for *vary-1*.

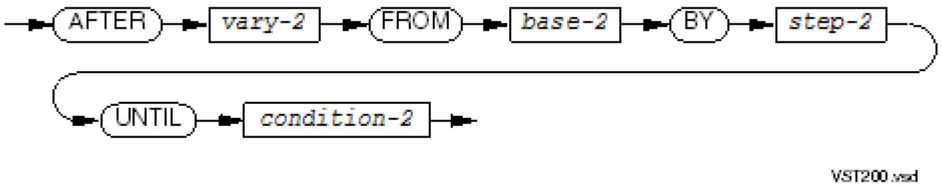
step-1

is a numeric literal or the identifier of a numeric data item. It is the increment that is to be added to *vary-1* each time control returns from the end of the range of the PERFORM. The value of *step-1* must not be 0.

condition-1

is any conditional expression.

after-phrase



specifies additional inner loops of the PERFORM statement control logic. Up to six AFTER phrases can be used in an out-of-line PERFORM statement. In COBOL, the AFTER phrase is not permitted in the in-line PERFORM statement.

vary-2

is a numeric data item or an index. It is an iteration variable—a variable whose value changes each time the process executes the code in an inner loop.

base-2

is a numeric literal, an index-name, or the identifier of a numeric data item. It is the initial value for *vary-2*.

step-2

is a numeric literal or the identifier of a numeric data item. It is the increment that is to be added to *vary-2* each time control returns from the end of the range of the PERFORM. The value of *step-2* must not be 0.

condition-2

is any conditional expression.

imperative-statement

is defined in Imperative Statement (page 240).

END-PERFORM

ends the scope of the PERFORM statement, causing the PERFORM to be a delimited-scope statement. If the PERFORM statement does not end with an END-PERFORM phrase, it is an out-of-line PERFORM. If the PERFORM statement ends with an END-PERFORM phrase, it is an in-line PERFORM.

Usage Considerations:

- No Execution With TEST BEFORE if the Condition Value is TRUE

  If the TEST AFTER phrase is not present and the value of the condition in the VARYING phrase is TRUE when control first arrives at the PERFORM statement, the PERFORM range is not executed.

- Values of Data Items

  The PERFORM VARYING statement initializes and augments the values of one or more data items in an orderly manner.

  When *vary-1* or *vary-2* is an index-name, it is initialized and subsequently augmented according to the rules of the SET statement.

  When *vary-1* or *vary-2* is the name of a numeric data item, it is initialized either according to the rules of the SET statement (if the associated *base-1* or *base-2* is an index-name) or according to the rules of the MOVE statement (if *base-1* or *base-2* is not an index-name). In either case, subsequent augmentation (with the BY phrase) occurs in the manner described later.

- Execution of PERFORM VARYING with TEST BEFORE Specified or Implied

  When no AFTER phrase list appears, *vary-1* is initialized with the value of *base-1* at the beginning of execution of the PERFORM statement. If *condition-1* evaluates to FALSE, the first cycle is performed; if TRUE, no cycles are performed. After the completion of each cycle, the value of *vary-1* is augmented by *step-1*, and *condition-1* is evaluated again to determine whether or not to perform another cycle. Whenever the evaluation of *condition-1* results in TRUE, execution of the statement terminates. Figure 10-7 illustrates the execution of a PERFORM VARYING statement with a TEST BEFORE phrase and without an AFTER phrase list.

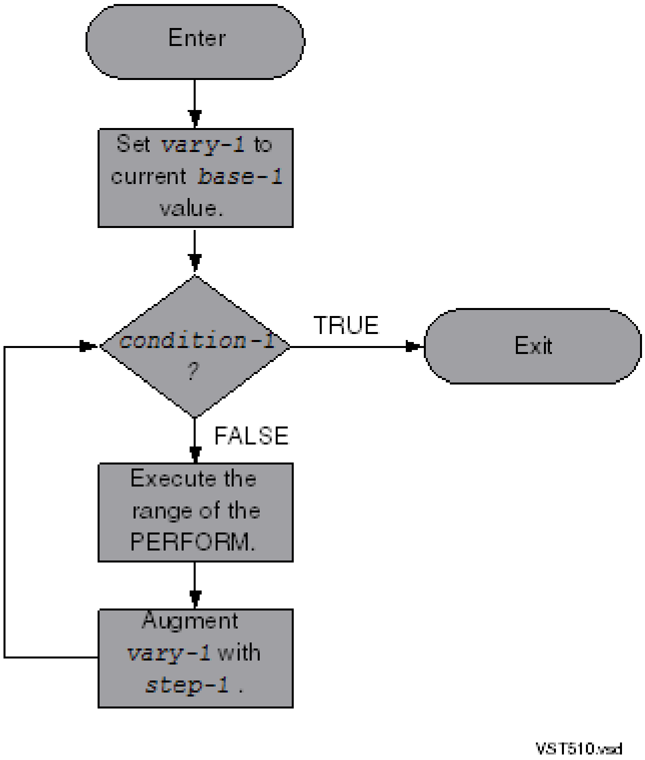
  After execution of the PERFORM statement terminates, *vary-1* has the value assigned, either by initialization or augmentation, at the point where the evaluation of *condition-1* gave a result of TRUE.

When one AFTER phrase appears, `vary-1` and `vary-2` are initialized with the values of `base-1` and `base-2`, respectively, at the beginning of execution of the PERFORM statement. Then either:

— If `condition-1` is TRUE, execution of the statement terminates without ever proceeding to the inner loop (without performing any cycles).

— If the initial value of `condition-1` is FALSE, execution proceeds to the inner loop of the PERFORM statement logic.

Figure 10-7 Execution of a PERFORM VARYING Statement With a TEST BEFORE Phrase and Without an AFTER Phrase List


Enter
Set vary-1 to current base-1 value.
condition-1 ?
TRUE
Exit
FALSE
Execute the range of the PERFORM.
Augment vary-1 with step-1 .
VST510.vsd

Each iteration of the inner loop begins with an evaluation of `condition-2`:

— If its value is FALSE, a cycle is performed, `vary-2` is augmented by the specified increment or decrement (the value of `step-2`), and control returns to the top of the loop.

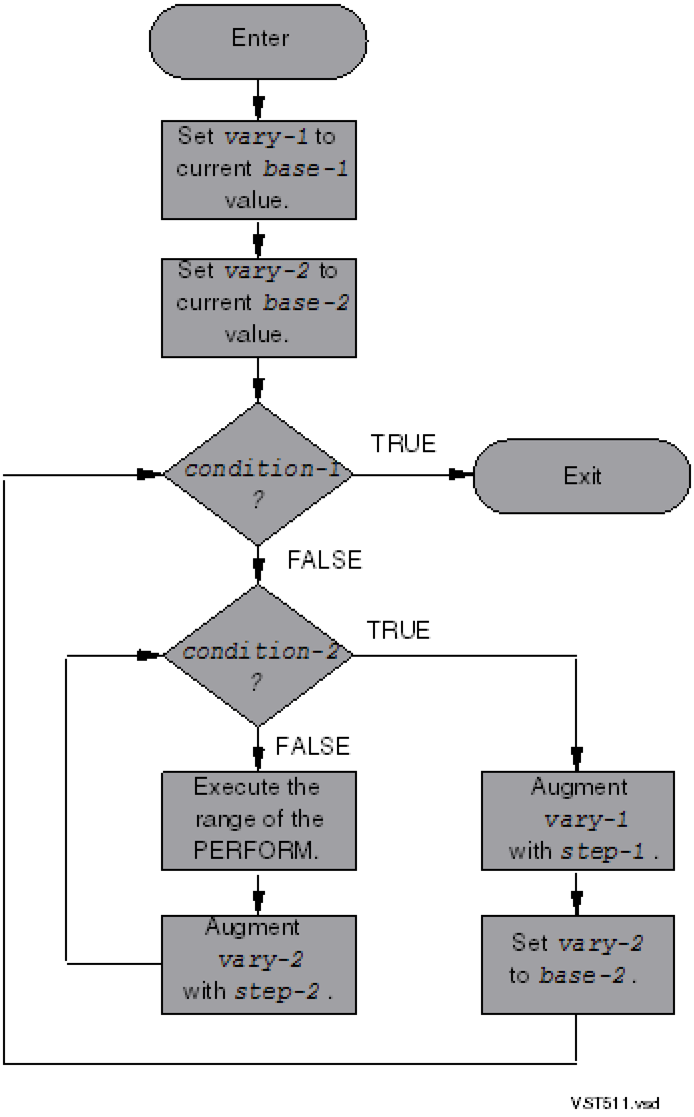
— If the value of `condition-2` is TRUE, no cycle is performed, `vary-1` is augmented by the specified increment or decrement (the value of `step-1`), `vary-2` is initialized to `base-2`, and the inner loop terminates.

After each termination of the inner loop, the process evaluates `condition-1` again:

— If the value of `condition-1` is FALSE, execution proceeds to the inner loop, as described in the preceding text.

— If the value of `condition-1` is TRUE, execution of the statement terminates.

Figure 10-8 Execution of a PERFORM VARYING Statement With a TEST BEFORE Phrase and One AFTER Phrase


Enter
Set vary-1 to current base-1 value.
Set vary-2 to current base-2 value.
condition-1 ?
TRUE
Exit
FALSE
condition-2 ?
TRUE
FALSE
Execute the range of the PERFORM.
Augment vary-1 with step-1 .
Augment vary-2 with step-2 .
Set vary-2 to base-2 .
VST511.vsd

After termination of the PERFORM statement, `vary-1` has the value assigned, either by initialization or augmentation, at the point where the evaluation of `condition-1` gave a result of TRUE. `vary-2` has the value assigned by its last initialization from the `base-2`.

When multiple AFTER phrases appear, the mechanism is the same as for one AFTER phrase except:

— In secondary AFTER phrases, `vary-2` is also initialized with the value of `base-2` in its associated FROM phrase at the beginning of execution of the PERFORM statement.

— In the inner loop (as described earlier), the step “a cycle is performed” is replaced by “execution proceeds to the next inner loop.”

— The logic of the next inner loop parallels that of the first inner loop except for:

  ◦ Its condition plays the role of `condition-2`.

  ◦ The next `vary-2` is augmented by the current value of its `step-2` after each cycle is performed.

  ◦ The next `vary-2` is initialized with the current value of the `base-2` in its associated FROM phrase when the value of its condition is TRUE.

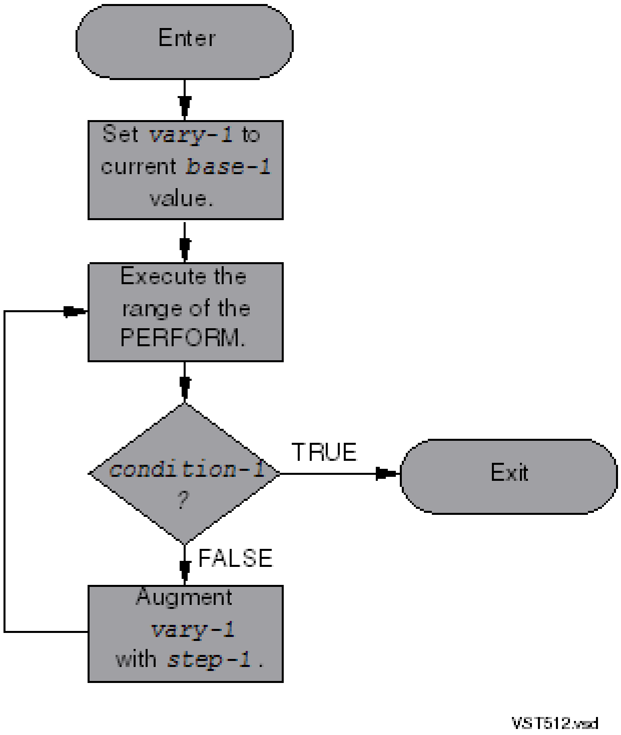
After termination of a PERFORM statement with multiple AFTER phrases, `vary-1` has the value assigned, either by initialization or augmentation, at the point where the evaluation of `condition-1` gave a result of TRUE, and each `vary-2` has the value assigned by its last initialization from its associated `base-2`.

- Execution of PERFORM VARYING with TEST AFTER Specified

  When no AFTER phrase list appears, `vary-1` is initialized with the value of `base-1` at the beginning of execution of the PERFORM statement. The first cycle is then performed. After the completion of each cycle, `condition-1` is evaluated to determine whether or not to perform another cycle. Whenever the evaluation of `condition-1` results in TRUE, execution of the statement terminates. If the evaluation of `condition-1` results in FALSE, the value of `vary-1` is augmented by the specified increment or decrement (the value of `step-2`), and another cycle is performed.

  After termination of the PERFORM statement, `vary-1` has the value it contained at the end of the last execution of the range of the PERFORM statement.

Figure 10-9 Execution of a PERFORM VARYING Statement With a TEST AFTER Phrase and Without an AFTER Phrase List


Enter
Set vary-1 to current base-1 value.
Execute the range of the PERFORM.
condition-1 ?
TRUE
Exit
FALSE
Augment vary-1 with step-1 .
VST512.vsd

When one AFTER phrase appears, `vary-1` and `vary-2` are initialized with the values of `base-1` and `base-2`, respectively, at the beginning of execution of the PERFORM statement.

Then the specified set of statements is executed. This is considered the inner loop. Each iteration of the inner loop ends by evaluating `condition-2`.

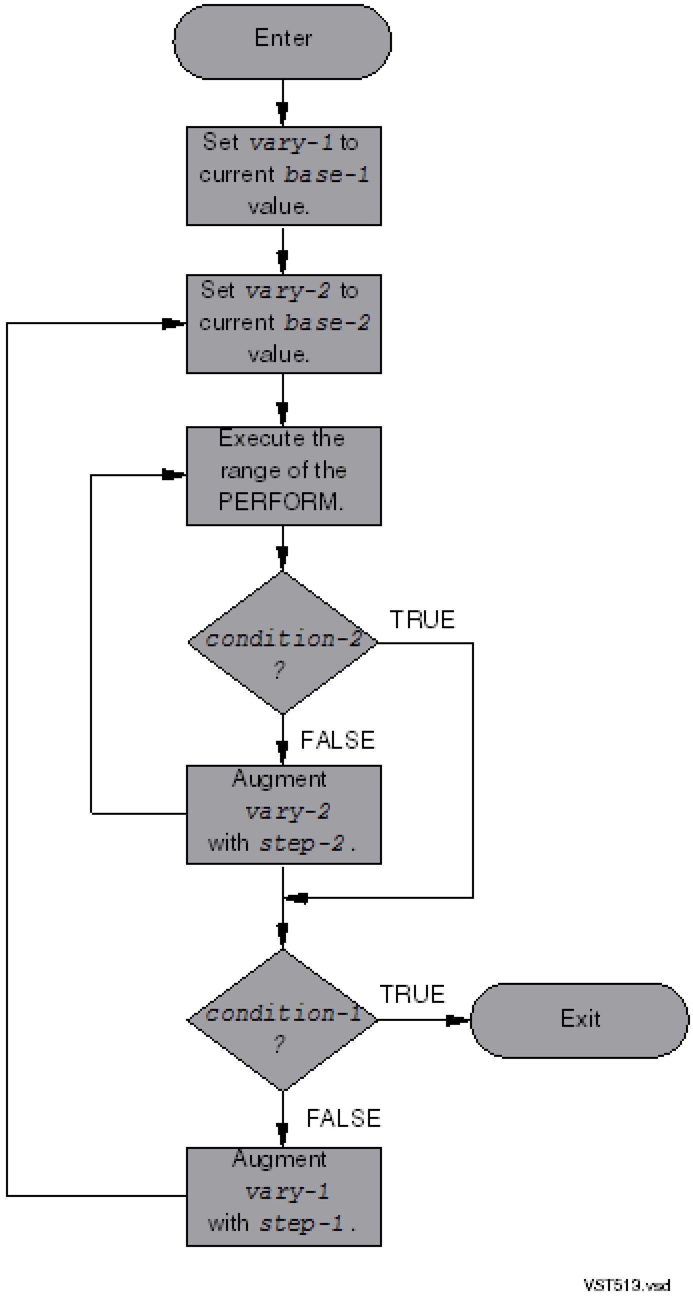
— If its value is FALSE, `vary-2` is augmented by the specified increment or decrement (the value of `step-2`), and a cycle is performed.

— If its value is TRUE, `condition-1` is evaluated.

  ◦ If the value of `condition-1` is FALSE, `vary-1` is augmented by the specified increment or decrement (the value of `step-1`), `vary-2` is initialized with the current value of `base-2`, and the inner loop is entered again.

  ◦ Whenever the evaluation of `condition-1` results in TRUE, execution of the statement terminates.

Figure 10-10 Execution of a PERFORM VARYING Statement With a TEST AFTER Phrase and One AFTER Phrase


Enter
Set vary-1 to current base-1 value.
Set vary-2 to current base-2 value.
Execute the range of the PERFORM.
condition-2 ?
TRUE
FALSE
Augment vary-2 with step-2 .
condition-1 ?
TRUE
Exit
FALSE
Augment vary-1 with step-1 .
VST513.vsd

After termination of the PERFORM statement, `vary-1` and `vary-2` have the values they contained at the end of the last execution of the specified set of statements.

When two or more AFTER phrases appear, the mechanism is the same as for one AFTER phrase except:

- In the second AFTER phrase, `vary-2` is also initialized with the value of `base-2` in its associated FROM phrase at the beginning of execution of the PERFORM statement.
- In the inner loop (as described earlier), the step "a cycle is performed" is replaced by "execution proceeds to the second inner loop."
- The logic of the second inner loop parallels that of the first inner loop except for:
  - Its condition plays the role of `condition-2`.
  - The second `vary-2` is increased by the current value of its `step-2` after each cycle is performed.
  - The second `vary-2` is initialized with the current value of the `base-2` in its associated FROM phrase when the value of its condition is TRUE.

  After termination of a PERFORM statement with two AFTER phrases, `vary-1` has the value assigned, either by initialization or increase, at the point where the evaluation of `condition-1` gave a result of TRUE, and each `vary-2` has the value assigned by its last initialization from its associated `base-2`.

- Restrictions for Index-Names

  When the `vary-1` or `vary-2` is an index-name, these restrictions apply:

  - The `base-1` or `base-2` operand must be a positive integer numeric literal, an index-name, or an identifier that designates an integer numeric data item.
  - The `step-1` or `step-2` operand must be an integer numeric literal or an identifier that designates an integer numeric data item.

  When the `base-1` or `base-2` operand is an index-name, these restrictions apply:

  - The associated `vary-1` or `vary-2` operand must be an index-name or an identifier that designates an integer numeric data item. It cannot designate a special register.
  - The associated `step-1` or `step-2` operand must be an integer numeric literal or an identifier that designates an integer numeric data item.

- Restrictions for Identifiers

  When the `vary-1` or `vary-2` operand is an identifier, and the associated `base-1` or `base-2` is not an index-name, these restrictions apply:

  - The `vary-1` or `vary-2` identifier must designate a numeric data item. It cannot designate a special register.
  - The associated `base-1` or `base-2` operand must be a numeric literal or an identifier that designates a numeric data item.
  - The associated `step-1` or `step-2` must be a numeric literal or an identifier that designates a numeric data item.

- Execution Cycles of PERFORM Statements (PERFORM Cycles)

  Execution of a PERFORM statement causes none, one, or more executions of its range. These executions are called PERFORM cycles. The number of PERFORM cycles depends upon the values of the operands of the PERFORM statement. The decision of whether to perform the first cycle or not occurs after any initialization specified in the VARYING and FROM phrases.

- See Usage Considerations in Unconditional PERFORM.

Example 10-53 PERFORM VARYING Statement Used to Display a List

```
WORKING-STORAGE SECTION.
01  COMMAND-DATA.
    05  FILLER PIC X(36)
        VALUE "ADD      - ADD A NEW RECORD".
    05  FILLER PIC X(36)
        VALUE "DELETE   - DELETE A RECORD".
      ...
01  COMMAND-TABLE REDEFINES COMMAND-DATA.
    05 COMMAND-ENTRY      PIC X(36)  OCCURS 10 TIMES.
01  COMMAND-NUMBERS.
    05  NO-OF-COMMANDS    PIC 99          VALUE 9.
    05  COMMAND-SUB       PIC 99 COMP     VALUE 1.

PROCEDURE DIVISION.
           ...
    PERFORM LIST-COMMANDS
       VARYING COMMAND-SUB FROM 1 BY 1
       UNTIL COMMAND-SUB GREATER THAN NO-OF-COMMANDS
          ...
LIST-COMMANDS.
    DISPLAY COMMAND-ENTRY(COMMAND-SUB)
    ...
```

In Example 10-54, each of the two paragraphs builds a table of numbers (rows) raised to powers (columns). Each paragraph fills the 5 columns of row 1, then of row 2, and so on.

Example 10-54 PERFORM VARYING Statement Used to Build a Table

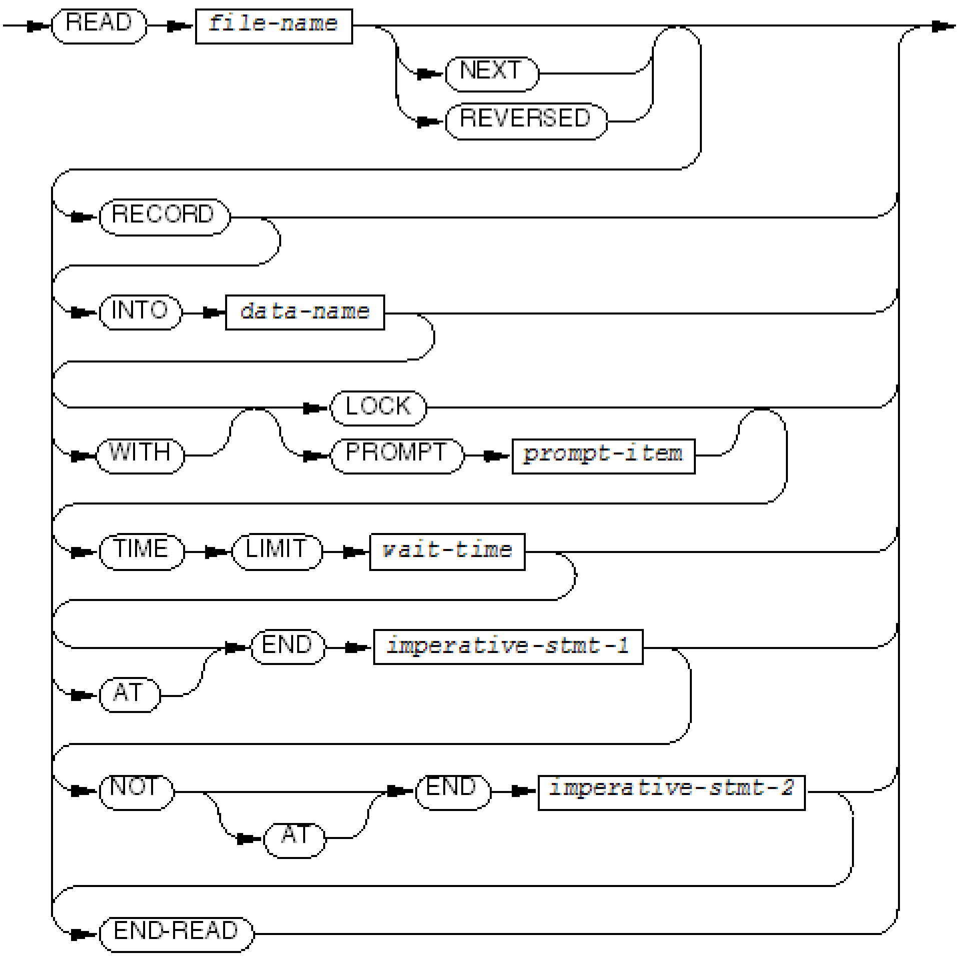
```
WORKING-STORAGE SECTION.
01 TWO-D-TABLE.
   03 OCCURS 10 TIMES.
      05 PWR  PICTURE 9(6) OCCURS 5 TIMES.
PROCEDURE DIVISION.
TWO-NESTED-PERFORMS.
  PERFORM VARYING R FROM 1 BY 1 UNTIL R > 10
     PERFORM VARYING C FROM 1 BY 1 UNTIL C > 5
        COMPUTE PWR (R, C) = R ** C
     END-PERFORM
  END-PERFORM.
SINGLE-PERFORM-WITH-AFTER.
  PERFORM VARYING R FROM 1 BY 1 UNTIL R > 10
          AFTER   C FROM 1 BY 1 UNTIL C > 5
      COMPUTE PWR (R, C) = R ** C
  END-PERFORM.
```

READ

READ copies one logical record from a file and stores it in a record area defined by your program.

READ for Sequential or Dynamic Access

READ for sequential or dynamic access reads the next record in the file. (For sequential access of line sequential files, see READ for Line Sequential Files (page 425).)



VST201.vsd

file-name

is the file description name of the file to retrieve a record from.

NEXT

indicates that the next record is to be read (that is, the record after the current record, according to the key of reference). NEXT is required for sequential reading of a relative, indexed, or queue file whose access is DYNAMIC.

REVERSED

indicates that the prior record is to be read (that is, the record before the current record, according to the key of reference). For restrictions, see Restrictions on Reversed.

data-name

is the identifier of the data area defined by your program to which the contents of the record area are transferred after the read operation is complete.

data-name cannot be an index-name or the identifier of an index data item. The transfer is conducted as if it were a move from an alphanumeric item to an alphanumeric item.

The INTO phrase is allowed only when one of:

- Only one record is associated with *file-name*.
- The record associated with *file-name* is defined as a data structure or as an elementary alphanumeric item and *data-name* is either a data structure or an elementary alphanumeric item.

LOCK

keeps other programs from accessing the record retrieved until an UNLOCKFILE statement, UNLOCKRECORD statement, or REWRITE UNLOCK statement executes. The file that *file-name* specifies must be associated with a disk device.

PROMPT

displays *prompt-item* to the file before a READ operation (as in a COBOL requester communicating with a server). The PROMPT phrase is permitted only for files whose organization is SEQUENTIAL and works only for terminals, processes, operator consoles, and communication system submanagers. If the PROMPT phrase is specified for other devices, such as disk files, it is ignored.

prompt-item

is a DISPLAY data item that starts at the beginning of a record of the file associated with *file-name*.

prompt-item is ignored if the file against which the READ is executed is open in INPUT, OUTPUT, or EXTEND mode, or if the file is assigned to a device not capable of supporting such operations (such as a disk file). For further information on the use of *prompt-item*, see Prompt Phrase.

wait-time

is the time interval, in seconds, in which the operation must complete. *wait-time* can be a literal or the name of a data item. In either case, it must have a value described with at most seven digits preceding any decimal point position. Any fractional portion is truncated to two decimal places.

If *file-name* was not opened with a TIME LIMITS phrase, including *wait-time* in the READ statement causes a run-time error.

imperative-stmt-1

is an imperative statement to be performed when the end of the file is encountered at the beginning of the read operation. This phrase is required if no USE statement is applicable for the file. If both a USE statement and an AT END phrase are present, only the AT END phrase is used.

imperative-stmt-2

is an imperative statement to be performed when the end of the file is not encountered at the beginning of the read operation.

END-READ

ends the scope of the READ statement, causing the READ to be a delimited-scope statement. If the READ statement does not end with an END-READ phrase, the presence of the AT END or the NOT AT END phrase causes the READ statement to be a conditional statement, which ends at the next period separator.

Usage Considerations:

- Restrictions on REVERSED

  REVERSED can be specified for a file only if all of these conditions are true:

  — The file is a structured disk file.

  — The file's file-control entry does not have a RESERVE clause with *number* greater than two.

  — The file's access is not RANDOM.

  — Prereading is not active for the file.

  If REVERSED is specified, then:

  — Neither LOCK nor PROMPT *prompt-item* can be specified.

  — REVERSED cancels the effect of GENERIC (a positioning mode that the START statement can specify).

REVERSED is not recommended if these conditions are true, because it is very inefficient:

  — File organization is RELATIVE.
  — The relative key is the key of reference.
  — Records are not contiguous.

- Action of the READ Statement (Sequential Read)

  Any READ statement that has a NEXT or REVERSED phrase, or is associated with a file for which the ACCESS MODE SEQUENTIAL clause is specified or implied, is a sequential READ statement. Any other READ statement is a random READ statement.

  The file identified by `file-name` must be open in the INPUT or I-O mode; if it is not, the read operation is unsuccessful with I-O status code “47.” If the file is described with fixed-length records and a smaller record is read, the read operation is successful with I-O status code “04.”

  In addition to the specific I-O status codes described later, the general codes “00,” “30,” and “90” can occur. For more information, see I-O Status Code (page 257).

  — Retrieval of a logical record

    The read operation retrieves a logical record from the file identified by `file-name` and places it into the record area associated with that file.

  — Alignment within record area

    If the size of the retrieved record is greater than the maximum record size defined for the file (as specified in the explicit or implicit RECORD clause in the file description entry for file name), the run-time routines truncate the retrieved record on the right before placing it in the record area.

    If the size of the retrieved record is less than the maximum record size defined for the file, the run-time routines left justify the record content within the record area. All character positions in the record area that are to the right of the last character in the retrieved record have undefined values—that is, they can contain any characters. If you want to verify that they are spaces, use the INTO phrase.

    When the read operation is attempted on a file with fixed-length records and the size of the retrieved record is greater than the maximum record size defined for the file or less than the minimum record size defined for the file (as specified in the explicit or implicit RECORD clause in the file description entry for the file), the read operation is unsuccessful with I-O status code “30” and the GUARDIAN-ERR register has the value 21.

  — Unsuccessful read operation

    Whenever the read operation terminates with an I-O status code greater than or equal to “30” (except “97”), execution of the READ statement is not successful. In this case the applicable USE procedure, if one exists, is executed.

    If the execution of a READ statement is unsuccessful for any reason, the key of reference and the contents of the record area associated with the file are undefined and the value of the depending item, if one is defined for the file, is not altered.

  — PROMPT phrase

    When the PROMPT phrase appears, and the device with which the file is associated is either a terminal or a process (typically, a server), and the file’s open mode is I-O, the value of `prompt-item` is sent to the file as part of the read operation.

    The PROMPT phrase (often used for the WRITE/READ action common in requesters) is effective only when the device with which the file is associated is either a terminal or a process and the open mode of the file is I-O.

When these conditions are met, the PROMPT phrase causes the value of `prompt-item` to be sent to the file before the actual read operation begins (as if a write operation preceded the read operation). The record area is then cleared to spaces for the length of the `prompt-item`. Characters being read from the terminal or process are then copied into the record area.

When the specified conditions are not met, the PROMPT phrase is ignored.

If the file has variable-length records, then the DEPENDING ON data item does not affect the length of the string written from the prompt-item. To write a variable-length `prompt-item`, use reference modification. You can also write a variable-length `prompt-item` by using an OCCURS DEPENDING phrase in the prompt-item, but then you cannot use reference modification on the `prompt-item` elsewhere in the program.

— Inability to establish a position for the read operation

The run-time routines examine the file position indicator at the start of the read operation. Certain settings of the file position indicator reflect that the read operation is unable to retrieve any record:

- If the file position indicator indicates that no valid next record has been established, the read operation terminates immediately with I-O status code “46.”
- If the file position indicator indicates that an optional file is not present, the run-time routines change the setting of the file position indicator to indicate that the at-end condition already exists. The read operation then terminates with I-O status code “10.” Execution then proceeds as described in the rule for the at-end condition (see the usage condition Retrieval).

- File-Status Data Item

You can declare a file-status data item for a file in its file-control entry. During each execution of a READ statement, this file-status data item is assigned a new value that reflects the outcome of the read operation.

The possible I-O status codes that result from successful read operations are:

I-O Status Code	Successful Read Operation
“00”	The read operation was unconditionally successful.
“02”	This is possible only when a file has the INSERTIONORDER attribute. The key value for the alternate key that is serving as the current key of reference is equal to the value of that same key in the record that is the next one in the file with respect to that key of reference.
“04”	The file is not described as having variable-length records (by having the RECORD CONTAINS `rec-1` TO `rec-2` CHARACTERS clause or a RECORD VARYING clause in its file description), and a record was read that was shorter than the maximum size. The execution of the READ statement is successful.
“97”	A locked record was read successfully. (You must have called SETMODE to allow locked records to be read.)

If, at the start of the read operation, the file position indicator is at the end-of-file mark or its value is not defined, execution of the READ statement is unsuccessful, and the file-status data item is set to a value other than “00.”

NOTE: After an unsuccessful execution of a READ statement, these values are unpredictable:

— The value of the file position indicator
— The contents of the current record area
— The key of reference

Any key value found in the current record area: that is, the value of the current key for indexed files or the value of the alternate key for any type of file-system file, because in HP COBOL sequential and relative files can have alternate keys.

The possible I-O status codes that result from unsuccessful read operations are:

I-O Status Code	Unsuccessful Read Operation
"10"	The end of file condition arose either in the normal course of events or because the program tried to read an optional file that was not present.
"30"	A permanent error exists.
"46"	The value of the file position indicator is undefined at the beginning of the execution of a READ statement.
"47"	The file is not open in the INPUT or I-O mode.
"90"	A logic error has occurred that is not covered by the "4x" file status codes, and no recovery is possible.
"91"	The file being read is a file in EDIT format, and some step in the read operation failed due to non-COBOL causes. The record is not read, and the execution of the READ statement is unsuccessful.

- Retrieval

  When the file position indicator has a value that permits positioning for the read operation, the run-time routines use that value to identify the record to be retrieved.

  In this topic, if the value of the file position indicator reflects a current key of reference that is a record key, the comparisons relate to the value of that key for records in the file; otherwise, the comparisons for a sequential file relate to the record number of the records in the file, and the comparisons for a relative file relate to the relative record number of the records in the file.

  If the file position indicator was established by a previous OPEN or START statement, then the record selected for retrieval is one of:

  — If NEXT is specified or implied, the record whose record number or key value is greater than or equal to the file position indicator
  — If REVERSED is specified, the record whose record number or key value is less than or equal to the file position indicator

  If the file position indicator was established by an earlier READ statement, and its setting does not reflect an alternate record key for which duplicates are allowed, then the record selected for retrieval is one of:

  — If NEXT is specified or implied, the first existing record in the file whose record number or key value is greater than the file position indicator
  — If REVERSED is specified, the first existing record in the file whose record number or key value is less than the file position indicator

If the file position indicator was established by an earlier READ statement, and the setting of the file position indicator reflects an alternate record key for which duplicates are not allowed, then the record selected for retrieval is one of:

— If NEXT is specified or implied, the first record in the file whose key value is either equal to the file position indicator and whose logical position within the set of duplicates is immediately after the record that was made available by that previous READ statement, or whose key value is greater than the file position indicator

— If REVERSED is specified, the first record in the file whose key value is either equal to the file position indicator and whose logical position within the set of duplicates is immediately prior to the record that was made available by that previous READ statement, or whose key value is less than the file position indicator

The execution of a START statement establishes a subset of the file's records that can be retrieved by subsequent sequential or dynamic READ statements. If the current statement is one in a sequence of such READ statements executed after a START statement for the same file (without the execution of an OPEN, START, or Random READ having intervened), then the record (if any) selected for retrieval by the preceding rules is tested to determine whether it is a member of the file subset established by the START statement. If it is not, then the record is disqualified and the effect is the same as if no record had been selected.

If the read operation is successful, the record selected by the file position indicator is retrieved and placed into the record area associated with the file.

If the file is a multiple-reel tape file, and the end-of-reel condition occurs during the retrieval operation, the run-time routines perform a reel-swap sequence, and the first record of the next reel is retrieved. (For information on the reel-swap sequence, see OPEN.) If there is no next reel, the at-end condition exists. In this case, the read operation terminates, and execution of the READ statement is unsuccessful.

The at-end condition exists because one of these conditions is true:

— The file is present but there is no record at the position specified by the file position indicator (such as the position immediately beyond the last record in the file).

— The file has an OPTIONAL phrase in its SELECT clause and, although open, is not actually present.

— The START GENERIC reached the end of a set of duplicates.

If the at-end condition exists, the read operation terminates, and execution of the READ statement is unsuccessful.

When the at-end condition exists, the I-O status code of the read operation is set to "10." If the AT END phrase is specified, control passes to the imperative statement in that phrase, and no USE procedure is executed. If neither the AT END nor the NOT AT END phrase is specified, and an applicable USE procedure exists, that procedure is executed.

- Relative Key Data Item of Relative Files

  A successful execution of the READ statement for a relative file also assigns the relative record number of the retrieved record to the file's relative key data item if the file description defines one. The relative key data item is optional for relative files being accessed sequentially.

- LOCK Phrase

  When the READ statement includes a LOCK phrase, the read operation also locks the retrieved record. The concept of record locking applies only to disk files and queued files. A successful record lock operation guarantees you exclusive access to the record until the program executes one:

  — A REWRITE statement with the UNLOCK option for the same record

  — An UNLOCKRECORD statement for the same record

  — An UNLOCKFILE statement for the file

If the program already holds a record lock for the record in question, the read operation notes this and preserves the lock.

If your HP COBOL process opens the same file twice (that is, treats the file as two separate files with separate file-control entries, each with its own *file-name*, executes a READ LOCK statement on one *file-name* and then executes another READ LOCK statement on the other *file-name*), the process deadlocks. This problem does not occur if the process opens the file only once or if you use time limits.

- Interaction of LOCKFILE and READ LOCK Statements

  If your process executes a READ LOCK statement on a file that it or any other process has locked with a LOCKFILE statement, or your process executes a LOCKFILE statement against a file that has an outstanding READ LOCK, the TIME LIMIT phrase determines what happens.

  If the second statement attempting to lock the file has a TIME LIMIT phrase, it keeps trying to lock the file until the time limit expires. Either it fails and then times out or it succeeds in locking the file.

  If the second statement attempting to lock the file has no TIME LIMIT phrase, it suspends execution until the statement succeeds because the contending lock is removed or until the program is terminated by an external agency such as the TACL command STOP.

- INTO Phrase

  When the READ statement includes an INTO phrase, the retrieved logical record is moved from the file record area to the data item specified in the INTO phrase.

  Any specified subscript or index evaluation involved occurs after the record is retrieved and placed in the record area and just before it is moved to the data item.

  The size of the sending operand in the implicit MOVE statement is the size of the record as placed into the record area.

  This move does not occur if the execution of the READ statement is unsuccessful for any reason. See INTO Phrase (page 257).

- GUARDIAN-ERR Special Register

  The GUARDIAN-ERR special register is updated each time a file-manipulating statement is executed. The value of GUARDIAN-ERR usually provides more specific information about the cause of an unsuccessful completion signaled by the file-status data item. For example, if the file status is “30” (permanent error), GUARDIAN-ERR contains the file system error number identifying the cause. See Diagnosing Input-Output Errors (page 261).

- Handling Exception Conditions

  The READ statement enables you to specify explicitly that when an at-end condition occurs, a particular statement is to be executed.

  For general exception handling, a group of statements called declaratives can be placed at the beginning of the Procedure Division to respond to error conditions arising for a single file or for all files open in the same input mode (INPUT, OUTPUT, I-O, or EXTEND). See USE AFTER EXCEPTION (page 491).

  The USE AFTER EXCEPTION statement specifies what to do when a file-manipulating statement encounters an exception condition. Declaratives can be called for at-end, invalid key, and other exception conditions.

  If, for example, the program includes a USE statement referring to the file being read, and a read operation is attempted when no next logical record exists, then the process executes either an AT END phrase or the USE statement:

  — If an AT END phrase is present, it is executed.

  — If no AT END phrase is present, the USE statement is executed.

If a READ operation encounters a recoverable permanent error (Status Key 1 equals 3, or Status Key 1 equals 9 but Status Key 2 does not equal 7), no at-end condition occurs; however, if a declarative is present for the file, that declarative is activated, then any NOT AT END phrase is executed. Either the declarative or the NOT AT END phrase can then determine the nature of the error (usually by checking GUARDIAN-ERR) and take appropriate action. See Recovering from Input-Output Errors (page 265).

After an at-end condition for a sequentially accessed file (other than $RECEIVE), you must close and reopen or reposition the file before any further operations can be done on it. You can reposition the file with the START statement, the routine COBOL85^REWIND or COBOL_REWIND_.

- Variable-Length Records

  An Enscribe structured file is always capable of containing variable-length records. The file has a stated maximum allowable record length, but records can vary from a length of 0 up to the stated maximum. If the file is written as variable-length records, the COBOL program can read it under a declaration of

  ```
  RECORD CONTAINS rec-1 TO rec-2 CHARACTERS
  ```

  or

  ```
  RECORD IS VARYING IN SIZE FROM rec-1 TO rec-2 CHARACTERS
         DEPENDING ON rec-size
  ```

  (although in COBOL you cannot explicitly state *rec-1* as 0). The only way to determine the length of the record read is to use the DEPENDING phrase of the VARYING clause. The contents of the record area beyond the data fetched by any given read is undefined.

  Other HP products are capable of writing records of length 0 in such files. A COBOL program can read a record of length 0; however, in an entry-sequenced file with fixed-length records, such a record is ignored and the next record is immediately read.

- Setting the Value of the DEPENDING Item (Variable-Length Records)

  When the file has variable-length records whose sizes are reflected by the *rec-size* item specified in the RECORD VARYING clause of the file description entry, the value associated with the *rec-size* at the beginning of a read operation is ignored. The read operation obtains the record and moves the number of character positions the record contains into *rec-size*.

- Use of *wait-time*

  wait-time must be either a numeric data item or a numeric literal, signed or unsigned, having a maximum of seven digits to the left of the decimal. Any fractional part to the right of the decimal is rounded to two decimal places; for example:

  ```
  05   WAIT-FILEX     PIC 9(7)V9(2) COMPUTATIONAL.
  ```

  A nonnegative value of *wait-time* indicates the time interval within which the operation must complete. If the record is locked or reading a process and the operation does not complete within that time interval, it is terminated, and no error message is generated. The file-status item is set to “30,” the GUARDIAN-ERR register is set to 40, and the value of the file position indicator becomes undefined.

  The file position indicator is undefined because it is not clear at what point of the read operation *wait-time* was exceeded. There is no guarantee that you can try the operation again.

  If the value of *wait-time* is -1 or the TIME LIMIT phrase is not present, no time limit is placed on the operation. The program can wait indefinitely for its request to complete. Any other negative value has the same effect as -1.

  If the TIME LIMIT phrase is specified with a nonnegative value and the file is not opened with time limits enabled, the program terminates with an I-O status code “90,” and a message (File is not opened for timed I-O) is delivered to the process’s home terminal.

The effect of declaratives on `time-limit` termination is:

- If there is no declarative procedure applicable to the file when the operation is abandoned, the process terminates, and an ABEND message is reported to the process's home terminal.
- If the applicable declarative procedure is present (but no AT END phrase is present) and the time interval expires, the declarative procedure is performed. Then program execution continues with the imperative statement in the NOT AT END phrase, if one is present, or otherwise with the statement following the one terminated.

  When a file is being read with APPROXIMATE positioning (see START (page 457)), the value used for `wait-time` must take into account that a read operation can take somewhat longer than expected. This can occur when a nonexistent record is sought, because the file system searches through the file looking for the next defined record before reporting the absence of the record sought. See the *Guardian Programmer's Guide* for more information on the action of READ.

- Concept of Next Record

  READ NEXT is used to read the next record of a file whose access mode is SEQUENTIAL or DYNAMIC. Files whose access mode is SEQUENTIAL can be read without the NEXT keyword, but each such read gets the "next record" in the file. The "next record" means "next existing record within the established key of reference."

  - If the last operation on the file was a start operation (or open operation, in the case of relative, indexed, or queue files), and the record selected by the file position indicator is still accessible through the file position indicator, that record is read.
  - If the last operation on the file was a read operation (either a READ KEY or a READ NEXT), or if the record selected by the file position indicator is no longer accessible through the file position indicator (due to deletion or a change of the alternate key), the file position indicator is updated to point to the next existing record in the file, and that record is read.
  - If the value of the file position indicator for the file is undefined when execution of the READ statement begins, the read operation is unsuccessful, and any file-status data item declared for the file in the file-control entry is updated to specify the reason.

- Use of the READ NEXT Statement on Relative, Indexed, or Queue Files

  The READ NEXT statement operates on a relative, indexed, or queue file only if the file is declared with ACCESS MODE SEQUENTIAL or ACCESS MODE DYNAMIC.

- Sequential Block Buffering and HP COBOL Fast I-O

  Sequential block buffering, enabled by the RESERVE clause of the FILE-CONTROL Paragraph, is an Enscribe feature that speeds the reading of a sequential, relative, indexed, or queue file by reading a block of records together into a memory buffer. HP COBOL Fast I-O is a variant of sequential block buffering that is even faster, because the run-time routines handle the record deblocking.

  For either of these features, the file's access must be sequential. Its file organization must be sequential, relative, indexed, or queue. See FILE-CONTROL Paragraph (page 127).

- Eight-Character Volume Names and HP COBOL Fast I-O

  If you use an eight-character volume name in this context, you do not get an error or warning, but you get normal output instead of fast output. (Input is not affected.)

Example 10-55 Reading a Sequential File

```
IDENTIFICATION DIVISION.
    ...
ENVIRONMENT DIVISION.
    ...
```

```
   SELECT INPUT-DATA
      ASSIGN TO "$TAPE"
      ORGANIZATION IS SEQUENTIAL
      ACCESS MODE IS SEQUENTIAL
      FILE STATUS IS INPUT-DATA-FILE-STATUS.
   ...
DATA DIVISION.
FILE SECTION.
FD INPUT-DATA.
01  INPUT-RECORD.
   ...
WORKING-STORAGE SECTION.
   ...
01  FILE-STATUSES.
    03 INPUT-DATA-FILE-STATUS PICTURE XX.
   ...
PROCEDURE DIVISION.
      ...
   READ INPUT-DATA
      AT END CLOSE INPUT-DATA
   END-READ
   IF ...
```

Example 10-56 Reading a Dynamic Indexed File

```
IDENTIFICATION DIVISION.
   ...
ENVIRONMENT DIVISION.
   ...
   SELECT MASTER-IN
   ASSIGN TO "$WOOSTR.BERTIE.MASTER"
   ORGANIZATION IS INDEXED
   ACCESS MODE IS DYNAMIC
   RECORD KEY IS CUSTOMER-NUMBER
   FILE STATUS IS MASTER-IN-FILE-STATUS.
   ...
DATA DIVISION.
FILE SECTION.
FD MASTER-IN
    LABEL RECORDS ARE OMITTED.
01  MASTER-IN-RECORD.
   ...
WORKING-STORAGE SECTION.
   ...
01  FILE-STATUSES.
    03 MASTER-IN-FILE-STATUS PICTURE XX.
   ...
PROCEDURE DIVISION.
      ...
   MOVE 1 TO CUSTOMER-NUMBER
   START MASTER-IN
      KEY IS LESS THAN CUSTOMER-NUMBER
   IF MASTER-IN-FILE-STATUS NOT EQUAL TO ZERO
      PERFORM START-ERROR-ROUTINE
   ELSE
      READ MASTER-IN NEXT RECORD
           AT END PERFORM WRITE-TOTALS-AND-QUIT
      END-READ
   END-IF
```

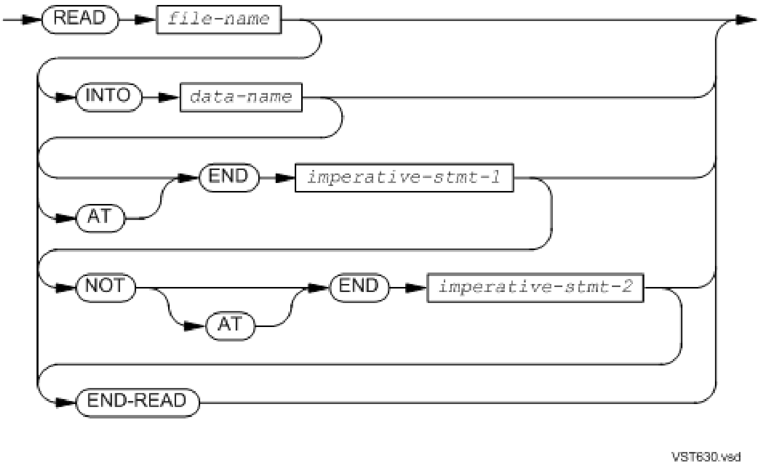
Example 10-57 Reading $RECEIVE With Timed Input-Output

```
IDENTIFICATION DIVISION.
   ...
```

```
ENVIRONMENT DIVISION.
   ...
SELECT REQUEST
   ASSIGN TO "$RECEIVE"
   ORGANIZATION IS SEQUENTIAL
   ACCESS MODE IS SEQUENTIAL
   FILE STATUS IS REQUEST-FILE-STATUS.
   ...
DATA DIVISION.
 FILE SECTION.
 FD REQUEST
     LABEL RECORDS ARE OMITTED.
 01  REQUEST-RECORD.
    ...
WORKING-STORAGE SECTION.
    ...
 01  FILE-STATUSES.
     03 REQUEST-FILE-STATUS PICTURE XX.
    ...
 01  WAIT-TIME PICTURE  PICTURE 9(5)V99 VALUE 30.
PROCEDURE DIVISION.
 DECLARATIVES.
 DECL SECTION.
    USE AFTER ERROR PROCEDURE ON REQUEST.
 DECL-ROUTINE.
    IF GUARDIAN-ERR NOT = 40
       STOP RUN.
 END DECLARATIVES.
MAIN-STUFF SECTION.
 MAIN-PROCESSING.
    ...
    OPEN INPUT REQUEST WITH TIME LIMITS
    READ REQUEST TIME LIMIT WAIT-TIME
    IF REQUEST-FILE-STATUS NOT = "00"
       IF GUARDIAN-ERR = 40
          PERFORM NO-MESSAGE
       END-IF
    END-IF
    ...
```

READ for Line Sequential Files

READ for line sequential files reads the next record in the file.



file-name

is the file description name of the file to retrieve a record from.

data-name

is the identifier of the data area defined by your program to which the contents of the record area are transferred after the read operation is complete.

data-name cannot be an index-name or the identifier of an index data item. The transfer is conducted as if it were a move from an alphanumeric item to an alphanumeric item.

The INTO phrase is allowed only when one of these conditions is true:

- Only one record is associated with file-name.
- The record associated with file-name is defined as a data structure or as an elementary alphanumeric item and data-name is either a data structure or an elementary alphanumeric item.

imperative-stmt-1

is an imperative statement to be performed when the end of the file is encountered at the beginning of the read operation. This phrase is required if no USE statement is applicable for the file. If both a USE statement and an AT END phrase are present, only the AT END phrase is used.

imperative-stmt-2

is an imperative statement to be performed when the end of the file is not encountered at the beginning of the read operation.

END-READ

ends the scope of the READ statement, causing the READ to be a delimited-scope statement. If the READ statement does not end with an END-READ phrase, the presence of the AT END or the NOT AT END phrase causes the READ statement to be a conditional statement, which ends at the next period separator.

See these usage considerations in READ for Sequential or Dynamic Access (page 414):

- In Action of the READ Statement (Sequential Read):
  - Retrieval of a logical record
  - Alignment within record area
  - Unsuccessful read operation
- File-Status Data Item
- Retrieval
- GUARDIAN-ERR Special Register
- Handling Exception Conditions
- Variable-Length Records
- Setting the Value of the DEPENDING Item (Variable-Length Records)
- Concept of Next Record
- Sequential Block Buffering and HP COBOL Fast I-O
- Eight-Character Volume Names and HP COBOL Fast I-O

READ for Random or Dynamic Access

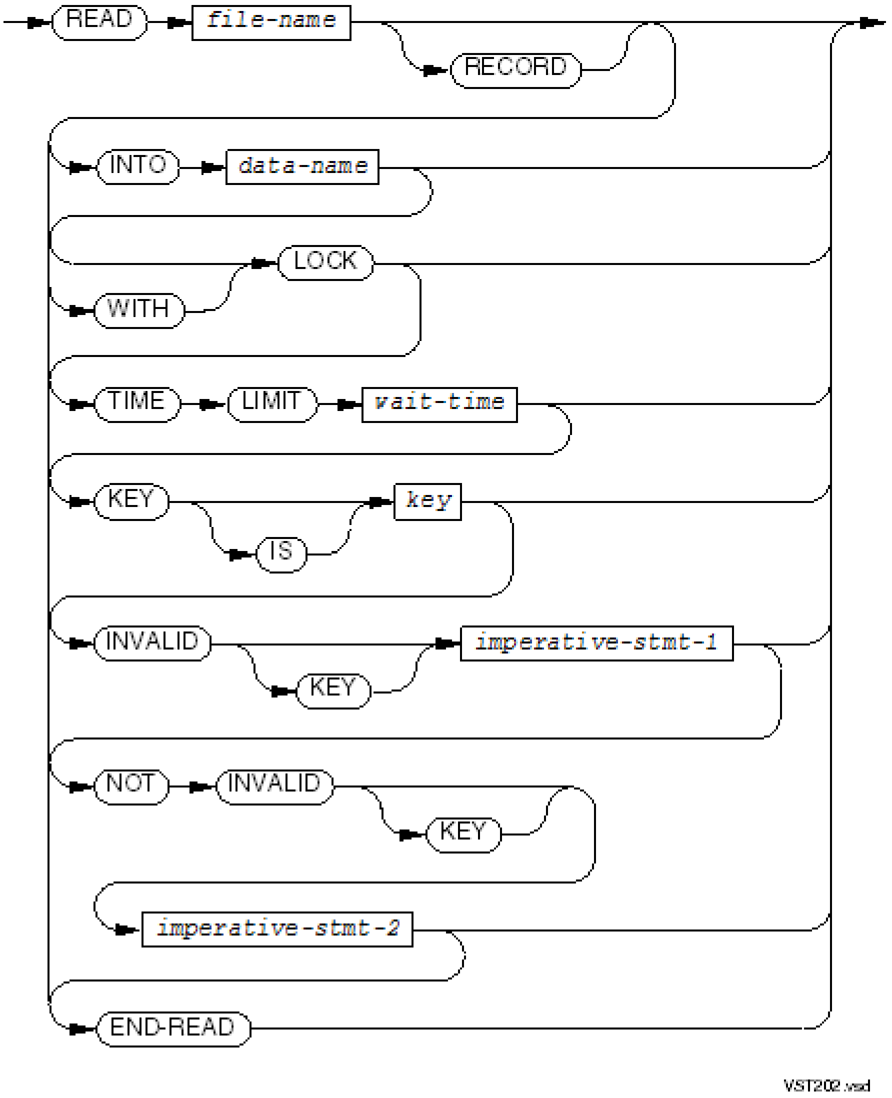
READ is for random or dynamic access reads a record from a file according to the value of a key, rather than according to the present value of the file position indicator.

For relative files, READ without NEXT sets the file position indicator to the item selected by RELATIVE KEY (if KEY is omitted) or to the specified alternate key.

For indexed and queue files, READ without NEXT sets the file position indicator to the item selected by either the key of reference (if KEY is omitted) or to the specified prime or alternate key.

When the selected record exists, the process places its contents in the record area. If it does not exist, an invalid-key condition occurs, and the process executes either the INVALID KEY statement or, if there is no INVALID KEY phrase, a USE AFTER EXCEPTION procedure.

When duplicate alternate key values are allowed, the order of records with equal values depends on the INSERTIONORDER parameter of the alternate key file: records with duplicate alternate key values are retrieved in either prime key order (the way NonStop systems software ordinarily works) or in the order in which they were inserted in the file (as specified in the 1985 ISO/ANSI COBOL standard).



`file-name`

is the file description name of the file to retrieve a record from.

`data-name`

is the identifier of the data area defined by your program to which the contents of the record area are transferred after the read operation is complete. `data-name` cannot be an index-name or the identifier of an index data item. The transfer is conducted as if it were a move from an alphanumeric item to an alphanumeric item.

LOCK

keeps other programs from using the record retrieved until an UNLOCKFILE statement, UNLOCKRECORD statement, or UNLOCK phrase is executed on the record.

wait-time

is the time interval, in seconds, in which the operation must complete. *wait-time* can be a literal or the name of a data item. In either case, it must have a value described with at most seven digits preceding any decimal point position. Any fractional portion is truncated to two decimal places.

If *file-name* was not opened with a TIME LIMITS phrase, including *wait-time* in the READ statement causes a run-time error.

key

is the key of reference. If *file-name* specifies a relative file, *key* must be an alternate key for that file. If file name specifies an indexed file, *key* can be the prime key or an alternate key. If *file-name* specifies a queue file, *key* must be the prime key for that file. Unlike the START statement, the KEY clause provides only APPROXIMATE positioning—not GENERIC positioning.

When the KEY phrase is absent, the relative key (for a relative file) or the prime record key (for an indexed or queue file) is established as the key of reference.

imperative-stmt-1

is an imperative statement to be performed when an invalid key is encountered at the beginning of the READ. It is required if no USE statement is applicable for the file. If both a USE statement and an INVALID KEY phrase are present, only the INVALID KEY phrase is used.

imperative-stmt-2

is an imperative statement to be performed when no invalid key is encountered at the beginning of the read operation.

END-READ

ends the scope of the READ statement, causing the READ to be a delimited-scope statement. If the READ statement does not end with an END-READ phrase, the presence of the INVALID KEY or the NOT INVALID KEY phrase causes the READ statement to be a conditional statement, which ends at the next period separator.

Usage Considerations:

- Action of the READ Statement (Read According to Key Value)

  The execution of this form of the READ statement proceeds:

  — For a file with indexed organization, or a file being accessed according to an alternate key, the run-time routines set the file position indicator to the first record in the file with a key data item value that matches the value in the key of reference. This record is then made available in the file's record area.

  — For a file with relative organization being accessed according to the relative record number, the run-time routines set the file position indicator to the record whose relative record number is contained in the data item named in the RELATIVE KEY clause for the file. This record is then made available in the file's record area.

  — In either case, if no such record exists in the file, the invalid-key condition exists, and the read operation is unsuccessful.

  A successful execution of the READ statement for a relative file, when the key of reference is an alternate record key, also assigns the relative record number of the retrieved record to the file's relative key data item.

  When the execution of the READ statement is successful and the access mode is dynamic, the key of reference established for the read operation is used for subsequent sequential-type

READ statements executed for the same file until the execution of some other statement explicitly establishes a different key of reference.

- File-Status Data Item

  If the file has an associated file-status data item, execution of the READ statement always assigns an appropriate I-O status code.

  One possible I-O status code that results from unsuccessful random or dynamic read operations is:

I-O Status Code	Unsuccessful Random or Dynamic Read Operation
"23"	The invalid-key condition exists, and the read operation is unsuccessful

For other possible I-O status codes representing successful and unsuccessful operations, see File-Status Data Item under READ for Sequential or Dynamic Access (page 414).

NOTE: After an unsuccessful execution of a READ statement, these values are unpredictable:

— The value of the file position indicator

— The contents of the current record area

— The key of reference

Any key value found in the current record area: that is, the value of the current key for indexed files or the value of the alternate key for any type of file-system file, because in HP COBOL, sequential and relative files can have alternate keys.

- Key of Reference to Specify a Record

  The program specifies which record is to be read by assigning the appropriate value to the data item established as the key of reference, before the READ statement is to be executed.

- Handling Exception Conditions

  For general exception handling, put declaratives at the beginning of the Procedure Division. They can respond to error conditions arising for a single file or for all files open in the same mode (INPUT, OUTPUT, I-O, or EXTEND). (See USE (page 491).)

  The USE AFTER STANDARD ERROR PROCEDURE statement specifies what to do when a file-manipulating statement encounters an error. Declaratives can be called for AT END, INVALID KEY, and other error conditions.

  If an invalid-key condition is encountered at the beginning of a read, and the READ statement contains the INVALID KEY phrase, control passes to `imperative-stmt-1` and no USE procedure is executed. If the READ statement does not contain an INVALID KEY phrase, but an applicable USE procedure exists, that procedure is executed.

  After an invalid-key condition arises for a relative, indexed, or queue file, sequential read operatons cannot be done on the file until you either:

  — Close and reopen the file

  — Execute a successful START statement on the file

  — Execute a successful random READ statement on the file

- Timed Input-Output Errors

  If a READ statement includes the TIME LIMIT phrase, and the I-O request exceeds the time interval indicated, the codes FILE STATUS 30 and GUARDIAN-ERR 40 are returned. For information about the special register GUARDIAN-ERR, see Diagnosing Input-Output Errors (page 261).

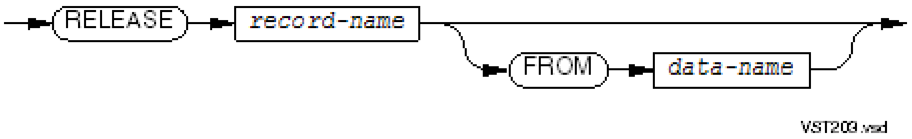
- See these usage considerations in READ for Sequential or Dynamic Access:
  - — File-Status Data Item
  - — LOCK Phrase
  - — Interaction of LOCKFILE and READ LOCK Statements
  - — INTO Phrase
  - — GUARDIAN-ERR Special Register
  - — Variable-Length Records
  - — Use of wait-time

Example 10-58 Reading a Random Indexed File

```
SELECT MASTER-IN ASSIGN TO "MASTER"
   ORGANIZATION IS INDEXED
   ACCESS MODE IS DYNAMIC
   RECORD KEY IS INVOICE-NUMBER.
    ...
PROCEDURE DIVISION.
        ...
   MOVE WS-INVOICE-NUMBER TO INVOICE-NUMBER
   READ MASTER-IN
       INVALID KEY PERFORM RANDOM-READ-ERROR-RTN
   END-READ
```

RELEASE

RELEASE, which must be within a SORT input procedure, sends the next input record to the sorting process.



record-name

is a record-name in a sort-merge file description (SD) entry.

data-name

is the identifier of the item containing the record to be sent. Before the record is sent, it is moved to *record-name*.

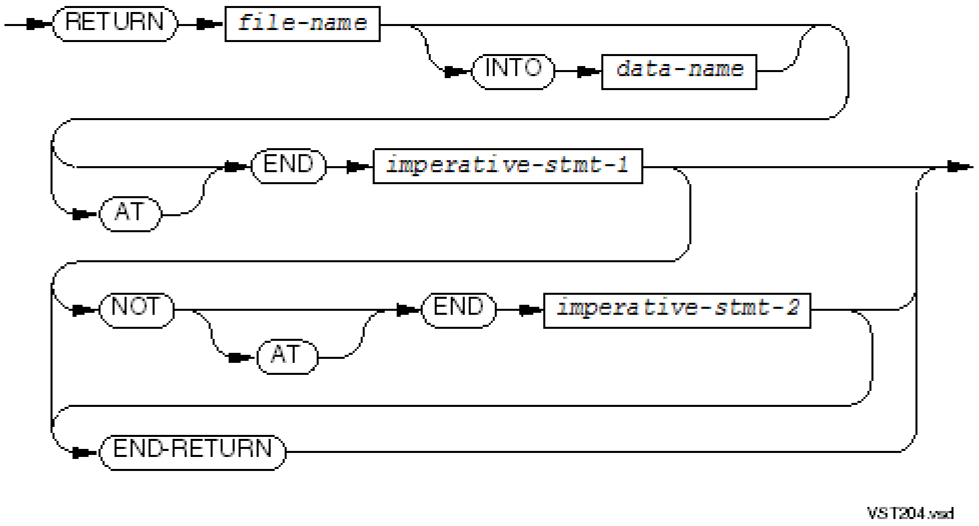
For more details and an example, see SORT (page 449).

REPLACE

REPLACE substitutes zero or more words of pseudotext for one or more words of pseudotext. You can use REPLACE in any division. For more information, see REPLACE Statement (page 516).

RETURN

RETURN, which must be within a SORT or MERGE output procedure, gets the next output record from the SORT or MERGE.



`file-name`

is a file name described by a sort-merge file description (SD) entry.

`data-name`

is the identifier of the area in your program (other than record area associated with `file-name`) where the record is stored.

`imperative-stmt-1`

is an imperative statement to be performed when the end of the file is encountered at the beginning of the return operation. This phrase is required.

`imperative-stmt-2`

is an imperative statement to be performed when the end of the file is not encountered at the beginning of the return operation.

END-RETURN

ends the scope of the RETURN statement, causing the RETURN to be a delimited-scope statement. If the RETURN statement does not end with an END-RETURN phrase, the presence of the AT END or the NOT AT END phrase causes the RETURN statement to be a conditional statement, which ends at the next period separator.

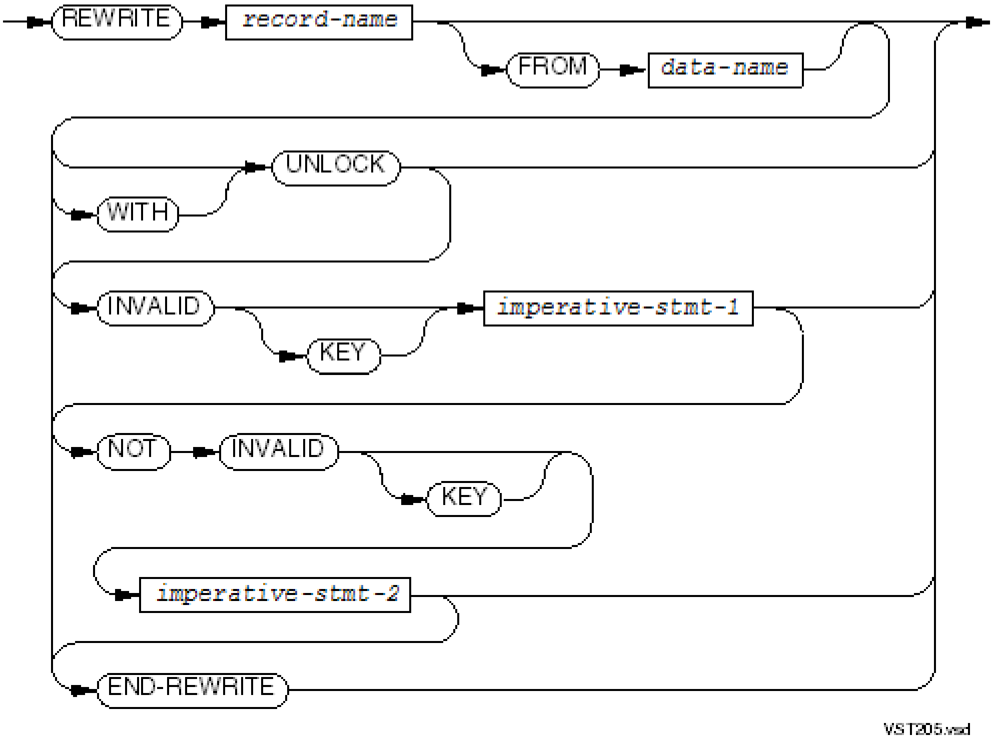
For more details and an example, see:

- SORT (page 449)
- MERGE

REWRITE

REWRITE replaces an existing record in a disk file that is open for I-O. REWRITE is not supported for line sequential files.

REWRITE for Sequential, Relative, Indexed, and Queue Files



record-name

is the record-name in a file description entry whose current contents replaces a record in the file. The file must be open in I-O mode.

data-name

is the identifier of the item that contains the new record instead of *record-name*. When this phrase is used, an implicit MOVE statement occurs to copy the data to *record-name* before the rewrite operation occurs. *data-name* cannot specify an index-name or an index data item. *data-name* cannot specify a data item allocated within the record area in which *record-name* is located.

UNLOCK

permits access by other processes to a record (after the rewrite operation) that was previously locked with a LOCK phrase.

imperative-stmt-1

is an imperative statement to be performed when an invalid-key condition is encountered by the REWRITE operation. It is required if no USE statement is applicable for the file. If both a USE statement and an INVALID KEY phrase are present, only the INVALID KEY phrase is used.

imperative-stmt-2

is an imperative statement to be performed when no invalid-key condition is encountered by the REWRITE operation.

END-REWRITE

ends the scope of the REWRITE statement, causing the REWRITE to be a delimited-scope statement. If the REWRITE statement does not end with an END-REWRITE phrase, the presence of the INVALID KEY or the NOT INVALID KEY phrase causes the REWRITE statement to be a conditional statement, which ends at the next period separator.

Usage Considerations:

- Action of the REWRITE Statement

  The rewrite operation releases a logical record to the run-time routines as a replacement for a record that exists in the file. The size of the logical record (the number of character positions in the record) is determined:

  — When the file has fixed length records (the file description entry includes a RECORD CONTAINS `n` CHARACTERS, or contains no RECORD clause at all), the size of the logical record is the fixed record size.

  — When the file has variable-length records whose sizes are controlled by the DEPENDING item specified in the RECORD clause (RECORD VARYING SIZE DEPENDING name), the value of the DEPENDING item determines the size of the logical record.

  — When the file has variable-length records but no DEPENDING item is specified in the RECORD clause of the file description entry (RECORD CONTAINS `m` TO `n` CHARACTERS), the size of the logical record is the size of the data item referenced by *`record-name`*. If the data item has a variable size because it contains a table that is described with an OCCURS DEPENDING clause, the current size of the item is used.

  When the file is described with the RECORD VARYING clause, the logical record size must not be greater than the maximum or less than the minimum number of character positions specified in that clause. (See I-O status code “44.”)

  The content of the logical record released by the rewrite operation is the left-justified value of the data item specified by *`record-name`*. If the logical record size is greater than the data item’s size, the value is logically extended on the right with arbitrary characters (that is, you cannot predict what the extending characters will be). If the logical record size is less than the data item’s size, the value is logically truncated on the right.

  The execution of a REWRITE statement, whether successful or not, does not normally affect the key of reference, the value of file position indicator, or the value of the depending item specified in the DEPENDING phrase of the RECORD clause associated with *`record-name`*; however, the value of the file position indicator can be left undefined in some cases when the I-O status is set to 30.

NOTE: The logical record released by a successful execution of the REWRITE statement is no longer available in the record area unless the file name associated with *`record-name`* is specified in a SAME RECORD AREA clause. The logical record is available to the program as a record of other files referenced in the SAME RECORD AREA clause as the associated output file, as well as the file associated with *`record-name`*.

- File-Status Data Item

  If the file has an associated file-status data item, execution of the REWRITE statement always assigns an appropriate I-O status code as its value. The status “00” reports a successful rewrite operation with no duplicate alternate keys written. The status code “02” reports a successful rewrite operation that created a duplicate alternate key value for at least one alternate record key for which duplicates are allowed (only when the file has the INSERTIONORDER attribute).

  The rewrite operation cannot succeed if the device associated with the file is not a disk device.

NOTE: After an unsuccessful execution of a REWRITE statement, these values are unpredictable:

— The value of the file position indicator

— The contents of the current record area

— The key of reference

Any key value found in the current record area: that is, the value of the current key for indexed files or the value of the alternate key for any type of file-system file, because in HP COBOL sequential and relative files can have alternate keys.

The possible I-O status codes for an unsuccessful rewrite are:

I-O Status Code	Unsuccessful Rewrite
“21”	The file’s access mode is sequential, the file organization is indexed, and the prime record key value of the logical record is not equal to the value of the prime record key of the last record read. The invalid-key condition arises.
“22”	One of the alternate key values in the logical record is equal to the value of that key in a record that already exists in the file, and the DUPLICATES phrase is not specified for that key. The invalid-key condition arises.
“23”	The file’s access mode is random or dynamic, and the specified relative key value or prime record key value does not correspond to that of any record existing in the file. The invalid-key condition arises.
“30”	The rewrite operation failed due to non-COBOL causes. The specified record might or might not have been rewritten. The run-time routines always return this status when the file is not assigned to a disk, in which case nothing is rewritten. The value of the file position indicator can be left undefined in some cases when the I-O status code is “30.” Whenever the rewrite operation terminates with an I-O status code greater than or equal to “30,” execution of the REWRITE statement is unsuccessful. In this case, the applicable USE procedure, if any, is executed.
“43”	The access mode is sequential, and the last input or output statement executed for the file was not a successful READ statement. The record is not released.
“44”	There are two possibilities: — The file is described with the RECORD VARYING clause, and the logical record size is greater than the maximum or less than the minimum number of character positions specified in that clause. The logical record is not released. — The file is a sequential file described with the RECORD VARYING clause, and the size of the logical record is not equal to the size of the record being replaced. The logical record is not released.
“49”	The file is not open in I-O mode. The rewrite operation terminates immediately.

- Access Mode and REWRITE

  — Sequential-access files

  When a sequential file has variable-length records, the number of character positions in the logical record specified by `record-name` must be equal to the number of character positions in the record being replaced. If this is not so, the rewrite operation is unsuccessful and terminates with I-O status code “44.”

  When the access mode of the file affected by the rewrite operation is sequential, the last input-output statement executed for the file must have been a successful READ statement. If it was not, the record is not released and the rewrite operation terminates with the I-O status code “43;” otherwise, the rewrite operation logically replaces the record that was accessed by the READ statement; therefore, for an indexed or queue file, the program logic must verify that the prime record key in the logical record has the same value as the prime record key of the record previously read.

  — Random-access or dynamic-access files

    For relative files with random or dynamic access, the record replaced is the one indicated by the value of the RELATIVE KEY item. If the old record does not exist, an invalid-key condition occurs.

    For indexed files with random or dynamic access or for files being accessed through alternate keys using random or dynamic access, the prime record key data item (the RECORD KEY item) selects the old record to be replaced. The value of alternate keys can differ in the new records, unless this would create duplicate key values for a key whose values must be unique.

- UNLOCK Phrase

  If the UNLOCK phrase appears, the rewrite operation also assures that the record is not in the locked state at the completion of the rewrite operation. If the record was not in the locked state at the beginning of the rewrite operation, no report of this is made to the program.

- Invalid-Key Condition

  Any of these circumstances can lead to an invalid-key condition:

  — The access mode is sequential, the file organization is indexed, and the prime record key value of the logical record is not equal to the value of the prime record key of the last record read. This results in a completion status of 21.

  — The access mode is random or dynamic, and the specified relative key value or prime record key value does not correspond to that of any record existing in the file. This results in I-O status code “23.”

  — One of the alternate key values in the logical record is equal to the value of that key in a record that already exists in the file, and the DUPLICATES phrase is not specified for that key. This results in an I-O status code of “22.”

  When the invalid-key condition exists, the rewrite operation does not occur and execution of the REWRITE statement is unsuccessful. The content of the record area is unaffected. If the INVALID KEY phrase is specified, control passes to the imperative statement in that phrase and no USE procedure is executed. If the INVALID KEY phrase is not specified but an applicable USE procedure exists, that procedure is executed.

  When the invalid-key condition does not exist, and the rewrite operation is successful, and the NOT INVALID KEY phrase is specified, control passes to the imperative statement in that phrase.

Example 10-59 REWRITE Statement for Indexed File

```
READ MASTER-IN WITH LOCK
IF NO-ERROR
   PERFORM GET-INPUT
   REWRITE MASTER-RECORD WITH UNLOCK
   IF NO-ERROR
      ADD 1 TO UPDATE-COUNTER
   ELSE UNLOCKRECORD MASTER-IN
   END-IF
ELSE
   ...
```

NOTE: Invalid-key conditions are taken care of by USE procedures instead of by INVALID KEY phrases on the READ and REWRITE statements.

SEARCH

SEARCH scans a table for an element that satisfies a condition. If SEARCH finds the element, SEARCH sets an index-name to the element's offset value

Before executing SEARCH, the program must:

- Initialize the relevant index-names and the table with appropriate values
- Determine what criteria define successful completion of the search operation

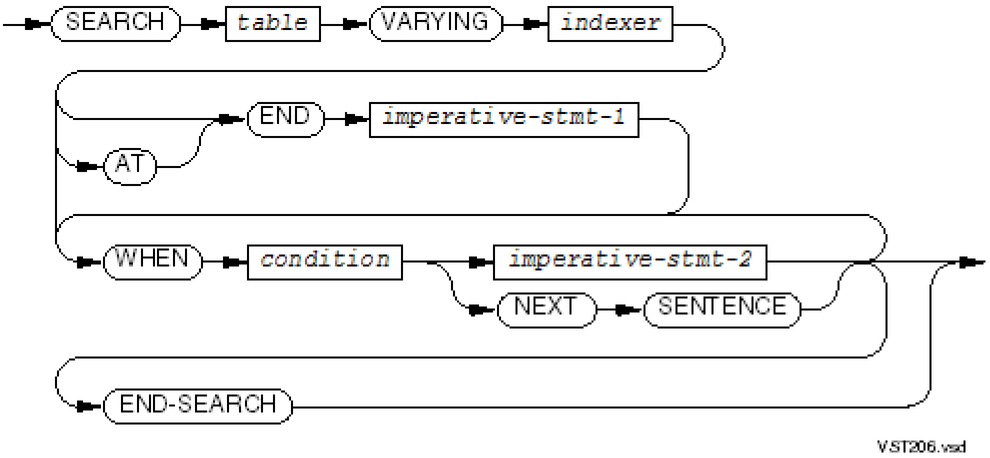
A search proceeds by successively selecting candidates from among the elements of the table and then evaluating the test conditions. For a variable-size table, only elements currently defined as a part of the table can be searched. The last of these is the one referenced by the maximum occurrence number, which is the value of the associated DEPENDING data item.

The search ends when a condition evaluates to TRUE or when every element of the table has been searched, whichever occurs first.

SEARCH operates on a single table dimension. To search an entire multi-dimensional table, the program must execute a SEARCH for each dimension, beginning with the outermost table level and ending with the innermost table level.

SEARCH VARYING

SEARCH VARYING performs a serial search beginning with the current index setting. SEARCH VARYING can also set the value of a data item or another index-name.



`table`

is the name of a data item described with an "OCCURS … INDEXED BY `index-1` … KEY …" clause. The search is done with `index-1`, which the search operation initializes to the middle of the table before beginning to scan. Only the name of `table` can appear, qualified if necessary, but not subscripted or reference modified.

`indexer`

is one of:

- One of the index-names in the INDEXED phrase in the definition of `table`
- An integer data item (which can, itself, be subscripted)
- An index data item

`indexer` cannot specify a special register. `indexer` cannot be subscripted by the first (or the only) index-name specified in the INDEXED phrase of the OCCURS clause of the definition of `table`.

When `indexer` is from the INDEXED phrase, that index is used to start the search. When it is a separate item, `index-1` from the INDEXED option is used to search `table`, and `indexer` is incremented by 1 when `index-1` is incremented.

The SEARCH statement does not initialize its index (either `indexer` or `index-1`). The value of the index at the time the search begins is used.

`imperative-stmt-1`

is an imperative statement to be executed when an at-end condition is detected during the search.

`condition`

specifies a conditional expression used to control the search for an element.

`imperative-stmt-2`

is an imperative statement to be executed when a condition is satisfied.

NEXT SENTENCE

specifies that control be passed directly to the end of the SEARCH statement. It is not recommended (see Usage Considerations).

END-SEARCH

ends the scope of the SEARCH statement, causing the SEARCH to be a delimited-scope statement. Without the END-SEARCH phrase, the SEARCH statement is a conditional statement, which ends at the next period separator.

Usage Considerations:

- CONTINUE is Recommended Over NEXT SENTENCE

  NEXT SENTENCE transfers control to the next period (.), while CONTINUE transfers control to END-SEARCH. Either `imperative-stmt-1`, `imperative-stmt-2`, or both can be CONTINUE statements.

- Execution

  The SEARCH VARYING statement specifies a serial search through elements of the table referenced by `table`. In general terms, the search operation successively increments the value of an index-name, which represents the occurrence number of an element in the table, until either some condition evaluates to TRUE or the occurrence number represented by the value of the index-name exceeds the maximum defined for the table.

  When the VARYING phrase is present, and if it specifies an index-name associated with `table` (by its OCCURS clause), then that index-name is used for the search. If it specifies an index-name not associated with `table`, or if it specifies a separate integer data item or an index data item, the primary index of `table` is used in the search (starting at its current setting), and these operations occur:

  — If you specify an index-name associated with another table, the process increments the occurrence number that index-name represents by the same amount and at the same time as it increments the occurrence number the primary index represents.

  — If you specify an index data item, the process increments the value of the item by the same amount and at the same time as it increments the index associated with `table`.

  — If you specify an integer data item, the process increments the value of the item by 1 at the same time as it increments the index associated with `table`.

  Without the VARYING phrase, the search varies the value of the primary index of `table` (starting at its current setting). Any other indexes specified with `table` remain unchanged.

If, when execution of the SEARCH statement begins, the value of the index-name corresponds to an occurrence number greater than the one that identifies the last element of the table, the at-end condition exists and the search operation terminates immediately; otherwise, the search operation proceeds:

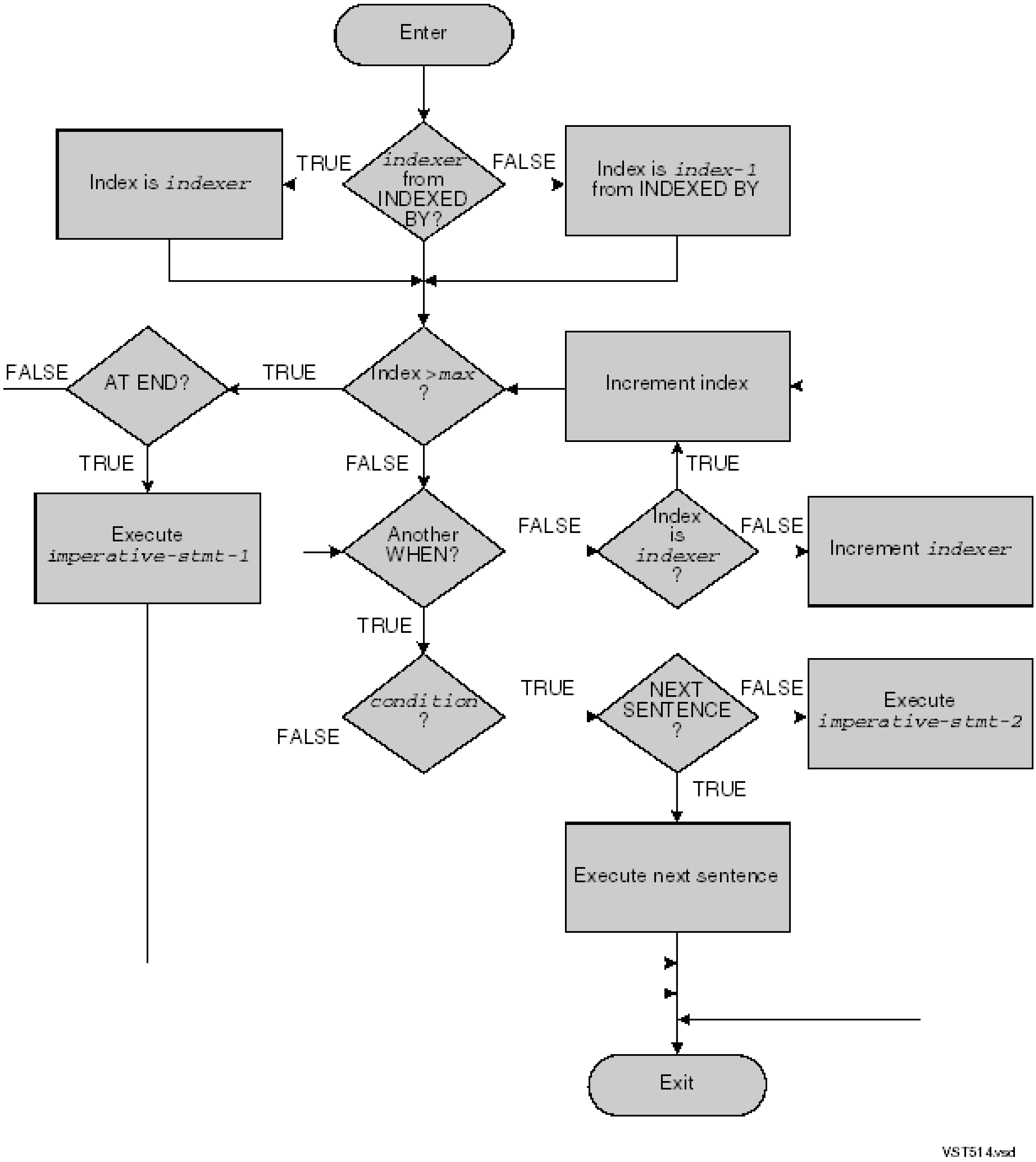
1. Each condition is evaluated, in the order in which they appear in the statement, until one of them evaluates to TRUE or all of them evaluate to FALSE. For each operand of a condition, the process of operand identification occurs just prior to its use each time the operand participates in the determination of that condition's value. For details, see Condition Evaluation Rules (page 287).
2. When a condition evaluates to TRUE, the search operation terminates immediately. If that condition is followed by the NEXT SENTENCE phrase, control passes to the next executable sentence; otherwise, the associated `imperative-stmt-2` is executed and control passes to the end of the SEARCH statement (unless the `imperative-stmt-2` explicitly transfers control elsewhere using a GO TO statement).
3. When all conditions evaluate to FALSE, the index-name value is incremented to correspond to the next occurrence number.
4. When the new value of the search index-name corresponds to an occurrence number greater than the one that identifies the last element of the table, the at-end condition exists and the search operation terminates immediately; otherwise, the search operation repeats from Step 1.

When the search operation terminates due to the at-end condition, `imperative-stmt-1` is executed, if the AT END phrase is specified. Control then passes to the end of the SEARCH statement (unless `imperative-stmt-1` explicitly transfers control elsewhere using a GO TO statement).

- Multidimensional Tables

  If `table` is an element of another table, you have a multidimensional table. Each dimension of the multidimensional table must be declared with an INDEXED phrase. A SEARCH statement uses only the first index of each dimension. To search an entire multidimensional table, you must execute several SEARCH statements. Before each execution of a SEARCH statement, you must execute a SET statement to adjust index-names to appropriate settings. The index-names must be used in the SEARCH statement's condition.

Figure 10-11 Execution of a SEARCH VARYING Statement With WHEN Phrases

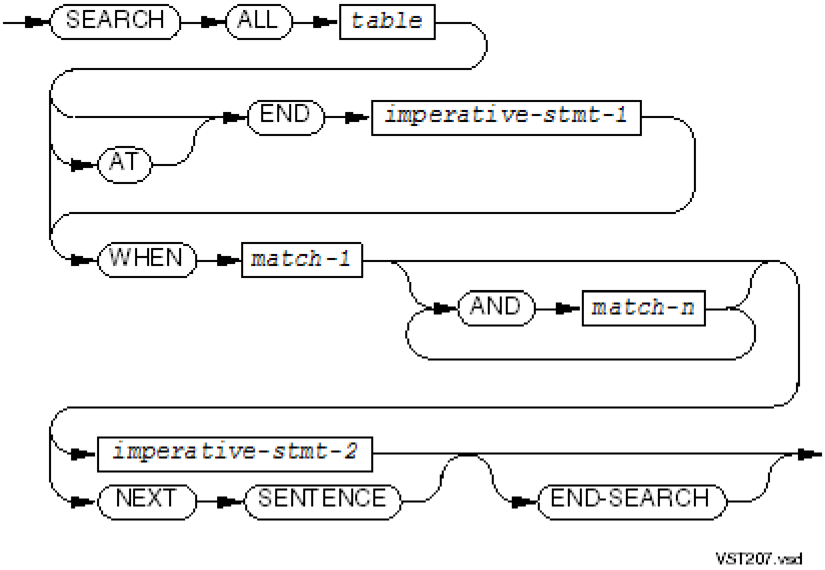


Example 10-60 SEARCH VARYING Statement

```
WORKING-STORAGE SECTION.
01  COMMANDS.
    05  FILLER          PIC X(6)  VALUE "ADD".
    05  FILLER          PIC X(6)  VALUE "DELETE".
    ...
01  COMMANDS-IN-TABLE REDEFINES COMMANDS.
    05  COMMAND-ENTRIES    PIC X(6)  OCCURS 6 TIMES
             INDEXED BY TABLE-INDEX.
    77  COMMAND-IN      PIC X(6).
    ...
PROCEDURE DIVISION.
   ...
   SET TABLE-INDEX TO 1
   SEARCH COMMAND-ENTRIES VARYING TABLE-INDEX
      AT END
         PERFORM COMMAND-ERROR-ROUTINE
      WHEN COMMAND-ENTRIES(TABLE-INDEX) = COMMAND-IN
         CONTINUE
   END-SEARCH
   ...
```

SEARCH ALL

SEARCH ALL performs a binary search on a table. The table must be declared with a KEY phrase in its OCCURS clause.



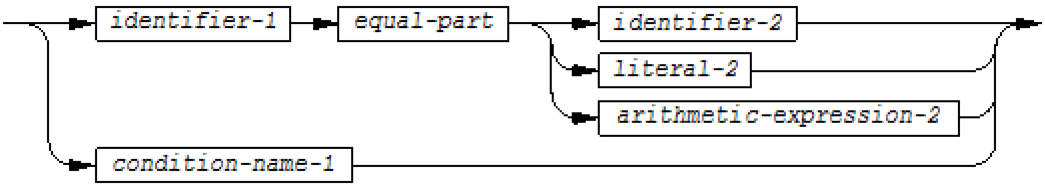
table

is the name of a data item described with an "OCCURS … INDEXED BY *index-1* … KEY …" clause. The search is done with *index-1*, which the search operation initializes to the middle of the table before beginning to scan. Only the name of *table* can appear, qualified if necessary, but not subscripted or reference modified.

imperative-stmt-1

is an imperative statement to be executed when an at-end condition is detected (the end of the search is reached with no entry satisfying all specified conditions).

match-1, match-n



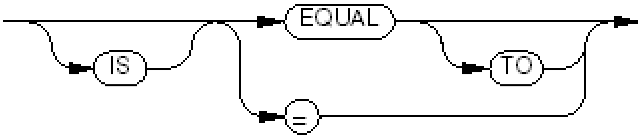
VST208.vsd

are matches that terminate the search operation.

identifier-1

is a data-name that is subscripted by the first index-name listed in the INDEXED clause that defines the indexes of *table*, along with any other subscripts that are required to identify the element in a multidimensional table. *identifier-1* must be defined as the first key of *table*. It can be qualified, but it cannot include a reference modifier.

equal-part



VST209.vsd

identifier-2

is any identifier.

literal-2

is any literal.

arithmetic expression-2

is any arithmetic expression.

condition-name-1

is a condition (a level-88 item) that terminates the search operation. *condition-name-1* must be defined as having a single value. The data-name with which *condition-name-1* is associated must be specified as the first in the KEY phrase of the OCCURS clause of *table*.

imperative-stmt-2

is an imperative statement to be executed when the condition that *condition-name-1* specifies is satisfied.

NEXT SENTENCE

specifies that control be passed directly to the end of the SEARCH statement. It is not recommended (see Usage Considerations).

END-SEARCH

ends the scope of the SEARCH statement, causing the SEARCH to be a delimited-scope statement. Without the END-SEARCH phrase, the SEARCH statement is a conditional statement, which ends at the next period separator.

Usage Considerations:

- CONTINUE is Recommended Over NEXT SENTENCE

  NEXT SENTENCE transfers control to the next period (.), while CONTINUE transfers control to END-SEARCH. Either *imperative-stmt-1*, *imperative-stmt-2*, or both can be CONTINUE statements.

- Binary Search

  HP COBOL performs a binary search when these conditions are met:

  — Each condition-name referenced in the WHEN clause must be defined as having a single value.

  — The data-name associated with a condition-name must appear in the KEY phrase in the OCCURS clause of the data definition of *table*.

  — Each *identifier-1* must be subscripted by the first index-name associated with *table*, along with other subscripts as required, and must be referenced in the KEY phrase in the OCCURS clause of the data definition of *table*.

  — The identifiers mentioned in the operands must not be referenced in the KEY phrase in the OCCURS clause associated with *table*, and must not be subscripted by the first index-name associated with *table*.

  — When a data-name in the KEY phrase of the OCCURS clause associated with *table* is referenced, or when a condition-name associated with such a data-name is referenced, all preceding data-names (or the associated condition-names) in the KEY phrase of that OCCURS clause must also be referenced.

  When a SEARCH ALL statement does not meet these conditions, the compiler issues a warning and does a linear search.

- Additional Syntactic Constraints

  If the WHEN phrase mentions either a data-name in the KEY phrase of the OCCURS clause of the definition of *table* or a condition-name associated with a data-name in that KEY phrase, all preceding data-names in the KEY phrase must also be mentioned in the WHEN phrase. To illustrate, suppose you have this OCCURS clause:

```
05 DEPT-TABLE OCCURS 500 TIMES
              ASCENDING KEY IS DIV
                               SEC
                               DEPARTMENT
              INDEXED BY NDEX, MDEX.
   07 DIV PIC 9(4).
   07 SEC PIC 9(3).
   07 DEPARTMENT PIC 9(3).
      88 R-AND-D VALUE 555.
```

  You can search on DIV, or DIV and SEC, or all three keys. You cannot search on SEC without searching on DIV, and you cannot search on DEPARTMENT without searching on DIV and SEC.

- Order of Table Elements

  Verify that the table is ordered as specified in the ASCENDING or DESCENDING phrase of the OCCURS clause of the table’s definition. If it is not, the result of the search is undefined.

- Execution

  The SEARCH ALL statement specifies a search through the elements of a table. The results of a SEARCH ALL statement are predictable only when both of these are true:

  — The data in the table are ordered in the manner described in the KEY phrase of the OCCURS clause that describes the table.

  — The condition specified in the WHEN phrase (called the “target condition”) evaluates to TRUE only during the consideration of exactly one of the table elements.

The search operation varies the value of the first index-name that appears in the INDEXED phrase of the OCCURS clause describing the table. The search operation insures that the varying value always corresponds to an occurrence number defined for the table.

The SEARCH ALL operation proceeds in a binary fashion, successively setting the index-name to correspond to different table elements and evaluating the target condition. For each operand of the target condition, the process of operand identification occurs just before its use each time the operand participates in the determination of the value of the target condition.

- When Exactly One Matching Element Is Found

  When the target condition evaluates to TRUE, the search all operation terminates and the value of the index-name corresponds to the element under consideration. If the NEXT SENTENCE phrase is specified, control passes to the next executable sentence; otherwise, `imperative-stmt-2` is executed and then control passes to the end of the SEARCH ALL statement (unless execution of `imperative-stmt-2` explicitly transfers control elsewhere using a GO TO statement).

  If more than one element satisfies the condition, the index can point to any one of them.

- When No Matching Element Is Found

  When the value of the target condition is FALSE, the search operation terminates with the at-end condition and the value of the index-name is undefined. If the AT END phrase is specified, `imperative-stmt-1` is executed. Control passes to the end of the SEARCH statement (unless execution of `imperative-stmt-2` explicitly transfers control elsewhere using a GO TO statement).

- Tables Defined With a DEPENDING Phrase

  For a variable-occurrence table, only those elements currently defined as a part of the table can be candidates. The last of these is the one specified by the maximum occurrence number, which is the value of the associated DEPENDING data item.

- Index Values

  During the search operation, the first index-name appearing in the INDEXED phrase of the OCCURS clause describing the table is varied so that its value always corresponds to an occurrence number defined for the table. The value of this index-name at the start of execution of the SEARCH statement is immaterial.

Example 10-61 SEARCH ALL Statement

```
WORKING-STORAGE SECTION.
01  COMMANDS.
    05  FILLER          PIC X(6)  VALUE "ADD".
    05  FILLER          PIC X(6)  VALUE "DELETE".
    05  FILLER          PIC X(6)  VALUE "EXIT".
    05  FILLER          PIC X(6)  VALUE "LIST".
        ...
01  COMMANDS-IN-TABLE REDEFINES COMMANDS.
    05  COMMAND-ENTRIES  PIC X(6)  OCCURS 6 TIMES
             ASCENDING KEY IS COMMAND-ENTRIES
             INDEXED BY TABLE-INDEX.
        ...
01  COMMAND-NUMBERS.
    05  COMMAND-INDEX     PIC 99  COMP VALUE 1.
    05  COMMAND-IN        PIC X(6).
        ...
PROCEDURE DIVISION.
        ...
   SEARCH ALL COMMAND-ENTRIES
      AT END
         PERFORM COMMAND-ERROR-ROUTINE
         GO TO GET-ANOTHER-COMMAND
      WHEN COMMAND-ENTRIES(TABLE-INDEX) = COMMAND-IN
         CONTINUE
   END-SEARCH
   SET COMMAND-INDEX TO TABLE-INDEX
        ...
```

SET

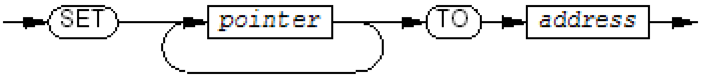
SET TO

SET TO sets the values of data item addresses, indexes, switches, or conditional variables.

Topics:

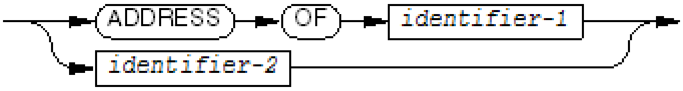
- POINTER Data Items
- Nonpointer Data Items

POINTER Data Items



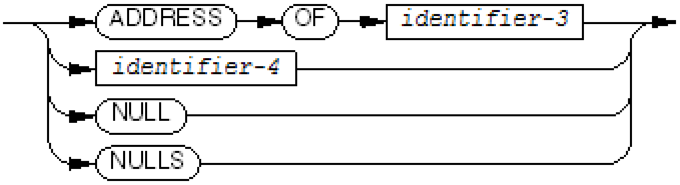
VST605.vsd

pointer



VST606.vsd

address



VST607.vsd

identifier-1

is a level-01 or level-77 data item in the Linkage Section, or a level-01 or level-77 BASED data item in any section, that does not have an ACCESS MODE STANDARD clause in its data description entry. If *pointer* is ADDRESS OF *identifier-1*, *identifier-1* is relinked so that subsequent references to *identifier-1* reference the item whose address is specified by *address*.

If *address* is ADDRESS OF *identifier-3*, then *identifier-1* is linked to the address of *identifier-3* (not the value of *identifier-3*).

If *address* is *identifier-4*, then *identifier-1* is linked to the address that *identifier-4* contains.

If *address* is NULL or NULLS, then *identifier-1* is linked to a null address guaranteed to point to no data item.

identifier-2

is a data item with USAGE POINTER. If *pointer* is *identifier-2*, the address specified by *address* is moved into *identifier-2*. This address is valid until the program terminates or returns control to its calling program.

identifier-3

is a data item of any level except 88, anywhere in the Data Division. The value of ADDRESS OF *identifier-3* is the address of *identifier-3*, not the value of *identifier-3*.

identifier-4

is a data item with USAGE POINTER.

NULL

NULLS

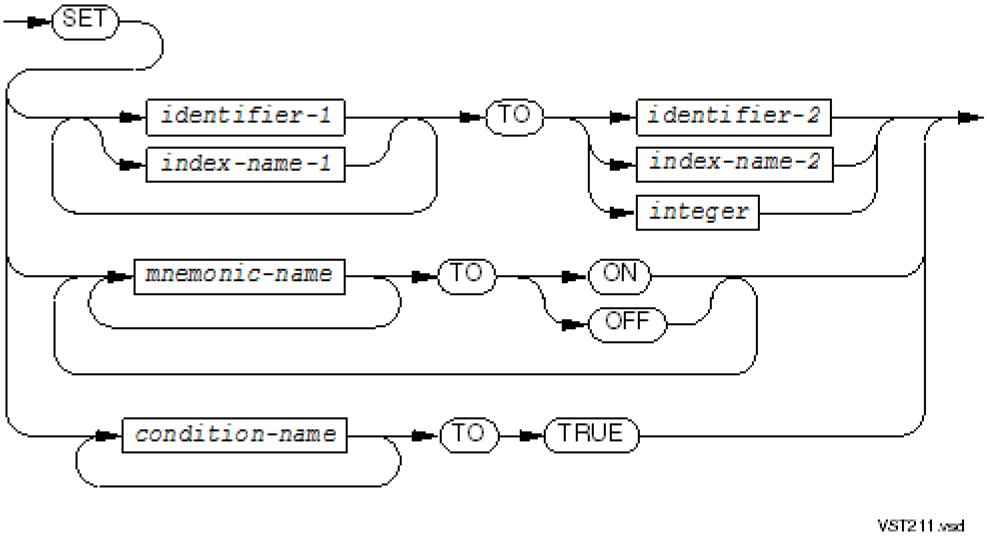
is a null address guaranteed not to point to any data item. A reference to a pointer whose value is NULL causes a trap 1 instruction failure.

If you redefine a VS COBOL II pointer variable as a COMPUTATIONAL field and perform an arithmetic operation on the field to change the value of the pointer, the pointer does not behave the same as it would in IBM/370 COBOL. The reason is that pointer format depends on machine architecture, and NonStop servers and IBM/370 machines have different architectures.

Nonpointer Data Items

SET TO can:

- Set an index-name to one of these values:
  - A value that corresponds to the occurrence number designated by another index-name
  - A value that corresponds to the occurrence number corresponding to the value of an integer literal or an integer data item
  - The value of an index data item
- Set the value of an index data item to the value of an index-name or the value associated with the contents of another index data item
- Set the value of an integer data item to the occurrence number of an index
- Set an external switch to ON or OFF
- Set a conditional variable to a value that makes an associated condition true



identifier-1

is the name of an integer elementary item or an index data item to be set.

`identifier-2`

is the name of an integer data item or an index data item.

`index-name-1`

is the name of an index to be set.

`index-name-2`

is the name of an index.

`integer`

is a numeric literal having no fractional part. If signed, it must have a positive value.

`mnemonic-name`

is the mnemonic name associated with an external switch in the SPECIAL-NAMES paragraph of the Environment Division. A SET statement can refer to any of the 15 external switches.

`condition-name`

is the condition-name (level-88 item) associated with a conditional variable.

Usage Considerations:

- Index-Names and Index Data Items

  An index-name is directly associated with a table and is declared in the INDEXED phrase of the table's data description entry. The value associated with an index-name is related only to the table with which it is defined.

  An index data item is a separate data item, not associated with any table, that is declared with the USAGE INDEX clause.

- Copying and Converting Values

  The SET statement performs copying between indexes and index data items. It also performs conversions from indexes to integer data items (occurrence numbers) or from integer data items (occurrence numbers) to indexes. This conversion is necessary because the values associated with indexes and index data items are machine-architecture related, whereas the occurrence numbers are not.

  Table 10-10 summarizes these rules:

  — The receiving item (`index-name-n`) is set to a value causing it to refer to the table element corresponding in occurrence number to the table element specified by the sending item (`identifier-1`, `index-name-1`, or `integer`). If the sending item is an index data item, or if it is an index-name that is related to the same table as the receiving item, no conversion occurs.

  — If the receiving item is an index data item, it can be set equal to the contents of either an index-name or another index data item. No conversion occurs.

  — If the receiving item is not an index data item, it can be set only to an occurrence number corresponding to the value of an index-name. The sending item cannot be a numeric integer literal or a numeric integer data item.

  — The assignment process is repeated for any other receiving data-names specified. Each time, the value of a sending data item is used as it was at the beginning of the execution of the statement. Any subscripting or indexing associated with the sending data item is evaluated immediately before the value of the receiving data item is changed.

Table 10-10 Valid SET TO Combinations

	Receiving Item		
Sending Item	**Integer Data Item**	**Index-Name**	**Index Data Name**
Integer literal	No (rule 3)	OK (rule 1)	No (rule 2)
Integer data item	No (rule 3)	OK (rule 1)	No (rule 2)

Table 10-10 Valid SET TO Combinations *(continued)*

	Receiving Item		
Sending Item	**Integer Data Item**	**Index-Name**	**Index Data Name**
Index-name	OK (rule 3)	OK (rule 1)	OK (rule 2)*
Index data item	No (rule 3)	OK (rule 1)	OK (rule 2)*

* No conversion occurs

- Index Values Before and After Execution

  If the sending item is an index-name, then before the execution of the SET TO statement, the value of the index must correspond to an occurrence number of an element in the associated table.

  If the receiving item is an index-name, then after the execution of the SET TO statement, the value of the index must correspond to an occurrence number of an element in the associated table.

- External Switches

  The SET statement modifies the status of the external switch associated with each specified *mnemonic-name*. When the SET statement changes the status of a switch to ON, any condition-name associated (in the SPECIAL-NAMES paragraph) with the ON setting of that switch evaluates to TRUE. When the SET statement changes the status of a switch to OFF, any condition-name associated (in the SPECIAL-NAMES paragraph) with the ON setting of that switch evaluates to FALSE.

- Condition Names

  The SET statement assigns to the conditional variable associated with the *condition-name* the value of the first literal in the VALUE clause of the definition of the *condition-name*.

  When more than one *condition-name* appears in the SET statement, the results are the same as if a separate SET statement had been written for each *condition-name* in the same order as specified in the SET statement.

- *integer* Out of Range

  In the statement

  ```
  SET index-name-n TO integer
  ```

  the value of *integer* must be at least zero and not greater than the maximum number of occurrences plus one. If the value of *integer* is outside this range, a warning message is produced. If the warning message is produced and subscript checking is active, the run unit terminates abnormally.

  If the value of *integer* is greater than 2,147,483,647 or less than -2,147,483,648, or if the product of *integer* and the occurrence length of the associated table (the length of one occurrence of the table) is not within that range, a run-time error occurs.

- Any Sending Item Out of Range

  If the value of any sending item (*identifier-1*, *index-name-1*, or *integer*) is greater than 2,147,483,647 or less than -2,147,483,648 and subscript checking is active, an error message is produced and the run unit terminates abnormally.

SET UP or SET DOWN

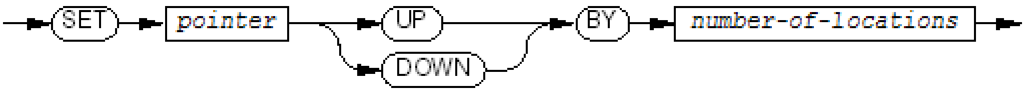
SET UP increments data items addresses or indexes; SET DOWN decrements them.

Topics:

- POINTER Data Items
- Nonpointer Data Items

POINTER Data Items

SET UP increments a pointer by an integral number of memory locations. SET DOWN decrements a pointer by an integral number of memory locations.

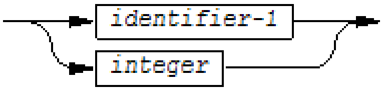


VST608.vsd

pointer

is a data item with USAGE POINTER. In these parameter descriptions, assume that the value of `pointer` is the address *p*.

number-of-locations



VST609.vsd

identifier-1

is either an elementary numeric data item described as an integer or a function that returns an integer value. If *number-of-locations* is *identifier-1*, and the value of *identifier-1* is *n*, then `pointer` is adjusted so that it references the address *p*+*n* bytes for SET UP or *p* -*n* bytes for SET DOWN.

integer

is a numeric literal whose value is an integer. If *number-of-locations* is *integer*, then `pointer` is adjusted so that it references the address *p*+*integer* bytes for SET UP or *p* -*integer* bytes for SET DOWN.

Usage Considerations:

- Do Not Use SET UP or SET DOWN for Record Pointers

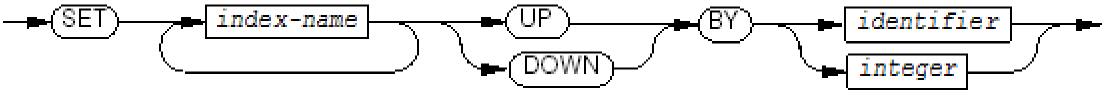
  Do not use SET UP or SET DOWN to point a record pointer at a different record. This would be syntactically correct, but the memory positions of records are not defined in HP COBOL and can vary between implementations.

- Incompatibility With IBM/370 Mainframe Pointer Variables

  If you redefine a VS COBOL II pointer variable as a COMP field and perform an arithmetic operation on the field to change the value of the pointer, the pointer does not behave the same as it would in IBM/370 COBOL. The reason is that pointer format depends on machine architecture, and HP and IBM/370 machines have different architectures.

Nonpointer Data Items

SET UP increments an index value by an integer amount. SET DOWN decrements an index value by an integer amount.



VST212.vsd

`index-name`

is the name of an index whose value is to be incremented or decremented.

`identifier`

is the identifier of an integer numeric item whose value UP adds to the index or DOWN subtracts from the index.

`integer`

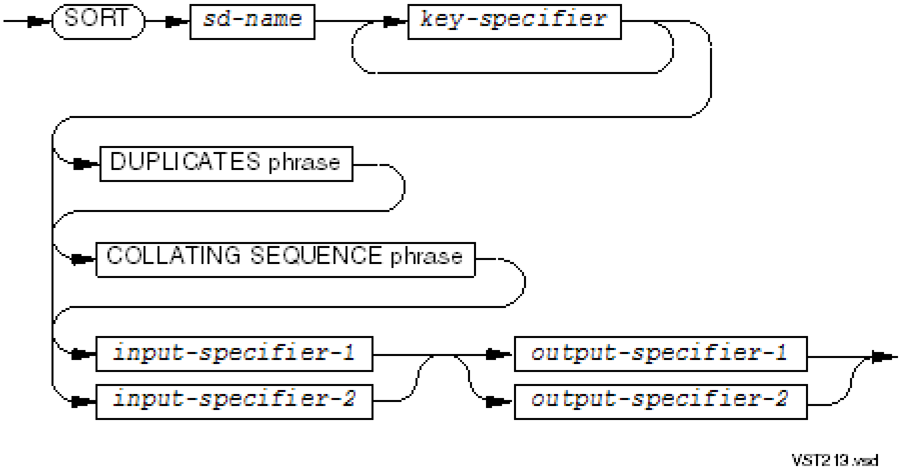
is an integer that UP adds to the index or DOWN subtracts from the index. It can be signed.

The value of each `index-name` is incremented (UP) or decremented (DOWN) by a value corresponding to the number of occurrences represented by `identifier` or `integer`. Each time, the identifier of an integer numeric item is used as it was at the beginning of the execution of the statement.

Both before and after the execution of the SET statement, the value of the index must correspond to an occurrence number of an element in the associated table.

SORT

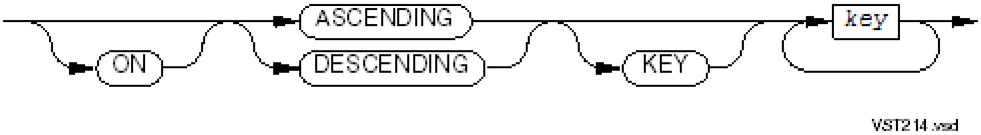
SORT orders a set of records according to one or more keys. The records can either be in a file or an input procedure can send them to SORT, one at a time. SORT can either write the sorted records to a file or it can send them to an output procedure, one at a time. SORT calls the FastSort utility.



`sd-name`

is the file name given in a sort-merge file description (SD) entry.

`key-specifier`

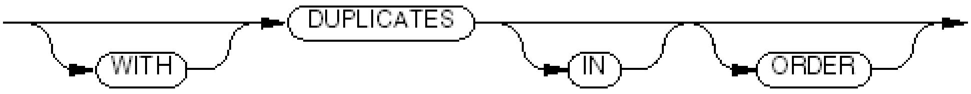


specifies one or more of the items described in the record descriptions of `sd-name`.

`key`

is used to sort the records in ascending or descending order. The first `key` is compared first, the second next, and so on. Two records equal in all sort keys are sorted by their original order in the file.

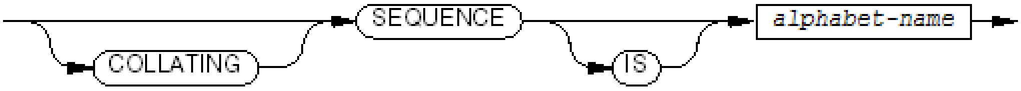
DUPLICATES phrase



VST215.vsd

specifies that, if two or more records contain equal values for all sort keys, their final order within the sort file is the order in which they were released to the sort file. If this phrase is omitted, the order of such duplicate records is arbitrary.

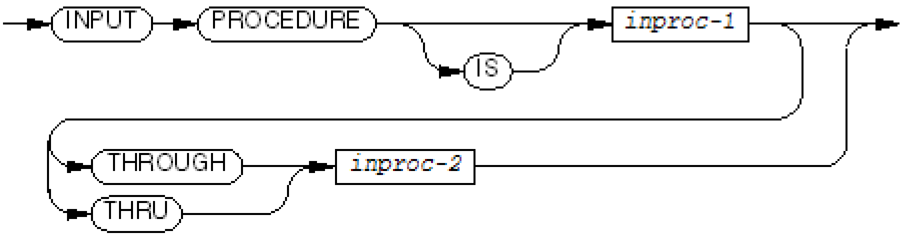
COLLATING SEQUENCE phrase



VST216.vsd

specifies a collating sequence for sorting. The alphabet-name must be associated with a sorting sequence in the SPECIAL-NAMES paragraph of the Environment Division (see SPECIAL-NAMES Paragraph (page 118)).

input-specifier-1



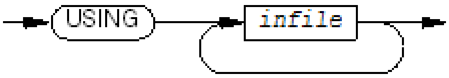
VST217.vsd

specifies a procedure that sends records one at a time to SORT using one or more RELEASE statements. The procedure *input-specifier-1* cannot terminate before it finishes sending records.

inproc-1, inproc-2

are sections or paragraphs of the Procedure Division.

input-specifier-2



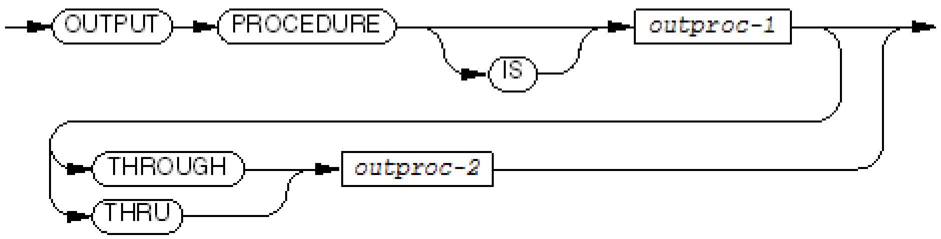
VST391.vsd

specifies one or more files (a maximum of 31) that contain the records to be sorted.

infile

is a file description name. During execution of the SORT statement, *infile* must either be closed or be open in another process or by another file description entry, but not in conflict with a following open for protected input.

output-specifier-1



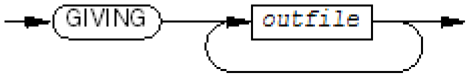
VST218.vsd

specifies a procedure that processes the sorted records, one at a time. Each record is returned by a RETURN statement. The procedure *output-specifier-1* cannot terminate before it finishes returning all the sorted records.

outproc-1, outproc-2

are sections or paragraphs of the Procedure Division.

output-specifier-2



VST392.vsd

specifies one or more files (a maximum of 31) where the sorted records are to be written.

outfile

is a file description name. During execution of the SORT statement, *outfile* must be closed and cannot be locked.

Usage Considerations:

- Using the Features of FastSort

  Three utility routines that enable you to control the features of FastSort (such as multi-processor parallel sorting) are:

  - COBOL85_SET_SORT_PARAM_VALUE_
  - COBOL85_SET_SORT_PARAM_TEXT_
  - COBOL85_RETURN_SORT_ERRORS_

  For descriptions of the preceding routines, see Chapter 14: Libraries and Utility Routines (page 607).

- How the Scratch File is Determined

  In this explanation, if a scratch file is specified but its value is all spaces, assume that no scratch file was specified.

  If COBOL_SET_SORT_PARAM_TEXT_ specifies a SCRATCH-FILE, then that file is the scratch file.

  If the SELECT statement associated with *sd-name* specifies the =_SORT_DEFAULTS DEFINE as the *define-name-literal*, then:

  - If the =_SORT_DEFAULTS DEFINE exists and specifies a scratch file, then that file is the scratch file.
  - If the =_SORT_DEFAULTS DEFINE exists but does not specify a scratch file, then a temporary file on the volume $SYSTEM is the scratch file.

  - If no =_SORT_DEFAULTS DEFINE exists, then a temporary file on the volume $SYSTEM is the scratch file.
  - If the SELECT statement associated with `sd-name` does not specify the =_SORT_DEFAULTS DEFINE as the `define-name-literal`, then the file that the SELECT statement specifies is the scratch file.

  For instructions for creating the =_SORT_DEFAULTS DEFINE, see the *FastSort Manual*.

- How the Volume of the Temporary Swap File is Determined

  The operating system assigns a swap file to swap pages in and out of memory while the compiler is running. The swap file mirrors all of the data areas that the compiler uses. The ideal swap file is a fast device that is neither busy nor mirrored. To redirect the swap file, give `define-name-literal` the value =_SORT_DEFAULTS.

  The swap file is a temporary file with a volume but no subvolume. If you specify a swap file, the `file-id` you specify is not used, only the volume. If you specify only the `file-id`, the default volume is used.

  In this explanation, if a swap file is specified but its value is all spaces, assume that no swap file was specified.

  If COBOL_SET_SORT_PARAM_TEXT_ specifies a SWAP-FILE, then the swap file is created on that file's volume.

  If the SELECT statement associated with `sd-name` specifies the =_SORT_DEFAULTS DEFINE as the `define-name-literal`, then:

  - If the =_SORT_DEFAULTS DEFINE exists and specifies a swap file, then the swap file is created on that file's volume.
  - If the =_SORT_DEFAULTS DEFINE exists but does not specify a swap file, then the swap file is created on the volume used for the scratch file.
  - If no =_SORT_DEFAULTS DEFINE exists, then the swap file is created on the volume used for the scratch file.
  - If the SELECT statement associated with `sd-name` does not specify the =_SORT_DEFAULTS DEFINE as the `define-name-literal`, then the swap file is created on the volume of the file that the SELECT statement specifies.

For instructions for creating the =_SORT_DEFAULTS DEFINE, see the *FastSort Manual*.

- Placement of SORT Statements

  A SORT statement cannot appear in the Declaratives Portion of the Procedure Division. It can appear anywhere in the other portion except within the range of a sort input procedure or a sort or merge output procedure.

- File Descriptions

  File `sd-name` must be described by a sort-merge file description entry (sort-merge file description entry) in the File Section of the Data Division. The file can be described as having fixed-length or variable-length records. Every file record must contain all of the sort key fields.

- Restrictions on Sort Keys

  Each `key` identifies a sort key data item and is subject to these restrictions:

  - The data item specified by `key` must be described within a record associated with `sd-name`. When `sd-name` has more than one record description, a sort key item can be defined within any one of those record descriptions.
  - No sort key data item can have a variable size (have a subordinate described with an OCCURS DEPENDING clause).
  - No sort key data item can be described with an OCCURS clause or can be subordinate to an item described with an OCCURS clause.

- Sort Input and Sort Output Files

  Files specified by *infile* and *outfile* must be defined as data files; that is, none of them can be defined in a sort-merge file description entry.

  If the file identified by *sd-name* has variable-length records, the size of the records contained in the file identified by *infile* must not be less than the size of the smallest record described for *sd-name*; likewise, the size must not be greater than the size of the largest record described for *sd-name*. If *sd-name* is described as having fixed-length records, the size of the records contained in the file described by *infile* must not be larger than the largest record defined for *sd-name*.

  If the file identified by *outfile* has variable-length records, the size of the records contained in the file identified by *sd-name* must not be less than the size of the smallest record described for *outfile* ; likewise, the size must not be greater than the size of the largest record described for *outfile*. If *outfile* is described as having fixed-length records, the size of the records contained in the file described by *sd-name* must not be larger than the largest record defined for *outfile*.

  If *infile* specifies a file whose SELECT clause does not include the OPTIONAL phrase, the file must be present at execution time.

  If an instance of *outfile* identifies an indexed or queue file, the major key must be associated with the ASCENDING phrase and the first instance of *key* must specify the same character positions in its record as are specified for the prime record key for that file.

  A COBOL program can sort to and from these types of files:

  — Disk files
  — Tape files
  — Multiple-reel tape files
  — Tape files on a multiple-file reel

  No more than one file name from a multiple-file reel can be in a SORT statement.

- Sort Input Procedures

  The INPUT PROCEDURE phrase defines a sort input procedure that extends from *inproc-1* to *inproc-2*, each of which must identify a Procedure Division section or paragraph. If THROUGH or THRU is omitted, the compiler assumes an *inproc-2* that specifies the same section or paragraph as *inproc-1*.

- Sort Output Procedures

  The OUTPUT PROCEDURE phrase defines a sort output procedure that extends from *outproc-1* to *outproc-2*, each of which must identify a Procedure Division section or paragraph. If THROUGH or THRU is omitted, the *compiler* assumes an *outproc-2* that specifies the same section or paragraph as *outproc-1*.

- Execution Phases

  The execution of the SORT statement consists of three phases:

  — The input phase transfers records to the sort file *sd-name*, either from one or more *infile* files specified in a USING phrase or from a sort input procedure specified in an INPUT PROCEDURE phrase.
  — The sort phase reorders these records according to the sort keys and the applicable collating sequence.
  — The output phase either transfers the sorted records to a set of one or more files specified by *outfile* in the GIVING phrase or returns them to the sort output procedure specified in an OUTPUT PROCEDURE phrase.

- Input Phase of Execution
  - USING phrase

    If the USING phrase appears, the input phase transfers all of the records in each input file `infile` to the sort file *sd-name*. When the SORT statement begins executing, each input file must be either closed but not locked, or open but not in conflict with a following open for protected input (that is, the same file connector cannot be open).

    For each input file, the input phase implicitly opens it in the input mode, executes implicit "READ *infile*NEXT AT END ..." statements to retrieve the records and implicit release operations to release them to the sort file, then it implicitly closes the input file. The open and close operations are equivalent to OPEN INPUT and CLOSE statements without any optional phrases.

    All of these implicit functions are performed such that any associated declarative procedures are executed; however, the execution of a declarative procedure must not cause the execution of any statement that manipulates any of the input files or accesses the record area associated with any input file.
  - INPUT PROCEDURE phrase

    If the INPUT PROCEDURE phrase appears, the input phase executes an implicit PERFORM *procedure-group* statement (where *procedure-group* is procedure name, optionally followed by THROUGH or THRU and another procedure name). The sort input procedure thereby specified is responsible for releasing the records to be sorted by one or more executions of RELEASE statements. The sort input procedure must not open a file specified by `outfile`. Control automatically returns to the sort operation after completing the execution of the implicit PERFORM statement.

    The range of the implicit procedure must not cause the execution of a MERGE, RETURN, or SORT statement or a RELEASE statement that references any sort or merge file other than the one identified by *sd-name*.
- Sort Phase of Execution

  The sort phase reorders the records within the sort file according to their sort key values. The records of *sd-name* are first sorted in accordance with their values for the most significant (first listed) sort key. When two or more records have equal values for the current sort key, that group of records is then sorted in accordance with the record values for the next most significant key, and so on. When two or more records contain equal values for all sort keys, their final order within the sort file depends on the DUPLICATES phrase. If DUPLICATES is specified, their order will be the order in which they were released to the sort file. If DUPLICATES is not specified, their order is arbitrary.

  The *key* s are listed from left to right within the SORT statement in order of decreasing significance, without regard to how the list of keys is divided into ASCENDING KEY or DESCENDING KEY phrases; therefore, the data item specified by the first *key* is the most significant sort key, and the data item specified by the last *key* is the least significant sort key.

  The ordering implied by a sort key depends upon its KEY phrase:
  - When the sort key is specified in an ASCENDING KEY phrase, the sorted sequence is from the record having the lowest value in the sort key data item to the record having the highest value in the sort key data item.
  - When the sort key is specified in a DESCENDING KEY phrase, the sorted sequence is from the record having the highest value in the sort key data item to the record having the lowest value in the sort key data item.
  - In both cases, the determination of which value is higher is made according to the rules for comparison of operands in a relation condition.

The collating sequence that applies to the comparison of nonnumeric sort key data items is the one specified by the alphabet-name in the COLLATING SEQUENCE phrase. When this phrase is not specified, the program collating sequence applies.

- Output Phase of Execution
  - — GIVING phrase

    If the GIVING phrase appears, the output phrase transfers all records in the *sd-name* file to each output file *outfile*. Each output file must be in the closed state, but not in the locked state, when the output phase begins executing. First, the output phrase implicitly opens each output file in the output mode. The output phase then executes implicit return operations to retrieve the records from the sort file and implicit WRITE statements write operations to release them to each of the output files. Finally, the output phase implicitly closes each of the output files.

    The open and close operations are equivalent to OPEN OUTPUT and CLOSE statements having no optional phrases.

    All of these implicit functions are performed such that any associated declarative procedures are executed; however, the execution of a declarative procedure must not cause the execution of any statement that manipulates any of the output files or accesses the record area associated with any output file.

    When more than one instance of *outfile* is specified, each operation is performed on each file in the order that they appear in the GIVING phrase. That is, the output files are opened in order, each retrieved record is written to each file in order, and then the files are closed in order.

    If an output file is a relative file, the relative key data item is set to the value 1 for the first record returned, 2 for the second, and so forth. After the SORT statement finishes executing, the content of the relative key data item indicates the last record returned to the file.

    If an output file has fixed-length records, each record retrieved from the sort file containing fewer character positions than the fixed size specified for the output file is extended on the right with as many space characters as needed before it is written to the output file.

    The first attempt to write beyond the defined boundaries of an output file causes a boundary violation condition. This in turn causes the execution of the applicable USE procedure, if one exists.

    The sort operation automatically closes the sort file after the end of the output phase. Then the SORT statement terminates execution.

  - — OUTPUT PROCEDURE Phrase

    If the OUTPUT PROCEDURE phrase appears, the output phase executes an implicit PERFORM *procedure-group* statement (where *procedure-group* is the procedure name, optionally followed by THROUGH or THRU and another procedure name). The sort output procedure thereby specified is responsible for retrieving the sort file records by one or more executions of RETURN statements. The sort output procedure must not open the sort input file *infile*. Control automatically returns to the sort operation after execution passes beyond the last statement in the range of the PERFORM statement.

    The range of the output procedure must not cause the execution of a MERGE, RELEASE, or SORT statement or a RETURN statement that references any sort or merge file other than the one identified by *sd-name*.

- Sort Scratch File

  The FastSort utility determines the default scratch file size if the file-system file named in the file-control entry does not exist. If the file does exist, SORT uses it. You can use the File Utility Program CREATE or command interpreter CREATE command to create the scratch file on disk. If the sort input file is defined to be on tape, or if the sort receives its records from an input procedure, the default number of records in the scratch file is 50K.

SORT deletes the scratch file after it completes ordering the records unless you specify SAVE-SCRATCH in the routine COBOL85^SET^SORT^PARAM^VALUE or COBOL_SET_SORT_PARAM_VALUE_.

Example 10-62 Typical Use of Input and Output Procedures

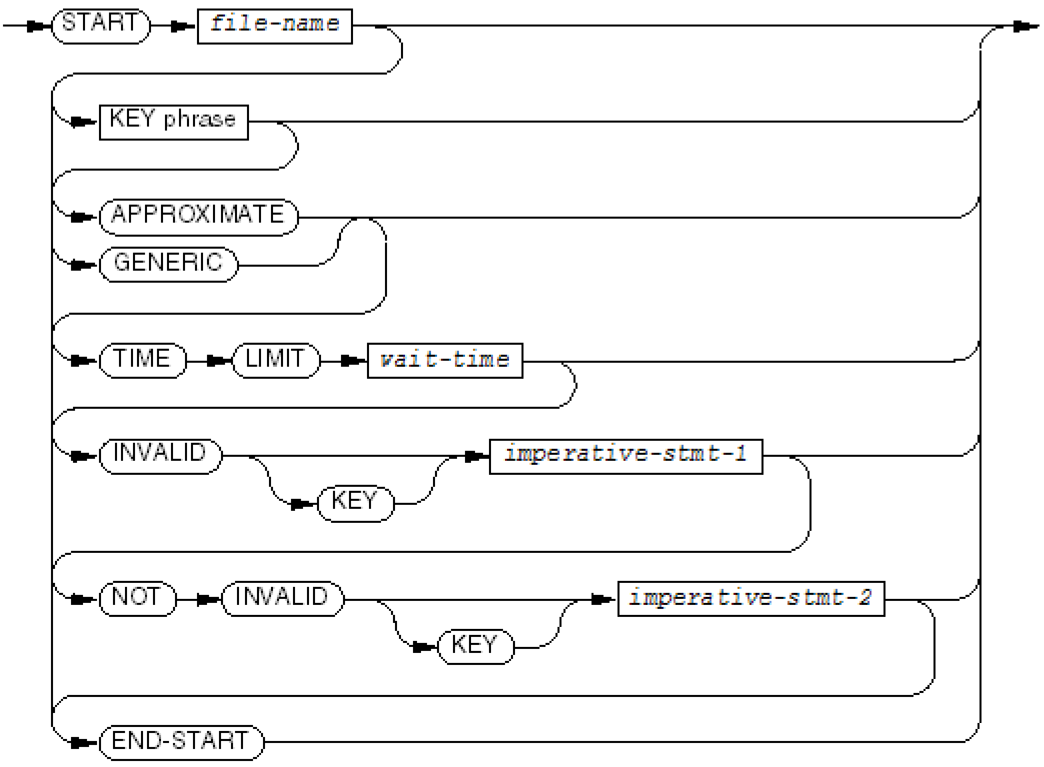
```
FILE-CONTROL.
   SELECT EMPLOYEE-MASTER ASSIGN TO "EMPMST"
          ORGANIZATION IS SEQUENTIAL
          ACCESS MODE IS SEQUENTIAL.
   SELECT SORT-WORK ASSIGN TO "SORTWORK".

DATA DIVISION.
FD EMPLOYEE-MASTER
    LABEL RECORDS ARE OMITTED
    RECORD CONTAINS 52 CHARACTERS.
01  EMPLOYEE-DETAIL.
    05  EMPLOYEE-NUMBER   PIC X(05).
    05  EMPLOYEE-NAME     PIC X(20).
    05  DEPT              PIC X(03).
    05  JOB-CLASS         PIC X(05).
    05  HOURLY-RATE       PIC 9(3)V99.
    05  DEDUCTIONS        PIC 9(3)V99.
    05  ANNUAL-SALARY     PIC 9(7)V99.

SD SORT-WORK
    RECORD CONTAINS 52 CHARACTERS.
01  SORT-RECORD.
    05  EMPLOYEE-NUMBER   PIC X(05).
    05  EMPLOYEE-NAME     PIC X(20).
    05  DEPT              PIC X(03).
    05  JOB-CLASS         PIC X(05).
    05  HOURLY-RATE       PIC 9(3)V99.
    05  DEDUCTIONS        PIC 9(3)V99.
    05  ANNUAL-SALARY     PIC 9(7)V99.
        ...

PROCEDURE DIVISION.
        ...
   IF NO-ERROR
       SORT SORT-WORK ON ASCENDING KEY EMPLOYEE-NAME
           OF SORT-RECORD
              INPUT PROCEDURE IS SORTIN-PROC
              OUTPUT PROCEDURE IS SORTOUT-PROC
        ...

SORTIN-PROC SECTION.
   READ EMPLOYEE-MASTER NEXT RECORD
      AT END GO TO SORTIN-EXIT
   END-READ
   IF WS-DEPT EQUAL DEPT OF EMPLOYEE-DETAIL
   OR WS-DEPT EQUAL SPACES
          RELEASE SORT-RECORD FROM EMPLOYEE-DETAIL
          GO TO SORTIN-PROC
   END-IF.

SORTIN-EXIT.
EXIT.

SORTOUT-PROC SECTION.
   RETURN SORT-WORK
      AT END GO TO SORTOUT-EXIT
   END-RETURN
   MOVE CORRESPONDING SORT-RECORD TO LIST-RECORD
   WRITE LIST-RECORD
   GO TO SORTOUT-PROC.
SORTOUT-EXIT.
   EXIT.
SORT-END SECTION.
   EXIT.
```

START

START positions a file, in sequential or dynamic access mode, for subsequent read operations. START cannot reference a file that is open for HP COBOL Fast I-O. If the START statement executes successfully, it establishes the key of reference and the initial record position needed for subsequent sequential record retrievals.

HP COBOL includes two extensions that are of particular interest if you are writing a server program:

Phrase	Description
GENERIC	Lets you position a file at the first record in a subset of a file that consists of all records that satisfy a certain (EQUALS) key relation. After such a START, you can execute sequential READ statements (READ NEXT, for dynamic-access mode) to read all records of the subset. When, eventually, a record does not satisfy the key relation, the run-time routine simulates an at-end condition.
POSITION	Lets you position a file at a specific point in a set of duplicate values of an alternate key (as specified in the KEY phrase). The `position-key` must be a unique value: a prime or relative key or a unique alternate key. By using this feature, a context-free server can return a series of groups of records to its requester. With each group, it returns the value of the unique key last used. The requester can then send both the alternate key value for the KEY phrase and the unique value for the POSITION phrase to the server, to specify where to begin the next group.

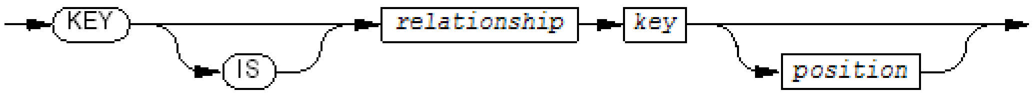


VST219.vsd

`file-name`

is the file description name of the file to position. It must have an access mode of sequential or dynamic and be opened for INPUT or I-O but not for HP COBOL Fast I-O.

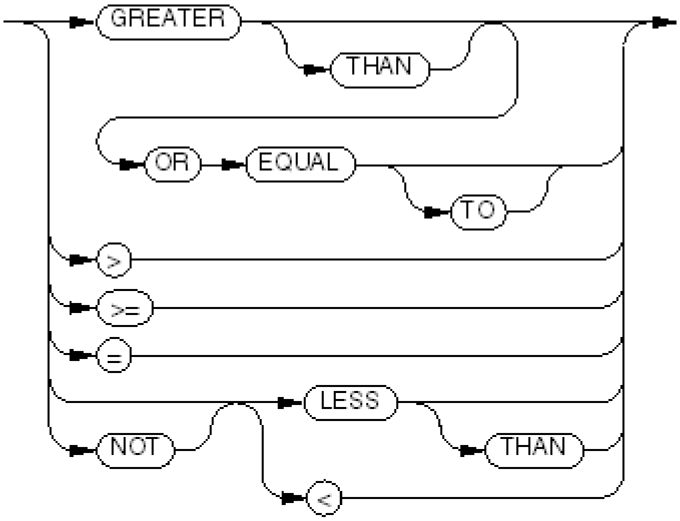
KEY phrase



VST220.vsd

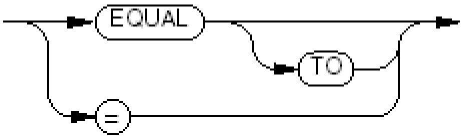
must appear if the file has sequential organization or if the file has relative organization but is not described with a RELATIVE KEY clause. If KEY phrase is omitted, the file's prime key is used.

`relationship`



VST221.vsd

If `position` is present, `relationship` is limited to:



VST223.vsd

`key`

determines, with `relationship`, where the file position indicator is to be set. The file position indicator is to be set such that the next record to be read from `file-name` contains a key value having the specified `relationship` to the current value of `key`.

The value of `key` is restricted by the presence or absence of `position`. If `position` is present, `key` must be either:

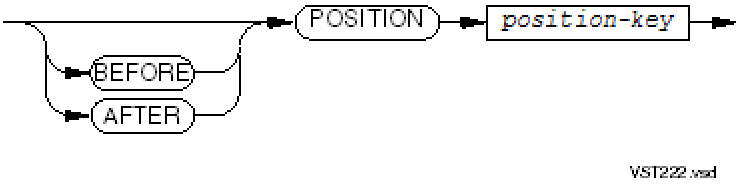
- A prime record key or leftmost subordinate of a prime key for an indexed or queue file
- An alternate key or leftmost subordinate of an alternate key for an indexed, queue, or relative file

If `position` is absent, `key` can be one of:

- The relative key for a relative file
- The prime record key or leftmost subordinate of the prime key for an indexed or queue file
- An alternate key or leftmost subordinate of an alternate key for a file of any organization

A reference modifier can be applied to `key`, but `leftmost-character-position` must be the constant 1. For reference modifier syntax, see Reference Modifier Syntax (page 102).

position



specifies that a combination of alternate key or prime key (specified by *key* in the KEY phrase) and *position-key* (specified either by the file prime key or by a unique alternate key) is to be used to position the file. This phrase is not permitted for files having sequential organization, or files having the INSERTIONORDER attribute.

BEFORE

specifies that the file position indicator is set to the record before the position defined by the values of *key* and *position-key*

.

NOTE: Only use BEFORE when the file will subsequently be read in reverse. If the file is not read in reverse, reading is slightly less efficient.

AFTER

specifies that the file-position indicator is set to the record after the position defined by the values of *key* and *position-key*.

If neither BEFORE nor AFTER is present, the file-position indicator is set to the record at the position defined by the values of *key* and *position-key*.

position-key

is either the file prime key or a unique alternate key. If *position-key* is the file prime key (the prime record key of an indexed or queue file or the relative key of a relative file), then *key* must refer to one of:

- The prime key
- A leftmost subordinate of the prime key
- An alternate key
- A leftmost subordinate of an alternate key

If *position-key* is a unique alternate key (a key described without the DUPLICATES clause), then *key* must refer to the same alternate key (or to a leftmost subordinate of that alternate key).

If *position-key* is a leftmost subordinate of a prime or alternate key, see GENERIC.

APPROXIMATE

is the default positioning mode. APPROXIMATE means that reading begins at the first record that satisfies the relationship that the KEY phrase specifies. Subsequent READ statements read the records in file key order or physical order, depending on the file's organization, until the end of the file is reached.

GENERIC

is an alternative positioning mode. If GENERIC is used, then *relationship* must be EQUAL, EQUAL TO, or =. Reading begins at the first record that satisfies the relationship and continues until the relationship is not satisfied (logical end of file).

GENERIC is usually used with a partial key; that is, a subordinate item of the key defined for the file. Use of the leftmost subordinate governs the action of the GENERIC clause, but the complete alternate key is used for positioning (as explained in START Statement With the POSITION Phrase).

wait-time

is the time interval, in seconds, in which the operation must complete. *wait-time* can be a literal or the name of a data item. In either case, it must have a value described with at most seven digits preceding any decimal point position. Any fractional portion is truncated to two decimal places.

If *file-name* was not opened with a TIME LIMITS phrase, including *wait-time* in the START statement causes a run-time error.

imperative-stmt-1

is an imperative statement to be executed when an invalid-key condition is encountered by the START operation. It is required if no USE statement is applicable for the file. If both a USE statement and an INVALID KEY phrase are present, only the INVALID KEY phrase is used.

imperative-stmt-2

is an imperative statement to be executed when no invalid-key condition is detected in the START operation.

END-START

ends the scope of the START statement, making the START statement a delimited-scope statement. If the START statement does not end with an END-START phrase, the presence of the INVALID KEY or the NOT INVALID KEY phrase makes the START statement a conditional statement, which ends at the next period separator.

Usage Considerations:

- Action of the START Statement

  The START statement proceeds in this manner:

  1. It establishes the specified key as the key of reference and searches the file for the first logical record whose value for that key satisfies the specified relation with respect to the comparison data item values.

     When the specified key is the prime record key or an alternate record key, the comparison data item is handled as if it were an alphanumeric data item (regardless of its actual description) and the normal nonnumeric comparison rules apply, except that the standard collating sequence for the first 128 characters is always used.

     NOTE: The program collating sequence does not apply to file key comparisons.

     When the size of the comparison data item in character positions is less than that of the file key, the comparison proceeds as if the record's key value were truncated on the right so as to reduce its size to the equivalent number of characters.

2. When the file does not contain a qualifying record, the invalid-key condition exists and the start operation terminates with the I-O status code “23.” When the search operation succeeds, the file position indicator is set to the value of the key of reference for the record found.

   The execution of the START statement does not alter the contents of the record area associated with `file-name` or the content of the depending item specified in the DEPENDING phrase of the RECORD clause that describes `file-name`.

NOTE: Use care in specifying the starting record. When you are starting a file with APPROXIMATE positioning (which is the default), the value you use for `wait-time` must take into account that a START can take somewhat longer than expected. This can occur when the START leads to a nonexistent record, because the file system searches through the file looking for the next defined record before reporting the absence of the record sought.

Suppose that your file contains alternate keys that can have duplicates, and one such alternate key is DEPT-NUM. If you want to start at the department whose number follows 5440, it is more efficient to use a START of the form

```
MOVE "5441" TO DEPT-NUM.
START MYFILE KEY IS NOT LESS THAN DEPT-NUM.
```

than it is to use

```
MOVE "5440" TO DEPT-NUM.
START MYFILE KEY IS GREATER THAN DEPT-NUM.
```

because the latter form causes the file system to sequentially read each record having the key value “5440” until it finds a record whose key exceeds 5440. The time saved depends upon the number of duplicates in the file.

See the *Guardian Programmer’s Guide* for more information on the action of READ (which is called by the START logic).

- File-Status Data Item

  If `file-name` has an associated file-status data item, execution of the START statement always assigns an appropriate I-O status code to it. The value “00” reports an unconditionally successful start operation. The value “97” reports a successful start operation that validated a position by reading a locked record.

  The values of the I-O status data item for unsuccessful start operations are:

I-O Status Code	Unsuccessful Start Operation
“23”	The file position indicator indicates that an optional input file is not present, the invalid-key condition exists, and the start operation terminates.
“30”	Either the time specified in the TIME LIMIT phrase elapsed before the start operation completed (indicated by the special register GUARDIAN-ERR having the value “40”), or the start operation failed for some non-COBOL reason. In either case, the value of the file position indicator becomes undefined.
“47”	The file identified by `file-name` is not open for INPUT or for I-O. The start operation terminates.
“90”	The program attempted to use the TIME LIMIT phrase when the associated OPEN statement does not specify TIME LIMITS.
“91”	The program attempted to use the POSITION phrase when the file has insertion-ordered alternate keys. The start operation fails.

- Invalid-Key Condition

  If the START statement contains the INVALID KEY phrase, control passes to the imperative statement in that phrase, and no USE procedure is executed. If the START statement does not contain an INVALID KEY phrase, but an applicable USE procedure exists, that procedure is executed.

- Key Data Item

  The data item specified by *`key`* is the comparison data item for the start operation. When this data item is a file key data item, that key is the specified key for the start operation; otherwise, the file key data item of which *`key`* is a leftmost subordinate specifies the file key for the start operation.

  The *`key`* data item can be reference-modified. It can be a level 66 (RENAMES) item.

  `key` can also be an item whose description contains a REDEFINES clause or subordinate to an item whose description contains a REDEFINES clause. The redefining data item and, in the latter case, the data item subordinate to the redefining item must contain the leftmost character positions of the redefined data item.

  When the KEY phrase does not appear, the start operation behaves as if the START statement includes the phrase KEY EQUAL TO *`the-key`*, in which *`the-key`* is either the relative key data item (for files with relative organization) or the prime record key data item (for files with indexed organization).

- TIME LIMIT Phrase

  If the TIME LIMIT phrase appears, the time limit operand is evaluated and rounded, if necessary, to include at most two fractional digits. When the result is negative, the TIME LIMIT phrase does not apply, and the operation is not subject to a time limit; otherwise, the result specifies the time interval, in seconds, within which the start operation must complete. The start operation fails if an acceptable record cannot be located within the time interval specified.

  If the TIME LIMIT phrase is specified with a nonnegative value, and the file is not opened with time limits enabled, the program terminates with an I-O status code “90,” and a message, “File is not opened for timed I-O,” is delivered to the process’s home terminal.

- Effect of Declaratives on Termination

  If there is no declarative procedure applicable to the file when the operation is terminated, the program terminates, and an error message is reported to the process’s home terminal.

  If the applicable declarative procedure is present (but no INVALID KEY phrase is present) and the time interval expires, the declarative procedure is performed. Then program execution continues with the imperative statement in the NOT INVALID KEY phrase, if one is present, or otherwise with the statement following the one terminated.

- Result of Successful Start Operation

  A successful start operation identifies a subset of the file’s records that can be retrieved by subsequent sequential READ statements. The initial record in the subset is the one located by the search.

  When the APPROXIMATE phrase appears, the subset includes all records that follow the initial record according to the key of reference.

  When the GENERIC phrase appears, the subset includes only those records whose value for the key of reference satisfies the specified relation with respect to the comparison data-item value. During the execution of subsequent READ NEXT statements, the at-end condition occurs when the run-time routines detect the end of the subset.

  When neither the APPROXIMATE nor the GENERIC phrase is present, the start operation behaves as if the APPROXIMATE phrase is present.

When the execution of the START statement is successful, the key of reference established for the start operation is also used for any subsequently executed READ NEXT statements for the file, until the execution of some statement explicitly establishes a different key of reference.

Whenever the start operation is successful, if a NOT INVALID KEY phrase is specified, control passes to the imperative statement in that phrase. If that imperative statement does not transfer control elsewhere using a GO TO statement, control then passes to the end of the START statement.

- Result of Unsuccessful Start Operation

  When the execution of a START statement is unsuccessful for any reason, the file position indicator is set to indicate that no valid next record has been established, and the key of reference for the file is undefined.

- START Statement Without the POSITION Phrase

  Execution of the START statement establishes the specified key as the key of reference for the file:

  — For sequential files, an alternate key

  — For relative files, the relative key item or an alternate key

  — For indexed files, a prime key or alternate key

  — For queue files, a prime key

  The start operation searches the file for the first record whose value for the key of reference satisfies the specified relation with respect to the comparison data item.

  When the specified key is the prime record key or an alternate record key, the comparison data item is handled as if it were an alphanumeric data item (regardless of its actual description) and the normal nonnumeric comparison rules (using the ASCII collating sequence) apply. When the size of the comparison data item in character positions is less than that of the file key, the comparison proceeds as if the record's key value were truncated on the right to reduce its size to the equivalent number of characters.

  When the file does not contain a qualifying record, the invalid-key condition exists, and the execution of the START statement is unsuccessful.

  When the search succeeds, the file position indicator for the file is set to point to the record found.

- START Statement With the POSITION Phrase

  A START statement that includes a POSITION phrase also serves to establish a key of reference and to establish a value for the file position indicator. If more than one record has the same alternate key, the POSITION phrase determines which record is read. If the program is reading according to a partial alternate key that has duplicate values in the file, but the complete alternate key values are unique, the POSITION phrase has no effect (even if the alternate key is specified with the DUPLICATES phrase).

  Suppose a requester asks for a set of records from a member of a context-free server class: all the records that have a certain alternate key value, or partial value. If the message buffer cannot hold all such records, the server can return the first group of records plus the value of the unique key with the reply. This value is that of the prime record key, the relative key, or the entire alternate key declared as having no duplicates. Then the requester can ask for another group, supplying the value of both the partial alternate key and the unique key, and the server can resume with the last record it had sent or (by using the AFTER phrase) the record beyond that. The server can remain context-free, because the information necessary to do the positioning and to establish the key of reference and the comparison length (for partial keys) accompanies the request.

- GENERIC Positioning Mode and the POSITION Phrase

  The choice of APPROXIMATE or GENERIC positioning mode, coupled with the KEY clause, govern where the next READ statement starts reading and what constitutes the end of the file (or the beginning of the file, if the file is being read in reverse). A common use of GENERIC involves reading all records that have a common value in the leftmost subordinate of a given alternate key; for example, all records that have a 10-character alternate key, where the first five characters of that key have the value "A5R32." Under the GENERIC mode, the program reads records until it encounters one that has something different in the first five characters for that alternate key, then the program receives an end-of-file condition instead of a new record value.

  Only if the file contains multiple records that have the same value for the complete alternate key (not just the leftmost subordinate) can the POSITION phrase be used as outlined in the general discussion earlier to process groups of records.

- BEFORE and AFTER Phrases

  If neither the BEFORE phrase nor the AFTER phrase is included, the execution of the START statement sets the file position indicator to point to the record uniquely identified by the two key values.

  If the BEFORE phrase is included, the execution of the START statement sets the file position indicator to point to the record preceding the one that the two key values uniquely identify.

  If the AFTER phrase is included, the execution of the START statement sets the file position indicator to point to the record following the one that the two key values uniquely identify.

- INSERTIONORDER Attribute Incompatible with POSITION Phrase

  The POSITION phrase can be used only if the file on which the START statement is operating has the NO INSERTIONORDER attribute; that is, records with duplicate values of an alternate key are delivered in principal key order.

- Next READ After a START with POSITION Might Fail

  When you use the POSITION phrase, there is no guarantee that the record you specify actually exists. In this aspect, the start-with-position operation differs from the ordinary COBOL start operation. The positioning operation merely sets the file position indicator to the file location where the record can be found if it exists; therefore, the invalid-key condition does not occur when the record does not exist. Instead, the first subsequent sequential READ statement encounters the at-end condition (if the physical end of the file has been reached, or if the logical end of file associated with the generic positioning mode has been reached).

- Determining Key Value of Last Record of an Indexed or Queue File

To determine the key value of the last record of an indexed or queue file, which is especially important when using the READ REVERSED statement, move HIGH-VALUES to the key and then use the START statement with the `relationship` LESS THAN. START LESS THAN positions the indexed file so that a READ NEXT or READ REVERSED statement accesses the last record in the file.

Table 10-11 Using the POSITION and KEY Phrases

Position-Key	Key	Length Used and Restriction	Effect
Relative key (ORGANIZATION RELATIVE)	Alternate or alternate(1:*n*)	length (entire alternate) + 4 < 254	Full value of alternate AND of relative used
	Relative key	4 (which is the length of the relative key)	Relative key used alone
Prime key (ORGANIZATION INDEXED)	Alternate or alternate (1:*n*)	length (entire alternate) + length (prime) < 254	Full value of alternate AND of prime used
	Prime key	length (prime)	Prime key used alone
Alternate key (ORGANIZATION RELATIVE or INDEXED)	Same alternate or alternate (1:*n*)	Full alternate must be unique (no DUPLICATES) < 254	Full value of alternate used alone

Example 10-63 START Statement for Indexed File

```
SELECT RECEIVABLES-MASTER ASSIGN TO "RECMAST"
       ORGANIZATION IS INDEXED
       ACCESS MODE IS DYNAMIC
       RECORD KEY IS INVOICE-NUMBER
              ALTERNATE RECORD KEY IS COMPANY-NAME
                   WITH DUPLICATES.
         ...
PROCEDURE DIVISION.
         ...
   OPEN I-O RECEIVABLES-MASTER
         ...
   MOVE LOW-VALUES TO COMPANY-NAME
   START RECEIVABLES-MASTER KEY NOT LESS THAN COMPANY-NAME
         INVALID KEY
           DISPLAY "ERROR STARTING READ FOR REPORT"
           GO TO REPORT-EXIT
   END-START.
GET-NEXT-RECORD.
   READ RECEIVABLES-MASTER NEXT RECORD
        AT END PERFORM...
```

Example 10-64 reads all records for employees whose last names start with `G`.

Example 10-64 START Statement With GENERIC Phrase for Sequential File

```
  SELECT INPUT-FILE ASSIGN TO "INFILE"
     ORGANIZATION IS SEQUENTIAL
     ACCESS MODE IS SEQUENTIAL
     ALTERNATE RECORD KEY IS EMP-NAME
        WITH DUPLICATES.
          ...
FD INPUT-FILE
    LABEL RECORDS ARE OMITTED
    RECORD CONTAINS 95 CHARACTERS
    DATA RECORD IS PERSONNEL-DATA.
01  PERSONNEL-DATA.
    05  DEPT-NO             PIC 9(5).
    05  EMPLOYEE-NO         PIC 9(7).
    05  EMP-NAME.
        10  LAST-NAME.
            15  FIRST-LETTER    PIC X.
            15  FILLER          PIC X(14).
        10  FIRST-NAME          PIC X(9).
    ...
```

```
PROCEDURE DIVISION.
       ...
  OPEN I-O INPUT-FILE
       ...
  MOVE "G" TO FIRST-LETTER
  START INPUT-FILE KEY = FIRST-LETTER GENERIC
     INVALID KEY GO TO START-ERROR-ROUTINE
  END-START
  READ INPUT-FILE
     AT END
```

Example 10-65, fragments of a server program, shows the use of a START statement with a POSITION phrase. The query the server gets through $RECEIVE includes a department number and a unique employee number. The server is to return an array containing information about all employees in the specified department.

At the first call, the requester sends the chosen department number but an employee number of zero. If there are more than ten employees in a department, the server reports only the first ten it found. The requester can then send another query, but this time it includes the employee number of the last array entry of the previous reply. By using this technique, the server remains context-free, able to serve another requester.

When the server executes the START statement, the key of reference used with the GENERIC phrase is still the department number, but the presence of a unique record key value of EMP-NO in the AFTER POSITION phrase says to resume reading along the DEPT-NO path after the record for the specified employee.

Example 10-65 START Statement With POSITION Phrase

```
    SELECT EMP-FILE ASSIGN TO "EMPL1093"
       ORGANIZATION IS INDEXED
       ACCESS MODE IS DYNAMIC
       RECORD KEY IS EMP-NO
       ALTERNATE RECORD KEY IS DEPT-NO WITH DUPLICATES
       ALTERNATE RECORD KEY IS EMP-NAME WITH DUPLICATES
       FILE STATUS IS EMP-STATUS.
    ...
DATA DIVISION.
  FILE SECTION.
     ...
  FD EMP-FILE.
  01  EMP-DATA.
      05 EMP-NO    PIC 9(6).
      05 EMP-NAME  PIC X(45).
      05 EMP-DEPT  PIC 9(7).
      ...
PROCEDURE DIVISION.
       ...
  OPEN INPUT  EMP-FILE
       ...
* Read $RECEIVE to get the query
    READ REC-IN ...
* If this is an initial request, do a nonposition start
    IF DEPT-NO OF EMP-LIST-REQUEST = ZEROS
       START EMP-FILE KEY = DEPT-NO OF EMP-DATA
             GENERIC
* Otherwise, resume after record last reply array ended with
    ELSE
        START EMP-FILE KEY = DEPT-NO OF EMP-DATA
            AFTER POSITION EMP-NO
            GENERIC
     END-IF
 * Zero the counter
 * Perform (with test after) until EOF or 10 employees found:
 *   Read EMP-FILE NEXT record
 *   If EOF, return array to requester with signal for EOF
 *   else add 1 to the counter
 *        copy info to the array
 *   end-if
```

In Example 10-65, suppose that department 1572 has 12 employees:

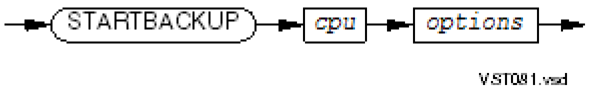
```
000131Smith                Jan                   0001572
001552Nguyen               Tracy                 0001572
001744Dietrich             Pat                   0001572
001745Wellhausen           Robin                 0001572
001746Thomas               Kim                   0001572
001991Chew                 Meredith              0001572
004451O'Hara               Flemming              0001572
005433Logan                Shannon               0001572
006112McClure              Beck                  0001572
009733Kinoshita            Lynn                  0001572
012255Bostrup              Stacy                 0001572
013146Tilden               Tex                   0001572
```

If the requester sends a request with an EMP-NO value of zero for department 1572, the server returns the records for the first 10 employees (131 through 9733) in response. If the requester sends EMP-NO as 9733 in the second request for department 1572, the START AFTER positions the file to resume reading after that number, so the server returns the last 2 records in the file.

STARTBACKUP

NOTE: Do not use this directive in the OSS environment.

STARTBACKUP defines options for handling process pairs and starting the backup process.



cpu

is the processor module where the backup process is to run. The *cpu* parameter is either an integer numeric literal or an identifier that designates an integer numeric data item. The value associated with *cpu* must not be less than 0 nor greater than 15 and cannot designate the same processor in which the process executing the STARTBACKUP statement is running.

options

is either an integer numeric literal or an identifier that designates an integer numeric data item. The value must be 0, 1, 2, or 3. The significance of *options* is:

0	The fault-tolerant facility is to read and process system messages, and the primary process is to terminate abnormally if a trap condition occurs. If this option is specified and the primary process is stopped (because of a command interpreter STOP command or by a third process), the backup process stops. The non-CRE environment run-time processes relevant system messages, such as CPU-DOWN, as part of this activity. The occurrence of a trap condition in the primary process causes it to fail (see the explanation of ARMTRAP in the *Guardian Programmer's Guide*.) Additionally, if the backup process fails, the primary process automatically starts a new backup process when it is possible to do so.
1	This option is the same as option 0 except that if the primary process stops, the backup process takes over processing of the application.

2	This option is the same as option 1 except that if the primary process encounters a trap condition, it enters the DEBUG procedure instead of being terminated abnormally. (For an explanation of traps and the DEBUG procedure, see the *Guardian Programmer's Guide*.)
3	The primary process, rather than the fault-tolerant facility, reads the $RECEIVE file and takes appropriate action for system messages. In addition, if the primary process encounters a trap condition, it enters the DEBUG procedure. When this option is in force and the backup process fails, the primary process must re-execute the STARTBACKUP statement to reestablish the backup. (This is considered an advanced option because it requires direct calls to the operating system. If you use this option, see the information on interprocess communication and checkpointing in the *Guardian Programmer's Guide*.)

Usage Considerations:

- When STARTBACKUP Statement Has No Effect

  In the OSS environment, or in the Guardian environment when PARAM NONSTOP OFF is active, the STARTBACKUP statement has no effect.

- NONSTOP Compiler Directive

  The STARTBACKUP statement can appear in a source program only when the compiler finds the NONSTOP directive before it finds an Identification Division header (see NONSTOP (page 566)).

- Backup Process and Takeover Points

  When the process is qualified to execute as a process pair, successful execution of the STARTBACKUP statement establishes and initializes a backup process. The STARTBACKUP statement does not itself establish a valid takeover point; this is a function of the CHECKPOINT, OPEN, or CLOSE statement.

  Code a CHECKPOINT statement after each STARTBACKUP statement so that the new backup process will have all the data it needs for a possible takeover.

- Creation and Maintenance of a Backup Process

  Execution of the STARTBACKUP statement assigns a value to the special register PROGRAM-STATUS, reflecting the success or failure of the creation of the backup process. PROGRAM-STATUS is a record with two fields, PROGRAM-STATUS-1 and PROGRAM-STATUS-2:

  ```
  01  PROGRAM-STATUS.
      02  PROGRAM-STATUS-1    PIC X.
      02  PROGRAM-STATUS-2    PIC XXX.
  ```

  Automatic backup process maintenance (for levels below 3) does not take effect until after the successful execution of a STARTBACKUP statement.

  For PROGRAM-STATUS values, see Table 33-1: Values for PROGRAM-STATUS When STARTBACKUP Has Option 0, 1, or 2 (page 969) and Table 33-2: Values for PROGRAM-STATUS When STARTBACKUP Has Option 3 (page 970).

- Unnamed Processes

  If the process to which the STARTBACKUP statement applies was not initiated as a named process (that is, with the NAME option of the RUN command), the STARTBACKUP statement returns a specific value in the special register PROGRAM-STATUS. In the CRE, the value is 4922; in the non-CRE environment, it is 4013.

- STARTBACKUP Options

  For most applications, use option 0 or 1. For either of these options, a STARTBACKUP statement must execute once during the initialization phase of the program (check the outcome of the STARTBACKUP execution by examining the PROGRAM-STATUS variable). At each successive checkpoint, the fault-tolerant facility checks the state of the backup process and processor. If the backup process is inoperable, the fault-tolerant facility re-creates the backup process. If the backup's processor is operable, the fault-tolerant facility re-creates the backup process immediately. If the backup's processor is inoperable, the fault-tolerant facility re-creates the backup process when backup's processor becomes operable.

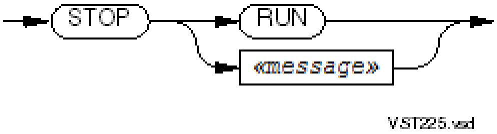
- What Fault-Tolerant Facility Checkpoints

  The fault-tolerant facility checkpoints the global-data portion of the globals-heap segment, the main stack, and all open files when the STARTBACKUP statement is executed; however, your program must contain a CHECKPOINT statement to verify the creation of a backup process and to establish a valid takeover point in the program code (rather than in the run-time library).

STOP

STOP halts the execution of a run unit, either permanently or only to display a message.

NOTE: The 1985 COBOL standard classifies the latter use as **obsolete**, so you are advised not to use it.



RUN

halts the execution of a run unit and transfers control to the operating system, closing any open files.

message

NOTE: The 1985 COBOL standard classifies *message* as **obsolete**, so you are advised not to use it. Instead, use a DISPLAY statement followed by an ACCEPT statement.

is a nonnumeric literal string. Execution stops temporarily and *message* is sent to the home terminal. When any response or carriage return is entered, execution resumes.

The routine COBOL85^COMPLETION or COBOL_COMPLETION_ provides the application program a means of reporting a completion code to the operating environment and terminating execution.

This statement suspends processing until the home terminal sends a carriage return:

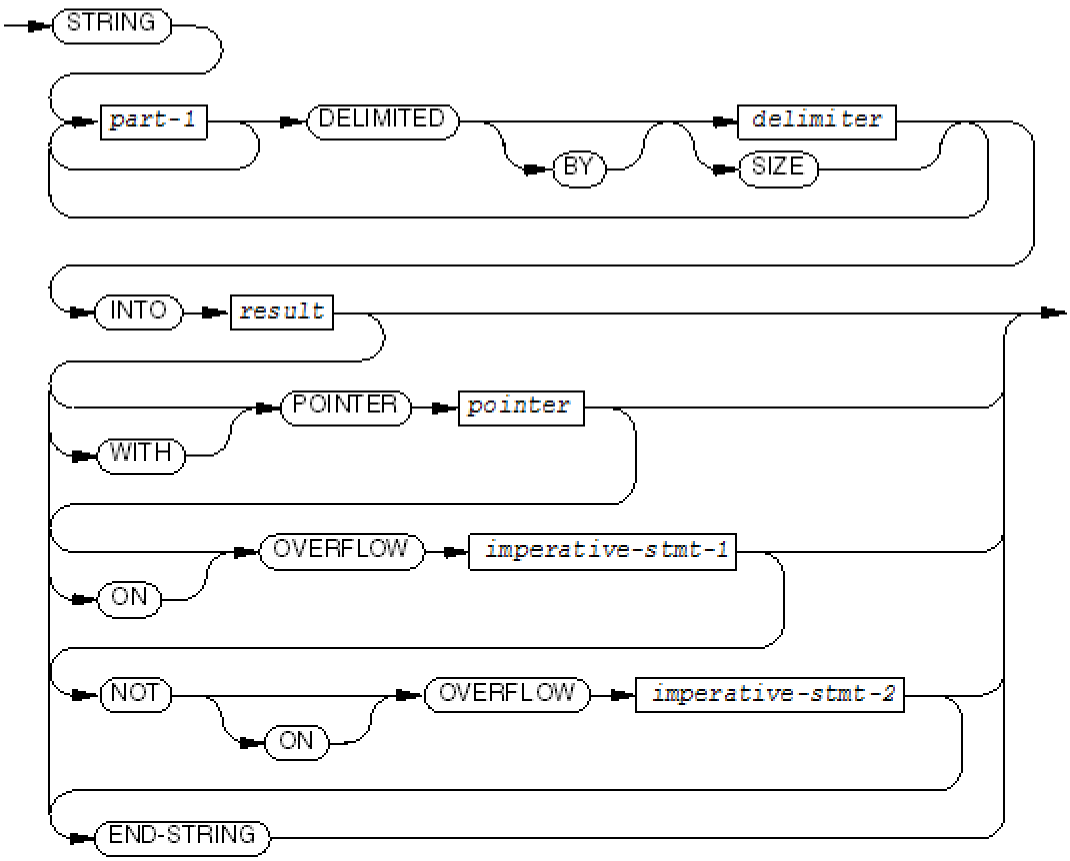
```
STOP "Press Return to show you are out there.".
```

This sequence of statements is equivalent:

```
DISPLAY "Press Return to show you are out there.".
ACCEPT USER-REPLY.
```

STRING

STRING combines some or all the characters from two or more data items into one other data item.



VST226.vsd

part-1

is an alphanumeric literal or identifier of a DISPLAY data item. The concatenation of their values is stored in the data item named by *result*. If a data item is numeric, it must be described as an integer without *P* in its PICTURE character-string. When a figurative constant is used, it represents a one-character nonnumeric literal.

delimiter

is an alphanumeric literal or the identifier of a DISPLAY data item. It specifies the portion of *part-1* that is moved. It also specifies that all characters up to but not including the value of *delimiter* are moved.

If the literal is a figurative constant, it represents the equivalent one-character nonnumeric literal.

If the data item is numeric, it must be described as an integer whose PICTURE character-string does not include *P*.

SIZE

specifies that all the characters in *part-1* are moved.

result

is an identifier for an alphanumeric data item where the characters chosen from the *part-1* s are stored. If result is an elementary item, its PICTURE character-string must not contain editing symbols or a JUSTIFIED clause. Reference modification is not permitted.

Storing begins at the position identified by the initial value of *pointer*. When *pointer* is not used, the leftmost position is the beginning place. An internal index keeps track of the next available position in *result*. *result* cannot reference a special register.

pointer

is the identifier of an integer data item whose PICTURE character-string does not include *P*. The initial value of *pointer* (which you must set) is the position in *result* to which the first character of data will be moved. The value of the first position is 1. At the conclusion of STRING, *`pointer`*'s value is its initial value plus the total number of characters moved. *`pointer`* cannot reference a special register.

`imperative-stmt-1`

is an imperative statement to be executed when the internal index points past the end of *`result`*. OVERFLOW also occurs if the initial value of *`pointer`* is less than 1 or greater than the length of *`result`*. If no OVERFLOW statement is given, control passes to the next statement after STRING.

`imperative-stmt-2`

is an imperative statement to be executed when the internal index does not point past the end of *`result`*.

END-STRING

ends the scope of the STRING statement, causing the STRING to be a delimited-scope statement. If the STRING statement does not end with an END-STRING phrase, the presence of the OVERFLOW or the NOT OVERFLOW phrase causes the STRING statement to be a conditional statement, which ends at the next period separator.

Usage Considerations:

- Operand Identification

  For each identifier, the process of operand identification occurs only once, at the beginning of the execution of the STRING statement.

- Initialization

  The execution of the STRING statement begins in this manner:

  1. The data item specified by *`result`* is established as the receiving item.
  2. When the POINTER phrase appears, the data item specified by *`pointer`* is established as the pointer variable.

     If the initial value of *`pointer`* is less than 1 or greater than the size of the receiving item, then the overflow condition exists immediately and no string operation occurs; otherwise, the initial value of *`pointer`* determines the relative character position within result at which the first sending value is assigned. The leftmost character position is designated to be 1.

     When the POINTER phrase does not appear, the string operation presumes an initial relative character position of 1.

- String Operation Cycle

  The string operation transfers data from each *`part-1`*, in the same order as they are specified in the STRING statement. The steps involved in the transfer of a sending value are:

  — Determining the transfer string

  If the applicable DELIMITED phrase specifies *`delimiter`*, then the value of the current *`part-1`* is examined character by character from left to right until either of these is found: the end of the value, or a sequence of contiguous characters that matches the value of *`delimiter`*.

  If a match is found, the transfer string consists of the portion of the sending value preceding the character that matches the first character of the delimiter; otherwise, the transfer string consists of the entire sending value.

If the applicable DELIMITED phrase specifies SIZE, then no delimiter exists and the transfer string always consists of the entire sending value.

— Copying the transfer string to the receiving item

Assignment of the transfer string to the receiving item (*result*) proceeds on a character by character basis. Beginning with the leftmost character, each character of the transfer string is moved into the receiving item character position designated by the pointer value, which is incremented by 1 after each character is moved.

— Overflowing the size of the receiving item

Overflow occurs if not all of the characters of a transfer string can be moved to the receiving item. The overflow condition exists when the pointer value exceeds the size of the receiving item during assignment of any transfer string or after completing assignment of any transfer string except the last one. After assignment of the last transfer string, the pointer value can be one character past the end of the receiving item, and overflow does not occur unless there are leftover characters in the transfer string.

If the overflow condition exists, the string operation terminates immediately. Any remaining characters in the current transfer string are ignored; any sending values not yet processed are ignored.

- State of the Rest of *result*

  After termination of the string operation, only the portion of the data item specified by *result* that was referenced as described earlier is changed. That is, any portions preceding the first character assigned or following the last character assigned retain their previous value.

  When the execution of a STRING statement specifying a POINTER phrase terminates, either normally or due to an overflow condition, the contents of the data item specified by *pointer* contains a value equal to its initial value plus the total number of characters transferred into the data item specified by *result*.

- Overflow Condition

  When an overflow condition exists, execution of the STRING statement terminates at that point. If the OVERFLOW phrase is specified, the imperative statement in that phrase is executed; otherwise control passes directly to the end of the STRING statement.

- Overlapping Operands

  If any part of the storage area referenced by *result* or of that referenced by *pointer* is the same as the storage area referenced by any other identifier appearing in the STRING statement, the execution of the STRING statement will produce unpredictable results.

- National Data Items and National Literals

  If any of the data items *part-1*, *delimiter*, and *result* is a national data item or national literal, then all of them must be national items.

In Example 10-66, STRING builds a single data item from several data items. Also see the examples in UNSTRING.

Example 10-66 STRING Statement

Input:

```
WORKING-STORAGE SECTION.
77  PART-1    PIC X(10).
77  PART-2    PIC X(26).
77  PART-3    PIC X(10).
77  RESULT-1  PIC X(80)  VALUE SPACES.
77  COUNT-1   PIC 99.
PROCEDURE DIVISION.
```

```
A10-START.
   DISPLAY "ENTER PART-1 (MAX 10 NUMERIC CHARACTERS)"
   ACCEPT PART-1
   INSPECT PART-1 REPLACING ALL " " BY "0"
   IF PART-1 NOT NUMERIC DISPLAY "NOT NUMERIC"
      GO TO A10-START
   END-IF
   DISPLAY "ENTER PART-2 (MAX 26 CHARACTERS)"
   ACCEPT PART-2
   DISPLAY "ENTER PART-3 (MAX 10 CHARACTERS)"
   ACCEPT PART-3
   MOVE 1 TO COUNT-1
   STRING PART-1 DELIMITED BY ZERO
          SPACE DELIMITED BY SIZE
          PART-2 DELIMITED BY SPACE
          SPACE DELIMITED BY SIZE
          PART-3 DELIMITED BY SPACE
               INTO RESULT-1 WITH POINTER COUNT-1
   END-STRING
   DISPLAY "PART-1 = " PART-1
   DISPLAY "PART-2 = " PART-2
   DISPLAY "PART-3 = " PART-3
   DISPLAY "COUNT-1 = " COUNT-1
   DISPLAY "RESULT-1 AFTER STRING = " RESULT-1
   STOP RUN.
```

Output:

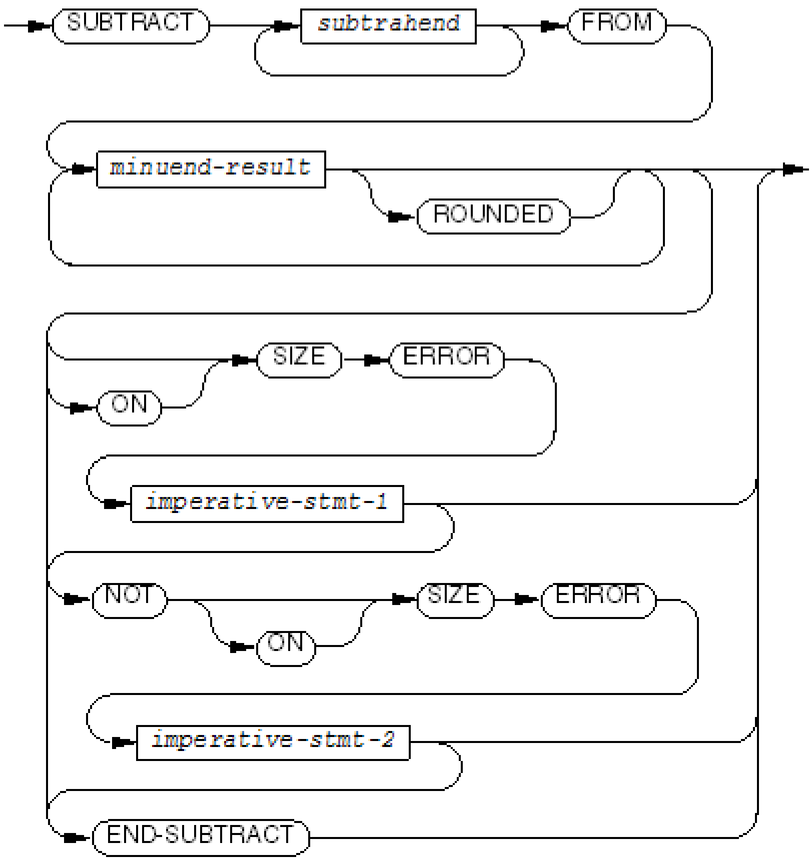
```
>RUN rununit
ENTER PART-1 (MAX 10 NUMERIC CHARACTERS)
?1234 67
ENTER PART-2 (MAX 26 CHARACTERS)
?MINNIE MOUSE
ENTER PART-3 (MAX 10 CHARACTERS)
AAAAA
PART-1 = 1234067000
PART-2 = MINNIE MOUSE
PART-3 = AAAAA
COUNT-1 = 18
RESULT-1 AFTER STRING = 1234 MINNIE AAAAA
```

SUBTRACT

SUBTRACT computes differences between numeric values

SUBTRACT FROM

SUBTRACT FROM subtracts one or more values from the values of one or more identifiers and stores the results in the identifiers.



VST227.vsd

subtrahend

is the identifier of a numeric elementary data item or a numeric literal.

minuend-result

is the identifier of a numeric elementary data item.

ROUNDED

specifies that the result is to be rounded before being stored.

imperative-stmt-1

is an imperative statement to be executed when a size error is detected in the subtraction or in storing the result.

imperative-stmt-2

is an imperative statement to be executed when no size error is detected in the subtraction or in storing the result.

END-SUBTRACT

ends the scope of the SUBTRACT statement, causing the SUBTRACT to be a delimited-scope statement. If the SUBTRACT statement does not end with an END-SUBTRACT phrase, the presence of the SIZE ERROR or the NOT SIZE ERROR phrase causes the SUBTRACT statement to be a conditional statement, which ends at the next period separator.

Usage Considerations:

- Mathematics

  The values of all the *subtrahend* s are totaled. That sum is subtracted from each *result*, and the result of each such subtraction is then stored after rounding, if ROUNDED was specified as the new value of the *minuend-result*. For example, the statement

  ```
  SUBTRACT A B C  FROM  D E
  ```

means store D - (A + B + C) in D and store E - (A + B + C) in E.

- Specifying the Same Data Item for More Than One Result

  If more than one *minuend-result* specifies the same data item, the final value of that item reflects multiple subtractions of the intermediate sum. For example,

  ```
  SUBTRACT A FROM B B
  ```

  means the final value of B is (B - A) - A.

- Operand Identification

  For each *minuend-result*, the process of operand identification occurs just prior to the “subtract and store” operation; therefore, in

  ```
  SUBTRACT A, B, C FROM I, I, X(I)
  ```

  the subscript is not evaluated until the sum of A, B, and C is subtracted twice from I.

- Arithmetic Operations

  See Arithmetic Operations (page 267) for information on data conversion and alignment, intermediate results, multiple results (and subscript evaluation), and incompatible data.

- Precision

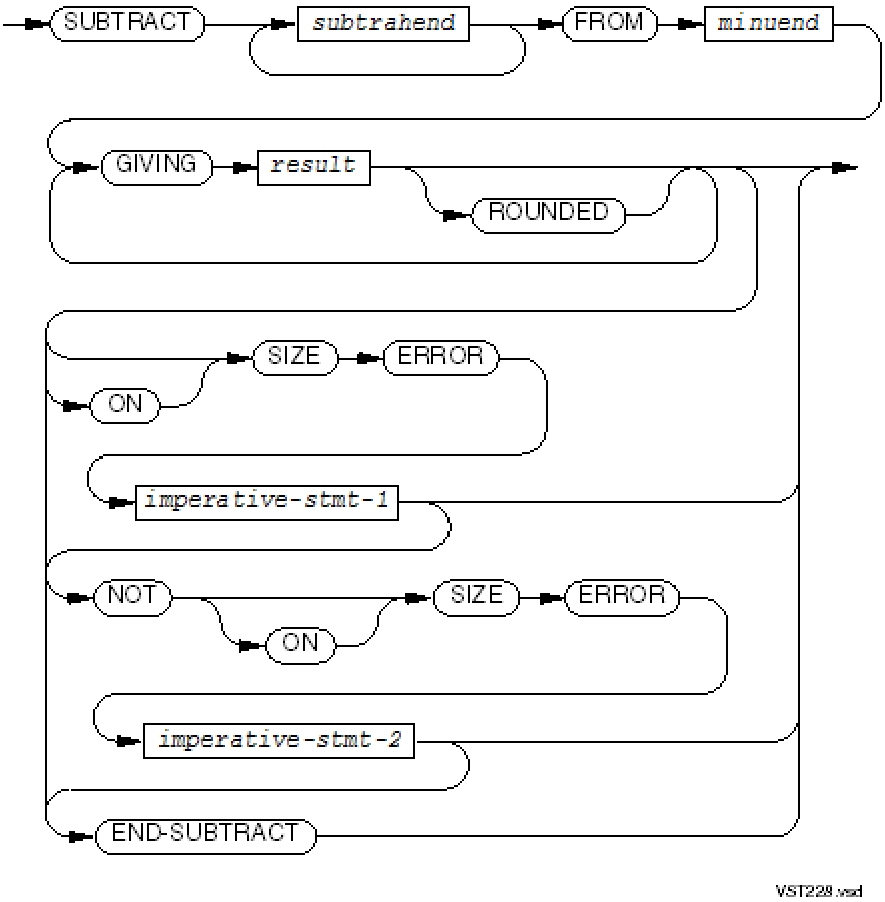
  For any series of items involved in a subtraction, when their decimal points are aligned, the composite picture must involve not more than 18 digits of representation, or a size error condition can result (which can cause a run unit to terminate abnormally with an arithmetic overflow condition). For information on precision of subtraction, see ADD and SUBTRACT Statements (page 272).

- ROUNDED, SIZE ERROR, and NOT SIZE ERROR Phrases

  See ROUNDED Phrase (page 254) and SIZE ERROR Phrase (page 254) for information on these phrases.

SUBTRACT GIVING

SUBTRACT GIVING subtracts one or more values from a value of one identifier and stores the results in another set of identifiers.



`subtrahend`
is a numeric literal or the identifier of a numeric elementary data item.

`minuend`
is a numeric literal or the identifier of an elementary numeric data item.

`result`
is the identifier of a numeric or numeric edited elementary data item.

ROUNDED
specifies that the result is to be rounded before being stored.

`imperative-stmt-1`
is an imperative statement to be executed when a size error is detected in the subtraction or in storing the result.

`imperative-stmt-2`
is an imperative statement to be executed when no size error is detected in the subtraction or in storing the result.

END-SUBTRACT
ends the scope of the SUBTRACT statement, causing the SUBTRACT to be a delimited-scope statement. If the SUBTRACT statement does not end with an END-SUBTRACT phrase, the presence of the SIZE ERROR or the NOT SIZE ERROR phrase causes the SUBTRACT statement to be a conditional statement, which ends at the next period separator.

Usage Considerations:

- Mathematics

  The values of all the `subtrahend`s are totaled. That sum is subtracted from `minuend`, and the result of this subtraction is then stored after rounding, if ROUNDED was specified as the new value of each `result`. For example, the statement

  ```
  SUBTRACT A B C  FROM  D GIVING  E
  ```

  means store D - (A + B + C) in E.

- Changing Operand Values

  The SUBTRACT GIVING statement does not change the value of any subtrahend or the minuend, unless one is also named as a result.

- Arithmetic Operations

  See Arithmetic Operations (page 267) for information on data conversion and alignment, intermediate results, multiple results (and subscript evaluation), and incompatible data.

- Precision

  For any series of items involved in a subtraction, when their decimal points are aligned, the composite picture must involve not more than 18 digits of representation, or a size error condition can result (which can cause a run unit to terminate abnormally with an arithmetic overflow condition). For information on precision of subtraction, see ADD and SUBTRACT Statements (page 272).

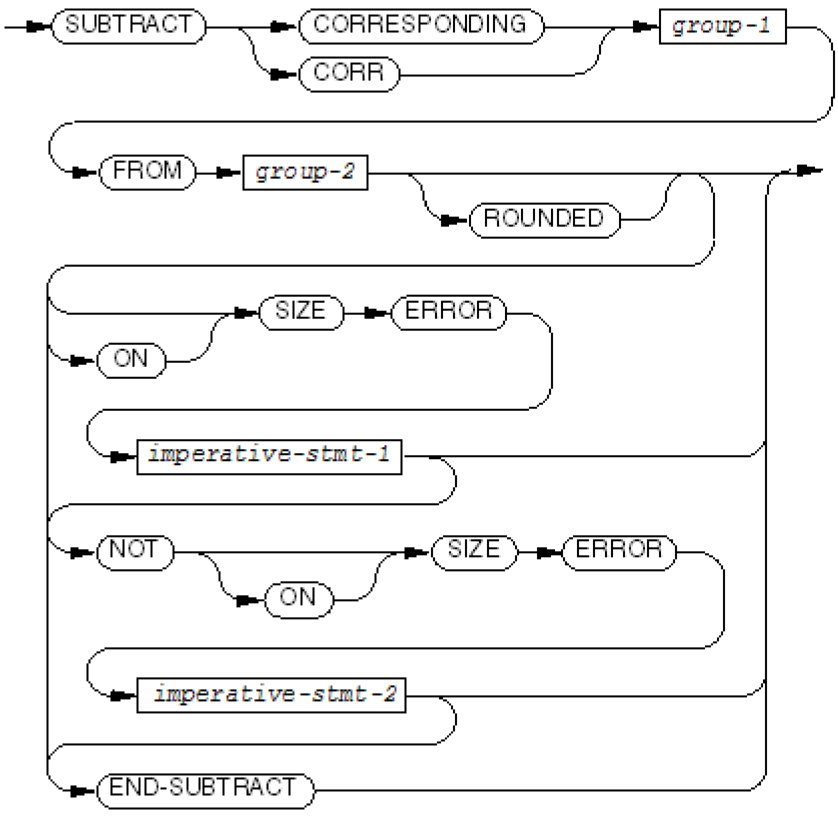
- ROUNDED and SIZE ERROR Phrases

  See ROUNDED Phrase (page 254) and SIZE ERROR Phrase (page 254) for information on these phrases.

SUBTRACT CORRESPONDING

SUBTRACT CORRESPONDING subtracts elements of one data structure from corresponding elements of another data structure.

CAUTION: SUBTRACT CORRESPONDING is not recommended, because minor changes to one data structure can change the correspondence between its elements and those of the other data structure, and this is difficult to detect.



VST229.vsd

group-1

is the identifier of a data structure in which some or all of the elementary items are numeric.

group-2

is the identifier of a data structure that has one or more elementary numeric items. For each elementary numeric item in *group-2* that corresponds to such an item in *group-1*, the difference between the values of the two items replaces the value of *group-2*.

The composite picture of any pair of items aligned by decimal points must not involve more than 18 digits of representation.

ROUNDED

specifies that each difference be rounded before being stored.

imperative-stmt-1

is an imperative statement for the compiler to execute when it detects a size error in any subtraction or in storing the result.

imperative-stmt-2

is an imperative statement to be executed when no size error is detected in any subtraction or in storing the result.

END-SUBTRACT

ends the scope of the SUBTRACT statement, causing the SUBTRACT to be a delimited-scope statement. If the SUBTRACT statement does not end with an END-SUBTRACT phrase, the presence of the SIZE ERROR or the NOT SIZE ERROR phrase causes the SUBTRACT statement to be a conditional statement, which ends at the next period separator.

Usage Considerations:

- Definition of Correspondence

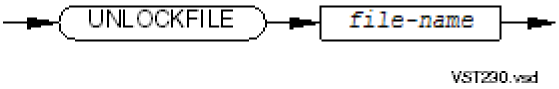
  Groups of data correspond if they have the same names and qualifier names, beyond the group-names in the SUBTRACT statement, and if they meet restraints explained under CORRESPONDING Phrase (page 253).

- Problem Inherent in Using SUBTRACT CORRESPONDING

  The SUBTRACT CORRESPONDING statement can save keystrokes, but can cause problems when someone adds a name to a data structure: the name might be included in an unintended subtraction operation.

UNLOCKFILE

UNLOCKFILE restores access to any records in a file previously locked with LOCKFILE or to those records locked with a LOCK phrase on previous READ statements. Other processes can then read or write records, depending on the file's open mode.



file-name

is the file description name of a file.

Usage Considerations:

- Action of the UNLOCKFILE Statement

  A successful UNLOCKFILE statement releases the exclusive file-access guarantee obtained by a previously executed LOCKFILE statement specifying the same *file-name*. It also releases all exclusive record-access guarantees for that file obtained by previously executed READ LOCK statements. If the file is not locked, or no records of the file are locked, no error occurs.

- File-Status Data Item

  If *file-name* has an associated file-status data item, execution of the UNLOCKFILE statement always assigns an appropriate I-O status code as its value. The status code "00" reports a successful UNLOCKFILE operation.

  Whenever the statement terminates with an I-O status code greater than or equal to "30," execution of the UNLOCKRECORD failed. The possible I-O status codes for an unsuccessful UNLOCKFILE operation are:

I-O Status Code	Unsuccessful UNLOCKFILE Operation
"30"	The unlock operation failed due to non-COBOL causes. The file might or might not now be unlocked.
"42"	The file was not open.

- Disk Files Only

  The UNLOCKFILE statement has no effect unless the file specified is a disk file.

- Key of Reference, File Position Indicator, and Record Area

  The key of reference, the file position indicator, and the record area are not affected by the execution of an UNLOCKFILE statement, whether it succeeded or not.

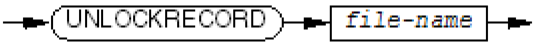
In Example 10-67, the UNLOCKFILE statement precedes a CLOSE statement to unlock the file before the process closes it.

Example 10-67 UNLOCKFILE Statement

```
 FD IN-MASTER-FILE
     ...
 PROCEDURE DIVISION.
     ...
   IF UPDATE-FILE
       OPEN INPUT IN-MASTER-FILE
       LOCKFILE IN-MASTER-FILE
       PERFORM UP-DATE-MASTER
          UNTIL DONE
       UNLOCKFILE IN-MASTER-FILE
       CLOSE IN-MASTER-FILE
   ELSE
     ...
 UP-DATE-MASTER.
* Perform the update, during which time no other process
* can read any record in that file.
     ...
```

UNLOCKRECORD

UNLOCKRECORD restores access by other processes to the last record read (the record selected by the file position indicator). The record was locked by execution of a READ LOCK statement.



VST231.vsd

file-name

is the file description name of a file.

Usage Considerations:

- Action of the UNLOCKRECORD Statement

  The UNLOCKRECORD statement affects only the record designated by the file position indicator for the specified file; therefore, in normal use, it follows a successful read operation or a successful read-then-rewrite sequence. A successful UNLOCKRECORD statement releases the exclusive record-access guarantee obtained by a previously executed READ LOCK statement. If the record is not locked, no error occurs.

- UNLOCKRECORD Statement Immediately After START Statement

  If an UNLOCKRECORD statement is the first input-output statement after a START statement, a run-time error occurs.

- File-Status Data Item

  If *file-name* has an associated file-status data item, execution of the UNLOCKRECORD statement always assigns an appropriate I-O status code as its value. The status “00” reports a successful UNLOCKRECORD operation.

Whenever the statement terminates with an I-O status code greater than or equal to “30,” execution of the UNLOCKRECORD failed. The possible I-O status codes for an unsuccessful UNLOCKRECORD operation are:

I-O Status Code	Unsuccessful UNLOCKFILE Operation
“30”	The unlock operation failed due to non-COBOL causes. The record might or might not now be unlocked.
“42”	The file was not open.

- Disk Files Only

  The UNLOCKRECORD statement has no effect unless the file specified is a disk file.

- Key of Reference, File Position Indicator, and Record Area

  The key of reference, the file position indicator, and the record area are not affected by the execution of an UNLOCKRECORD statement, whether it succeeded or not.

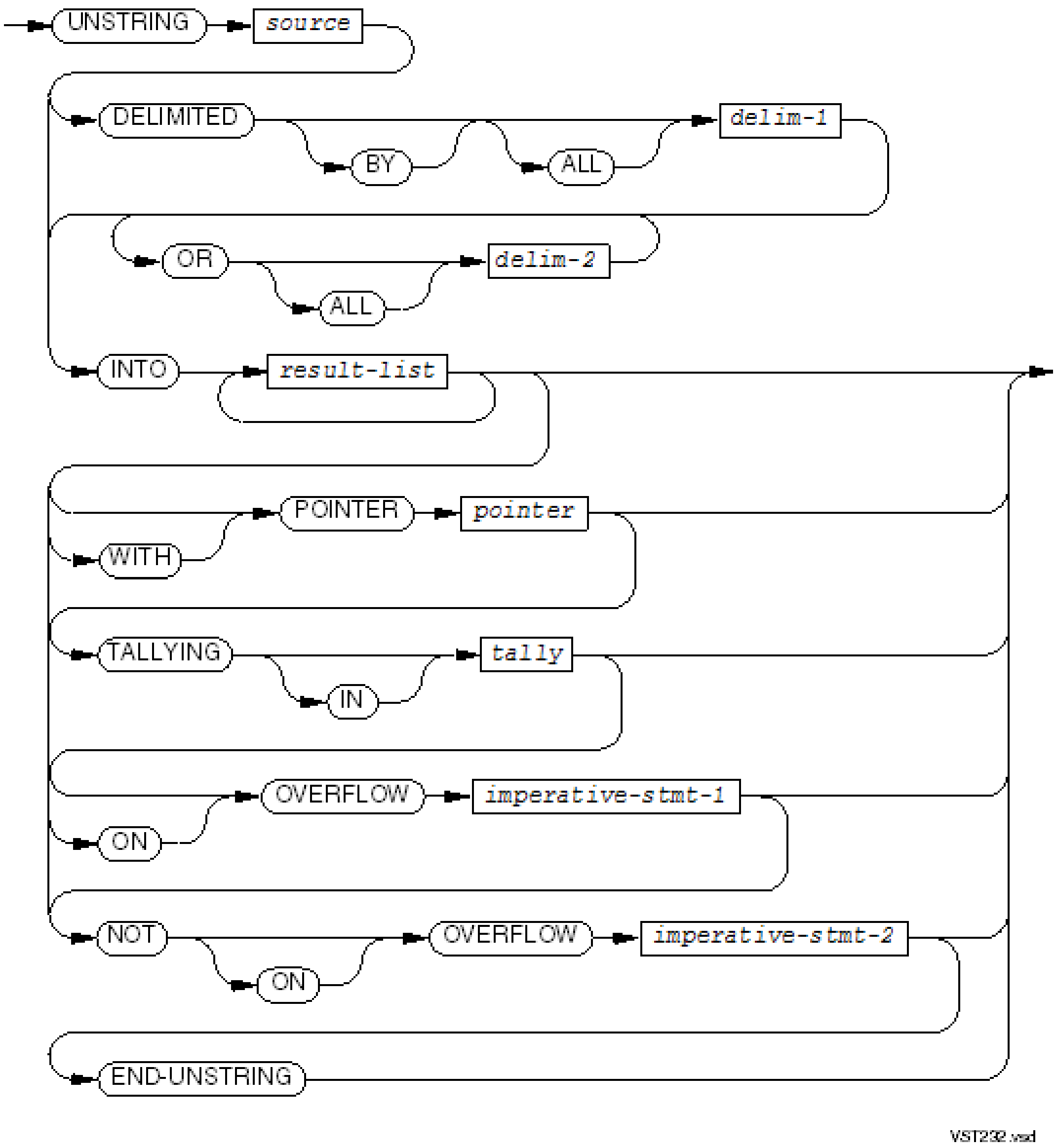
UNSTRING

UNSTRING partitions a data item (the source) into strings of consecutive characters and stores those strings into other data items (the results). You can determine the partitioning in two ways:

- Use the size of the result data items.
- Use the presence of specific character sequences in the source (like spaces in a person’s name or commas and decimal points in an edited number).

When the statement executes, it stores these into data items of your choice:

- Each part
- For each part, the delimiter that ended that part (optional) and the number of characters stored (optional)
- A count of the parts stored (optional)
- A character-count pointer to specify where in the item the unstring operation terminates (optional)



`source`
: is the identifier of an alphanumeric data item containing a sequence of characters that the unstring operation is to separate into one or more sequences of characters and store them in one or more `result` items. `source` can be qualified or subscripted, but cannot include reference modification.

ALL
: causes the unstring operation to handle all consecutive occurrences of the value of `delim-1` as if they were only one occurrence of `delim-1`. Without ALL, the unstring operation handles only the first occurrence of the value of `delim-1` as the delimiter.

`delim-1`
: is the identifier of an alphanumeric data item or an alphanumeric literal. If it is a figurative constant, it must represent a single character.

The delimiter `delim-1` marks the end of a portion of the value of `source`. The value that the unstring operation stores into a `result` item does not include the value of `delim-1`.

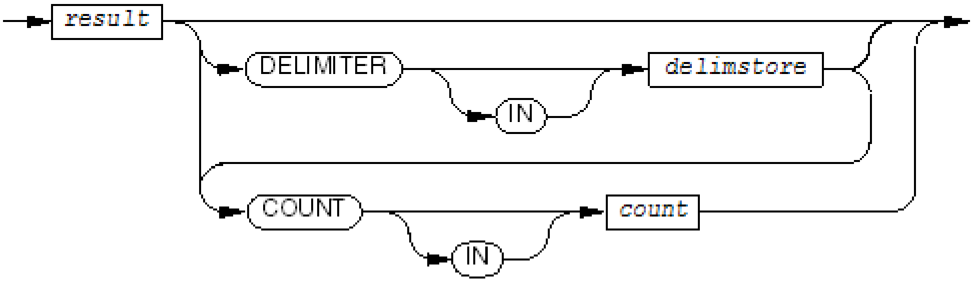
ALL
: causes the unstring operation to handle all consecutive occurrences of the value of `delim-2` as if they were only one occurrence of `delim-2`. Without ALL, the unstring operation handles only the first occurrence of the value of `delim-2` as the delimiter.

delim-2

is the identifier of an alphanumeric data item or an alphanumeric literal. If it is a figurative constant, it must represent a single character.

The delimiter *delim-2* marks the end of a portion of the value of *source*. The value that the unstring operation stores into a *result* item does not include the value of *delim-2*.

result-list



VST239.vsd

result

specifies the identifier of an alphanumeric, alphabetic, or numeric DISPLAY elementary data item into which the unstring operation copies characters from *source*. If *result* identifies a numeric data item, the PICTURE clause that describes it cannot contain any *P* s.

result cannot reference a special register.

An UNSTRING statement can have at most 127 *result* fields.

delimstore

is the identifier of an alphanumeric data item. You can only use the DELIMITER phrase when you also use the DELIMITED phrase. The unstring operation copies the character or characters of *source* that matched the delimiter specified in the DELIMITED phrase to *delimstore*. *delimstore* cannot reference a special register.

count

is the identifier of an integer data item. You can only use the COUNT phrase when you also use the DELIMITED phrase. The unstring operation stores in *count* the number of characters it moved to *result*. If it did not move any characters, the unstring operation does not alter the value of *count*, so you must initialize *count* to a known value if you want to use it to determine whether the UNSTRING statement moved any characters to *result*.

When the character-string from *source* is longer than *result*, the unstring operation truncates it to fit into *result*, and stores into *count* the length *source* had before it was truncated.

pointer

is the identifier of an integer data item. When you omit this phrase, the unstring operation begins at the first character of the *source* data item. When you include this phrase, the integer value in *pointer* specifies the position in the source data item where the unstring operation begins. When the unstring operation completes, the integer value in *pointer* specifies the position just beyond the last character processed by the unstring operation (delimiter or portion copied to a *result* data item).

pointer must reference a data item that is capable of containing a numeric value equal to one plus the size of the item referenced by *source*. *pointer* cannot reference a special register.

tally

is the identifier of an integer data item to which the unstring operation adds the number of *result* data items it stored. The UNSTRING statement does not initialize *tally* but adds to its current value. If you want *tally* to begin at a given number, set it before the UNSTRING statement executes.

tally cannot reference a special register.

imperative-stmt-1

is an imperative statement to be executed when overflow occurs; that is, when these conditions exist:

- The initial value of *pointer* is less than 1 or greater than the length of *source*.
- All data-receiving areas have been acted upon, and *source* still contains unexamined characters.

If you do not specify an OVERFLOW clause and an overflow occurs, control passes to the next statement after UNSTRING.

imperative-stmt-2

is an imperative statement to be executed when overflow does not occur.

END-UNSTRING

ends the scope of the UNSTRING statement, causing the UNSTRING to be a delimited-scope statement. If you omit the END-UNSTRING phrase, the presence of the OVERFLOW or the NOT OVERFLOW phrase makes the UNSTRING statement a conditional statement, which ends at the next period separator.

Usage Considerations:

- Purpose

  In general terms, the UNSTRING statement partitions the value of the sending area (referenced by *source*) into a sequence of strings of consecutive characters and stores them in the receiving areas (referenced by a list of one or more instances of *result*).

- Execution of the UNSTRING Statement

  The execution of the UNSTRING statement consists of these three phases:

1. Initialization phase

   In the initialization phase, the UNSTRING statement performs these initialization operations before beginning to cycle through the list of INTO phrases:

   — It establishes the *source* item as the sending area. Even if the item has a variable size (is defined with the OCCURS DEPENDING clause), the initial size is used as the sending area size for the duration of statement execution.
   — If the UNSTRING statement includes the POINTER phrase, the initialization establishes the *pointer* item as the pointer variable. The initial value of this item determines the relative character position within the sending area at which the unstring operation begins. The first character position is 1.
   — This enables you to start the unstring operation at some character other than the first character in the *source* item, and to retain a record of where the unstring operation terminated. (A later UNSTRING statement could then begin its operation at the point where another one left off.)
   — If the initial value of the *pointer* item is less than one or greater than the size of the *source* item, then the overflow condition exists immediately and no unstring operation occurs.
   — If the UNSTRING statement does not include the POINTER phrase, the unstring operation begins at the first character position in the sending area.
   — If the UNSTRING statement includes a TALLYING phrase, the initialization establishes *tally* as the tallying variable.
   — Initialization establishes the first *result* item as the current receiving area.

2. UNSTRING cycle phase

   In the UNSTRING cycle phase, the unstring operation consists of one or more cycles. Each cycle examines characters in the "sending area" one by one. The sending area is the portion of the source item beginning at the current character position and ending at the last character of the source item. Each cycle assigns an appropriate string of characters to the current *result*.

   The identification of the item to be copied depends on the presence or absence of the DELIMITED phrase, and on when the unstring operation reaches the end of the *source* item.

   If the statement includes the DELIMITED phrase, the examination proceeds left to right until it encounters either a delimiter string or the end of the sending area. A delimiter string is a contiguous set of characters, beginning with the character under examination, whose value exactly matches the value of any of the constants or data items specified in the DELIMITED phrase.

   If your program has these data items:

```
05 U PIC X(32) VALUE "The UNSTRING statement is handy."
*                     ....'....'....'....'....'....'..
*                         5   10   15   20   25   30
01         .
   03 WORD-1 PIC X(8).
   03 WORD-2 PIC X(10).
   03 WORD-3 PIC X(7).
   03 WORD-4 PIC X(3).
   03 WORD-5 PIC X(8).
   03 WORD-6 PIC X(9).
```

   And if you execute

```
UNSTRING U DELIMITED BY SPACE
            INTO WORD-1 WORD-2 WORD-3
                 WORD-4 WORD-5 WORD-6
```

   the first item to be copied is "The" and the second is "UNSTRING" and so on.

If you specify two or more delimiters, the unstring operation compares their values with the sending area in the same order as they appear in the phrase. If a match occurs, the corresponding set of characters in the sending area forms the delimiter string; any delimiters not yet tested are ignored. If no delimiter value matches the sending area at the current position, the unstring operation repeats the delimiter search beginning with the next character of the sending area. No character in the sending area can be considered a part of more than one delimiter.

If you execute

```
UNSTRING U DELIMITED BY SPACE OR "I"
           INTO WORD-1 WORD-2 WORD-3
           WORD-4 WORD-5 WORD-6
```

the first item to be copied is "The," the second item is "UNSTR," the third is "NG," and so on.

Two special cases are consecutive delimiters and multicharacter delimiters.

If ALL was not specified and the unstring operation encounters two consecutive delimiters, it interprets the sending item as zero if *result* is numeric. If *result* is not numeric, the unstring operation interprets the sending item as spaces.

Each instance of *delim-1* or *delim-2* represents one delimiter. When a delimiter contains two or more characters, all of the characters must be present in contiguous positions in the *source* item, and in the order given, to be recognized as a delimiter.

If you omit the DELIMITED phrase, the examination proceeds left to right until the number of characters examined equals the size of the current receiving area.

If you execute

```
UNSTRING U INTO WORD-1 WORD-2 WORD-3
             WORD-4 WORD-5 WORD-6
```

the first item to be copied is "The UNST," the second item is "RING state," and so on.

If the *result* item is numeric and its operational sign occupies a separate character position, the number of characters the unstring operation examines (when you omit the DELIMITED phrase) is one less than the size of the item.

If the unstring operation reaches the end of the *source* item before it detects a matching delimiter (for the DELIMITED phrase), or before it fills all the receiving items, the examination terminates with the last character examined.

If you execute

```
UNSTRING U DELIMITED BY SPACE
           INTO WORD-1 WORD-2 WORD-3
                WORD-4 WORD-5 WORD-6
```

The fifth item to be copied is "handy." There is no sixth item to be copied.

The unstring operation handles the set of characters thus examined (excluding the delimiter string, if any) as an elementary alphanumeric data item and copies it to the current receiving area in accordance with the rules for the MOVE statement. When the set contains no characters (that is, if a delimiter string begins at the very first character examined in this cycle), the unstring operation moves a null value to the current receiving area. If the current receiving area is described as numeric or numeric-edited, the value 0 is moved; otherwise the value spaces is moved.

If you include a DELIMITER phrase in the INTO phrase for this cycle, the unstring operation handles the set of characters in the delimiter string as an elementary alphanumeric data item and copies it to *delimstore* in accordance with the rules for the MOVE statement. If the delimiting condition is the end of the sending area (that is, there is no delimiter string), then the unstring operation fills the delimiter data item with spaces.

If the delimiter is described as a figurative constant with the ALL qualifier, only one occurrence of the unqualified figurative constant is moved.

If you execute

```
UNSTRING U DELIMITED BY SPACE OR "."
           INTO WORD-1 DELIMITER IN DEL-1
                ...
                WORD-5 DELIMITER IN DEL-5
```

you get

```
"The     "
```

in WORD-1,

```
" "
```

in DEL-1, and so on down to

```
"handy "
```

in WORD-5 and

```
"."
```

in DEL-5.

If you include the DELIMITER phrase in the INTO phrase for this cycle, the unstring operation advances the current character position in the sending area to the first character following the delimiter string located in this cycle.

If the specification of the matched delimiter includes the keyword ALL and the portion of the sending area following the delimiter string contains one or more repetitions of that set of characters, then the unstring operation advances the current character position past all repetitions; therefore the unstring operation considers two or more contiguous occurrences of the matched delimiter string as equivalent to a single occurrence in determining the beginning character for the next examination cycle.

If you include a COUNT phrase in the INTO phrase for this cycle, the unstring operation assigns the numeric value equal to the number of characters examined (excluding the delimiter string, if any) to *count* in accordance with the rules for an elementary move operation.

If you execute

```
UNSTRING U DELIMITED BY SPACE OR "."
           INTO WORD-1 COUNT IN COUNT-1
                ...
                WORD-5 COUNT IN COUNT-5
```

you get

```
"The     "
```

in WORD-1, but 3 in COUNT-1, and you get

```
"stateme"
```

in WORD-3 but 9 in COUNT-3.

If you include the POINTER phrase, the unstring operation increments the value of the *pointer* item by one for each character examined as a part of this cycle. This includes each character in the value copied to the receiving area, each character in the delimiter string (if any), and each character in any contiguous repetitions of the delimiter string; therefore the resulting value of the pointer variable reflects the relative character position within the sending area at which the next cycle begins.

If the sending area still contains any unexamined characters and the current receiving area is not the last one, then the unstring operation establishes the next *result* item as the new current receiving area and begins another unstring cycle.

If the sending area still contains any unexamined characters but the current receiving area is the last one, then the overflow condition exists and the unstring operation terminates.

If the sending area does not contain any unexamined characters, then the unstring operation terminates normally and any remaining receiving areas are ignored.

When the execution of an UNSTRING statement with a TALLYING phrase terminates, either normally or due to an overflow condition, `tally` contains a value equal to its initial value plus the number of data receiving items that were assigned new values.

If an overflow condition arises, execution of the UNSTRING statement terminates at that point.

3. OVERFLOW/NO OVERFLOW processing phase (optional)

   If you include an OVERFLOW phrase, the imperative statement in that phrase is executed; otherwise control is transferred to the next executable statement in the normal way.

- Operand Identification

  For each identifier, the process of operand identification occurs only once, at the beginning of the execution of the UNSTRING statement.

- Operand Overlap

  Neither the storage area referenced by *pointer* nor that referenced by *tally* can be the same as or overlap the storage area referenced by any other identifier appearing in the UNSTRING statement.

  The storage area referenced by *result*, *delimstore*, and *count* must not overlap or be the same as any of the storage areas referenced by *source*, *delim-1*, or *delim-2*.

  Violation of these rules produces unpredictable results.

- National Data Items and National Literals

  If any of the data items *delim-1*, *delim-2*, *result*, or *delimstore* is a national data item or national literal, then all of them must be national items.

In Example 10-68, UNSTRING breaks a data item into a collection of data items. UNSTRING uses the MOVE statement rules in transferring values into shorter data items.

Example 10-68 UNSTRING Statement

Input:

```
DATA DIVISION.
WORKING-STORAGE SECTION.
77  SOURCE-STRING PIC X(18)   VALUE "12345 MICKEY ABCDE".
77  UNPART-1      PIC X(3).
77  UNPART-2      PIC X(3).
77  UNPART-3      PIC X(3).
PROCEDURE DIVISION.
A10-START.
 UNSTRING SOURCE-STRING DELIMITED BY " "
     INTO UNPART-1 UNPART-2 UNPART-3
 DISPLAY "UNPART-1 = " UNPART-1
 DISPLAY "UNPART-2 = " UNPART-2
 DISPLAY "UNPART-3 = " UNPART-3
 STOP RUN.
```

Output:

```
>RUN RUNUNIT
UNPART-1 = 123
UNPART-2 = MIC
UNPART-3 = ABC
```

In Example 10-69, UNSTRING separates a name, entered as it would be typed on a mailing envelope, into a last name and a remainder. You could use this mechanism to build records that can be sorted on last and first name.

Example 10-69 UNSTRING Statement

```
 WORKING-STORAGE SECTION.
 01  NAMES-TABLE.
     03 NAMES PIC X(50) OCCURS 10 TIMES
                        INDEXED BY NAME-INDEX,
                                   FIRST-NAME-INDEX.
01  WORK-GROUP.
     03 NAME-COUNT   PIC 99 COMP.
     03 WHOLE-NAME   PIC X(50).
     03 LAST-NAME    PIC X(50).
     03 REST-OF-NAME PIC X(50).
     03 POINTER-1    PIC 99 COMP.
 ...
 PROCEDURE DIVISION.
 WHATS-NEXT.
*  PROMPT FOR A NAME AS IT WOULD BE TYPED ON AN ENVELOPE
     DISPLAY "Enter a name (or a space to terminate):".
     ACCEPT WHOLE-NAME|
     IF WHOLE-NAME = SPACES
        STOP RUN
     END-IF
*  INITIALIZE
     MOVE SPACES TO NAMES-TABLE, REST-OF-NAME
    MOVE 0 TO NAME-COUNT
*  GET EACH CONSECUTIVE BLOCK OF CHARACTERS ENDING IN ONE
*  OR MORE SPACES INTO A NAME (I) AND COUNT THE BLOCKS.
     UNSTRING WHOLE-NAME
              DELIMITED BY ALL " " INTO
              NAMES (1) NAMES (2) NAMES (3)
             NAMES (4) NAMES (5) NAMES (6)
              NAMES (7) NAMES (8) NAMES (9) NAMES (10)
              TALLYING IN NAME-COUNT

*  LAST ONE COPIED IS LAST-NAME
     MOVE NAMES (NAME-COUNT) TO LAST-NAME
*  IGNORE LEADING SPACES
     IF NAMES (1) = SPACES
        SET FIRST-NAME-INDEX TO 2
    ELSE
        SET FIRST-NAME-INDEX TO 1
     END-IF

*  CONCATENATE OTHER NAMES INTO REST-OF-NAME
     MOVE 1 TO POINTER-1
     PERFORM COLLECT-REST
             VARYING NAME-INDEX FROM FIRST-NAME-INDEX BY 1
             UNTIL NAME-INDEX = NAME-COUNT
     DISPLAY "Last name is......" LAST-NAME
     DISPLAY "Rest of name is..." REST-OF-NAME
     DISPLAY " "
     GO TO WHATS-NEXT.
 COLLECT-REST.
     STRING NAMES (NAME-INDEX) DELIMITED BY SPACE
            INTO REST-OF-NAME
            POINTER POINTER-1.
```

This is a sample run of Example 10-69 with a typical name and a single-word name:

```
Enter a name (or a space to terminate):
?Abraham Moyer
Last name is......Moyer
Rest of name is...Abraham
Enter a name (or a space to terminate):
```

```
?Johann Philip Geissinger
Last name is......Geissinger
Rest of name is...JohannPhilip
Enter a name (or a space to terminate):
?Copernicus
Last name is......Copernicus
Rest of name is...
```

The portions of the rest of the name are not separated by spaces. If you want spaces, make the STRING statement look like this:

```
STRING NAMES (NAME-INDEX) DELIMITED BY SPACE
       " " DELIMITED BY SIZE
           INTO REST-OF-NAME
           POINTER POINTER-1.
```

USE

USE defines debugging or exception-handling procedures beyond the standard techniques of the file system. USE also defines the conditions under which the procedure paragraphs following it are executed.

USE can appear only as the first statement of a section in the Declaratives Portion of the Procedure Division, and it must be the only statement in the sentence containing it. Any other statements that the section contains must be organized into sentences belonging to paragraphs of the section. Empty debugging declarative sections serve no purpose.

Each section containing a USE statement is a declarative procedure. A declarative procedure, together with any associated utility sections, forms a logically discrete area. (The other logically discrete area of a program is the remainder of the Procedure Division.)

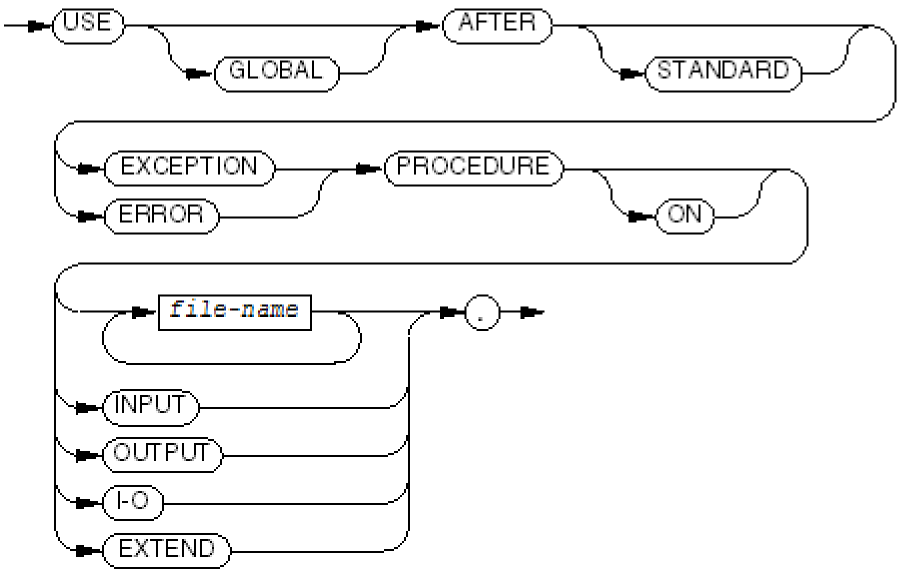
Control statements (ALTER, GO TO, and PERFORM) specified in one logically discrete area are, with a few exceptions, not permitted to refer to procedure-names defined within another logically discrete area. For the detailed restrictions and permissions, see ALTER (page 302), GO TO (page 345), and PERFORM (page 399).

USE DEBUGGING

NOTE: The 1985 COBOL standard classifies USE DEBUGGING as **obsolete**. The compiler does not recognize it. For its description, see the *COBOL Manual for TNS and TNS/R Programs*.

USE AFTER EXCEPTION

USE AFTER EXCEPTION intercepts control when an input-output exception occurs. The input-output system executes USE AFTER EXCEPTION procedures after it completes the standard input-output error routine. It also executes USE AFTER EXCEPTION procedures when invalid-key or at-end conditions arise and no INVALID KEY or AT END phrases apply to the file. After the USE procedure executes, it returns control to the calling routine, unless the I-O status code is “4*x*” or “90,” in which case the process terminates.



VST235.vsd

GLOBAL

NOTE: Do not use GLOBAL in the Declaratives Portion (page 248).

applies the declarative exception procedure to the program in which the USE statement appears and to any programs nested within that program, unless such a nested program has its own declarative exception procedure for that particular exception.

EXCEPTION, ERROR

introduce the exception procedure and specify that the procedure is to be performed after the COBOL run-time input-output error routine, or when an invalid-key or at-end condition arises and no INVALID KEY or AT END phrase is present in the statement then executing.

file-name

is a file description name that is to use this USE statement when an exception occurs. A given *file-name* can occur in only one USE statement in a given program. Sort-merge file description names are not permitted.

INPUT

specifies that all files opened in INPUT mode that are not specified explicitly in another USE statement are to use this USE statement.

OUTPUT

specifies that all files opened in OUTPUT mode that are not specified explicitly in another USE statement are to use this USE statement.

I-O

specifies that all files opened in I-O mode that are not specified explicitly in another USE statement are to use this USE statement.

EXTEND

specifies that all files opened in EXTEND mode that are not specified explicitly in another USE statement are to use this USE statement.

Usage Considerations:

- Restrictions
  - Control statements (ALTER (page 302), GO TO (page 345), and PERFORM (page 399)) specified in one logically discrete area are, with a few exceptions, not permitted to refer

to procedure-names defined within another logically discrete area. For detailed restrictions and permissions, see ALTER, GO TO, and PERFORM.

- — Procedure names defined within a nondebugging declarative procedure (and procedure-names defined within utility sections associated with that declarative procedure) can be referred to by PERFORM statements located anywhere in the Procedure Division.
- — A declarative exception procedure in which the GLOBAL phrase is specified must not execute an EXIT PROGRAM statement.
- — Declarative exception procedures apply only to data files, not to sort-merge files.

- Implicit and Explicit File Reference

  A USE statement that mentions a file by name is said to make an explicit reference to the corresponding file. A USE statement that mentions an open mode (INPUT, I-O, OUTPUT, or EXTEND) makes reference to a file implicitly, according to the mode in which the program opened (or attempted to open) the file.

  It is usually best to write USE statements that explicitly reference individual files, because this gives you better control over the handling of exceptions. The declarative called by implicit reference has no simple way to determine the identity of the file that generated an exception. The management of this generic sort of exception handling can be difficult, particularly when program maintenance might introduce additional files, or in nested programs when a GLOBAL declarative is active.

- Precedence Rules for Nested Programs

  When a program contains other programs, these precedence rules apply. When the run-time routines detect an I-O exception, they apply these rules (in the order indicated) to select and perform only the first declarative procedure that qualifies.

  1. If the program in which the exception-causing statement occurred contains an appropriate declarative, use it.
  2. If no declarative in that program is appropriate, check the next containing program (the next one outward in the nesting). If the containing program contains an appropriate declarative in which the GLOBAL phrase is specified, use it.
  3. Repeat Item 2 until the outermost program has been checked. If no qualifying declarative procedure has been found, none is executed.

  An outer program can contain a GLOBAL declarative procedure to handle each file, but a nested program can contain an overriding GLOBAL or local declarative procedure that governs the handling of exceptions for certain files within the nested program or any programs it contains.

- File Status and GUARDIAN-ERR Special Register

  For information on file status and the special register GUARDIAN-ERR, see I-O Status Code (page 257). A USE AFTER EXCEPTION procedure can base its activity on the values of these data items.

In Example 10-70, two files use the same error routine and another file uses a separate error routine.

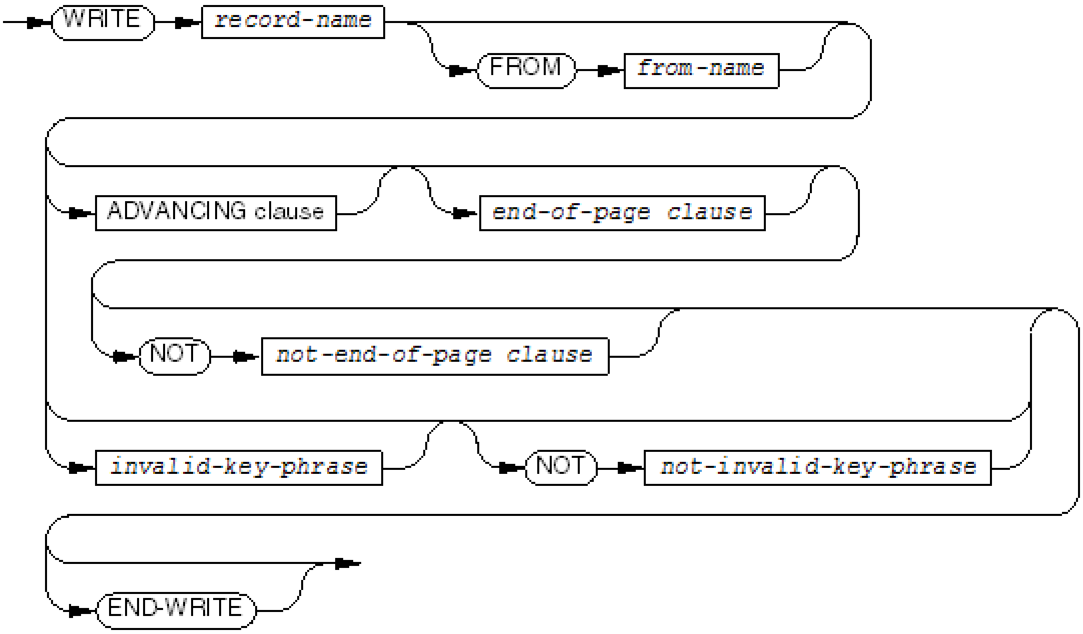
Example 10-70 USE AFTER EXCEPTION Statement

```
   ...
PROCEDURE DIVISION.
DECLARATIVES.
MASTER-FILES SECTION.
    USE AFTER EXCEPTION PROCEDURE ON MASTER-1 MASTER-2.
MASTER-ERROR-ROUTINE.
    IF FILE-STATUS....
      ...
DETAIL-FILE SECTION.
    USE AFTER EXCEPTION PROCEDURE ON DETAIL-IN.
DETAIL-ERROR-ROUTINE.
      ...
END DECLARATIVES.

MAIN-SECTION SECTION.
BEGIN-PROGRAM.
      ...
```

WRITE

WRITE delivers a record to its associated file.

WRITE for Sequential Files



VST236.vsd

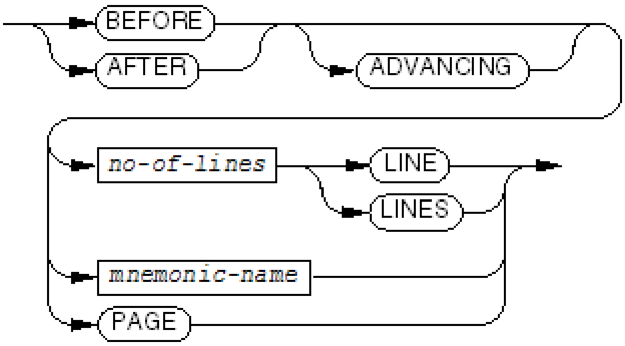
`record-name`

is a logical record described in the File Section of the Data Division. The `record-name` can be qualified by the name of the file with which the record is associated. The data written is the current contents of `record-name`.

`from-name`

is the identifier of a data area whose contents are to be moved to the record specified by `record-name` before the WRITE occurs. `from-name` must specify a data area other than that specified by `record-name`. It also cannot specify an index data item.

ADVANCING clause



VST237.vsd

advances the file (skips lines) before or after the record is written to it. The default ADVANCING clause is AFTER ADVANCING 1. The ADVANCING clause can be used only with files assigned to processes or printers.

BEFORE

specifies that the record is to be printed before any lines are skipped.

AFTER

specifies that the record is to be printed after any lines are skipped.

no-of-lines

is a numeric integer literal, or the identifier of a numeric integer data item, whose value is greater than or equal to zero and represents the number of lines to advance before or after the write operation.

NOTE: When *no-of-lines* has the value -1, HP COBOL advances to the top of a page; however, setting *no-of-lines* to -1 is not recommended, because it could interfere with later extensions to COBOL that would allow backspacing one line. To go to the top of a page, use BEFORE PAGE or AFTER PAGE.

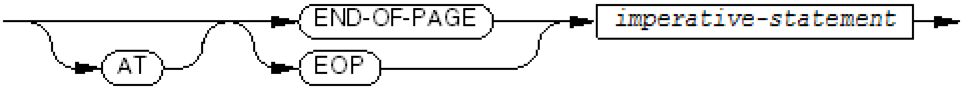
mnemonic-name

specifies a channel position on a carriage-control tape that directs the printer to skip lines (advancing to a particular channel is usually faster than printing lines of spaces). *mnemonic-name* is defined in a SPECIAL-NAMES paragraph for a printer (or a process simulating a printer) with the CHANNEL option. *mnemonic-name* is not permitted for a file described with a LINAGE clause.

PAGE

advances the printer to the top of the page before or after the write operation. PAGE and *end-of-page clause* cannot appear in the same WRITE statement.

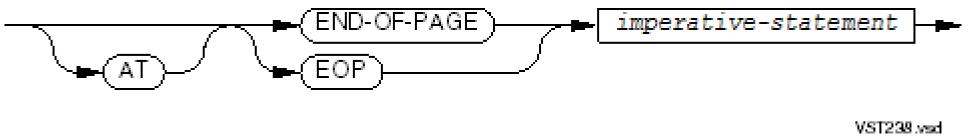
end-of-page clause



VST238.vsd

executes *imperative-statement* when the write operation encounters the end-of-page condition. The file description must include a LINAGE clause.

not-end-of-page clause

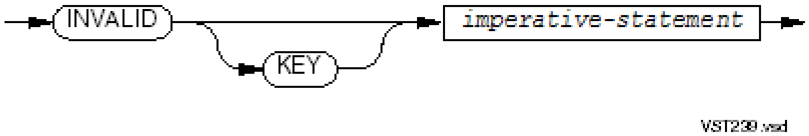


executes *imperative-statement* when the write operation does not encounter the end-of-page condition. The file description must include a LINAGE clause.

imperative-statement

is to be executed when the end-of-page condition is satisfied when LINAGE-COUNTER is either greater than the defined page length or is at a line in the footing area. In both cases, the line is written before *imperative-statement* is executed. If an attempt is made to write beyond the allowable area on a page, whether at or after the end-of-page condition, the page is automatically advanced (this is called the page overflow condition).

invalid-key-phrase

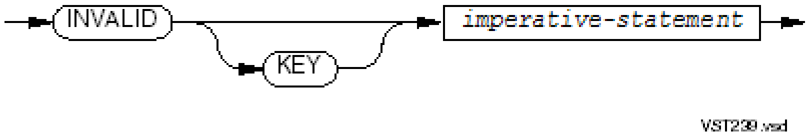


executes *imperative-statement* when the write operation encounters the invalid-key condition. The file description cannot include a LINAGE clause.

imperative-statement

is an imperative statement to be executed when an invalid-key condition arises because alternate keys are defined for the sequential file, and the write operation would create a duplicate key when the file definition did not specify that duplicates are allowed. If no INVALID KEY phrase is present, a USE procedure must be present for the file or files to be opened in OUTPUT mode.

not-invalid-key-phrase



executes *imperative-statement* when the write operation does not encounter the invalid-key condition. The file description cannot include a LINAGE clause.

imperative-statement

is an imperative statement to be executed when an invalid-key condition does not arise.

END-WRITE

ends the scope of the WRITE statement, causing the WRITE to be a delimited-scope statement. If the WRITE statement does not end with an END-WRITE phrase, the presence of the AT END-OF-PAGE, the NOT AT END-OF-PAGE, the INVALID KEY, or the NOT INVALID KEY phrase causes the WRITE statement to be a conditional statement, which ends at the next period separator.

Usage Considerations:

- Action of the WRITE Statement

  The write operation releases a logical record to the run-time routines for inclusion in the file. For a file of sequential organization, the order of the records in the file therefore corresponds to the order in which they are released.

  In general, the logical record consists of the value of the data item specified by *record-name*; however, when the file has fixed-length records, and the *record-name* item contains fewer than the defined number of character positions, the logical record is extended with arbitrary character values.

- Printer or Spooler Files

  Release of a logical record does not necessarily imply immediate transmission to the file. The record can be held in an internal buffer. The logical record is always transmitted to the file or spooler buffer immediately.

- Opening File Positions Printer at Top of First Page

  When you open a file, the printer is positioned at the top of the first page, as if BEFORE ADVANCING PAGE had been executed. If you use default advancing (AFTER ADVANCING 1), you get a blank line at the top of the first page. (You usually use BEFORE 1 before the first write operation.)

- Files That Are Not Printers or Spooler Files

  If a file is not a printer or spooler file, each logical record is written to the file immediately, any ADVANCING clause is ignored, and no control information (for advancing or forms control) or extra blank lines are written to the file.

- Writing to EDIT Files

  An HP COBOL program can write at most 239 characters to an EDIT file (a file with file code 101). The action of the WRITE statement depends on the existence of the file and on the form of OPEN used.

  — Existing EDIT file, OPEN EXTEND

    Each write operation appends a line to the file. The line numbers of the new lines begin at a value one greater than the last line number in the file, and are incremented by one.

  — Existing EDIT file, OPEN OUTPUT

    Existing records are deleted from the file. Each write operation appends a line to the file. Line numbers begin at one and are incremented by one.

  — File does not exist, OPEN OUTPUT or EXTEND

    For the write operation to create the file, the run-time environment must contain an ASSIGN command that specifies CODE 101. Each write operation appends a line to the file. Line numbers begin at one and are incremented by one.

    To make the file an EDIT file before writing it, use the COBOL_ASSIGN_ routine with *file-code* 101.

  — Program was compiled with the NONSTOP directive

    If a program was compiled with the NONSTOP directive, it can open an EDIT file for input, but not for either form of output.

- File-Status Data Item

  If the file has an associated file-status data item, execution of the WRITE statement always assigns an appropriate I-O status code. The value “00” reports a successful write operation.

The I-O status codes that result from an unsuccessful write operation are:

I-O Status Code	Unsuccessful Write Operation
"22"	An alternate record key value of the logical record equals that of a record that already exists in the file and duplicate values are not allowed for that key (all access modes).
"30"	The write operation failed due to non-COBOL causes. The specified record might or might not have been written.
"34"	A boundary violation exception exists when execution of a WRITE statement would require exceeding the record storage capacity of the file. The logical record is not released.
"44"	When the file is described with the RECORD VARYING clause, the logical record size must be neither greater than the maximum nor less than the minimum number of character positions specified in that clause. This requirement is not met, and the logical record is not released.
"48"	Either the file has sequential organization and is not open in I-O, OUTPUT, or EXTEND mode; or the open mode is I-O, and the file device is not a terminal, process, or $RECEIVE.
"90"	The WRITE statement contains an ADVANCING phrase, and the value of no-of-lines is a negative number other than -1.

- Invalid-Key Condition

  When the sequential WRITE statement is used to write to an indexed file, or to write to a file with alternate keys, the invalid-key condition arises when the I-O status code is "21" or "22."

- Reel-Swap Sequence for Multiple-Reel Tape Files

  If execution of a WRITE statement for a multiple-reel tape file exhausts the capacity of the current reel, the run-time routine automatically performs a reel-swap sequence. You can use the COBOL_SPECIAL_OPEN_ routine to notify the program.

- Page Headers and Trailers

  If you want a page trailer, describe the footing area with a FOOTING value less than or equal to the LINES value in the LINAGE clause. Then use an END-OF-PAGE phrase to write a page trailer; eject the page, if necessary; and write the next page header. If you want only a page header, omit the FOOTING phrase from the LINAGE clause.

  Multiple end-of-page conditions can occur during the production of one logical page if several successive WRITE statements cause printing or spacing within that logical page's footing area.

- Page Overflow

  An automatic page overflow condition exists whenever the execution of a write statement cannot be fully accommodated within the current page body of a printer file described with a LINAGE clause. This situation arises when the execution of the WRITE statement would cause the LINAGE-COUNTER to assume a value greater than the number of lines in the current page body.

  In this case, the process prints the record on the logical page before or after advancing the device to the first line following the top margin of the next logical page.

  If the END-OF-PAGE phrase appears in the WRITE statement, then its imperative statement is executed after both the print and advancing operations are completed.

  When execution of a WRITE statement causes both the end-of-page and the page overflow conditions to occur, only the actions for the page overflow condition occur.

- Buffered Cache

  Buffered cache, enabled by the RESERVE clause of the FILE-CONTROL paragraph, speeds the writing of disk files. This technique buffers records up in cache rather than writing them immediately to disk.

  Do not use buffered cache in applications that require each record to be actually written to disk before execution of the next statement in the program.

  See FILE-CONTROL Paragraph (page 127).

- Variable-Length Records

  An Enscribe structured file can have variable-length records. See READ for Sequential or Dynamic Access (page 414).

  When you write to a file that is defined as having variable-length records, the length of the record written depends on whether the file is declared with

  ```
  RECORD CONTAINS rec-1 TO rec-2 CHARACTERS
  ```

  or

  ```
  RECORD IS VARYING IN SIZE FROM rec-1 TO rec-2 CHARACTERS
    DEPENDING ON rec-size
  ```

  form of the RECORD CONTAINS clause.

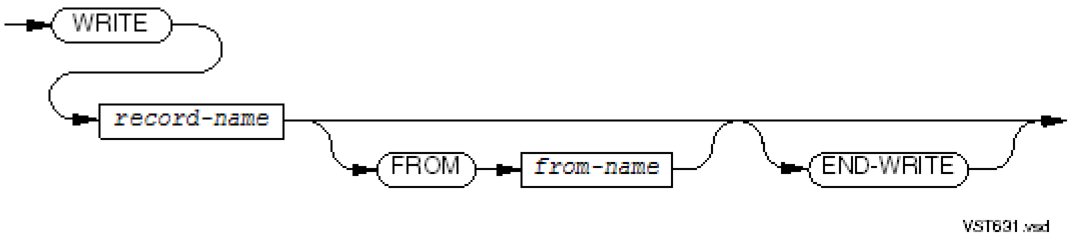
  In the former case, the length of the record specified in the WRITE statement is the number of characters written to the file system file.

  In the latter case, the length of the record written is the value present in the *rec-size* data item specified in the DEPENDING clause at the time the WRITE statement is executed.

Example 10-71 ADVANCING Phrase

```
ENVIRONMENT DIVISION.
      ...
 INPUT-OUTPUT SECTION.
 FILE-CONTROL.
     SELECT MASTER-RANDOM-FILE
     ASSIGN TO "$MARKT.PLATZ.RPT"
     ORGANIZATION IS SEQUENTIAL
     ACCESS MODE IS SEQUENTIAL
     FILE STATUS IS M-R-STATUS.
 ...
 DATA DIVISION.
 FILE SECTION.
 ...
 FD MKT-REPORT
     LABEL RECORDS ARE OMITTED
     RECORD CONTAINS 132 CHARACTERS

     LINAGE IS 60 LINES
        WITH FOOTING AT FOOTLINE
        LINES AT TOP    TOPLINES
        LINES AT BOTTOM BOTTOMLINES.
 01  PRINT-LINE-OUT        PIC X(132).
      ...
 WORKING-STORAGE SECTION.
 01 LINAGE-STUFF.
    03 FOOTLINE    PIC 99 VALUE 45.
    03 TOPLINES    PIC 99 VALUE 0.
    03 BOTTOMLINES PIC 99 VALUE 6.
    ...
 PROCEDURE DIVISION.
      ...
 WRITE-DETAIL.
    WRITE PRINT-LINE-OUT FROM DETAIL-LINE
       AT EOP
          PERFORM
          ADD 1 TO PAGE-COUNTER
          MOVE PAGE-COUNTER TO PAGE-NUMBER
*         The next WRITE statement advances to the third
*         line after the one just written, leaving two lines
*         of spaces and printing on the third line.
          WRITE PRINT-LINE-OUT FROM PAGE-NUMBER-LINE
              AFTER ADVANCING 3 LINES
*         The next WRITE statement advances to the top of
*         the next page (issues a forms-control code to skip
*         to channel 1.)
          WRITE PRINT-LINE-OUT FROM DETAIL-HEADER
                AFTER ADVANCING PAGE
          MOVE SPACES TO PRINT-LINE-OUT
          WRITE PRINT-LINE-OUT
          END-PERFORM
    END-WRITE
    ...
```

WRITE for Line Sequential Files



record-name

is a logical record described in the File Section of the Data Division. The *record-name* can be qualified by the name of the file with which the record is associated. The data written is the current contents of *record-name*.

from-name

is the identifier of a data area whose contents are to be moved to the record specified by *record-name* before the WRITE occurs. *from-name* must specify a data area other than that specified by *record-name*. It also cannot specify an index data item.

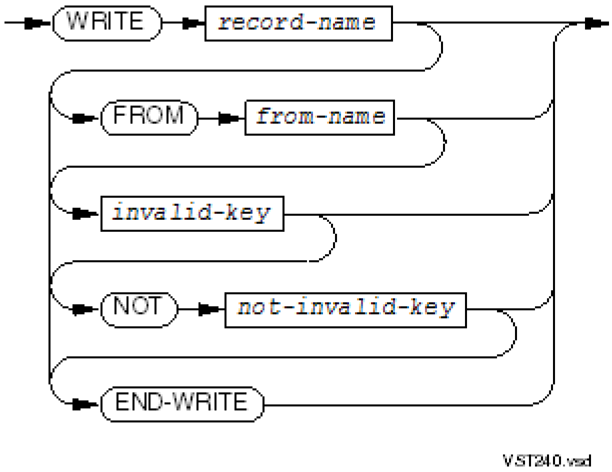
END-WRITE

ends the scope of the WRITE statement, causing the WRITE to be a delimited-scope statement. If the WRITE statement does not end with an END-WRITE phrase, the presence of the AT END-OF-PAGE, the NOT AT END-OF-PAGE, the INVALID KEY, or the NOT INVALID KEY phrase causes the WRITE statement to be a conditional statement, which ends at the next period separator.

Usage Considerations:

- See these usage considerations in WRITE for Sequential Files (page 494):
  - — Action of the WRITE Statement
  - — File-Status Data Item
  - — Reel-Swap Sequence for Multiple-Reel Tape Files
  - — Buffered Cache

WRITE for Relative, Indexed, and Queue Files



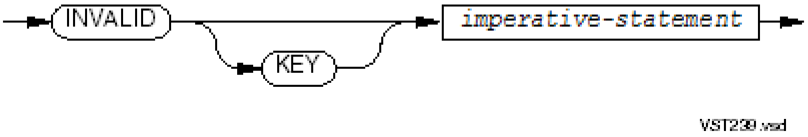
record-name

is a logical record described in the File Section of the Data Division. The *record-name* can be qualified by the name of the file with which the record is associated.

from-name

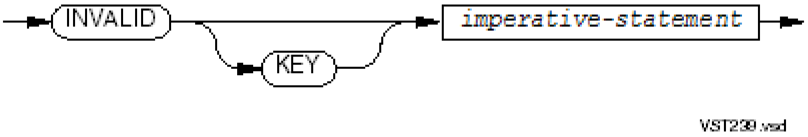
is the identifier of a data area whose contents are to be moved to the record specified by *record-name* before the WRITE occurs. *from-name* must specify a data area other than that specified by *record-name*. It also cannot specify an index data item.

invalid-key



executes *imperative-statement* when an invalid-key condition arises. This phrase is required if no USE statement is applicable for the file.

not-invalid-key



executes *imperative-statement* when no invalid-key condition arises.

END-WRITE

ends the scope of the WRITE statement, causing the WRITE to be a delimited-scope statement. If the WRITE statement does not end with an END-WRITE phrase, the presence of the INVALID KEY or the NOT INVALID KEY phrase causes the WRITE statement to be a conditional statement, which ends at the next period separator.

Usage Considerations:

- Action of the WRITE Statement

  The write operation releases a logical record to the run-time routines for inclusion in the file. For a file of relative or indexed organization, the order of the records in the file is determined by the relative record number or the prime record key, respectively.

  In general, the logical record consists of the value of the data item specified by *record-name*. Records in a relative, indexed, or queue file can be any length from zero up to the maximum length specified when the file was created.

  Release of a logical record does not necessarily imply immediate transmission to the file. For example, the actual transmission of a logical record to the file can be deferred until some time after completion of the WRITE statement.

- File-Status Data Item

  If the file has an associated file-status data item, execution of the WRITE statement always assigns an appropriate I-O status code. The value “00” reports a successful write operation.

The I-O status codes that result from an unsuccessful write operation are:

I-O Status Code	Unsuccessful Write Operation
“21”	The file is defined to have indexed organization and sequential access mode, and the prime record key value of the logical record is less than or equal to the prime record key value of the most recently released record.
“22”	One of: — The relative record number to be associated with the logical record equals that of a record that already exists in the file (random access or dynamic access only). — The prime record key value of the logical record equals that of a record that already exists in the file (random access or dynamic access only). — An alternate record key value of the logical record equals that of a record that already exists in the file and duplicate values are not allowed for that key all access modes.
“24”	A boundary violation exception exists when execution of a WRITE statement would require exceeding the record storage capacity of the file. The logical record is not released.
“30”	The write operation failed due to non-COBOL causes. The specified record might or might not have been written.
“44”	When the file is described with the RECORD VARYING clause, the logical record size must be neither greater than the maximum nor less than the minimum number of character positions specified in that clause. This requirement is not met, and the logical record is not released.
“48”	Either the file has sequential organization, and is not open in I-O, OUTPUT, or EXTEND mode; or the open mode is I-O, and the file device is not a terminal, process, or $RECEIVE.

- Variable-Length Records

  An Enscribe structured file can have variable-length records. See READ for Sequential or Dynamic Access (page 414).

  When you write to a file that is defined as having variable-length records, the length of the record written depends on whether the file is declared with

  ```
  RECORD CONTAINS rec-1 TO rec-2 CHARACTERS
  ```

  or

  ```
  RECORD IS VARYING IN SIZE FROM rec-1 TO rec-2 CHARACTERS
    DEPENDING ON rec-size
  ```

  form of the RECORD CONTAINS clause.

  In the former case, the length of the record specified in the WRITE statement is the number of characters written to the file system file.

  In the latter case, the length of the record written is the value present in the `rec-size` data item specified in the DEPENDING clause at the time the WRITE statement is executed.

- Sequential Access

  For files of relative organization, records are released in relative number order, beginning at 1. When the file has an associated relative key data item, that item is set to the current record number at each successful release.

  For files of indexed organization, the program is responsible for setting the prime record key data item to a desired value prior to the execution of a WRITE statement for that record. In the case of sequential access, the records must be released in ascending order of prime record key value.

- Relative and Dynamic Access

  For files of relative organization, the relative key data item must be set to the desired record number before the WRITE occurs. This can be done in either of two ways:

  - Set the key item to the relative record number.
  - Set the key item to -1 to have the record written at the end of the file or to -2 to have the record written in any available file position.

  When the second method is used, the relative key data item must be defined as a signed integer numeric item; at the completion of the write operation, the relative key data item is set to the file position number used.

  For files of indexed organization, the program is responsible for setting the prime record key data item to a desired value prior to the execution of a WRITE statement for that record.

- Invalid-Key Conditions

  This condition occurs for relative files when any of these conditions is true:

  - The RELATIVE KEY item defines an existing record or the alternate key is duplicated without DUPLICATES option (I-O status code “22”).
  - The file is physically full or the relative key data item points outside the file’s boundary (I-O status code “24”).
  - The value intended for the RELATIVE KEY item does not fit into the RELATIVE KEY item.

  This condition occurs for indexed or queue files when any one of these is true:

  - Keys are not in ascending order in sequential access (I-O status code “21”).
  - The prime key is duplicated, or the alternate key is duplicated without DUPLICATES option (I-O status code “22”).
  - The file is physically full, or the key specifies a point outside the file’s boundary (I-O status code “24”).

Example 10-72 INVALID KEY Phrase

```
ENVIRONMENT DIVISION.
     ...
INPUT-OUTPUT SECTION.
FILE-CONTROL.
    SELECT MASTER-REL-FILE
    ASSIGN TO "$MARKT.PLATZ.DT"
    ORGANIZATION IS RELATIVE
    ACCESS MODE IS RANDOM
    RELATIVE KEY IS VKF-NR
    FILE STATUS IS M-R-STATUS.
...
DATA DIVISION.
FILE SECTION.
...
FD MASTER-REL-FILE
    LABEL RECORDS ARE OMITTED
    RECORD CONTAINS 180 TO 250 CHARACTERS.
01  DOMESTIC-SALE          PIC X(180).
01  FOREIGN-SALE           PIC X(250).
     ...
WORKING-STORAGE SECTION.
01  VKF-NR PICTURE 97.
01  J-W-D.
      03 NATION      PIC X(15).
      03 SUBDIVISION PIC X(15).
      03 CITY        PIC X(15).
       ...
PROCEDURE DIVISION.
     ...
AUSLAND.
     ...
   ADD 1 TO VKF-NR
   WRITE FOREIGN-SALE FROM J-W-D
         INVALID KEY PERFORM RECOVER-M-R-BAD-KEY
   END-WRITE
   ...
```

11 Source Text Manipulation

Source manipulation comprises the COPY statement, COPY libraries, and the REPLACE statement. You can use the COPY and REPLACE statements in any division of the source program.

The COPY statement summons source text from a COPY library and delivers merged text to the compiler. Its optional REPLACING phrase replaces every occurrence of a specified portion of library text with a specified portion of new text when it copies library text into a source program. You can specify more than one such replacement pair in a REPLACING phrase. With or without the REPLACING phrase, the COPY statement does not change the COPY library or the source file.

The COPY statement appears in the listing unless a NOSHOWCOPY, NOLIST, or SUPPRESS directive is active. The library text appears in the listing unless a NOLIST or SUPPRESS directive is active.

In the Guardian environment, a COPY library is a file in the EDIT format. It contains one or more sections of zero or more text lines, each preceded by a SECTION directive line and succeeded by either another SECTION directive line or the end of the file.

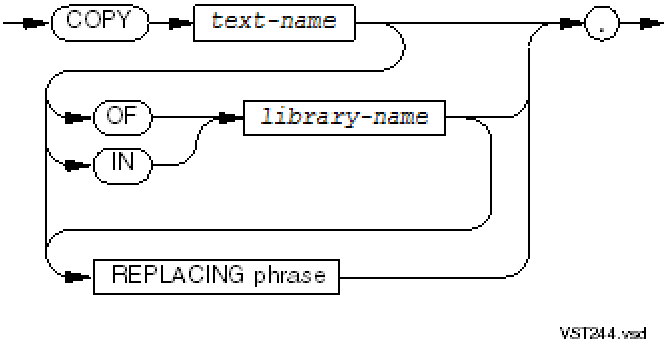
In the OSS environment, a COPY library is an ASCII text file.

The REPLACE statement replaces source program text. It is useful for establishing names for constants and abbreviations for words and phrases and for overcoming the introduction of new reserved words into the language.

- COPY Statement
- REPLACING Phrase
- COPY Libraries
- REPLACE Statement

COPY Statement

COPY summons source text from a file set up as a COPY library. In many systems, one section of code or data is common to several programs. Such a section can be written once, kept in a COPY library, and inserted into each program at compile time by COPY statements. In HP COBOL, the COPY statement is a mechanism for summoning text that is managed by the Data Definition Language (DDL) compiler. (Another such mechanism is the SOURCE directive.)



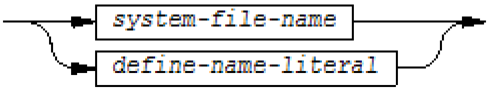
COPY

is a reserved word that cannot be split across source program lines.

text-name

is the name of a section in a COPY library file. It is a COBOL word (1 to 30 letters, digits, and hyphens but not all digits) that appears in a SECTION directive in the file.

library-name



VST245.vsd

is the name of the COPY library file that contains the text to be copied.

If *library-name* is not specified, a default library file is selected:

- In the Guardian environment:
  1. If a COPY library is named in the COBOL85 command (see Starting a Compilation (page 538)), then that file is the default library file.
  2. If the condition in item 1 is not true and the file COPYLIB exists on the current volume and subvolume, then that COPYLIB file is the default library file.
  3. If the conditions in item 1 and item 2 are not true and a DEFINE of class SEARCH named =_SOURCE_SEARCH exists, then the default library file is the first file named COPYLIB in the search list defined by =_SOURCE_SEARCH. (See =_SOURCE_SEARCH (page 538).)
  4. If none of the preceding conditions are true, the COPY statement is in error.
- In the OSS environment, the -Wcopylib flag specifies the default COPY library.

If *library-name* is specified, these definitions apply:

system-file-name

is the name of an EDIT file (code 101), OSS ASCII text file (code 180), or PC file. If *system-file-name* is a PC file name or does not begin with a dollar sign ($), backward slash (\), or number sign (#), then it must be enclosed in quotation marks unless it forms a COBOL word. For more information about operating system file names, see the *Guardian Procedure Calls Reference Manual*.

define-name-literal

is a nonnumeric literal that represents the name of a DEFINE of class MAP that is associated with an EDIT file. Quotation marks must enclose *define-name-literal*. For more information on DEFINE names, see DEFINEs (page 601).

If the file-system file name identified by *define-name-literal* is qualified (the subvolume and any other qualifiers are specified), then that specific file is the COPY library. If the file-system file name identified is not qualified, the file selected as the COPY library depends on the presence or absence of a DEFINE of class SEARCH named =_SOURCE_SEARCH. If =_SOURCE_SEARCH exists, the search list that it specifies is used to locate the COPY library. If =_SOURCE_SEARCH does not exist, only the current volume and subvolume is searched for the COPY library.

REPLACING phrase

specifies text-words for the compiler to replace with other specified text-words when it copies text. See REPLACING Phrase.

Usage Considerations:

- Where COPY Statements Can Be Used

  With these exceptions, a COPY statement can occur anywhere in the source text that a character-string or separator can occur:

  — A COPY statement cannot appear within the body of another COPY statement.

  — A COPY statement cannot appear within source text that is introduced by another COPY statement or a REPLACE statement.

- If the word COPY appears either in a comment-entry or in a place where a comment-entry can appear, it is considered part of the comment-entry, not as the keyword that begins a COPY statement.
- The keyword COPY must be preceded by a space character, unless it immediately follows the indicator field.
- A COPY statement cannot appear on the same line as an SQL/MP or SQL/MX statement.
- All four characters of the keyword COPY must appear on the same source text line. The remainder of the statement can extend across additional program text lines, in accordance with the continuation conventions of the reference format.

- How the Compiler Processes a COPY Statement

  The compiler processes a source program that includes COPY statements as though the compiler included a preprocessor that performs these operations in this order:

  1. Locates each COPY statement
  2. Replaces the COPY statement with the appropriate (and possibly edited) library text
  3. Passes the resulting text to the compiler proper

  The effect of processing a COPY statement is to copy the specified library text into the source program. The copied text logically replaces the entire COPY statement, beginning with the keyword COPY and ending with the punctuation character period.

  If the source line on which the COPY statement begins contains other text preceding the word COPY, the compiler attempts to combine that portion of the line with the first library text line.

  If the source line on which the COPY statement ends contains other text following the terminating period, the compiler attempts to combine that portion of the line with the last library text line.

- Compiler Directives and the COPY Statement

  A compiler directive line cannot appear between the keyword COPY and the period separator that terminates the statement, unless it is part of pseudo-text (see REPLACE Statement).

- Compiler Directives in Library Text

  Library text can include compiler directives, which the compiler obeys when it analyzes the copied text.

  If a TANDEM or ANSI format directive occurs as a qualifier on the SECTION directive in the library file, the specified formatting is active only for the copied text. When the copying is complete, the previous formatting is again active.

  If a TANDEM or ANSI format directive occurs within the library text, its effect (which overrides the effect of any format directive that is a qualifier on the SECTION directive) persists until the copying is complete.

- Debugging Lines and the COPY Statement

  If the COPY statement itself begins on a debugging line (a line that has a `D` or `d` in the indicator field), all text that the copy operation introduces into the source program, except comment and compiler directive lines, appears on debugging lines.

  Because debugging lines and continuation lines are mutually exclusive, the compiler cannot introduce a continued text-word into the source text when the preceding rules require it to appear on debugging lines.

- Sensitivity to Reference Format

  Because the compiler analyzes pseudo-text without the benefit of any contextual information, you must observe the COBOL reference format rules carefully. In particular, the compiler does not identify a comma, semicolon, or period character as a separator unless it is followed by at least one space character.

Similarly, the compiler interprets the character sequence X/9 as one text-word (presumably a PICTURE character-string) rather than as three text-words; however, the compiler always considers a left parenthesis, right parenthesis, or colon character to be a separator unless it appears within a nonnumeric literal.

Another reason to carefully observe the reference format rules is that, when the compiler is performing replacement editing, it analyzes library text without the benefit of any contextual information.

- Including COPY Statements in a Listing

  The SHOWCOPY directive determines whether the COPY statement itself appears in the listing (see SHARED (page 576)). If you do not specify NOSHOWCOPY, NOLIST, or SUPPRESS, the compiler lists the COPY statement as a comment followed by the copied text.

In Example 11-1, EMPLOYEE-DETAIL of the COPY statement is not qualified because the COPY library is named COPYLIB and resides on the current volume and subvolume for the compile process.

Example 11-1 COPY Statement

Contents of COPY library COPYLIB:

```
?SECTION EMPLOYEE-DETAIL
01  EMP-DATA-IN.
    05  EMP-NO       PIC X(05).
    05  EMP-NAME     PIC X(20).
    05  DEPT         PIC X(03).
    05  JOB-CLASS    PIC X(05).
    05  HOURLY-RATE  PIC 9(3)V99.
    05  DEDUCTIONS   PIC 9(3)V99.
    05  SALARY       PIC 9(7)V99.
```

Source COBOL code:

```
  ...
DATA DIVISION.
FILE SECTION.
FD  EMP-MASTER
COPY EMPLOYEE-DETAIL.
FD  LIST-OUT
    ...
```

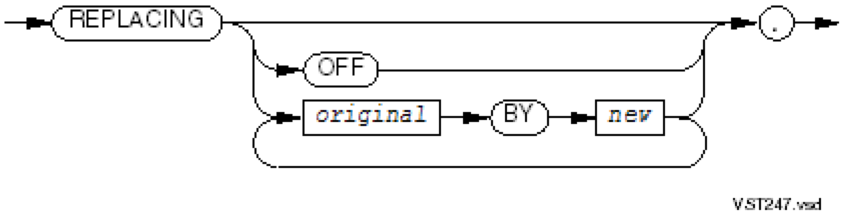
Source listing produced by compiler (lines from the COPY library are marked by < in the compilation listing):

```
          ...
      DATA DIVISION.
      FILE SECTION.
      FD  EMP-MASTER
     *COPY EMPLOYEE-DETAIL.
<     01  EMP-DATA-IN.
<         05  EMP-NO       PIC X(05).
<         05  EMP-NAME     PIC X(20).
<         05  DEPT         PIC X(03).
<         05  JOB-CLASS    PIC X(05).
<         05  HOURLY-RATE  PIC 9(3)V99.
<         05  DEDUCTIONS   PIC 9(3)V99.
<         05  SALARY       PIC 9(7)V99.

      FD  list-out
           ...
```

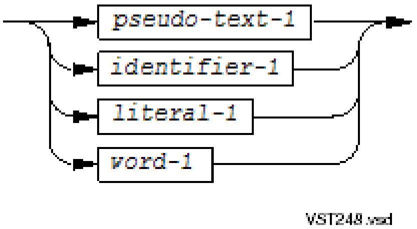
REPLACING Phrase

The REPLACING phrase of the COPY statement directs the compiler to replace every occurrence of a portion of library text with a replacement portion when it copies library text into a source program.

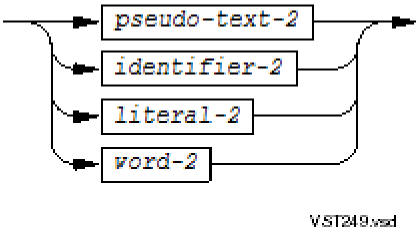
You can specify more than one pair of such portions for the compiler to replace when it executes a COPY statement. The compiler searches for each original portion in the order in which you declared them in the REPLACING phrase.



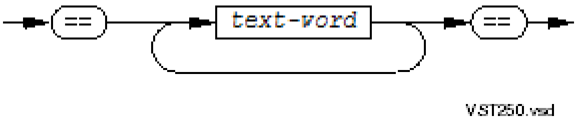
original



new

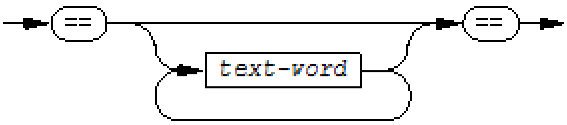


pseudo-text-1



contains at least one *text-word* other than a comma or semicolon separator. A character-string within *pseudo-text-1* can continue on the next line, but both characters of the pseudo-text delimiter (==) must be on the same line.

pseudo-text-2



VST251.vsd

can be null. A character-string within *pseudo-text-2* can continue on the next line, but both characters of the pseudo-text delimiter (==) must be on the same line.

text-word

is any character-string or separator, except space.

identifier-1, identifier-2

are identifiers of data items.

literal-1, literal-2

are literals (but not national literals). Neither can be a concatenation expression.

word-1, word-2

are COBOL words.

Usage Considerations:

- Effect of REPLACE on Literals

  The REPLACE statement does not affect literals; for example,

  ```
  REPLACING ==Year== BY ==Month==
  ```

  does not change the original *text-word*

  ```
  "End of Year"
  ```

  to the new *text-word*

  ```
  "End of Month"
  ```

- Matching Text in a COPY Library

  The compiler searches for portions of text in the order you specify in the REPLACING phrase. If you want to replace a sequence of text-words with something and a particular text-word with something else, specify the sequence first. For example, if you want to replace all occurrences of "XXX" with "BALANCE" and all occurrences of "XXX OF YYY" with "BALANCE OF BUDGET-REC," the REPLACING phrases must be in this order:

  ```
  REPLACING "XXX OF YYY" BY "BALANCE OF BUDGET-REC"
            "XXX" BY "BALANCE"
  ```

  If you specify "XXX" first, the compiler first changes the "XXX" to "BALANCE," producing "BALANCE OF YYY;" therefore, it can never find the sequence "XXX OF YYY."

  For purposes of matching, the compiler handles *identifier-1*, *word-1*, and *literal-1* as pseudo-text containing only *identifier-1*, *literal-1*, or *word-1*, respectively.

- Comparison Operation

  The compiler determines which characters of source text to replace by comparing *pseudo-text-1*, *identifier-1*, *literal-1* to text-words in the COPY library. This is how the comparison operation works:

  1. The compiler copies any separator comma, semicolon, and space that precedes the leftmost library text-word into the source program.
  2. Starting with the leftmost library text-word, the compiler compares all the text-words in the first *pseudo-text-1*, *identifier-1*, *literal-1*, or *word-1* to an equal number of contiguous text-words in the library. During the comparison, the compiler handles each occurrence of a separator comma or semicolon and each sequence of one

or more space separators as a single space. The compiler ignores any comment or directive line in the library text or in *pseudo-text-1*. The REPLACING phrase operand matches the library text if the two sequences of text-words are equal, character for character.

3. If no match occurs, the compiler repeats the comparison with each successive *pseudo-text-1*, *identifier-1*, *word-1*, or *literal-1*, if any, in the REPLACING phrase until a match occurs or until it has compared each operand in the phrase to the leftmost library text-word or text-words.
4. If the compiler compares all the REPLACING phrase operands without finding a match, it copies the leftmost library text-word into the source program. Then the compiler uses the next successive library text-word as the leftmost text-word and starts the comparison cycle again with the first *pseudotext-1*, *identifier-1*, *literal-1*, or *word-1* in the REPLACING phrase.
5. Whenever a match occurs between *pseudo-text-1*, *identifier-1*, *literal-1*, or *word-1*, the compiler copies the corresponding *pseudo-text-2*, *identifier-2*, *literal-2*, or *word-2* into the source program. The compiler places the text-words into the program according to the rules of the reference format specified in the COPY statement or, if not specified, the reference format active before the statement. Then the compiler uses the library text-word immediately following the matching library text as the leftmost text-word and starts the comparison cycle again with the first *pseudo-text-1*, *identifier-1*, *literal-1*, or *word-1* in the REPLACING phrase.
6. The comparison operation ends after the rightmost text-word in the library text participates either in a match or as a leftmost text-word in a complete comparison cycle.

- Comment Lines in Replacement Text

  The compiler copies any comment line in *pseudo-text-2* into the source program unchanged.

- Debugging Lines

  You can put debugging lines in library text and in *pseudo-text-2* or in *pseudo-text-1*. Text-words within a debugging line participate in the comparison cycle as though the indicator area did not contain a *D* or *d*.

  If a portion of the library text is replaced by a *pseudo-text-2*, any text-words of the replacement text specified on debugging lines appear on debugging lines in the resulting source text.

  If a portion of the library text that begins on a debugging line is replaced, all text-words of the replacement text appear on debugging lines in the resulting source text.

  If library text specified on debugging lines is copied without replacement, it appears on debugging lines in the resulting source text.

  A debugging line cannot contain embedded SQL/MP or SQL/MX statements.

Example 11-2 COPY Statement With REPLACING Phrases

```
COPY CUSTOMER REPLACING ==FULL NAME== BY ==Able X. Baker==
                        ADDRESS BY STREET-ADDRESS.
```

COPY Libraries

A COPY library is either a file in the EDIT format or an OSS ASCII text file. Its text can be merged into a source program during compilation using a COPY statement. If the COPY statement contains a REPLACING phrase, specified portions of library text can be replaced by specified new text when the COPY statement copies the library text into the source program. With or without the REPLACING phrase, the COPY statement does not change the COPY library.

Specifying Library Names

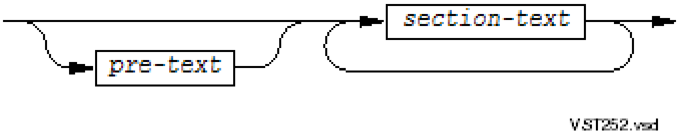
There are three places you can specify the name of the library from which a COPY statement is to collect text:

- The COPY statement itself can include an IN `library-name` phrase (see COPY Statement), in which the name can be a file-system file name or (in the Guardian environment) an alphanumeric literal containing a DEFINE name.
- The COBOL85 command that initiates the compilation can include a library name to be used whenever a COPY statement does not include a library name (see Starting a Compilation (page 538)). On a command line, the library name parameter is either a file-system file name or (in the Guardian environment) a DEFINE name, but neither can be enclosed in quotation marks.
- In the OSS environment, you can specify the default COPY library with the -Wcopylib flag.

If no name is specified on either the command line or the COPY statement, the compiler uses the name COPYLIB.

If *`library-name`* is not fully qualified with volume and subvolume, the current default volume or subvolume is used.

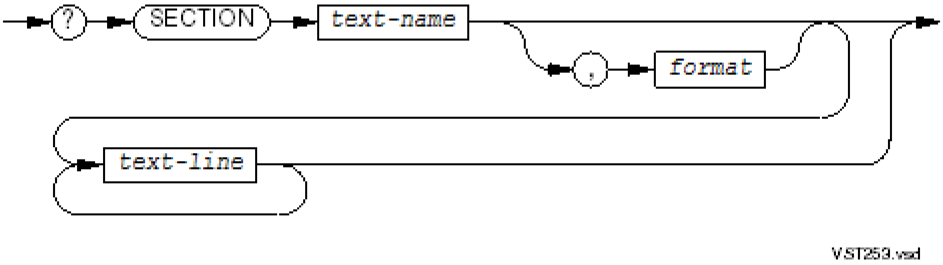
Library Format



`pre-text`

is one or more text lines. These lines are never copied. When a library file begins with *`pre-text`*, this text can be comments about the library's sections. The compiler examines *`pre-text`* only to look for a question mark (?) in column 1 followed by zero or more spaces followed by the directive COLUMNS (in any combination of cases). If it finds such a directive, it attempts to translate and use it.

`section-text`



SECTION

is the SECTION directive described in SECTION (page 574).

`text-name`

is the name of the section, the name that identifies the portion of the library file to be copied. It is a COBOL word (1 to 30 letters, digits, and hyphens but not all digits).

`format`

is the keyword TANDEM or ANSI.

`text-line`

is a line of source text. There is no limit on the number of such lines. No text line can begin with "?SECTION."

Usage Considerations:

- COLUMNS Directive

  The library can contain at most one COLUMNS directive. The COLUMNS directive must precede all SECTION directives and must appear alone in its compiler directive line. Furthermore, the line's question mark must always be in column 1 (even if the ANSI source text format applies).

- SECTION Directive

  Each SECTION directive must appear alone in a compiler directive line and the line's question mark must always be in column 1 (even if the ANSI source text format applies). Each SECTION directive demarcates an individual text unrelated to any other contents of the source library.

  A SECTION directive cannot appear anywhere in the text of a COPY statement (between the word COPY and its terminating period in the input stream) or a REPLACE statement (between the word REPLACE and its terminating period in the input stream); however, the compiler detects and reports its appearance within pseudo-text only if that pseudo-text is introduced into the source as the result of expanding the COPY statement or by the editing effects of the REPLACE statement.

  During program compilation, the compiler identifies a section by locating the SECTION directive whose `text-name` matches the `text-name` specified in the COPY statement. For more information, see SECTION (page 574).

- Copying Sections of Text

  The compiler copies text starting at the line after the SECTION directive line and continues until it recognizes another SECTION directive or reaches the end of the file.

- No COPY Statements in COPY Library Sections

  The COPY statement cannot be within a section of a COPY library.

- Reference Format

  The compiler assumes that the reference format of the library text is the same as that of the line containing the COPY statement, unless you specify a different format in the SECTION directive. (For information on Tandem format, see Reference Format for Source Program Lines (page 54). For information on ANSI format, see Chapter 17: ANSI Reference Format (page 711).)

  If you specify a reference format in a SECTION directive, that format is active only during execution of the COPY statement. After the compiler copies a section, it uses the reference format previously active.

- Compiler Directives in a COPY Library

  Source text in a COPY library can contain compiler directives, which the compiler obeys when it analyzes the copied text.

- Compiler Builds a Directory of the COPY Library

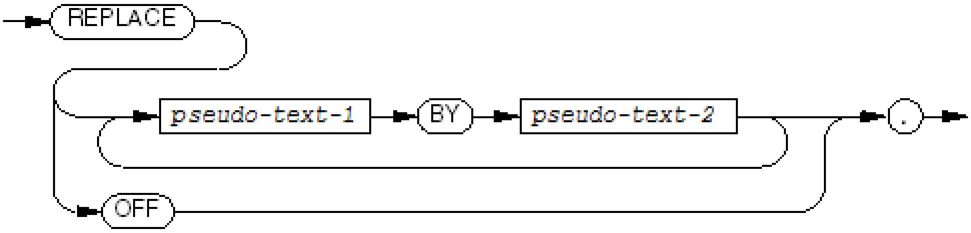
  The first time the compiler reads from a COPY library, it begins to build a directory for itself that enables it to locate the individual sections. It reads through the library, building its directory, until it finds the selected section. After copying it, it resumes normal compilation. For subsequent COPY statements, it checks its directory. If the chosen section is in the directory, the compiler can go right to it; otherwise, it resumes reading the library where it left off, and continues as before. If you put sections that most programs use in the front of the library and those that few programs use at the end, compilation speed is improved.

- Adding Source Text to a COPY Library

  Since COPY libraries are files in the EDIT format or OSS ASCII files, you can add a section of source text to a library by editing it. You can put a SECTION directive line and the source text that follows it before, between, or after sections already in the library.

REPLACE Statement

REPLACE substitutes zero or more words of pseudo-text for one or more words of pseudo-text.

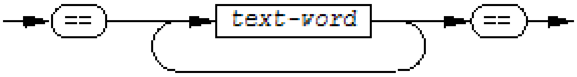


VST254.vsd

REPLACE

is a reserved word that cannot be split across source program lines.

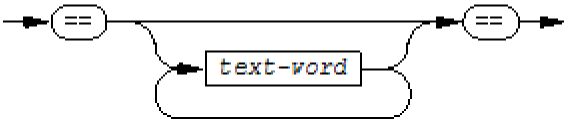
pseudo-text-1



VST250.vsd

contains at least one *text-word* other than a comma or semicolon separator. A character-string within *pseudo-text-1* can continue on the next line, but both characters of the pseudo-text delimiter (==) must be on the same line.

pseudo-text-2



VST251.vsd

can be null. A character-string within *pseudo-text-2* can continue on the next line, but both characters of the pseudo-text delimiter (==) must be on the same line.

text-word

is any character-string or separator except space. It cannot be a concatenation expression.

OFF

marks the end of the scope of replacement of the most recent REPLACE statement.

Usage Considerations:

- Comment Lines and Directive Lines in Pseudo-Text

  Pseudo-text can contain comment lines and directive lines (that is, the text-words in pseudo-text can be organized as comment lines and directive lines).

- Where REPLACE Statements Can Be Used

  With these exceptions, a REPLACE statement can occur anywhere in the source text that a character-string can occur:

  — A REPLACE statement cannot appear within the body of another REPLACE statement.

  — A REPLACE statement cannot appear within source text introduced by another REPLACE statement.

  — If the word REPLACE appears either in a comment-entry or in a place where a comment-entry can appear, it is considered part of the comment-entry, not as the keyword that begins a REPLACE statement.

  — The keyword REPLACE must be preceded by a period separator, except when it begins the first statement of a separately compiled program.

  — A REPLACE statement cannot appear on the same line as an SQL/MP or SQL/MX statement.

  All seven characters of the keyword REPLACE must appear on the same source text line. The remainder of the statement can continue across additional program text lines, in accordance with the continuation conventions of the reference format.

- Persistence

  A given occurrence of the REPLACE statement is active from the point at which you specify it until the next occurrence of a REPLACE statement or the end of the separately compiled program, respectively.

- REPLACE Statements and the Listing

  The listing the compiler produces shows the lines containing REPLACE statements like any other lines. When any replacement occurs, the image in the listing is that of the text after replacement.

- Order of Processing

  Conceptually, the compiler processes any REPLACE statements in a source program after it has processed any COPY statements.

- Comparison Operation

  The compiler determines which sequences of text-words to replace by comparing `pseudo-text-1` to text-words in the source text. This is how the comparison operation works:

  1. Starting with the leftmost source program text-word and the first `pseudo-text-1`, the compiler compares the word or words of `pseudo-text-1` with the equivalent number of contiguous text-words in the source program.
  2. During the comparison, the compiler handles each occurrence of a separator comma or semicolon and each sequence of one or more space separators as a single space. The compiler ignores any comment or directive line in the source text or in `pseudo-text-1`. The `pseudo-text-1` matches the source text if the two sequences of text-words are equal, character for character.
  3. If no match occurs, the compiler repeats the comparison with each successive occurrence of `pseudo-text-1`, if any, in the REPLACE statement until a match occurs or until it has compared each operand in the `pseudo-text-1` to the leftmost source program text-word or text-words.

4. If the compiler compares all the occurrences of `pseudo-text-1` without finding a match, it considers the next text-word of the source program as the leftmost source program text-word and starts the comparison cycle again with the first occurrence of `pseudo-text-1`.
5. Whenever a match occurs between `pseudo-text-1` and the source program text, the compiler replaces the matched text in the source program with the corresponding `pseudo-text-2`. The compiler then considers the source program text-word immediately following the rightmost text-word that participated in the match to be the leftmost source program text-word, and starts the comparison cycle again with the first occurrence of `pseudo-text-1`.
6. The compiler continues its comparison operation until either the rightmost text-word in the source program text (within the scope of the REPLACE statement) has participated in a match or has been considered as a leftmost text-word and participated in a complete comparison cycle.

- Comment Lines and Blank Lines in Replacement Text

  The matching operation ignores any comment lines or blank lines occurring in the source program text. The compiler determines the sequence of text-words in the source program text and in `pseudo-text-1` by the rules for reference format.

  The replacement operation copies comment lines or blank lines in `pseudo-text-2` into the source program text without change whenever it copies `pseudo-text-2` into the source program.

  The compiler does replace a comment line or blank line in source program text if that comment line or blank line appears within the sequence of text-words that match `pseudo-text-1`.

- Debugging Lines

  You can put debugging lines in `pseudo-text-1` or in `pseudo-text-2`. text-words within a debugging line participate in the comparison cycle as though the indicator area did not contain a `D` or `d`.

  If a portion of the apparent source text is replaced, any text-words of the replacement text specified on debugging lines appear on debugging lines in the final source text.

  If a portion of the apparent source text that begins on a debugging line is replaced, all text-words of the replacement text appear on debugging lines in the final source text.

  If the REPLACE statement itself begins on a debugging line, all replacement text, except comment and compiler directive lines, appears on debugging lines.

  Because debugging lines and continuation lines are mutually exclusive, the compiler cannot introduce a continued text-word into the source text when the preceding rules require it to appear on debugging lines.

- Sensitivity to Reference Format

  Because, when a REPLACE statement is in force, the compiler analyzes pseudo-text (and source text) without the benefit of any contextual information, you must observe the COBOL reference format rules carefully. In particular, the compiler does not identify a comma, semicolon, or period character as a separator unless it is followed by at least one space character.

  Similarly, the compiler interprets the character sequence X/9 as one text-word (presumably a PICTURE character-string) rather than as three text-words; however, the compiler always considers a left parenthesis, right parenthesis, or colon character to be a separator unless it appears within a nonnumeric literal.

  The REPLACE statement in Example 11-3 enables you to declare a name for a constant in your program, then use that name in various places in a program. While OFFICES could have been declared as a data-item for this purpose, SQ-FT-SIZE could not have.

Of course, with either EDIT or PS Text Edit (TEDIT), you could easily change all instances of reserved words in a source program to another word; but if your data dictionary contains any fields whose names have become reserved words, you might find the REPLACE statement a handy tool. The REPLACE operation occurs conceptually after the COPY operation, but before the remainder of the compilation.

Example 11-3 REPLACE Statement

```
DATA DIVISION.
REPLACE ==OFFICES== BY ==10==
        ==SQ-FT-SIZE== BY ==5==.
...
01 OFFS.
   03 OFFICE-INFO OCCURS OFFICES TIMES.
      05 DISTRICT    PICTURE 99.
      05 SQUARE-FEET PICTURE S9(SQ-FT-SIZE).
...
PROCEDURE DIVISION.
...
   PERFORM REPORT-OFFICE OFFICES TIMES.
...
```

12 Program Compilation

The compiler can run at a high PIN (a process identification number greater than 255) and can be requested by other processes running at high PINs.

The compiler accepts one source file as input. That source file can use SOURCE directives and COPY statements to read text from other source files. The source file that is input to the compiler and the source files from which it reads text can contain one or more source programs, each consisting of HP COBOL statements, comment lines, and compiler directives (instructions to the compiler). Compiler directives can also appear on the compiler command line.

Compiler directives specify the source format, control listing features, control selective compilation of portions of the source code, and request compilation options; therefore, they affect the output of the compiler. By default, if the source file has no errors, the compiler outputs a listing and an object file.

The NOLIST and SUPPRESS directives can suppress all or parts of the listing file.

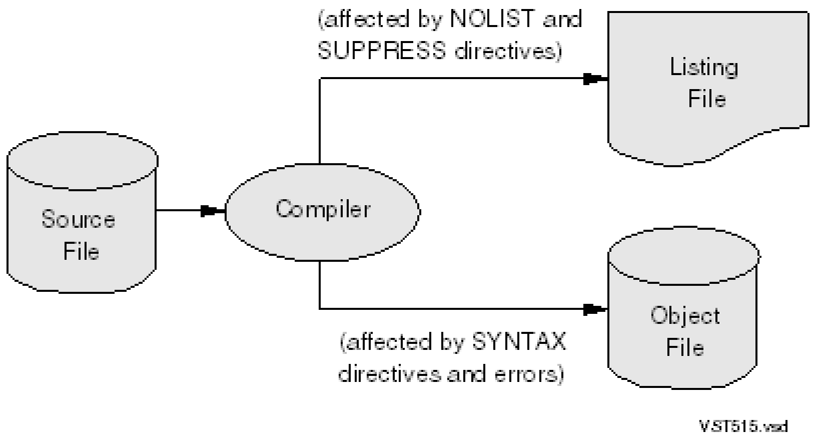
The ECOBOL compiler produces an object file that can either be input to the `eld` utility or executed.

The SYNTAX directive can suppress all or parts of the object file. Syntax errors in the source file can also suppress all or parts of the object file:

Syntax errors are in:	Compiler produces:
All programs in the compilation	No object file
Some programs in the compilation	No object file
No programs in the compilation	Complete object file

The rest of this section applies primarily to the Guardian environment. If you are compiling HP COBOL programs in the OSS environment, see Chapter 20: Using HP COBOL in the OSS Environment.

Figure 12-1 Compiler Input and Output



Compiler Input

Input to the compiler is always a single source file, but that source file can read text from other source files (using SOURCE directives and COPY statements). The source file that is input to the compiler and the source files from which it reads text can contain one or more source programs. One of these source programs can be a main program. The source programs can call each other, and, under some circumstances, they can also call non-COBOL programs. The source file that is input to the compiler is also called a compilation unit.

Main Programs

A main program is either compiled with the MAIN directive (see MAIN) or it has no Linkage Section (see Absent Linkage Section (page 193)). A loadfile must contain exactly one main program. It can also contain other programs. When the loadfile is executed, the NonStop operating system calls the main program. Then the main program can call the other programs and the other programs can call each other. The NonStop operating system does not call the other programs.

In Example 12-1, the MAIN directive makes A-PROGRAM a main program. If A-PROGRAM and A-FRIEND are compiled in the same compilation unit, A-FRIEND is not a main program. If A-FRIEND is compiled alone, it is a main program, because it does not have a Linkage Section.

Example 12-1 Main Program and Another Program

```
?MAIN A-PROGRAM
 IDENTIFICATION DIVISION.
 PROGRAM-ID. A-PROGRAM.
 ENVIRONMENT DIVISION.
 ...
 SPECIAL-NAMES.
   FILE "$SYSTEM.COBSYS.BFILES" IS B-FILE.
 DATA DIVISION.
 ...
 PROCEDURE DIVISION.
 ...
   CALL "A-FRIEND"
 ...
   CALL "B-FRIEND" IN B-FILE
 ...
   STOP RUN.
 END PROGRAM A-PROGRAM.
 IDENTIFICATION DIVISION.
 PROGRAM-ID. A-FRIEND.
 ENVIRONMENT DIVISION.
 ...
 SPECIAL-NAMES.
   FILE "$SYSTEM.COBSYS.BFILES" IS B-FILE,
   FILE "$SYSTEM.TALSYS.TFILES" IS T-FILE.
 DATA DIVISION.
 ...
 PROCEDURE DIVISION.
 ...
   CALL "HOME" IN BFILE
 ...
   ENTER TAL "SPECIALIST" IN T-FILE
 ...
   STOP RUN.
 END PROGRAM A-FRIEND.
```

Calling and Called Programs

Many languages have declarations that distinguish main programs from subprograms. COBOL does not use the term “subprogram.” Instead, it defines a program as a “calling program” if it contains a CALL or ENTER statement and defines a program as a “called program” if it is the object of a CALL or ENTER statement. A program can be both a calling program and a called program. The distinction is one of usage, not of explicit declaration. In Example 12-1, A-PROGRAM is a calling program and A-FRIEND is both a called program and a calling program.

If a called COBOL program is to receive parameters, it must have a Linkage Section (see Linkage Section (page 191)).

The statement a COBOL program uses to call another program depends on the language in which the called program was written.

Table 12-1 Statements for Calling Programs

Language of Called Program	Statement for Calling Program
HP COBOL	CALL
HP C	ENTER or X/Open CALL
HP C++	ENTER or X/Open CALL
pTAL	ENTER or X/Open CALL

Called programs are not required to be in the main program's compilation unit (see Compilation Units).

How an HP COBOL Program Calls a Non-COBOL Program

Any HP COBOL program can call a non-COBOL program with the ENTER statement. An HP COBOL program that was compiled with the PORT directive can also call a non-COBOL program with the X/Open CALL statement. The difference between the ENTER statement and the X/Open CALL statement is that the ENTER statement attempts to coerce the actual parameters into the types of the formal parameters, while the X/Open CALL statement reports an error if the types of the actual and formal parameters do not match.

Just as HP COBOL object programs can be read from an object file and included in the target file during the (optional) linking phase of ECOBOL compilation, object programs from other languages can also.

NOTE: You cannot put any combination of TNS, TNS/R, or TNS/E object files into a single object file.

Appendix B: Data Type Correspondence, shows the correspondence between HP COBOL data items and those of the other languages with which an HP COBOL program can interact.
HP COBOL index names are entirely internal to their own program; they cannot be written, read, or passed as parameters. Index data items in HP COBOL correspond to 32-bit integers.

Compilation Units

The source file that is input to the compiler is also called a compilation unit. A compilation unit contains one or more "separately compiled programs."A separately compiled program is a program whose source text can be submitted to the compiler independently of any other source text. Each such program can include nested programs, and any program can call other separately compiled programs.

Submitting a sequence of separately compiled programs to the compiler as a compilation unit (in a single compilation step) is called "stacked compilation."

The end of each separately compiled program is ordinarily marked by an END PROGRAM statement, although it can be marked with an ENDUNIT compiler directive with equivalent effect. The last (or only) program in a compilation unit does not require an END PROGRAM statement or an ENDUNIT directive. Separately compiled programs in a compilation unit can be in any order.

If there are compilation errors in some, but not all, programs in a compilation unit, the ECOBOL compiler produces no object file.

Example 12-2: Compilation Unit shows a compilation unit in which a main program and two called programs are included in the input file, contained in file CSOURCE.

The command to compile the compilation unit in Example 12-2: Compilation Unit with the ECOBOL compiler is:

```
ECOBOL /IN CSOURCE/ COBJECT
```

COBJECT is called the target file. It contains the object programs that the compiler produces.

Example 12-2 Compilation Unit

```
  IDENTIFICATION DIVISION.
  PROGRAM-ID.  CPGM1.
*                      CALLED PROGRAM 1.
  ...
  DATA DIVISION.
  ...
  LINKAGE SECTION.
  01 LS-NAMES.
     03 LS-A-NAME    PICTURE X(30).
     03 LS-B-NAME    PICTURE X(30).
  ...
  END PROGRAM CPGM1.

  IDENTIFICATION DIVISION.
  PROGRAM-ID.  CALLER.
*                      THE MAIN PROGRAM FOR THIS COMPILATION.
  ...
  DATA DIVISION.
  ...
  WORKING-STORAGE SECTION.
  01 WS-PARTS.
     03 WS-FIRST-PART        PICTURE X(30).
     03 WS-SECOND-PART       PICTURE X(30).
  01 WS-PRODUCER.
     03 WS-PRODUCER-NAME     PICTURE X(60).
     03 WS-PRODUCER-ADDRESS  PICTURE X(60).
  PROCEDURE DIVISION.
  CALL "CPGM1" USING WS-PARTS
  CALL "CPGM2" USING WS-PARTS WS-PRODUCER
  ...
  END PROGRAM CALLER.


  IDENTIFICATION DIVISION.
  PROGRAM-ID.  CPGM2.
*                      CALLED PROGRAM 2.
  ...
  DATA DIVISION.
  ...
  LINKAGE SECTION.
  01 LS-SUBASSY.
     03 LS-SUBASSY-1  PICTURE X(30).
     03 LS-SUBASSY-2  PICTURE X(30).
  01 LS-MFGR.
     03 LS-MFGR-NAME  PICTURE X(60).
     03 LS-MFGR-ADDR  PICTURE X(60).
  ...
```

Called programs are not required to be in the main program's compilation unit. They can reside in their own files and be compiled to produce separate object files. Such object files need not be loadfiles. They can exist solely as resources for binding or linking (see Object File Creation (page 533)).

Example 12-3: Calling Programs That Are in a Separate File calls program units that are in a separate file, C-ARCHIVE. When the main program is compiled, the two called programs are included in the resulting target file.

Example 12-3 Calling Programs That Are in a Separate File

```
PROGRAM-ID. CALLER.
  ...
SPECIAL-NAMES.
  FILE "$MYVOL.MYSUB.COBJECT" IS C-ARCHIVE.
    ...
  CALL "CPGM1" IN C-ARCHIVE
    ...
  CALL "CPGM2" IN C-ARCHIVE
```

Compilation Details

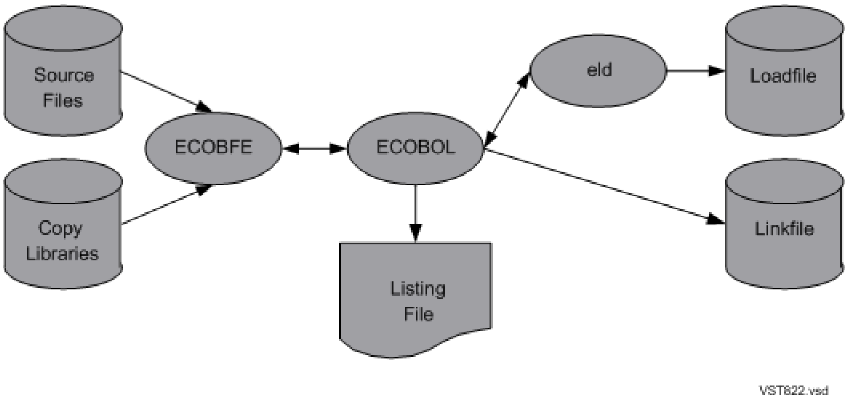
Processes Involved in Compilation

The ECOBOL compiler consists of a driver process, ECOBOL, and a sequence of subordinate processes.

For a successful compilation, the ECOBOL compiler driver calls ECOBFE, using temporary files to pass information between them. If you specify the RUNNABLE or SEARCH directive when creating a loadfile (CALL-SHARED) or a DLL library (SHARED), the ECOBOL compiler also calls the `eld` utility.

The symbol table information in the object file can be used later by the symbolic debuggers.

Figure 12-2 ECOBOL Compilation of Single Program Unit



Temporary File Placement

The ECOBOL compilers and their supporting processes create and use temporary files.

By default, the compiler creates its temporary files on the current default volume. If called by the compiler, the process ECOBFE and the linker create their temporary files on the current default volume.

The PARAM SWAPVOL command specifies the volume on which the compiler and its processes will create temporary files (if possible). It does not determine where the operating system creates the compiler’s own swap file. For details, see PARAM SWAPVOL.

The RUN option SWAP specifies one volume for both the temporary files that the compiler and its processes will create and for the compiler’s own swap file. For more information on SWAP and other RUN options, see the *TACL Reference Manual*.

CALL and ENTER Statement Processing: Overview

Any TNS/E HP COBOL program can call TNS/E HP COBOL programs with CALL statements. Any HP COBOL program can call non-COBOL programs with ENTER statements. An HP COBOL program that compiled with the PORT directive can also call non-COBOL programs with X/Open CALL statements.

In the CALL statement, the called program must be a COBOL program. It can be part of the same compilation unit as the calling program, or it can be an external reference to a COBOL program outside the compilation unit (but in the same object file). All ENTER statements cause external references, because they call programs compiled from source languages other than COBOL.

To generate the proper code to call a separately compiled program, the compiler needs a description of the parameter list of that program. Each CALL or ENTER statement can include qualification that specifies where the compiler must find the program. Although the compiler must search for unqualified programs, qualification significantly restricts the activity of the compiler.

Qualified References

Each CALL or ENTER statement can include a *`file-mnemonic`* that identifies the object file where the linker must look for the called program, resolving the external reference and including the called program's object code in the target file. You must associate this *`file-mnemonic`* with the object file's file-system file name in the SPECIAL-NAMES paragraph in the Environment Division that governs the calling program. (Either the calling program is not nested, and contains an Environment Division; or the calling program is nested, and its outermost containing program contains an Environment Division.)

If the compiler does not find the program in the specified object file, it reports an error and creates no code or data blocks for the program unit. Example 12-1 contains three examples of qualified references.

How the Compiler Resolves Unqualified References

When a program name is not qualified and the program it names is not in the compilation unit (such as the call to A-FRIEND in Example 12-1) the compiler resolves the reference in one of these ways:

- If you provided one or more search lists, the compiler examines those files in sequence for program names that correspond to the external references.
- If you did not provide search lists, or if the compiler did not find the necessary external references on them:

You create primary and tertiary search lists with the SEARCH, LIBRARY, and CONSULT directives, respectively. (The ECOBOL compiler does not recognize the LIBRARY directive or have a user library, the secondary search list.) You can also use the predefined SEARCH DEFINEs to specify one or more subvolumes to be searched for unqualified files (see Predefined SEARCH DEFINEs).

For more information, see:

- Primary Search List
- Tertiary Search List
- ECOBEXT File
- ECOBEX0 and ECOBEX1 Files

Primary Search List

SEARCH directives define the primary search list, an ordered list of object files. Each SEARCH directive adds one or more files to the primary search list. The compiler adds files to the search list in the order that they appear in the SEARCH directives. When trying to resolve unqualified external references, the compiler searches the files of the primary search list in the order that they appear. If the compiler resolves an external reference from the primary search list, the object program that the compiler finds is bound into the target file.

Tertiary Search List

CONSULT directives define the tertiary search list. Each CONSULT directive adds one or more files to the tertiary search list. The compiler adds files to the search list in the order that they appear in the CONSULT directives. When trying to resolve unqualified external references, the compiler searches the files of the tertiary search list in the order that they appear. If the compiler resolves an external reference from the tertiary search list, the object program that the compiler finds is not bound into the target file. Programs resolved from the tertiary search list do not have to be in the system library. They can be elsewhere and can be bound in later or can be referenced in a run-time library.

ECOBEXT File

If you did not provide search lists, or if the compiler did not find the necessary external references on them, in ZCOBDLL, or in ZCREDLL, it uses the ECOBEXT file to perform parameter validation, but does not import any object code. Programs resolved from ECOBEXT are available to the program at load time.

When you install the ECOBOL compiler, the ECOBEXT file is stored on the subvolume $SYSTEM.SYSTEM. If you move the ECOBEXT file to another subvolume, use the CONSULT directive to tell the compiler where to find it.

NOTE: The preceding paragraph applies only to the NonStop system. For the locations of files on the PC, see NonStop COBOL for TNS/E (ETK) (page 983).

ECOBEX0 and ECOBEX1 Files

Each RVU of the ECOBOL compiler includes three files: ECOBEX0, ECOBEX1, and ECOBEXT. These represent the most recent, next most recent, and third most recent RVUs of the Guardian environment, respectively. If you need access to a routine that was added (or to a parameter that was added to an existing routine) in the most recent version of the Guardian environment, include a CONSULT directive specifying ECOBEX0 in your compilation. The compiler then validates all calls of Guardian system routines from ECOBEX0 instead of from ECOBEXT.

When you install the ECOBOL compiler, the ECOBEX1 and ECOBEXT files are stored on the subvolume $SYSTEM.SYSTEM. If you move either file to another subvolume, use the CONSULT directive to tell the compiler where to find it.

NOTE: The preceding paragraph applies only to the NonStop system. For the locations of files on the PC, see NonStop COBOL for TNS/E (ETK) (page 983).

CALL and ENTER Statement Processing: Detailed Explanation

In processing CALL and ENTER statements, the compiler attempts to:

1. Find the called program (see Finding the Called Program) or the entered program (see Finding the Entered Program)
2. Validate the parameters in the calling program against those expected by the called program (see CALL (page 303) and ENTER (page 330))
3. Generate any necessary instructions to present the parameters from the calling program in the form that the called program expects (see Presenting Parameters to the Called Program)
4. Generate any necessary instructions to deliver a returned value (from the GIVING clause of an ENTER statement) in the form that the calling program expects (see Delivering the Returned Value)
5. Bind or link the object code for the called program into the target file or postpone the binding or linking until the first time the loadfile is loaded for execution (see Linking the Object Code)

Finding the Called Program

When the compiler reaches the end of a separately compiled program, it identifies all program names that were explicitly referenced by CALL statements in that program. If a CALL statement specified an identifier instead of an explicit name, the compiler cannot validate the parameters and does not try to locate the program for that CALL. You are responsible for the conformance of the parameters.

A compilation source text consists of one or more separately compiled HP COBOL programs. Each such program can include nested programs. Suppose separately compiled program S includes program P, which contains the statement CALL A. The compiler follows this procedure:

1. If program P directly contains a program named A, the compiler always chooses that program.

   In Example 12-4, the compiler issues a warning if a *file-mnemonic* appears.

2. If program S contains a common program named A, and also contains (directly or indirectly) program P, then if P calls A, the compiler chooses the common program A.

   In Example 12-5, the compiler issues a warning if a *file-mnemonic* appears. If common program A contains program P (directly or indirectly), the compiler leaves P's reference to A unresolved, and item 4 applies.

3. If neither item 1 or item 2 apply, and a previously seen separately compiled program has the name A, the compiler chooses that program.
4. If item 1, item 2, and item 3 do not apply, the compiler conducts a search for program A.

   If *file-mnemonic* appears, the compiler tries to find a separately compiled program unit in the file associated with the *file-mnemonic*. If the compiler cannot find such a program unit, it reports an error and delivers no object code for program S to the target file.

   If the compiler finds no separately compiled program A in the object files of the search lists, the compiler expects a separately compiled program A to appear somewhere within the source text, and also expects all of program A's parameters to have EXTENDED-STORAGE access mode (the default).

   If the compiler does find a program unit for A in the search lists, it resolves the reference with the first such program unit in the list. If the compiler discovers a separately compiled program A later in the source text, it replaces its previous resolution with the new one. If the new program A and the original program A have different numbers of parameters, or if their parameters have different access modes, the compiler reports an error and delivers no object code for program A to the target file.

5. When the compiler reaches the end of the source text, it issues a warning if it was unable to resolve any program references. You must supply the missing programs (using the linker) before you execute the loadfile.

Example 12-4 For Step 1 of Finding the Called Program

```
IDENTIFICATION DIVISION.
PROGRAM-ID. P. ...
PROCEDURE DIVISION. ...
CALL A ...
...
   IDENTIFICATION DIVISION.
   PROGRAM-ID. A.
   ...
   END PROGRAM A.
   IDENTIFICATION DIVISION.
   PROGRAM-ID. B. ...
   END PROGRAM B.
END PROGRAM P.
```

Example 12-5 For Step 2 of Finding the Called Program

```
IDENTIFICATION DIVISION.
PROGRAM-ID. S. ...
PROCEDURE DIVISION. ...
   IDENTIFICATION DIVISION.
   PROGRAM-ID. A COMMON. ...
   END PROGRAM A.
   IDENTIFICATION DIVISION.
   PROGRAM-ID. P. ...
     CALL A ...
   END PROGRAM P.
END PROGRAM S.
```

Finding the Entered Program

When it encounters an ENTER or X/Open CALL statement, the compiler follows this procedure:

1. If the compiler has already resolved the reference to the specified *routine-name* during the current compilation, the compiler resolves the current reference to that program.

2. If the compiler has not yet encountered the `routine-name` during the current compilation, the compiler conducts a search for the program:
   a. If `file-mnemonic` appears, the compiler tries to find a program with the specified name in the file associated with `file-mnemonic`. If the compiler finds the program, it verifies that `language` is the language in which the program was written. If the compiler does not find such a program, it reports an error.
   b. If no `file-mnemonic` appears:

      The compiler looks for the program in these files in this order:

      i. Primary search list
      ii. Tertiary search list
      iii. ZCOBDLL
      iv. ZCREDLL
      v. ECOBEXT
   c. If the compiler cannot find the program in any of the places discussed earlier, it reports an error and delivers no object code for the calling program.

NOTE: If `routine-name` specifies an HP C or HP C++ function whose name includes lowercase letters, the call fails, because the compiler automatically converts all names to uppercase.

Table 12-2 summarizes the preceding information.

Table 12-2 Resolution of External References

	External reference found in ...				
Form *	**Current Source File**	**Primary Search List**	**Tertiary Search List**	**ECOBEXT, ZCOBDLL, or ZCREDLL**	**Target File Created?**
CALL “P” IN MNEM	Search list not used—CALL is qualified				
CALL “P”	Yes	Irrelevant	Irrelevant	Irrelevant	Yes
	No	Yes	Irrelevant	Irrelevant	Yes
	No	No	Irrelevant	Irrelevant	Yes, with un-resolved externals; warning issued
ENTER `language` “R” IN MNEM	Search list not used—ENTER is qualified				
ENTER [`language`] “R”	Impossible	Yes	Irrelevant	Irrelevant	Yes
	Impossible	No	Irrelevant	Irrelevant	No, error reported
ENTER [TAL] “R” IN MNEM	Search list not used—ENTER is qualified				
ENTER [TAL] “R”	Impossible	Yes	Irrelevant	Irrelevant	Yes
	Impossible	No	Yes	Irrelevant	Yes
	Impossible	No	No	Yes	Yes

Table 12-2 Resolution of External References *(continued)*

	External reference found in ...				
Form *	Current Source File	Primary Search List	Tertiary Search List	ECOBEXT, ZCOBDLL, or ZCREDLL	Target File Created?
	Impossible	No	No	No	No, error reported

* Bracketed keywords are optional.

Presenting Parameters to the Called Program

A formal parameter in the called program can have one of two attributes: value or reference. Depending on the description of the actual parameter in the calling program and the description of the formal parameter in the called program, the compiler generates code to convert the value of the actual parameter to match the description of the formal parameter.

Parameter	Explanation
VALUE	When a formal parameter in the called routine has the VALUE attribute and is numeric, the compiler generates any code necessary to evaluate the actual parameter, convert the value to the expected representation and scaling, and pass the value to the called program. When a formal parameter has the VALUE attribute but is nonnumeric, the compiler generates code to pass the value to the called program. When an actual parameter in the calling program is a literal or an arithmetic expression, the corresponding formal parameter must have the VALUE attribute.
REFERENCE	When a formal parameter has the REFERENCE attribute, the actual parameter supplies access to an object in the calling program. When an actual parameter is an identifier, the compiler generates code to pass the address of the associated data item to the called program. In this case, the calling program and the called program must agree on the interpretation of the data item value. When an actual parameter is a file name, the compiler generates code to pass the address of the file control block. Use this option only where specified for a program supplied by HP.
OMITTED	When the parameter in the calling program is specified by the reserved word OMITTED, the compiler generates no code to convert or pass anything. The calling program can use OMITTED only when the called program has the EXTENSIBLE or VARIABLE attribute. The calling program and the called program must agree on which operands can be OMITTED.

Delivering the Returned Value

When an ENTER statement has a GIVING phrase, the compiler generates code to deliver the value that the program returns to a data item in the calling program. The generated code performs any necessary scaling of the result value, then performs an assignment according to the rules for an elementary move operation.

Linking the Object Code

When the compilation source text includes a separately compiled program, and the compiler reports no error against that program, the target file includes the object code for that program. The object code for nested programs is inseparable from the code for their outermost separately compiled program.

When a separately compiled HP COBOL program or a non-COBOL program is available in object form, and the compiler locates it using a *`file-mnemonic`* or primary search list, the target file includes the object code for that program.

When the ECOBOL compiler finds a program in the ZCOBDLL, ZCREDLL, or ECOBEXT file, the compiler validates parameters, but does not bind the object code into the target file.

#RECEIVE Blocks

When the linker creates a run unit, it compares the #RECEIVE blocks of all the programs and determines which argument values are to be used whenever another HP COBOL object file opens $RECEIVE.

Table 12-3 How Final Values of #RECEIVE Arguments Are Chosen

#RECEIVE Argument	Final Value
TABLE OCCURS	Maximum value specified by a TABLE OCCURS argument in any #RECEIVE block
SYNCDEPTH	Maximum value specified by a SYNCDEPTH argument in any #RECEIVE block
REPLY CONTAINS	Maximum value specified by a REPLY CONTAINS argument in any #RECEIVE block
REPORT	Logical OR
QUEUE DEPTH	Maximum value specified by a QUEUE DEPTH argument in any #RECEIVE block*

* HP COBOL does not have a QUEUE DEPTH argument, so the value is assumed to be 1.

The first HP COBOL external file to open $RECEIVE determines the values of the #RECEIVE arguments. Subsequent opens of $RECEIVE fail if they specify incompatible values.

#RECEIVE Argument	Subsequent value is incompatible with first value if ...
TABLE OCCURS	subsequent value > first value
SYNCDEPTH	subsequent value > first value
REPLY CONTAINS	subsequent value > first value
REPORT	subsequent value <> first value
QUEUE DEPTH	first value = 1 and subsequent value <>1 or first value > 1 and subsequent value <= 1 or subsequent value > first value

If an internal file opens $RECEIVE, the values of the #RECEIVE arguments depend on whether the program that describes the internal file has a RECEIVE-CONTROL paragraph. If so, the #RECEIVE block associated with the program's RECEIVE-CONTROL paragraph is used as the internal file's #RECEIVE block (even in a TNS program whose RECEIVE-CONTROL paragraph is EXTERNAL). If the program does not have a RECEIVE-CONTROL paragraph, the #RECEIVE arguments have these default values:

#RECEIVE Argument	Default Value
TABLE OCCURS	1
SYNCDEPTH	1
REPLY CONTAINS	0
REPORT	None
QUEUE DEPTH	1

If an external file opens $RECEIVE, its #RECEIVE block is the one associated with the run unit, if one exists; otherwise, the #RECEIVE arguments have the default values.

When sharing $RECEIVE, be aware that:

- If one opener requests system messages and another does not, the one that does not might get them anyway.
- The ERROR CODE and MESSAGE SOURCE phrases of the RECEIVE-CONTROL paragraph are always local to the program unit. They apply to all READ and WRITE requests in that program unit, even if $RECEIVE was opened in another program unit.
- Opening a file that references $RECEIVE uses the #RECEIVE arguments specified by the program that contains the OPEN statement, even if the file is EXTERNAL. If the file is EXTERNAL, the ERROR CODE and MESSAGE SOURCE clauses of the RECEIVE-CONTROL paragraph in other programs are ignored.

Compiler Output

The output of a compilation can include a listing (to an existing file or spooler) and an object file, but it always includes compilation statistics and a completion code. You can manipulate the object file.

Listing Creation

The listing is added to the end of the listing (OUT) file. The listing file can be either a disk file or nondisk file (see Starting a Compilation). If it is a disk file, it must exist before you start the compilation.

To create a listing file, use the command:

```
FUP CREATE disk-file-name, TYPE E, REC 132[, EXT pages]
```

disk-file-name

is a disk file name (see the *Guardian Procedure Calls Reference Manual*).

pages

is the number of extended segment pages.

TYPE E

specifies an entry-sequenced file.

REC 132

specifies 132-character records (partial lines are space-filled on the right through column 132).

EXT *pages*

is necessary only if *disk-file-name* exceeds the standard number of allocated segments.

For more information, see Source Program Listing (page 781).

Object File Creation

The object file that the compiler produces is either a linkfile or a loadfile.

The compiler produces a loadfile if you specify the directive RUNNABLE (page 570). If the compilation unit contains no main program, executing the object file causes it to terminate immediately with an error message. The compiler produces a linkfile by default (if you do not specify the RUNNABLE directive). Linkfiles that the compiler produces can be input to the `eld` utility. Loadfiles that the compiler produces cannot be input to the `eld` utility. For information on the `eld` utility, see the *eld Manual*.

Object File Size

The Guardian ECOBOL compiler specifies the maximum allowable size of the generated object file in terms of extents. Extents are measured in pages of 2K bytes each. The default extent size specified by the ECOBOL compiler is 32 pages. The maximum allowed number of extents for a

Guardian file is 900, for a maximum object file size of approximately 56MB. For more information about Guardian extents, see the *Guardian Programmer's Guide*.

If you expect that an object file will exceed the 56MB limit, you can specify the OBJEXTENT directive on the ECOBOL command line. The OBJEXTENT directive allows you to increase the size of the extents used for the object file, up to a maximum of 1070 pages, for a maximum object file size of approximately 2043MB.

Generating Instrumented Object Code for Use With the Code Coverage Tool

The Code Coverage Tool evaluates the code coverage provided by application test cases. The tool uses information provided by a specially-instrumented object file to produce a report that indicates which functions and blocks were executed, and how many times each was executed.

Using the Code Coverage Tool requires a special compilation to produce an object file containing the required instrumentation. To create such an object file, specify the COLUMNS directive on the compiler command line.

NOTE: The Code Coverage Tool is intended for data generation and collection in a test environment only. The use of instrumented object code is not recommended for production environments. Applications compiled with code coverage instrumentation will experience greatly reduced performance. See the Caution in Chapter 16: Debugging Tools (page 707) for more information.

For details on using the Code Coverage Tool, see the *Code Coverage Tool Reference Manual*.

Manipulating Object Files

You can use a TNS/E HP COBOL, TNS/E HP C, TNS/E HP C++, or EpTAL object file as input to the linker to create a new target file. Using the linker, you can add code blocks and data blocks compiled from TNS/E HP COBOL, TNS/E HP C, TNS/E HP C++, or EpTAL source files. For details, see the *eld Manual*.

Compilation Statistics

Statistics are printed at the end of every compilation:

1. Linker statistics (if an object file was produced)
2. Compiler statistics

Example 12-6 is the result of a compilation that did not specify the RUNNABLE directive.

Example 12-7 is the result of a compilation that specified the RUNNABLE directive.

Example 12-6 ECOBOL Compiler Statistics

```
COBOL - T0356H01 - (20DEC2004)
No failures detected.
No errors detected.
No warnings reported.
No remarks issued.
Maximum symbol table size = 10884 bytes

Object file: opt2o
Compiler driver: \DRP12.$SYSTEM.SYSTEM.ECOBOL
COBOL DLL:       \DRP12.$SYSTEM.ZDLL031.ZCOBDLL
CRE DLL:         \DRP12.$SYSTEM.ZDLL031.ZCREDLL
ECOBEXT:         \DRP12.$SYSTEM.SYSTEM.ECOBEXT
Compiler statistics
  phase     CPU seconds elapsed time file name
  ECOBFE            0.9     00:00:05 \DRP12.$SYSTEM.SYSTEM.ECOBFE
  total             0.9     00:00:06
All processes executed in CPU 01 (NSE-P)
Swap volume: \DRP12.$SYSTEM
```

Example 12-7 ECOBOL Compiler and Linker Statistics

```
COBOL - T0356H01 - (20DEC2004)
No failures detected.
No errors detected.
No warnings reported.
No remarks issued.
Maximum symbol table size = 10884 bytes

eld - TNS/E Native Mode Linker - T0608H01 - 03DEC07
Copyright 2004 Hewlett-Packard Company

eld command line:
   \drp12.$system.system.eld -o opt2o opt2o \DRP12.$SYSTEM.ZDLL031.ZCOBDLL
   \DRP12.$SYSTEM.ZDLL031.ZCREDLL

**** INFORMATIONAL MESSAGE **** [1019]:
   Using DLL \DRP12.$SYSTEM.ZDLL031.ZCOBDLL.
**** INFORMATIONAL MESSAGE **** [1019]:
   Using DLL \DRP12.$SYSTEM.ZDLL031.ZCREDLL.
**** INFORMATIONAL MESSAGE **** [1530]:
   Using the zimpimp file $SYSTEM.SYS00.ZIMPIMP.

Output file: opt2o (program file)
Output file timestamp: Jan 6 12:18:24 2005

No errors reported.
No warnings reported.
3 informational messages reported.
Elapsed Time: 00:00:01
Object file: opt2o
Compiler driver: \DRP12.$SYSTEM.SYSTEM.ECOBOL
COBOL DLL:       \DRP12.$SYSTEM.ZDLL031.ZCOBDLL
CRE DLL:         \DRP12.$SYSTEM.ZDLL031.ZCREDLL
ECOBEXT:         \DRP12.$SYSTEM.SYSTEM.ECOBEXT
Compiler statistics
  phase     CPU seconds   elapsed time   file name
  ECOBFE            0.9   00:00:06       \DRP12.$SYSTEM.SYSTEM.ECOBFE
  ELD               0.2   00:00:02       \DRP12.$SYSTEM.SYSTEM.ELD
  total             1.1   00:00:08
All processes executed in CPU 01 (NSE-P)
Swap volume: \DRP12.$SYSTEM
```

Completion Codes

The compiler reports an appropriate completion code when it terminates execution. An HP COBOL program can set its own completion code at termination by calling the COBOL_COMPLETION_ routine.

Table 12-4 Completion Codes

Code	Termination	Explanation
0	Normal	No diagnostic messages were issued. The object file is complete and valid (unless a SYNTAX directive suppressed its creation).
1	Normal	At least one compiler warning occurred. The object file is complete and presumed to be valid (unless a SYNTAX directive suppressed its creation).
2	Normal	At least one compiler error occurred. No object file was created.
3	Abnormal	The compiler exhausted one of its internal resources (such as symbol table space) or it was refused some external service (such as access to a file) before it completed its task. No object file was created.
5	Abnormal	During internal consistency checking, the compiler discovered a logic error, or else one of the compiler's server processes reported a failure and was terminated abnormally. No object file was created.
8	Normal	At least one compiler warning occurred. The name specified for the target (object) file could not be used, so another name was chosen. This name is reported in the summary. The object file is complete and presumed to be valid.

Running the Compiler

This topic explains how to run the compiler, including:

- PARAM Commands
- Predefined SEARCH DEFINEs
- Starting a Compilation
- Terminating a Compilation

PARAM Commands

If you want to use a PARAM command, you must enter it before you give the command to run the compiler. When the compiler terminates execution, it reports an appropriate completion code.

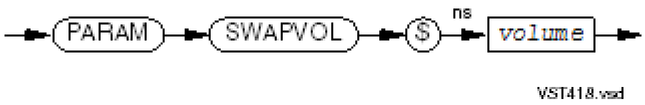
NOTE: PARAM commands for files in a user library are ignored.

Table 12-5 PARAM Commands Accepted by Compiler

Command	Description
PARAM SWAPVOL	Specifies the volume on which the ECOBOL compiler and its processes will create temporary files.
PARAM SYMBOL-BLOCKS	Specifies the number of 256 KB blocks that the compiler needs for its symbol dictionary. The number of lines allowed for each SQL/MP statement is 500 times the number specified by PARAM SYMBOL-BLOCKS.

PARAM SWAPVOL

The PARAM SWAPVOL command specifies the volume on which the compiler and its processes will create temporary files (if possible). It does not determine where the operating system creates the compiler's own swap file—the Kernel-Managed Swap Facility (KMSF) does that. For more information, see the *Kernel-Managed Swap Facility (KMSF) Manual*.



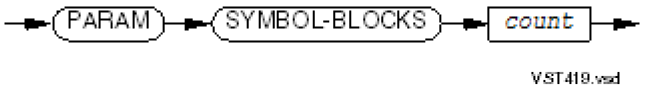
`$volume`

is a dollar sign ($) immediately followed by one to seven alphanumeric characters. The first alphanumeric character must be alphabetic. The `$volume` is the name of the volume on which the temporary files are to be created.

The `$volume` must exist on the system on which the compiler resides. If the compiler cannot create its first temporary file on the specified volume, compilation proceeds with temporary files created as though no PARAM SWAPVOL command were active.

PARAM SYMBOL-BLOCKS

The PARAM SYMBOL-BLOCKS command specifies how much space the compiler allocates for its symbol dictionary, local label table, and embedded SQL/MP or SQL/MX statements.



`count`

is an integer in the range 1 through 40, which affects space allocation. The default value of `count` is 4.

Table 12-6 How the PARAM SYMBOL-BLOCKS Command Affects Space Allocation

	Space Allocated for Item	
Item	**When PARAM SYMBOL-BLOCKS command is specified ...**	**Default**
Symbol dictionary	`count` 256-KB blocks	Four 256-KB blocks
Embedded SQL/MP statements	`count` times 500 lines times number of embedded SQL/MP statements, plus extra space for REPLACE statements	500 lines of SQL/MP text per statement

If the default value for `count` produces a failure in compilation, increase it by one. If that still is not enough, increase it by one more, and so on.

If the summary listing at the end of a compilation indicates that the maximum symbol table size is less than 256 KB, you might be able to reduce the system resources required for later compilations by specifying PARAM SYMBOL-BLOCKS 1.

Predefined SEARCH DEFINEs

The compiler recognizes two predefined DEFINEs with CLASS attribute SEARCH:

- =_SOURCE_SEARCH (page 538)
- =_OBJECT_SEARCH (page 538)

These allow you to specify one or more subvolumes for the compiler to search for unqualified source text files and object files, respectively. ("Unqualified" means that the file name does not contain a volume and subvolume.) If you use these predefined DEFINEs, you do not need to specify DEFINEs in the compiler command or in the source text.

To add a predefined SEARCH DEFINE, use the ADD DEFINE command. For example:

```
ADD DEFINE =_SOURCE_SEARCH, CLASS SEARCH, SUBVOL0 (=_DEFAULTS,$VOL1.SUB2,$VOL1.SUB3)
```

In this example, the compiler is to search the default subvolume first, then $VOL1.SUB2, and finally $VOL1.SUB3.

=_SOURCE_SEARCH

=_SOURCE_SEARCH tells the compiler where to search for unqualified source text files specified by:

- COPY statements
- SOURCE directives
- The *copy-library* parameter of the compilation command

The =_SOURCE_SEARCH DEFINE does not affect compiler searches in these cases:

- When the source text file name is qualified
- When the program uses the default COPY library, COPYLIB, for COPY text

=_OBJECT_SEARCH

=_OBJECT_SEARCH tells the compiler where to search for unqualified object files specified by:

- SPECIAL-NAMES paragraph
- CALL statements
- ENTER statements
- CONSULT directives
- LIBRARY directives
- SEARCH directives

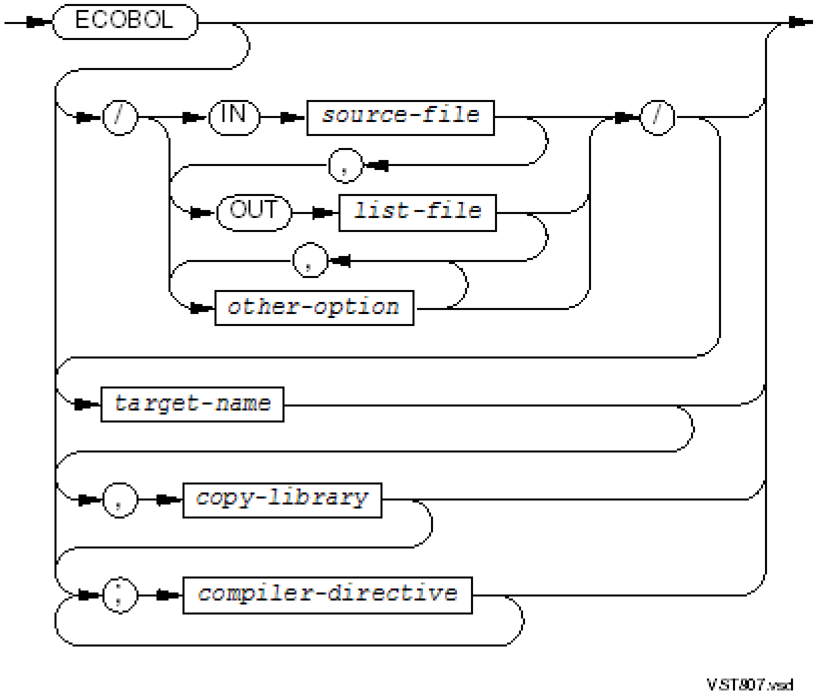
The =_OBJECT_SEARCH DEFINE does not affect:

- The compiler search when the object file name is qualified
- The file ECOBEXT, which is assumed to be on the same subvolume as the compiler.

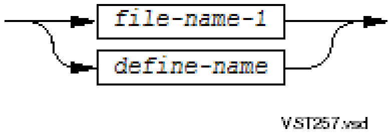
Starting a Compilation

This section applies to the Guardian environment. To compile an HP COBOL program in the OSS environment, see Chapter 20: Using HP COBOL in the OSS Environment (page 721).

To run the ECOBOL compiler, use this compiler command:



source-file



is a file containing HP COBOL statements, comment lines, and compiler directives. It must be a disk file, terminal, magnetic tape unit, or process. The compiler reads *source-file* as 132-byte records. The default is the current command interpreter IN file (usually the home terminal). *source-file* must be an EDIT file.

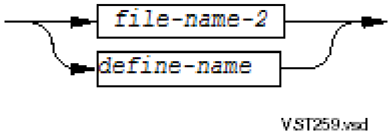
file-name-1

is a disk file name (if *source-file* is a disk file) or nondisk file name (if *source-file* is a terminal, magnetic tape unit, or process). For syntax, see the *Guardian Procedure Calls Reference Manual*.

define-name

is the name of a DEFINE established in the current run-time environment.

list-file



is the destination to which the compiler directs its output. It must be a disk file, terminal, magnetic tape unit, process, line printer, or spooler collector (possibly qualified with a location name). It must already exist and meet the criteria in Listing Creation. If *list-file* is unstructured, each record is 132 characters (partial lines are space-filled on the right).

`file-name-2`

is a disk file name (if `list-file` is a disk file) or nondisk file name (if `list-file` is a terminal, magnetic tape unit, process, line printer, or spooler collector). For syntax, see the *Guardian Procedure Calls Reference Manual*.

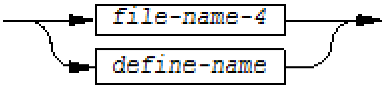
`define-name`

is the name of a DEFINE established in the current run-time environment.

`other-option`

is any other command interpreter RUN option (see the description of the RUN command in the *TACL Reference Manual*).

`target-name`



VST261.vsd

is the name of a disk file on which the single target object file is to be produced. This name is used only if all of these are true:

- The compiler is instructed to produce a target file. This means that the SYNTAX directive is not specified.
- The compilation and binding are successful.
- The source file contains only one main program. If it contains more than one, you must specify an object file for each of them (`target-name-1` through `target-name-n`).

The default `target-name` is:

`\default-system.$default-volume.default.subvolume.RUNUNIT`

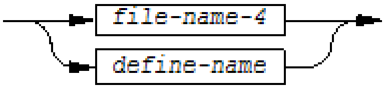
`file-name-3`

is a disk or nondisk file name. For syntax, see the *Guardian Procedure Calls Reference Manual*.

`define-name`

is the name of a DEFINE established in the current run-time environment.

`obj-1` ... `obj-n`



VST261.vsd

is a list of names of disk files on which the multiple target object files are to be produced. Each has the same syntax as `target-name`. These restrictions apply:

- You must specify at least as many object file names as there are main programs delivered to the compiler. You can specify more, but not fewer. If you specify more, unused ones are ignored.
- The `source-file` must be an EDIT file.
- The `list-file` cannot be a magnetic tape file. `list-file` is opened and closed for each separately compiled program. If `list-file` is a spooler, one job exists for the compiler command and one for each separately compiled program in the `source-file`.
- Each directive that applies to more than one separately compiled program in the `source-file` (for example, ICODE or NOBLANK) must be specified as a `compiler-directive` in the compiler command.
- Each separately compiled program in the `source-file` must end with an END PROGRAM statement or an ENDUNIT compiler directive.

- In the *source-file*, compiler directives must be contained in separately compiled programs. They cannot appear before, between, or after them.
- A COPY or SOURCE statement cannot bring in the beginning or end of a separately compiled program.

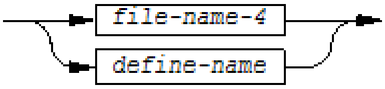
file-name-3

is a disk or nondisk file name. For syntax, see the *Guardian Procedure Calls Reference Manual*. The default *file-id* of *file-name-3* is COPYLIB.

define-name

is the name of a DEFINE established in the current run-time environment.

copy-library



VST261.vsd

is the name of a COPY library, a disk file in EDIT format, which is to be the default COPY library (for any COPY statement in the source program that does not specify a library from which to copy).

file-name-4

is a disk file name. For syntax, see the *Guardian Procedure Calls Reference Manual*. The default *file-id* of *file-name-3* is COPYLIB.

define-name

is the name of a DEFINE established in the current run-time environment.

compiler-directive

is any directive described in Compiler Directives.

Examples of Commands That Run the Compiler

Each of these commands initiates compilation of the program contained in MYSRC (on the default system, volume, and subvolume), directing the listing to $SPOOL (a spooler collector):

```
ECOBOL /IN MYSRC,OUT $SPOOL,PRI 140/MYPROG;INNERLIST
```

The compiler is to run at a priority of 140. The name of the target file, if the compilation succeeds, is to be MYPROG on the default system, volume, and subvolume. The directive specifies that a listing of the mnemonic version of the generated code is to be produced.

The next command initiates compilation of the program contained in file COBOLPRG on system \MM, volume $DEV, subvolume PYRL and directs that the listing output be discarded:

```
ECOBOL /IN \MM.$DEV.PYRL.COBOLPRG,OUT/; SETTOG 3
```

If the compilation succeeds, the loadfile is to be named RUNUNIT. The directive sets a compilation toggle that presumably governs the inclusion or exclusion of certain portions of the source text.

Terminating a Compilation

There are two ways to terminate a compilation before normal completion:

- Press Break and type STOP.
- Press Break and type:

  ```
  STATUS *, TERM
  ```

  Find the processor (*cpu*) and process number (*process*) for the compiler. Type:

  ```
  STOP cpu,process
  ```

With either method, the supporting processes stop when the compiler stops.

NOTE: If you expect to have multiple compilations active at one time, give each ECOBOL process a different name with the command interpreter NAME parameter. Then you can stop the named compilation process of your choice using the second method.

For possible values of the system completion code upon the termination of a compilation, see Completion Codes.

Compiler Directives

Compiler directives are used to specify the source format, to control listing features, to control selective compilation of portions of the source code, and to request compilation options.

Topics:

- Where Compiler Directives Are Allowed
- Categories of Compiler Directives
- One for each compiler directive, listed alphabetically, beginning with ANSI

Where Compiler Directives Are Allowed

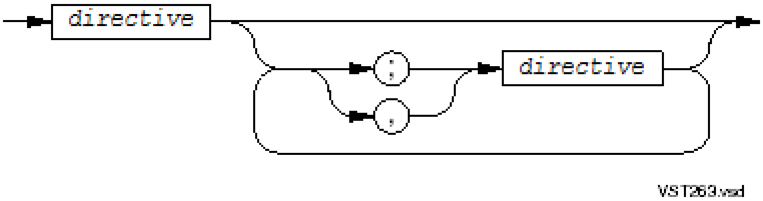
In general, one or more directives can be entered in the directive field of the TACL command to run the compiler (see Starting a Compilation) or can be included in the source text on lines beginning with a question mark (?) in the indicator area. Restrictions on these general rules are:

- Each of the directives ENDUNIT, IF, IFNOT, and ENDIF must be either on a directive line of its own or be the last of a sequence of directives.
- The SECTION directive must always be on a line of its own.
- The compiler accepts the SQL directive on the compiler command line, but not in the source program.

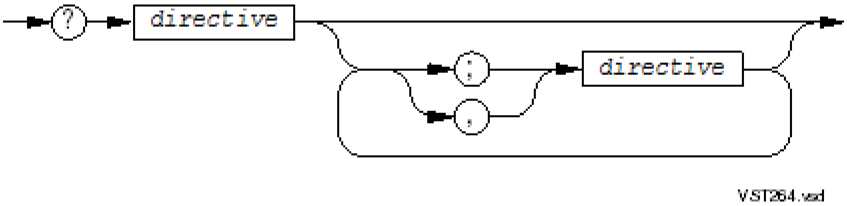
Directives in the source text override directives on the command line.

The general form of compiler directives is:

On the compiler command line:



In the program source text:



?

is a source text format indicator (like the asterisk and the slash) and is not part of the compiler directive.

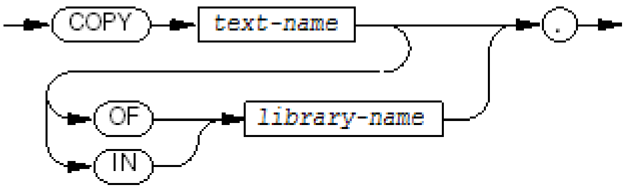
The question mark must be in the indicator area (column 1 for Tandem format and column 1 or 7 for ANSI format).

directive

is one of the directives listed in this section.

With these exceptions, compiler directive lines can appear at any point in the source text, including those portions that a COPY statement retrieves from a source library file.

- Compiler directive lines cannot occur between the elements of a multiline COPY statement. The syntax of the COPY statement, without the REPLACING phrase, is:



VST265.vsd

If, for example, the word COPY is on one source line and *text-name* is on a subsequent source line, no compiler directive lines can occur on any intervening source line.

- These directives have restrictions on their location within the source text:
  - CODECOV
  - COMPILE
  - ELD
  - GLOBALIZED
  - MAIN
  - MIGRATION-CHECK
  - NONSTOP
  - OBJEXTENT
  - OPTIMIZE
  - STANDARD
  - SYMBOLS
  - SYNTAX
  - UL
- The COLUMNS directive can appear in a COPY library, but only once, before the first SECTION directive
- The SQL directive cannot appear in a source program

Categories of Compiler Directives

Compiler directives fall into these categories:

- Table 12-7: Source Text Manipulation Directives (page 544)
- Table 12-8: Input Format Control Directives (page 544)
- Table 12-9: Listing Control Directives (page 544)
- Table 12-10: Code-Generation Control Directives (page 545)
- Table 12-11: Resolution and Binding Control Directives (page 546)
- Table 12-12: Miscellaneous Control Directives (page 547)
- Table 12-13: ENDUNIT Directive (page 547)

Table 12-7 Source Text Manipulation Directives

Directive(s)	Default	Action
SECTION		Marks beginning of portion of text in an EDIT file, which compiler reads in response to COPY statements and SOURCE directives
SHOWFILE and NOSHOWFILE	NOSHOWFILE	Switches identification of source file in compiler listing on or off
SOURCE*		Reads an entire EDIT file, or one or more sections of an EDIT file
SETTOG*		Turns toggles on
RESETTOG*		Turns toggles off
IF and IFNOT*		Enables or suppresses compilation of subsequent lines if a specified toggle has been set; disables it otherwise
ENDIF*		Marks end of toggle block

* Must be the last directive on its line.

Table 12-8 Input Format Control Directives

Directive	Default	Action
COLUMNS		Specifies logical length of source lines when TANDEM directive is present
TANDEM	TANDEM	Specifies Tandem source format
ANSI	TANDEM	Specifies ANSI source format

Table 12-9 Listing Control Directives

Directive(s)	Default	Action
CODE and NOCODE[1]	NOCODE	Ignored (warning issued)
CROSSREF and NCROSSREF[1]	NOCROSSREF	Ignored (warning issued). For a cross-reference listing, use the `enoft` utility with the XREFPROC flag (see the *enoft Manual*).
DIAGNOSE-74 and NODIAGNOSE-74	NODIAGNOSE-74	Enables or suppresses warnings about statements that might behave differently for HP COBOL and COBOL 74
DIAGNOSE-85 and NODIAGNOSE-85	NODIAGNOSE-85	Enables or suppresses warnings about source constructs that might behave differently for COBOL85 compiler and ECOBOL compiler
DIAGNOSEALL and NODIAGNOSEALL	NODIAGNOSEALL	Enables or suppresses warnings for multiple references to undefined identifiers (instead of the first one only)
FIPS and NOFIPS	NOFIPS	Enables or suppresses identification of statements as required by the Federal Information Processing Standard (FIPS)
FMAP		Enables listing of source file map with fully qualified name and timestamp for IN file and every SOURCE and COPY file
HEADING		Specifies page heading text.
ICODE and NOICODE[1]	NOICODE	Ignored (warning issued)

Table 12-9 Listing Control Directives *(continued)*

Directive(s)	Default	Action
INNERLIST and NOINNERLIST	NOINNERLIST	Enables or suppresses mnemonic code listing after each source program statement
LINES		Specifies lines per page
LIST and NOLIST	NOLIST	Enables or suppresses compiler listing
LMAP and NOLMAP[1]	LMAP	Ignored (warning issued)
MAP and NOMAP	NOMAP	Enables or suppresses symbol map
MIGRATION-CHECK	No warnings issued	Enables warnings when STANDARD 1985 is specified and compiler detects a user-defined word that is a COBOL-2002 reserved word
SHOWCOPY and NOSHOWCOPY	SHOWCOPY	Enables or suppresses listing of COPY statements as comments
SUBSET		Enables or suppresses flags on extensions or obsolete elements
SUPPRESS and NOSUPPRESS	NOSUPPRESS	Suppresses or enables all but leader, diagnostics, and trailer text in compiler listing
WARN and NOWARN	WARN	Enables or suppresses warnings

[1] Described in the *COBOL Manual for TNS and TNS/R Programs*

Table 12-10 Code-Generation Control Directives

Directive(s)	Default	Action
BLANK and NOBLANK	NOBLANK	Enables or suppresses default initialization of data-items whose declarations lack VALUE clauses
CALL-SHARED	CALL-SHARED	Generates shared code [position-independent code (PIC)] (see also SHARED)
CANCEL and NOCANCEL	CANCEL	Controls whether code is generated to initialize data for a program that does not have the INITIAL attribute
CHECK		Controls the level of run-time checking included in the loadfile
CODECOV	No code coverage instrumentation in generated object code	Controls whether instrumented object code is generated for use with the Code Coverage Tool
COMPACT and NOCOMPACT[1]	COMPACT	Ignored (warning issued)
COMPILE	COMPILE	Generates object code (compare to SYNTAX)
FLOAT and NOFLOAT[1]	FLOAT	Ignored (warning issued)
GLOBALIZED	Generate non-preemptable object code	Directs compiler to generate preemptable object code
INSPECT and NOINSPECT	INSPECT	Selects the debugger to be used if the program is initiated with RUND or if the DEBUG command is issued against it
LARGEDATA[1]		Ignored (warning issued)
LESS-CODE[1]		Ignored (warning issued)

Table 12-10 Code-Generation Control Directives *(continued)*

Directive(s)	Default	Action
NON-SHARED[1]	CALL-SHARED	Ignored
OBJEXTENT (Guardian only)	OBJEXTENT 32	Determines the size of the extents allocated for the generated object file
OPTIMIZE		Specifies the level of object code optimization the compiler must provide
PORT and NOPORT	NOPORT	Causes BINARY/COMPUTATIONAL items to be byte-aligned and allows the CALL statement to behave as it does in X/Open and XPG4
RUNNABLE		Causes the ECOBOL compiler to produce a loadfile
SAVEABEND and NOSAVEABEND	NOSAVEABEND	Specifies that if the program terminates abnormally, its state is saved for future examination with the selected debugger
SHARED	CALL-SHARED	Generates shared code (PIC) for a DLL (see also CALL-SHARED)
SYMBOLS and NOSYMBOLS	NOSYMBOLS	Controls whether a symbol table is included in the target file
SYNTAX	COMPILE	Suppresses generation of object code
TRAP2 and NOTRAP2[1]	TRAP2	Ignored (warning issued)
TRAP2-74 and NOTRAP2-74[1]	TRAP2-74	Ignored (warning issued)
UL		Specifies that the resulting object code will be in a user library

[1] The compiler ignores and issues warnings for these directives.

Table 12-11 Resolution and Binding Control Directives

Directive(s)	Default	Action
CONSULT and NOCONSULT	NOCONSULT[1]	Specifies a list of object files from which the compilation can resolve unqualified external references, but not bind in the specified routines
ELD		Passes one or more linker options to the `eld` utility
LD[1]		Ignored (warning issued)
LIBRARY[1]		Ignored (warning issued)
NLD[1]		Ignored (warning issued)
SEARCH and NOSEARCH	NOSEARCH[1]	Specifies a list of object files from which the compilation can resolve unqualified external references and bind or link in the specified routines

[1] The compiler ignores and issues warnings for these directive.

Table 12-12 Miscellaneous Control Directives

Directive(s)	Default	Action
ENV[1]		Useless, but accepted with COMMON option; error reported for OLD option; warning issued for LIBRARY option
ERRORFILE		Specifies a file that is to contain information about any compilation errors or warnings (it can be used with the FIXERRS TACL macro)
ERRORS		Specifies maximum number of errors allowed
HEAP[1]		Ignored (warning issued)
HIGHPIN[1]		Ignored (warning issued)
HIGHREQUESTERS[1]		Ignored (warning issued)
MAIN		Specifies the main program in an input stream
NONSTOP[1]		Specifies program is to run as a process pair
PERFORM-TRACE		Provides additional information if run-time error 148 occurs.
RUNNAMED		Specifies that the run unit is to be run as a named process
SAVE[1]		Specifies which initialization message information is saved
SQL and NOSQL	NOSQL[2]	Tells the compiler to expect SQL/MP statements in the compilation unit
SQLMEM[1]		Ignored (warning issued)
STANDARD	STANDARD 1985	Specifies whether the compiler is to apply the COBOL-1985 or COBOL-2002 standard
SUBTYPE[1]		Specifies that the program runs as a process with the designated subtype number

[1] These directives are not available in the OSS environment.

[2] The compiler ignores and issues warnings for these directives.

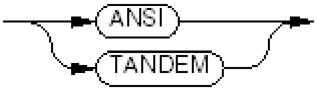
Table 12-13 ENDUNIT Directive

Directive	Action
ENDUNIT	Marks the end of the Procedure Division in a source file that contains multiple, separately compiled COBOL programs.*

* This directive must be the last or the only directive on its line. If any other directive appears on the line after it, the compiler reports an error and ignores the additional directives.

NOTE: In the list of directives, "**Default:**" identifies the default for the compiler directive itself, not for its optional parameter(s). This default applies if a program does not contain the compiler directive at all.

ANSI



VST266.vsd

ANSI

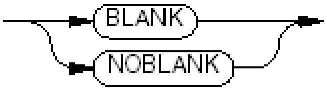
specifies that subsequent source text is in ANSI reference format, which is described in Chapter 17: ANSI Reference Format (page 711).

TANDEM

specifies that subsequent source text is in Tandem reference format, which is described in Reference Format for Source Program Lines (page 54).

Default:	TANDEM
Placement:	Anywhere
Scope:	ANSI or TANDEM within a section of text obtained from a copy library or source library is effective only for the length of that text section. When the compiler reverts to the source file where it found the COPY verb or SOURCE directive, the previously active reference format applies.
Dependencies:	None
References:	TANDEM

BLANK and NOBLANK



VST267.vsd

BLANK

adds an implicit VALUE SPACES clause to the description of:

- Every data item in the Working-Storage Section and Extended-Storage Section, except those to which an explicit VALUE clause or an EXTERNAL clause applies
- Every data item in the File Section, except those to which an EXTERNAL clause applies

BLANK increases the size of the object file and can increase execution time.

NOBLANK

prevents the addition of implicit VALUE SPACES clauses.

Default:	NOBLANK
Placement:	Anywhere
Scope:	The last BLANK or NOBLANK in the program is applies.
Dependencies:	None

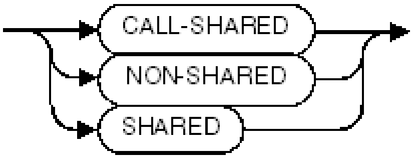
Program units compiled with BLANK (or for which BLANK is the default) include code that performs implicit and explicit initializations. Program units compiled with NOBLANK (or for which NOBLANK is the default) do not provide this automatic initialization, and any data item without a VALUE clause has an undefined initial value.

NOBLANK is recommended for initial programs and programs that declare very large data items that do not have initial values assigned, because it saves significant amounts of compilation time, object program storage space on disk, and run-time initialization time. If you must initialize very large data items to spaces, use VALUE clauses, INITIALIZE statements, or MOVE statements of the form

```
MOVE SPACES TO X
```

VALUE clauses are recommended.

CALL-SHARED



VST802.vsd

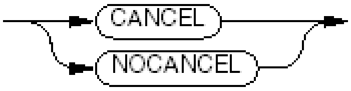
CALL-SHARED
: generates shared code (PIC).

NON-SHARED
: has no effect. (For the NMCOBOL compiler, it generates nonshared code (non-PIC)).

SHARED
: generates shared code (PIC) for a DLL.

Default:	CALL-SHARED
Placement:	Anywhere
Scope:	The last CALL-SHARED or SHARED in the compilation unit applies to the entire compilation unit.
Dependencies:	• If RUNNABLE is active, CALL-SHARED uses the linker to create a PIC executable object file; otherwise, CALL-SHARED creates a PIC linkfile. • Do not use with UL.
References:	• SHARED (page 576) • RUNNABLE • UL

CANCEL and NOCANCEL



VST383.vsd

CANCEL
: generates code that initializes the program the first time the program is called after being canceled by a CANCEL statement.

NOCANCEL

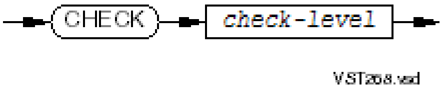
prevents the generation of code that initializes the program the first time the program is called after being canceled by a CANCEL statement.

Default:	CANCEL
Placement:	Anywhere
Scope:	Applies to the program that contains it. CANCEL and NOCANCEL do not apply to initial programs, which are initialized every time they are called.
Dependencies:	None

If a program is not an initial program and is never referenced by a CANCEL statement, specify NOCANCEL to reduce the size of the program's object code. Do not specify NOCANCEL for a program that is referenced by a CANCEL statement. If you do, the program might not execute in the manner expected.

CHECK

CHECK controls the level of run-time checking that a program performs.



check-level

is an integer in the range 0 through 15. For the meanings of the different values of *check-level*, see Table 12-14.

Default:	CHECK 1
Placement:	Anywhere
Scope:	In each separately compiled program, the last CHECK in the program unit determines the check level for the code block being produced.
Dependencies:	None

Table 12-14 CHECK Levels

check-level	What is Checked	Comments
0	Nothing	CHECK 0 results in the fastest execution time.
1 (default)	Nothing	CHECK 1 might have a different meaning in future versions of HP COBOL. For the fastest execution (and no subscript checking) in current and future versions of HP COBOL, specify CHECK 0.
2	Validity of subscripts and indexes*	
3	Validity of subscripts, indexes, and reference modifiers*	
4	Validity of subscripts, indexes, reference modifiers, and certain data conversions*	Specifies the maximum level of checking that HP COBOL currently provides.

Table 12-14 CHECK Levels *(continued)*

check-level	What is Checked	Comments
5-15	Validity of subscripts, indexes, reference modifiers, and certain data conversions*	5-15 might have different meanings in future versions of HP COBOL. For the maximum level of checking that HP COBOL currently provides, specify CHECK 4. For maximal checking, specify CHECK 15 (which could increase the program's run-time overhead in future versions of HP COBOL if additional checking levels are implemented).

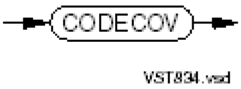
* The compiler generates extra code in order to check the validity of these items.

At run time, if the subscript or reference modifier used in a statement is out of range, or if an invalid DISPLAY data item is part of implicit or explicit conversions from DISPLAY to COMP or NATIVE, an error is reported, and the execution terminates.

CODECOV

CODECOV directs the compiler to generate instrumented object code for use with the Code Coverage Tool. For detailed information about the Code Coverage Tool, see the *Code Coverage Tool Reference Manual*.

NOTE: Instrumented object code can experience greatly reduced performance. Therefore, the CODECOV directive should be used only in a test environment. See the caution under Chapter 16: Debugging Tools (page 707), which indicates how CODECOV affects debugging applications. For information on instrumented object code, see Generating Instrumented Object Code for Use With the Code Coverage Tool (page 534).

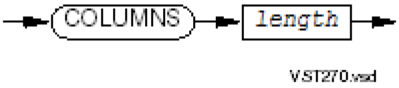


Default:	No code coverage instrumentation is included in the generated object file.
Placement:	On the command line
Scope:	Applies to the compilation unit
Dependencies:	None

COLUMNS

COLUMNS causes the compiler to ignore any text beyond a certain column in subsequent input records.

COLUMNS applies only to text being read under Tandem reference format. Text being read under ANSI reference format ignores any text beyond column 72.



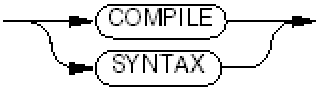
`length`

is an integer whose value is at least 12. Any value greater than 132 is considered to equal 132.

Default:	COLUMNS 132
Placement:	• On a directive line, COLUMNS must begin with a question mark (?) in column 1, regardless of any active ANSI. • In a COPY library or a source library, COLUMNS must be the only directive on its line and must precede all SECTION directives in that library.

Scope:	When the compiler shifts from reading the primary source file to reading a COPY library (in compliance with a COPY statement) or a source library (in compliance with a SOURCE directive), it saves the current (default or specified) logical length for source lines. That length applies to all source lines in the COPY library or source library, unless a COLUMNS directive occurs in the COPY library file (as mentioned in the preceding item). In this case, the compiler reverts to the saved logical length when it resumes reading text from the primary source file.
Dependencies:	COLUMNS works only if TANDEM is active.
References:	• ANSI • SOURCE • TANDEM

COMPILE



VST272.vsd

COMPILE

compiles the program unit and includes its code and data blocks in the target file being created.

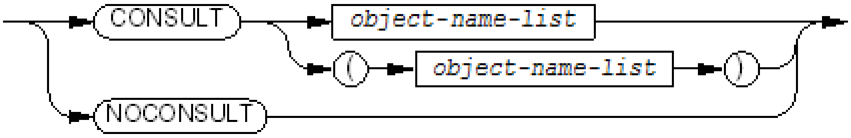
SYNTAX

checks the syntax of the source text. No target file is produced.

Default:	COMPILE
Placement:	Outside the boundaries of a separately compiled program; that is, not between the Identification Division header of a separately compiled program and its end, which is marked by one of: • The corresponding END PROGRAM statement • ENDUNIT • The end of the source file
Scope:	The last COMPILE or SYNTAX in the compilation unit applies to the entire compilation unit.
Dependencies:	None
References:	ENDUNIT

CONSULT and NOCONSULT

NOTE: The compiler ignores and issues a warning for the NOCONSULT directive.

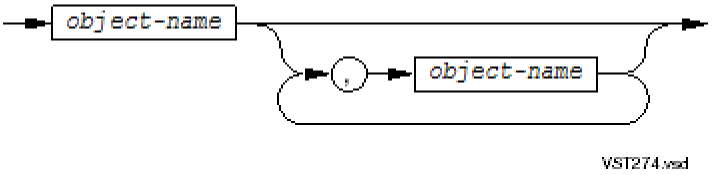


VST273.vsd

CONSULT

adds the files that `object-name-list` specifies to the tertiary search list. (See Tertiary Search List.)

object-name-list



can be continued onto subsequent lines, but the left parenthesis must appear on the same directive line as the keyword CONSULT. Each continuation line for a CONSULT directive must have a question-mark (?) in the indicator area. If a CONSULT directive spans multiple lines, they must be consecutive (no blank lines, comment lines, or program text lines can intervene).

object-name

is the name of a TNS/E object file. It can be either a file-system file name or (in the Guardian environment) a DEFINE name. If the file-system file name is not fully qualified with system, volume, and subvolume names, the compiler uses the current default system, volume, and subvolume names to complete the qualification. If *object-name* is in the OSS environment, it is an OSS pathname.

object-name must specify either:

- A linkfile
- An archive file
- A DLL
- An import library

When *object-name* is an archive file, the entire archive file is generally not linked into the target object file. Only those member files are linked that contain procedures named either in the statement CALL (page 303) or in the statement ENTER (page 330) in the COBOL program being compiled. Whenever a procedure is needed from a member file, the entire member file is linked.

When *object-name* is an import library, the compiler assumes that the member programs were compiled without the directive NONSTOP. If this is not true, the compiler issues warning 369, which you can ignore.

If an HP COBOL program references *object-name* either in the statement CALL (page 303) or in the statement ENTER (page 330), *object-name* must have been compiled with the SYMBOLS directive (see SYMBOLS and NOSYMBOLS).

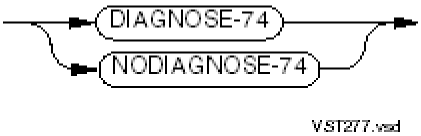
NOCONSULT

is ignored.

Default:	NOCONSULT
Placement:	Anywhere
Dependencies:	None

If more than one CONSULT is present, the object names are appended to the list in the order they are read; however, if any object name specified is already in the list, it retains its current position in the list.

DIAGNOSE-74 and NODIAGNOSE-74



DIAGNOSE-74

causes the compiler to issue warnings when it encounters source constructs that could cause the program to produce different results than it would if it were compiled with the HP COBOL 74 compiler.

NODIAGNOSE-74

prevents the compiler from issuing warnings when it encounters source constructs that could cause the program to produce different results than it would if it were compiled with the HP COBOL 74 compiler.

Default:	NODIAGNOSE-74
Placement:	Anywhere
Scope:	Applies until its opposite overrides it
Dependencies:	None

The warning messages issued begin with:

```
Logic may differ from COBOL 74 -
```

The possible suffixes, explained in Usage Considerations, are:

- Size of index data item
- Implied value of Figurative Constant
- Precision of division operation
- Action when program not found
- Search algorithm
- Size of receiving operand
- Identification of statement operand
- Assignment order of PERFORM operands
- Alphanumeric sender

Usage Considerations:

- Size of Index Data Item

  A data item within a record description has an explicit USAGE INDEX clause. Indexes occupied two bytes in COBOL 74, but occupy four bytes in HP COBOL.

  You must convert any files that include records in which index data items occur.

- Implied Value of Figurative Constant

  A MOVE statement attempts to assign a value of

  ```
  ALL literal
  ```

  to a numeric or numeric-edited item, or a relation condition involves such a value.

  You must change the literal to deliver the expected value. If X has a picture of 99.9, COBOL 74 handled

  ```
  MOVE ALL "6" TO X
  ```

  as meaning

  ```
  MOVE 6 TO X
  ```

HP COBOL handles the same MOVE as meaning

```
MOVE 66.0 TO X
```

- Precision of Division Operation

  The division operator (/) occurs in an arithmetic expression. In COBOL 74, a division could discard high-level digits without producing a diagnostic. In HP COBOL, a fatal arithmetic overflow can occur.

- Action When Program Not Found

  The OVERFLOW clause is present in a CALL statement. HP COBOL accepts EXCEPTION and NOT EXCEPTION in the CALL statement.

  You must replace the OVERFLOW clauses in CALL statements with EXCEPTION clauses.

- Search Algorithm

  The SEARCH ALL statement in COBOL 74 performed a linear search. In HP COBOL, it performs a binary search.

  You must verify that the table being searched is properly organized (strictly ascending or descending order) so that a binary search is feasible.

- Size of Receiving Operand

  The logic of COBOL 74 and HP COBOL differ in some cases when a value is being assigned to a data structure that contains an item described with an OCCURS DEPENDING phrase.

  One instance is that of an UNSTRING statement assigning a value through an INTO phrase that does not include a DELIMITER phrase, where the DEPENDING item is within the receiving data structure. COBOL 74 uses the computed size of the receiving item, but HP COBOL uses the maximum size.

  Another, more general case is when a MOVE, UNSTRING, or some other value-delivering statement stores a value into a data structure that contains an item described with an OCCURS DEPENDING clause, where the DEPENDING item is not part of the receiving (group) item. COBOL 74 used the maximum size of the receiving item in performing the storage operation in all cases but the one mentioned in the previous paragraph. HP COBOL uses the computed size in all cases where the DEPENDING item is not part of the receiving item.

  You must check that the value-assigning statement will do what you want it to do.

- Identification of Statement Operand

  In COBOL 74, STRING and UNSTRING statements deferred some subscript evaluations until just before the subscripted item was used. HP COBOL evaluates all subscripts once at the beginning of the statement's execution.

  You must check that the STRING and UNSTRING statements do what you want them to do.

- Assignment Order of PERFORM Operands

  In COBOL 74, when a PERFORM VARYING statement controlled multiple loops by using the AFTER phrase, the inner loop base was set before the outer loop index was incremented (or decremented). In HP COBOL, these steps are reversed. When the inner loop logic uses the value of the outer loop's index, the behavior is different. For example, the statement:

  ```
  PERFORM p1 VARYING x FROM 1 BY 1 UNTIL x = 5
              AFTER y FROM x BY 1 UNTIL y = 5.
  ```

  Under COBOL 74 rules, the preceding statement performs these steps in this order:

  1. Sets x to 1, sets y to 1
  2. Runs y from 1 through 4
  3. Sets y to the value of x (still 1), sets x to 2
  4. Runs y from 1 through 4

5. Sets y to the value of x (now 2), sets x to 3
6. Runs y from 2 through 4
7. Sets y to the value of x (now 3), sets x to 4
8. Runs y from 3 through 4
9. Terminates

In HP COBOL, the same statement performs these steps in this order:

1. Sets x to 1, sets y to 1
2. Runs y from 1 through 4
3. Sets x to 2, sets y to the value of x (now 2)
4. Runs y from 2 through 4
5. Sets x to 3, sets y to the value of x (now 3)
6. Runs y from 3 through 4
7. Sets x to 4, sets y to the value of x (now 4)
8. Runs y with the value 4
9. Terminates

You must revise the logic of any PERFORM VARYING containing an AFTER phrase that is controlled by the index of the outer loop. The simplest maneuver is probably to separate the PERFORM into two separate PERFORM statements, because you cannot use an arithmetic expression in the FROM phrase.

- Alphanumeric Sender

  When left-justified numbers with trailing spaces are moved from an alphanumeric data item to a numeric data item described as USAGE DISPLAY or to a numeric edited data item, the result differs between COBOL 74 and HP COBOL. For example, given these descriptions:

  ```
  01 ITEM-A      PIC X(5)  VALUE "05   ".
  01 ITEM-B      PIC 9(5).
  ```

  The statement

  ```
  MOVE ITEM-A TO ITEM-B
  ```

  results in

  ```
  "00005"
  ```

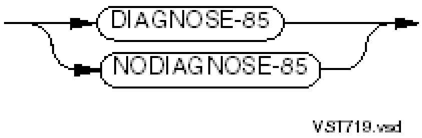
  in ITEM-B for COBOL 74 and

  ```
  "05   "
  ```

  in ITEM-B for HP COBOL.

  If you cannot avoid moving left-justified numbers with trailing spaces between these data types, use an UNSTRING statement with a DELIMITED BY SPACES phrase rather than a MOVE statement.

DIAGNOSE-85 and NODIAGNOSE-85



DIAGNOSE-85

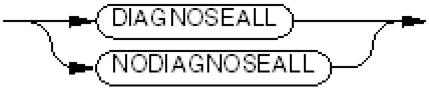
causes the ECOBOL compiler to issue warnings when it encounters source constructs that could cause the program to produce different results than it would if it were compiled with the COBOL85 compiler.

NODIAGNOSE-85

prevents the ECOBOL compiler from issuing warnings when it encounters source constructs that could cause the program to produce different results than it would if it were compiled with the COBOL85 compiler.

Default:	NODIAGNOSE-85
Placement:	Anywhere
Scope:	Applies until its opposite overrides it
Dependencies:	None

DIAGNOSEALL and NODIAGNOSEALL



VST470.vsd

DIAGNOSEALL

causes the compiler to issue a warning each time it encounters an undefined identifier, even if it already issued a warning for a previous use of that identifier.

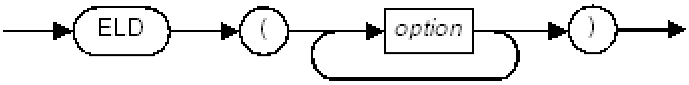
NODIAGNOSEALL

causes the compiler to issue a warning the first time it encounters an undefined identifier, but not if it already issued a warning for a previous use of that identifier.

Default:	NODIAGNOSEALL
Placement:	Anywhere
Scope:	Applies until its opposite overrides it
Dependencies:	None

ELD

ELD passes one or more linker options to the `eld` utility.



VST157.vsd

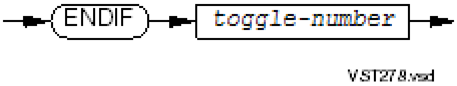
Default:	None
Placement:	In the command line
Scope:	Applies to the compilation unit
Dependencies:	None

`option`

is an `eld` option. For information about `eld` options, see the *eld Manual*.

ENDIF

ENDIF terminates the effect of a preceding IF or IFNOT directive.



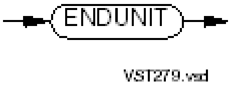
toggle-number

is the *toggle-number* specified in a preceding IF or IFNOT directive.

Default:	None
Placement:	Must be either on a directive line of its own or be the last of a sequence of directives.
Dependencies:	Requires a preceding IF or IFNOT directive with the same *toggle-number*
References:	IF and IFNOT

ENDUNIT

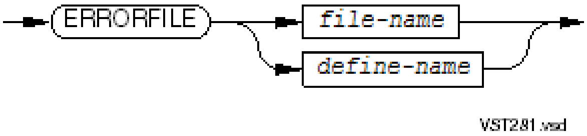
ENDUNIT signals the end of a program unit. Each independent HP COBOL program must end with either an ENDUNIT directive or an END PROGRAM statement; otherwise, programs appear to be nested but lacking their required END PROGRAM statement.



Default:	The compiler detects the end of a Procedure Division by encountering either an END PROGRAM statement or the end of the file.
Placement:	Must be either on a directive line of its own or be the last of a sequence of directives.
Scope:	Applies to program unit
Dependencies:	None

ERRORFILE

ERRORFILE specifies an error-logging file that is to contain information about any compilation errors or warnings. The error-logging file is not an EDIT file. You can use it only with the FIXERRS Macro (page 708).



file-name

is the name of the error-logging file. It must be a disk file name.

define-name

is the name of a MAP DEFINE that refers to a disk file. The compiler uses this file as the error-logging file.

Default:	None
Placement:	Either on the compiler command line or in the source file before the Identification Division.
Scope:	Applies to the compilation unit
Dependencies:	None

If the specified file does not already exist, the compiler creates an entry-sequenced file using the specified file name and the file code 106.

If the specified file is an existing error file identified by file code 106, the compiler replaces the contents of the existing file with the new error file.

If the specified file is an existing file but is not an error file, the compiler terminates compilation and displays the message

```
ERRORFILE not created
```

The error file contains one record for each error or warning message that the compiler found during compilation. Each error record contains this information:

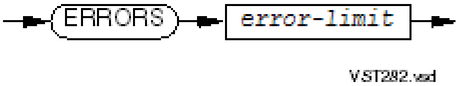
- The fully qualified, local file name of the file in which the compiler found the error or warning

  If you specified a DEFINE name for the name of the source file, the name of the file to which the DEFINE evaluated appears in this entry.

- The EDIT line number in which the error or warning occurred
- The column number in the line where the compiler detected the error or warning
- The error or warning message text

ERRORS

ERRORS sets the maximum number of severe errors allowed during compilation. If this limit is exceeded, the compilation terminates.



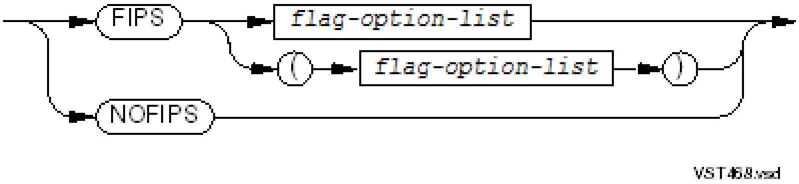
`error-limit`

is an integer in the range 0 through 32767.

Default:	ERRORS 100
Placement:	Anywhere
Scope:	Applies to the compilation unit
Dependencies:	None

FIPS and NOFIPS

FIPS and NOFIPS determine whether the compiler identifies language elements as required by the Federal Information Processing Standard (FIPS).



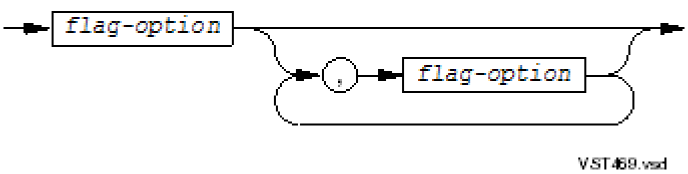
FIPS

identifies the language elements specified by `flag-option-list`.

NOFIPS

prevents identification of language elements which violate the FIPS.

flag-option-list



flag-option

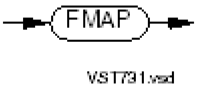
Value	Elements Identified
OBSOLETE	Obsolete language elements
ABOVEMIN	Language elements above the minimum subset
ABOVEINTER	Language elements above the intermediate subset
LEVEL1COM *	Communication language elements
ABOVELEVEL1COM *	Communication language elements above level 1 of communication
LEVEL1DEB	Debug language elements
ABOVELEVEL1DEB *	Debug language elements above level 1 of Debug
REPORTWRITER *	Report Writer language elements
LEVEL1SEG	Segmentation language elements
ABOVELEVEL1SEG	Segmentation language elements above level 1 of segmentation
NONSTANDARDEXT	Nonstandard extensions to HP COBOL

* This option does not affect compilation because the compiler does not support this module. If you use this option, the compiler issues a warning and ignores the option.

Default:	NOFIPS
Placement:	Anywhere
Scope:	Applies until its opposite overrides it
Dependencies:	FIPS is incompatible with STANDARD 2002

FMAP

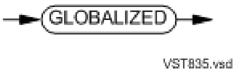
FMAP causes the compiler to produce a source file map, which shows the fully qualified name and timestamp of the IN file and each file specified by a SOURCE directive or COPY statement.



Default:	The compiler does not produce a source file map.
Placement:	Anywhere
Scope:	Applies to the compilation unit
Dependencies:	NOLIST and SUPPRESS do not suppress the source file map that FMAP produces.
References:	• LIST and NOLIST • SUPPRESS and NOSUPPRESS

GLOBALIZED

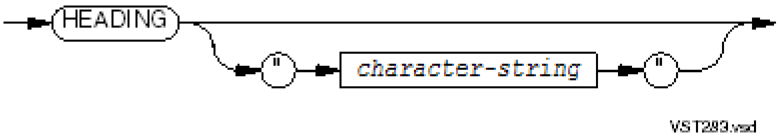
The GLOBALIZED directive directs the compiler to generate preemptable object code. Preemptable object code allows named references in a DLL to resolve to externally-defined code and data items instead of to the DLL's own internally-defined code and data items. By default, the compiler generates non-preemptable object code. Non-preemptable code is more efficient than preemptable code, and results in faster compilation and execution. GLOBALIZED is required only when compiling code that will be linked into a globalized DLL.



Default:	Generate non-preemptable object code
Placement:	On the command line
Scope:	Applies to the compilation unit
Dependencies:	None

HEADING

HEADING replaces or sets to spaces the heading portion of the standard top-of-page line that appears on each page of the compilation listing.



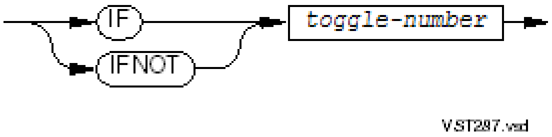
`character-string`

is a string of one or more ASCII characters. If a quotation mark character is part of the string, it must be represented as two contiguous quotation marks. The string is used in all subsequent top-of-page lines.

If `character-string` is omitted, the heading portion of the top-of-page lines is set to all spaces.

Default:	Standard top-of-page line
Placement:	Anywhere
Scope:	Applies until another HEADING overrides it
Dependencies:	None

IF and IFNOT



IF

enables compilation of subsequent source text if the toggle with `toggle-number` is set; disables it otherwise.

IFNOT

suppresses compilation of subsequent source text if the toggle with `toggle-number` is set; enables it otherwise.

toggle-number

is an integer in the range 1 through 15.

Default:	None
Placement:	Must be either on a directive line of its own or be the last of a sequence of directives
Dependencies:	Requires a preceding IF or IFNOT directive with the same *toggle-number*
References:	IF and IFNOT

The toggles are turned on and off by use of the SETTOG and RESETTOG directives. If no SETTOG directive has been issued, a toggle is off.

COPY and REPLACE statements are not affected by this directive; they are still processed and expanded. In contrast, a SOURCE directive is ignored when it is part of the ignored text.

If a given toggle is not set, any directives between an IF for that toggle and its subsequent ENDIF are parsed but are not acted upon; if a given toggle is set, any directives between an IFNOT for that toggle and its subsequent ENDIF are parsed but are not acted upon.

Example 12-8 IF Directive

```
?IF 1
    ADD A B C GIVING D
?ENDIF 1
?IFNOT 1
    COMPUTE D = A + B + C
?ENDIF 1
```

In Example 12-8, if toggle 1 is turned on, the ADD statement is compiled; if toggle 1 is turned off, the COMPUTE statement is compiled.

NOTE: The IFNOT directive does not behave like an ELSE clause of the IF directive. Each IF and each IFNOT must have its own ENDIF directive.

Example 12-9 Nested IF and IFNOT Directive Scopes

IF and IFNOT scopes can be nested. Given this program fragment:

```
        01 MASTER-RECORD.
           03 HEADER              PICTURE X(30).
?IF 1
           03 ACCOUNT-ID          PICTURE X(50).
?IF 2
           03 ACCOUNT-ID-REDEF REDEFINES ACCOUNT-ID.
              05 ACCOUNT-GROUP    PICTURE X(20).
              05 ACCOUNT-CLASS    PICTURE X(10).
              05 ACCOUNT-NUMBER   PICTURE X(20).
?ENDIF 2
           03 TRAILER             PICTURE X(100).
?ENDIF 1
```

The compiler sees this, depending on the setting of toggles 1 and 2:

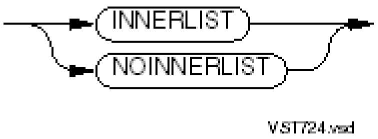
```
Toggles On   Source Lines Compiled

None         01 MASTER-RECORD.
                  03 HEADER               PICTURE X(30).

1 only       01 MASTER-RECORD.
                  03 HEADER               PICTURE X(30).
                  03 ACCOUNT-ID           PICTURE X(50.
                  03 TRAILER              PICTURE X(100).

2 only       01 MASTER-RECORD.
                  03 HEADER               PICTURE X(30).

1 and 2      01 MASTER-RECORD.
                  03 HEADER               PICTURE X(30).
                  03 ACCOUNT-ID           PICTURE X(50).
                  03 ACCOUNT-ID-REDEF REDEFINES ACCOUNT-ID.
                       05 ACCOUNT-GROUP   PICTURE X(20).
                       05 ACCOUNT-CLASS   PICTURE X(10).
                       05 ACCOUNT-NUMBER  PICTURE X(100).
                  03 TRAILER              PICTURE X(100).
```

Lines whose compilation IF or IFNOT suppresses still appear in the compiler listing. To keep them from appearing, use NOLIST and LIST directives as the code fragments in Example 12-10 show.

Example 12-10 Omitting Uncompiled Lines from Compiler Listing

```
?IF n
?NOLIST
?ENDIF n

?IFNOT n
* Lines compiled if toggle n is not set go here
?ENDIF n

?IF n
?LIST
?ENDIF n

?IFNOT n
?NOLIST
?ENDIF n

?IF n
* Lines compiled if toggle n is set go here
?ENDIF n

?IFNOT n
?LIST
?ENDIF n
```

INNERLIST and NOINNERLIST



INNERLIST

lists, immediately after each source statement, the generated machine instructions in mnemonic form.

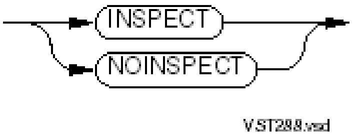
NOINNERLIST

suppresses the listing of generated machine instructions.

Default:	NOINNERLIST
Placement:	Anywhere

Scope:	Applies until its opposite overrides it
Dependencies:	INNERLIST works only if LIST is active and SUPPRESS is not.
References:	• LIST and NOLIST • SUPPRESS and NOSUPPRESS

INSPECT and NOINSPECT



INSPECT

turns on the Inspect attribute for the debugging tool for the compilation unit.

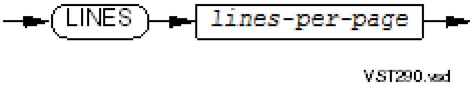
NOINSPECT

turns off the Inspect attribute for the debugging tool for the compilation unit.

Default:	INSPECT
Placement:	Anywhere
Scope:	The last INSPECT or NOINSPECT in the compilation unit applies to the compilation unit.
Dependencies:	NOINSPECT and SAVEABEND override each other (whichever is last is active).
References:	SAVEABEND and NOSAVEABEND

LINES

LINES determines the number of lines to be listed on each page of the output file. Whenever the next line of a listing would cause a page to exceed the specified line count, the compiler ejects the page and begins a new listing page. It prints the standard page heading at the top of the new page, followed by two blank lines, and then the next line of the listing.

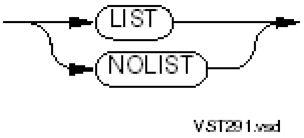


lines-per-page

is an integer in the range 10 through 32767.

Default:	LINES 60
Placement:	Anywhere
Scope:	Applies until another LINES overrides it
Dependencies:	LINES works only if paging applies to the compilation list device.

LIST and NOLIST



LIST

lists each source image in the output file.

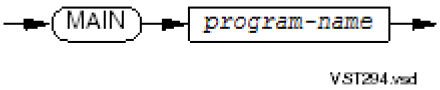
NOLIST

suppresses the listing of each source image in the output file.

Default:	LIST
Placement:	Anywhere
Scope:	Applies until its opposite overrides it
Dependencies:	SUPPRESS overrides LIST (and therefore CODE, CROSSREF, ICODE, LMAP, MAP, SHOWCOPY, and SQL). NOLIST suppresses INNERLIST.
References:	• SUPPRESS and NOSUPPRESS • INNERLIST and NOINNERLIST • MAP and NOMAP • SHARED (page 576) • SQL and NOSQL

MAIN

MAIN identifies the main program unit.



program-name

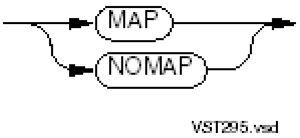
is the program-name (specified in the PROGRAM-ID paragraph) of the program that is the main entry point of the target file being produced by the compilation.

Default:	Every program unit that does not have a Linkage Section is compiled as a main program. If more than one program unit of a compilation unit qualifies as a main program, the compiler reports this as an error, compiles the first qualifying program unit as the main program, and produces multiple object files.
Placement:	Before the Identification Division header of the first program unit in the compilation unit.
Scope:	Applies to the program that contains it
Dependencies:	None

When MAIN appears, only the program unit indicated (the one beginning with a PROGRAM-ID paragraph that specifies the same program-name) is compiled as a main program. All other program units in the compilation unit are compiled as called programs, whether or not they include a Linkage Section.

If no PROGRAM-ID paragraph in the compilation unit specifies the same program-name as that specified in MAIN, the compiler reports that the resulting target file has no main program. It is therefore not executable.

MAP and NOMAP



MAP

lists a symbol table listing, which appears after the source program listing for the program unit (see Symbol Table Listing (page 788)).

NOMAP

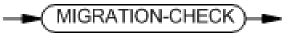
suppresses the symbol table listing.

Default:	NOMAP
Placement:	Anywhere
Scope:	The last MAP or NOMAP applies.
Dependencies:	MAP works only if LIST is active and SUPPRESS is not.
References:	• LIST and NOLIST • SUPPRESS and NOSUPPRESS

MIGRATION-CHECK

MIGRATION-CHECK causes the compiler to issue a warning message when it encounters a user-defined COBOL word that is a reserved word in the COBOL-2002 standard. MIGRATION-CHECK applies only when used with the STANDARD 1985 directive.

Use the DIAGNOSEALL and NODIAGNOSEALL directives to control whether the compiler issues a message for every occurrence of a given user-defined word or only for the first occurrence, respectively.



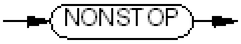
VST837.vsd

Default:	No warnings issued
Placement:	In the command line or before the Identification Division of the first program unit in the source text.
Scope:	Applies to the compilation unit
Dependencies:	Applies only when STANDARD 1985 is enabled.
References:	• STANDARD (page 582) • DIAGNOSEALL and NODIAGNOSEALL

NONSTOP

NOTE: Do not use this directive in the OSS environment.

NONSTOP causes the compiler to accept the STARTBACKUP verb. The STARTBACKUP and CHECKPOINT verbs enable you to run a program as a process pair.



VST296.vsd

Default:	The program cannot run as a process pair (even if it contains a PARAM NONSTOP ON command).
Placement:	Must appear before the Identification Division of the first program unit in the source text.
Scope:	Applies to the program unit
Dependencies:	PARAM NONSTOP OFF command overrides NONSTOP.
References:	PARAM Command

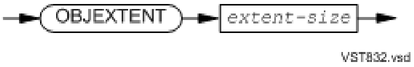
When a program is compiled without this directive, a PARAM NONSTOP command is not enough to make the program run as a process pair.

If NONSTOP is absent, the compiler reports an error if it finds a STARTBACKUP verb.

NOTE: Chapter 33: Fault-Tolerant Processes (page 963), explains how to use the HP fault-tolerant facility. It is your responsibility to design the program to make use of the CHECKPOINT and STARTBACKUP statements to operate in a fault-tolerant manner. The presence of the NONSTOP directive and CHECKPOINT and STARTBACKUP statements do not guarantee fault tolerance.

OBJEXTENT

The OBJEXTENT directive specifies the size of the extents allocated for the generated object file.



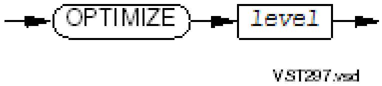
extent-size

specifies the number of pages to use for the primary and secondary extents of the generated object file. `extent-size` is an integer in the range 2 through 1070. Use this directive to increase the capacity of the generated object file. For more information about extents, see the *Guardian Programmer's Guide*

Default:	OBJEXTENT 32
Placement:	On the Guardian command line
Scope:	Applies to the compilation unit
Dependencies:	You cannot specify OBJEXTENT and the linker option `-NS_extent_size` on the same command line

OPTIMIZE

OPTIMIZE specifies the extent to which the compiler attempts to optimize the code it emits.



`level`

is 0, 1, or 2. Optimization level has only a small effect on affects compilation speed. The effect of each level is:

Level	Effect
0	Code is not optimized. Provided in case other optimization levels cause errors. Supports symbolic debugging; data is always in memory.
1 (default)	Code is optimized within statements and across statement boundaries. The resulting code is more efficient than that produced by lower levels of optimization. Supports symbolic debugging; data is not always in memory.
2	Code is optimized within statements and across statement boundaries, and the resulting code is more efficient than code produced by lower levels.

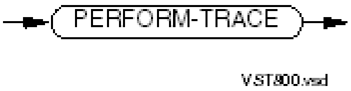
NOTE: Code generated under OPTIMIZE 1 is more difficult to debug than code generated under OPTIMIZE 0. Optimization under OPTIMIZE 2 obscures statement boundaries. Before you attempt to use a symbolic debugger to debug a program, do either:

- Be certain that the program was compiled with OPTIMIZE 0.
- Use the output generated by the INNERLIST directive to be certain of statement boundaries.

Default:	OPTIMIZE 1
Placement:	Outside the boundary of a separately compiled program
Scope:	The optimization level active at the beginning of a separately compiled program determines the level of optimization for that program and any programs it contains.
Dependencies:	None

PERFORM-TRACE

PERFORM-TRACE gives additional information if run-time error 148, “PERFORM nesting too deep,” occurs.



If the program was compiled with the PERFORM-TRACE directive and run-time error 148 occurs, this additional information is given:

```
Line numbers of PERFORMs from latest to earliest
Line Number    File Number
llllll.lll     nnn
...
```

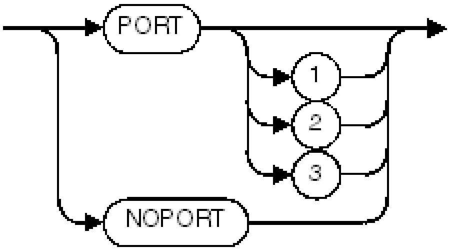
The file number *nnn* is the number of the source file. The `source-file` specified in the compilation command has file number 001. A file number is shown for each file referenced by the directive SOURCE or a COPY Statement (page 507). To see the file numbers, compile the program with SHOWFILE directive (see SHOWFILE and NOSHOWFILE).

Default:	The compiler does **not** provide additional information if run-time error 148 occurs.
Placement:	Anywhere
Scope:	Applies to the compilation unit
Dependencies:	None

PORT and NOPORT

PORT and NOPORT determine:

- How BINARY/COMPUTATIONAL data items are aligned
- Whether the CALL statement in a program behaves as defined in X/Open and XPG4 specification.



VST600.vsd

PORT

aligns BINARY/COMPUTATIONAL data items on byte boundaries (unless they are described with the SYNCHRONIZED clause, which aligns them on 2-byte boundaries).

PORT also allows the CALL statement in a program to behave as it does in X/Open and XPG4; that is, to call non-COBOL routines, pass parameters by value, and return the values of functions.

1

- Allows CALL to call programs other than COBOL programs
- Prevents the compiler from issuing a warning when the Linkage Section does not having a matching item in the PROCEDURE DIVISION header
- Causes the external variable RETURN-CODE to be automatically defined

2

prevents binary items from being aligned on word boundaries.

3

does what both PORT 1 and PORT 2 do. PORT 3 is equivalent to PORT.

NOPORT

aligns BINARY/COMPUTATIONAL data items according to standard alignment rules and prevents the CALL statement in a program from calling non-COBOL routines, passing parameters by value, and returning the values of functions.

Default:	NOPORT
Placement:	Anywhere
Scope:	Applies to the compilation unit
Dependencies:	None

Usage Considerations:

- NATIVE-*n* Data Items and Pointers

  The PORT directive does not affect the alignment of NATIVE-*n* data items or pointers.

- Run-Time Performance

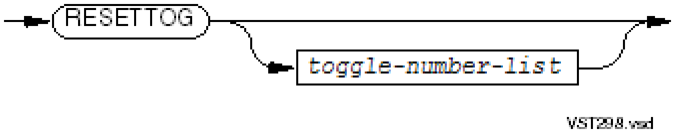
  The PORT directive can severely impact n HP COBOL program's run-time performance.

- Passing Parameters to Programs Not Compiled With PORT

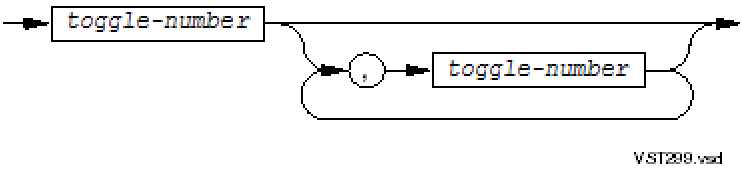
  If a program compiled with the PORT directive calls and passes parameters to a program not compiled with the PORT directive (or compiled with the NOPORT directive), serious problems can arise from differences in the way BINARY/COMPUTATIONAL data items are aligned in the two programs.

RESETTOG

RESETTOG turns off all specified toggles, which are used by the IF directive.



toggle-number-list



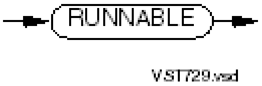
toggle-number

is an integer in the range 1 through 15. If *toggle-number* is omitted, all toggles are turned off.

Default:	None
Placement:	Must be either on a directive line of its own or be the last of a sequence of directives
Scope:	Applies until SETTOG overrides it
Dependencies:	SETTOG overrides it.
References:	SETTOG

RUNNABLE

RUNNABLE causes the compiler to use the linker to produce a loadfile if there are no compilation errors.



Default:	The compiler does **not** use the linker to produce a loadfile if there are no compilation errors.
Placement:	Anywhere
Scope:	Applies to the compilation unit

Dependencies:	SYNTAX overrides RUNNABLE. You must specify RUNNABLE to use RUNNAMED, SAVEABEND, NOSAVEABEND, or SUBTYPE in the compiler. With CALL-SHARED (the default), RUNNABLE uses the linker to produce a PIC loadfile. With SHARED, RUNNABLE uses the linker to produce a PIC library file (DLL).
References:	• SYNTAX • RUNNAMED • SAVEABEND and NOSAVEABEND • SUBTYPE • CALL-SHARED • SHARED (page 576)

If you run the compiler with the CALL-SHARED or SHARED directive, and without the directive RUNNABLE or SYNTAX, and no compilation errors occur, you can produce a loadfile by running the `eld` utility separately (see the *eld Manual*).

RUNNAMED

RUNNAMED specifies that the run unit is to be run as a named process.

VST300.vsd

Default:	None
Placement:	Any of: • Anywhere in a source program • In a linker session (specify `-change`)
Scope:	Applies to the program in which or before which it appears, to all subsequent programs in the compilation unit, and to all target files
Dependencies:	RUNNAMED is appropriate only if the program was compiled with RUNNABLE (because a linkfile is not a run unit).
References:	RUNNABLE

If your program is not compiled with the RUNNAMED directive and you run the program with the command

```
RUN ru
```

the process has no process name. If your program is compiled with the RUNNAMED directive or you run the program with the command

```
RUN ru/NAME/
```

the operating environment gives it a timestamp name. The RUNNAMED directive causes the operating environment to give the process a timestamp name without your having to include "/NAME/" in the RUN command.

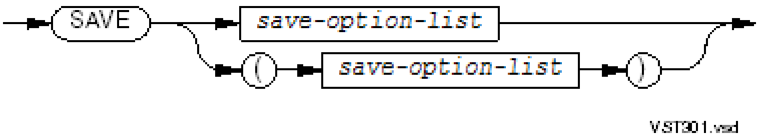
Naming the process allows any requester that uses this run unit as a server to use the old method of process handling. The old method of process handling does not work for processes that run with a high PIN.

You can specify RUNNAMED for any run unit.

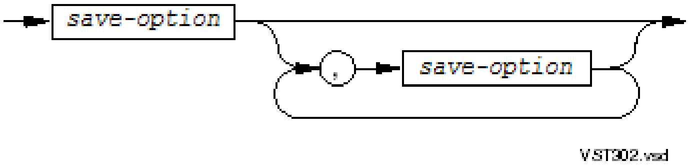
SAVE

NOTE: Do not use this directive in the OSS environment.

The SAVE directive specifies which initialization messages are stored for possible program manipulation.

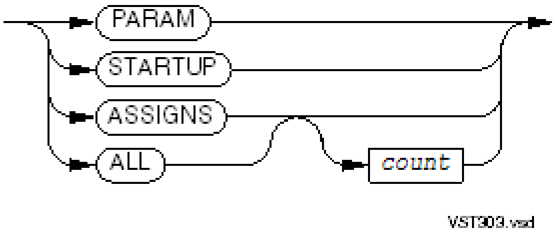


save-option-list



save-option-list cannot be continued onto subsequent lines, even if it is enclosed in parentheses.

save-option



PARAM

specifies that the PARAM message is to be saved.

STARTUP

specifies that the startup message is to be saved.

ASSIGNS

specifies that all ASSIGN messages are to be saved.

count

is an unsigned integer in the range 1 through 100, an estimate of the number of messages to be saved. The default is 19.

The compiler uses *count* to estimate process data-space requirements for the saved ASSIGN messages. The saved ASSIGN messages are in the data block #CRE_HEAP. If there is not enough space for all the ASSIGN messages held by the command interpreter, the program terminates with a run-time error message.

ALL

specifies that all messages are to be saved.

Default:	The compiler does not save initialization messages.
Placement:	Anywhere

Scope:	Applies to the compilation unit
Dependencies:	None

Whenever an HP COBOL program begins execution, its creator sends it a series of initialization messages:

- A startup message (always)
- One PARAM message (if any parameters are active)
- An ASSIGN message for each assignment active

You cannot directly manipulate saved messages as COBOL data items, but you can use utility routines to retrieve and modify their contents (see Saved Message Utility (SMU) Overview (page 619)).

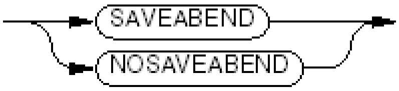
All combinations of SAVE options are legal, as are multiple SAVE directives. Redundant options are ignored, except that the last-seen count overrides any previous ones. The directive

```
SAVE ALL 17, ASSIGNS 10
```

is equivalent to

```
SAVE ALL, ASSIGNS 10
```

SAVEABEND and NOSAVEABEND



VST304.vsd

SAVEABEND

generates a saveabend file if the process terminates abnormally. Sets the Inspect attribute.

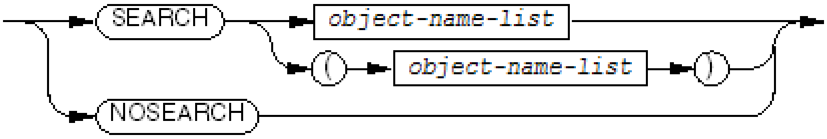
NOSAVEABEND

does not generate a saveabend file if the process terminates abnormally.

Default:	NOSAVEABEND
Placement:	Anywhere
Scope:	The last SAVEABEND or NOSAVEABEND in the compilation unit applies to the entire compilation unit.
Dependencies:	SAVEABEND and NOINSPECT override each other (whichever is last is active). SAVEABEND and NOSAVEABEND are appropriate only if the program was compiled with RUNNABLE.
References:	• INSPECT and NOINSPECT • RUNNABLE

SEARCH and NOSEARCH

NOTE: The compiler ignores the NOSEARCH directive and issues a warning.



VST305.vsd

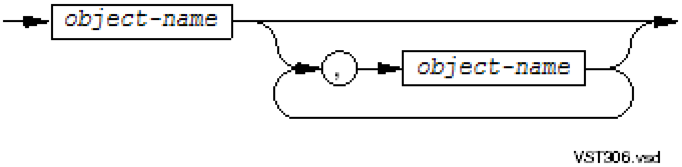
SEARCH

adds the files that *object-name-list* specifies to the primary search list. (See Primary Search List.)

NOSEARCH

is ignored.

object-name-list



can be continued onto subsequent lines, but the left parenthesis must appear on the same directive line as the keyword SEARCH. Each continuation line for a SEARCH directive must have a question-mark (?) in the indicator area. If a SEARCH directive spans multiple lines, they must be consecutive (no blank lines, comment lines, or program text lines can intervene).

object-name

is a name that designates a TNS/E object file—either a file-system file name or (in the Guardian environment) a DEFINE name. In the OSS environment, *object-name* must be an OSS pathname. If the file-system file name is not fully qualified with system, volume, and subvolume names, the compiler uses the current default system, volume, and subvolume names or those specified by an =_OBJECT_SEARCH DEFINE (see =_OBJECT_SEARCH (page 538)).

The linker links the entire file designated by *object-name*.

object-name must designate either a linkfile or an archive file.

When an archive file is searched, the entire archive file is generally not linked into the target object file. Only those member files are linked that contain procedures named in CALL or ENTER statements in the COBOL program being compiled. Whenever a procedure is needed from a member file, the entire member file is linked.

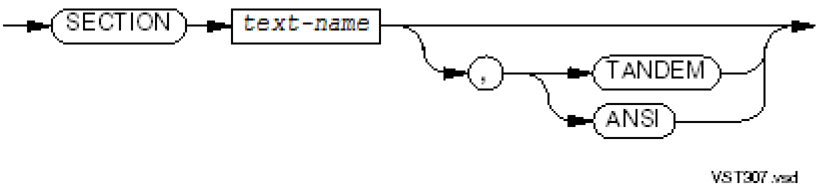
If an HP COBOL program references the object in a CALL or ENTER statement, the object must have been compiled with symbols.

Default:	NOSEARCH
Placement:	Anywhere
Scope:	Applies to the compilation unit
Dependencies:	None
References:	• RUNNABLE • CONSULT and NOCONSULT

If more than one SEARCH directive appears, the file names are appended to the list in the order read.

SECTION

SECTION identifies a section of text to be copied from either a COPY library by a COPY statement or from a source library by a SOURCE directive.



text-name

is a COBOL word (see COBOL Words (page 73)).

TANDEM

specifies that subsequent source text is in Tandem reference format, which is described in Reference Format for Source Program Lines (page 54).

ANSI

specifies that subsequent source text is in ANSI reference format, which is described in Chapter 17: ANSI Reference Format (page 711).

Default:	None
Placement:	Must be on a directive line of its own. If it appears in the text of the compilation source file (the IN file), it is ignored.
Scope:	Applies only to the section of text that it specifies
Dependencies:	None

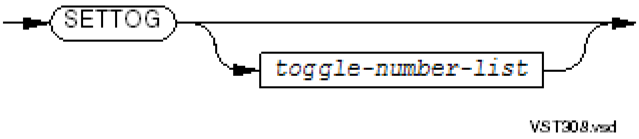
If you do not specify TANDEM or ANSI in the SECTION directive, the compiler assumes that the format of the library text is the same as the current source-text format. Although you can override this by inserting a compiler directive as the first line following the SECTION directive, the input format control directive (ANSI or TANDEM) of the SECTION directive is usually more convenient for this purpose. In either case, selecting Tandem or ANSI format as an option has an effect only for the duration of the fetching of that portion of library text. Upon completion of the fetching operation, the compiler reverts to the previous format. (See COPY Statement (page 507) and SOURCE).

If two or more SECTION directives in a text library contain the same `text-name`, the text following the first one in the library (the one on the edit line that has the lowest line number) is the text fetched by a COPY statement or SOURCE directive.

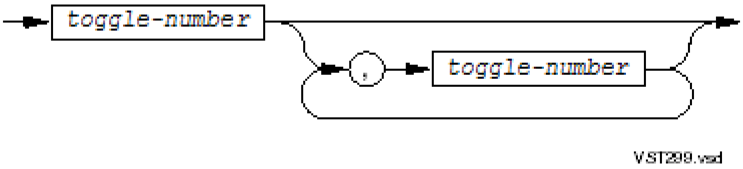
If any SECTION directive in a text library contains a `text-name` that is not a legal COBOL word, the first time the compiler attempts to fetch any member from the library, the compiler reports an error, even if the offending `text-name` is not the one sought.

SETTOG

SETTOG turns on all specified toggles, which the IF directive uses.



toggle-number-list



If *toggle-number-list* is omitted, all toggles are turned on.

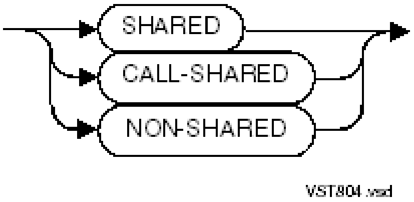
toggle-number

is an integer in the range 1 through 15.

Default:	None
Placement:	Must either be on a directive line of its own or be the last of a sequence of directives.
Scope:	Applies until RESETTOG overrides it
Dependencies:	None
References:	RESETTOG

SETTOG or RESETTOG on the command line does not override any such directives within the source stream. Command line directives are the exact equivalent of a directive in the source stream immediately before the first line of the source.

SHARED



SHARED

generates shared code (PIC) for a DLL.

CALL-SHARED

generates shared code (PIC).

NON-SHARED

is ignored.

Default:	CALL-SHARED
Placement:	Anywhere
Scope:	The last SHARED or CALL-SHARED in the compilation unit applies to the entire compilation unit. The last SHARED, CALL-SHARED, or NON-SHARED in the compilation unit applies to the entire compilation unit.
Dependencies:	• If RUNNABLE is active, SHARED uses the linker to create a PIC library file (DLL); otherwise, SHARED creates a PIC linkfile. • Do not use with CALL-SHARED or UL.
References:	• CALL-SHARED • RUNNABLE • UL

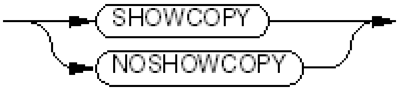
If you put an HP COBOL program in a DLL, you must export the program name when you build the DLL; otherwise, other programs cannot call the HP COBOL program. To export the HP COBOL program name, use one of these `ld` commands:

Environment	ld Commands

Guardian	`eld (-export` *`program-name`* `)`
	or
	`eld (-export_all)`

OSS or PC	`-Weld="-export` *`program-name`* `"`
	or
	`-Weld="-export_all"`

SHOWCOPY and NOSHOWCOPY



VST310.vsd

SHOWCOPY

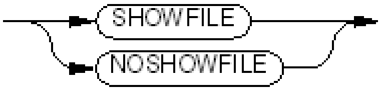
displays COPY statements as comments in the listing before the copied text.

NOSHOWCOPY

suppresses the display of COPY statements.

Default:	SHOWCOPY
Placement:	Anywhere
Scope:	Applies until its opposite overrides it
Dependencies:	SHOWCOPY works only if LIST is active and SUPPRESS is not.
References:	• LIST and NOLIST • SUPPRESS and NOSUPPRESS

SHOWFILE and NOSHOWFILE



VST311.vsd

SHOWFILE

displays the identity of the file from which source text is being read under the control of the SOURCE directive.

NOSHOWFILE

suppresses the display of the identity of the file from which source text is being read under the control of the SOURCE directive.

Default:	NOSHOWFILE
Placement:	Anywhere
Scope:	Applies until its opposite overrides it
Dependencies:	None

SHOWFILE applies only to source file transitions due to the effect of SOURCE directives. The compiler does not generate lines to report transitions caused by COPY statements.

Whenever the compiler begins reading a different source file in response to a SOURCE directive and whenever it resumes reading from the prior source file after a SOURCE directive's effect terminates, it inserts a line of this format into the listing:

```
Source file:  [n]  file-name  yyyy-mm-dd  hh:mm:ss
```

n

is the file ordinal assigned by the compiler to the source file. Every time the compiler encounters a SOURCE directive, it increments the value of *n* by 1.

file-name

is the fully qualified source file name.

yyyy-mm-dd

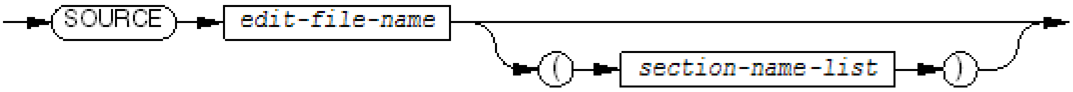
is the date when the source file was last modified.

hh:mm:ss

is the time when the source file was last modified.

SOURCE

SOURCE causes the compiler to begin accepting source text from a different EDIT file. The compiler can read either the entire file or one or more sections of the file (demarcated by SECTION directives). You can also use the predefined SEARCH DEFINEs to specify one or more subvolumes to be searched for unqualified files (see Predefined SEARCH DEFINEs).

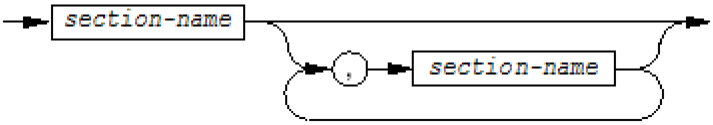


VST312.vsd

edit-file-name

is a file-system file name that identifies an accessible EDIT file. If any of the system, volume, or subvolume components are omitted, the compiler supplies the missing ones from its process defaults. *edit-file-name* can be the name of a DEFINE of the class MAP.

section-name-list



VST313.vsd

section-name-list can be continued onto subsequent lines, provided that the *edit-file-name* and the left parenthesis appear on the same directive line as the keyword SOURCE. Each continuation line for a SOURCE directive must have a question-mark (?) in the indicator area. If a SOURCE directive spans multiple lines, they must be consecutive (no blank lines, comment lines, or program text lines can intervene).

The default for *section-name-list* is the entire file.

`section-name`

is the name of a section in an EDIT file (a `text-name` in the directive SECTION).

Default:	None
Placement:	Must be the last directive on its line.
Dependencies:	None

When the compiler encounters a SOURCE directive, it suspends reading of the current source file and begins reading lines from the specified file. If no section-name list appears, the compiler incorporates the entire file into its stream of source text lines. If a section-name list does appear, the compiler incorporates text from the designated sections only, in the order in which the sections appear within the specified file (not in the order of the list). When the compiler reaches the end of the file specified in the SOURCE directive, or has copied all specified sections, it resumes reading from the prior source file.

Usage Considerations:

- Effect on Compilation Output Listing

  If a SHOWFILE directive is active, when the compilation encounters an input line containing a SOURCE directive, (assuming that the LIST and NOSUPPRESS directives are active) the compiler lists the directive line, then a line showing the fully qualified name of the new source file and the date and time it was last modified.

  When no SHOWFILE directive is active (the default condition), the directive line containing the SOURCE directive appears in the output, but the additional line does not.

  The text from the section summoned by the SOURCE directive is listed, subject only to the LIST and SUPPRESS directives.

- Nesting of SOURCE Directives

  The text introduced by a SOURCE directive can include one or more SOURCE directives. The compiler processes nested SOURCE directives when it encounters them. The maximum nesting depth is three; that is, at most three source files can be open at any given time as a consequence of SOURCE directives (source files open in response to COPY statements are not counted in this respect).

- Relationship Between SOURCE Directive and COPY Statement

  The effect of the SOURCE directive is neither a subset nor a superset of the effect of the COPY statement; they are completely independent.

  The SOURCE directive does not provide a REPLACING capability.

  The text that a COPY statement delivers can include one or more SOURCE directives. The text that a SOURCE directive delivers can include one or more COPY statements. A COPY statement is not permitted to deliver a SOURCE directive that delivers another COPY statement, because this would create the effect of nested COPY statements.

  A SOURCE directive cannot appear anywhere in the text of a COPY statement (between the word COPY and its terminating period in the input stream) or a REPLACE statement (between the word REPLACE and its terminating period in the input stream); however, the compiler detects and reports its appearance within pseudo-text only if that pseudo-text is introduced into the source as the result of expanding the COPY statement or by the editing effects of the REPLACE statement.

- SOURCE Directive Behavior in IF/IFNOT/ENDIF Sequences

  The compiler ignores a SOURCE directive specified within conditional text (within the scope of an IF or IFNOT directive) if the controlling condition is false.

- Interaction of SOURCE and COLUMNS Directives

  When a COLUMNS directive applies to the line or lines containing a SOURCE directive, the same logical line length attribute is the default for the lines introduced in response to that directive; however, a COLUMNS directive appearing within the designated file overrides the default. In this circumstance, the compiler honors the "inner" logical line length for the remainder of the text imported by the SOURCE directive, after which the "outer" logical line length becomes active again.

- Effect of ANSI and TANDEM Format Specifying Directives

  The source text format (ANSI or Tandem) which applies to the line or lines containing a SOURCE directive is the default for the lines introduced in response to that directive; however, ANSI or TANDEM directives appearing within lines read from the file (or ANSI or TANDEM options specified in SECTION directives) take effect when encountered. These then apply to subsequent lines until they are overridden, the end of the source file is reached, or the end of a selected section is reached (which causes the compiler to revert to the default source format before it resumes reading the next selected section). In all cases the "outer" source format attribute becomes active again after processing of the SOURCE directive completes.

- Diagnostic Messages and SOURCE Directives

  If the compiler cannot at least minimally process a SOURCE directive (because the edit-file-name specified is invalid, or the file cannot be accessed, or the file is not an EDIT file, and so forth), the compilation terminates.

  If the compiler is unable to access a section specified in the SOURCE directive (because the section-name is invalid or the section is not in the file) the compiler issues a diagnostic message, but does not terminate the compilation.

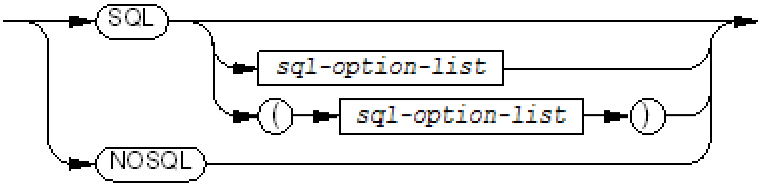
- Format of a Source Library

  A source library has the same format as a COPY library; in fact, the only difference between a source library and a COPY library is context—a source library is referenced by a SOURCE directive and a COPY library is referenced by a COPY statement. For the syntax of a source library, see Library Format (page 514).

  During program compilation, the compiler identifies a section by locating the SECTION directive whose `text-name` matches the `section-name` specified in the SOURCE directive.

SQL and NOSQL

NOTE: The compiler ignores the NOSQL directive and issues a warning.



VST315.vsd

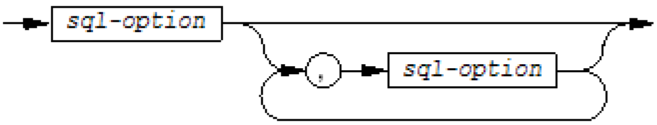
SQL

tells the compiler to expect SQL/MP statements in the compilation unit.

NOSQL

is ignored.

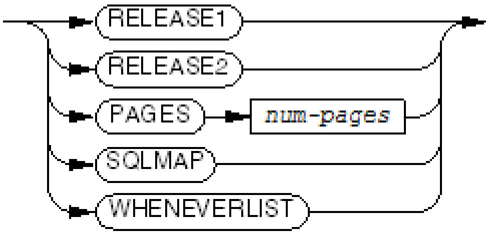
sql-option-list



VST316.vsd

tells the compiler how much memory to allocate to the SQL/MP compiler interface and what SQL/MP information to print to the listing file.

sql-option



VST317.vsd

RELEASE1

is for the COBOL85 compiler only—see the *COBOL Manual for TNS and TNS/R Programs*.

RELEASE2

causes the compiler to accept only SQL/MP Release 2 features; however, the resulting object file can be executed with SQL/MP Release 350 (or later).

If you do not specify RELEASE2, the ECOBOL selects the latest SQL/MP product version (350 or later).

PAGES

allocates `num-pages` of memory to the SQL compiler interface for processing SQL/MP statements. Each page is 2048 bytes.

num-pages

has a minimum (and default) value of 860 and the maximum value of 1000. If you allocate too few pages, an error results.

SQLMAP

includes information about SQL/MP statements in the listing file so that they can be matched with MEASURE output. MEASURE output identifies each SQL/MP statement by its Run-Time Data Unit (RTDU) name and SLT index. An RTDU is an internal SQL/MP data structure residing in the code file. An SLT index maps a single SQL/MP statement to a table in the RTDU. By default, the SQLMAP option is off.

WHENEVERLIST

lists the WHENEVER options that are active for each embedded SQL/MP statement printed to the listing file. WHENEVER options are a feature of SQL/MP. For information on them, see the *SQL/MP Reference Manual* or the *SQL/MX Programming Manual for C and COBOL*. By default, the WHENEVERLIST option is off.

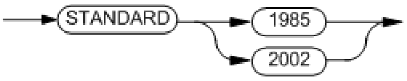
Default:	None. If the program contains SQL/MP statements, the SQL directive is required; otherwise, it is unnecessary.
Placement:	In the compiler command line.

Scope:	Applies to the compilation unit
Dependencies:	SQL works only if LIST is active and SUPPRESS is not.
References:	• LIST and NOLIST • SUPPRESS and NOSUPPRESS

STANDARD

STANDARD specifies whether the ECOBOL compiler is to apply the COBOL-1985 standard or the COBOL-2002 standard when compiling a program. STANDARD-1985 causes the compiler to follow the COBOL-1985 standard; the compiler does not reserve any of the COBOL words that have become reserved in the COBOL-2002 standard, and therefore, does not support any of the COBOL-2002 features that are not already supported as extensions.

STANDARD-2002 causes the compiler to reserve all COBOL words that are reserved in the COBOL-2002 standard; that is, those words cannot be used as user-defined words. Note that this does not mean that the compiler actually supports all features of COBOL-2002.



VST836.vsd

1985

Causes the compiler to apply the COBOL-1985 standard.

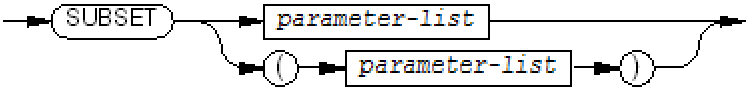
2002

Causes the compiler to apply the COBOL-2002 standard.

Default:	STANDARD 1985
Placement:	In the command line
Scope:	Applies to the compilation unit
Dependencies:	STANDARD 2002 is incompatible with the FIPS and SUBSET directives. Whichever directive is specified first takes effect; whichever is specified next causes the compiler to issue a warning and has no other effect.
References:	• FIPS and NOFIPS • SUBSET (page 582)

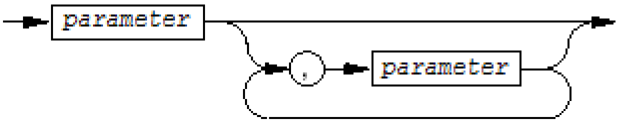
SUBSET

SUBSET causes the compiler to report in the output listing any source statement that contains syntax that is obsolete or does not conform to the restrictions of a specified subset.



VST319.vsd

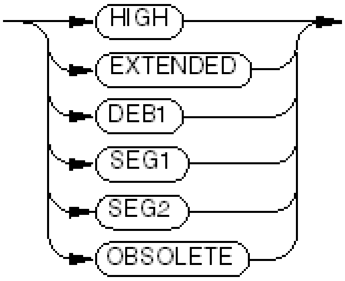
parameter-list



VST320.vsd

You cannot continue *parameter-list* onto subsequent lines, even if you enclose it in parentheses.

parameter



VST321.vsd

HIGH

reports syntax not defined in the High subset of ISO/ANSI COBOL (NUC 2, SEQ 2, REL 2, INX 2, IPC 2, STM 2), with respect to the required functional modules.

EXTENDED

accepts all defined syntax without comment (HP extensions plus those of HIGH), with respect to the required functional modules.

DEB1

accepts all syntax defined in level 1 of the Debug module without comment.

If DEB1 is not specified, the compiler reports syntax defined in level 1 of the Debug functional module (DEB 1).

Syntax defined in level 2 of the Debug module causes the compiler to report an error, because HP COBOL does not support level 2 of this module.

SEG1

accepts all syntax defined in level 1 of the Segmentation module (SEG 1) without comment, but reports syntax defined in level 2 (SEG 2).

If SEG1 is not specified, the compiler reports syntax defined in the Segmentation functional module.

Although HP COBOL does not support the Segmentation functional module, syntax that has no semantic effect is not diagnosed. This behavior simplifies the conversion of existing programs to HP COBOL.

SEG2

accepts all syntax defined in the Segmentation module without comment.

If SEG2 is not specified, the compiler reports syntax defined in level 2 of the Segmentation functional module.

OBSOLETE

accepts all obsolete syntax without comment unless it fails to conform to the specified subset; in that case, the compiler reports nonconformance rather than obsolescence.

If OBSOLETE is not specified, the compiler reports syntax classified as obsolete, regardless of whether it conforms to the specified subset in other respects.

The specified parameters can appear in any order. If two or more parameters are mutually incompatible, the last one overrides the others (redundancy is permitted). If neither HIGH nor EXTENDED is specified, EXTENDED is assumed by default.

Default:	None
Placement:	Must appear before the first COBOL source program (that is, before the first Identification Division header).
Scope:	The last valid SUBSET in the program applies.
Dependencies:	SUBSET is incompatible with the STANDARD 2002 directive

Usage Considerations:

- Invalid SUBSET Directives

  If a SUBSET directive appears in an invalid context, the compiler issues an appropriate message and, after analyzing the directive, ignores it and cancels the effect of any previously accepted SUBSET directive (since an error has occurred).

  Upon the occurrence of any error, the compiler cancels any active SUBSET directive and discards any messages not yet issued.

- Messages

  After the compiler has processed a separately compiled program that contains unauthorized syntax (but no actual errors), it displays the message

  ```
  *** The preceding program does not conform to the specified subset ***
  ```

  and then describes each nonconforming element.

  Each reported instance of nonconformance or obsolescence identifies the context of the syntax, describes its location in the source text, and characterizes the syntax as "obsolete" or "nonconforming." If an instance is both obsolete and nonconforming, only "obsolete" is reported. A message has either of these forms:

  ```
  context on line nnnnn at
    column xx contains obsolete syntax
  Syntax on line nnnnn at column xx is defined in level
  ```

  or

  ```
  context on line nnnnn at
  column xx contains nonconforming syntax
  Syntax on line nnnnn at column xx is defined in level
  ```

  context

  is a header (such as "Data Division header"), statement (such as "ADD statement"), clause (such as "LINAGE clause"), or the general phrase "Source text."

  nnnnn

  is a five-digit line number.

  xx

  is a two-digit column number.

  level

  is HIGH, EXTENDED, DEB1, SEG1, or SEG2, as appropriate.

Syntax on line *nnnnn* at column *xx* is defined in *level*

includes the location phrases only if they differ from the information in the first line.

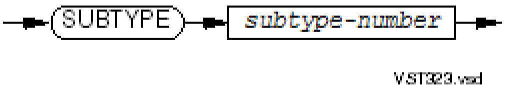
The messages generated in accordance with the SUBSET directive are issued as a block after each separately compiled program.

To report ...	Use the directive ...
Obsolete syntax	`SUBSET EXTENDED, DEB1, SEG2`
Syntax unlikely to be portable	`SUBSET HIGH, DEB1, SEG2, OBSOLETE`

SUBTYPE

NOTE: Do not use this directive in the OSS environment.

SUBTYPE causes the main program created by the current compilation to execute as a process of a specified subtype.



subtype-number

is an integer in the range 0 through 63.

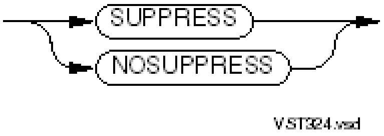
You can define the meaning and behavior of process subtypes 48 through 63.

HP defines the meaning and behavior of process subtypes 1 through 47 (see the *Guardian Programmer's Guide*).

Default:	SUBTYPE 0
Placement:	Anywhere
Scope:	The last SUBTYPE in the program applies.
Dependencies:	SUBTYPE works only with RUNNABLE.
References:	RUNNABLE

The compiler installs the value of *subtype-number* in the object file header. When the object file executes as a named process, it is assigned the process subtype specified by the value of *subtype-number*. Any user can create a named process that has a subtype in this range.

SUPPRESS and NOSUPPRESS



SUPPRESS

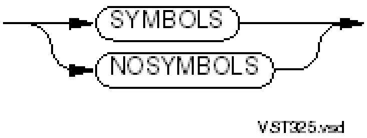
overrides the LIST directive, suppressing all compiler listing output except the compiler leader text, diagnostic messages, and the compiler trailer text.

NOSUPPRESS

overrides the SUPPRESS directive.

Default:	NOSUPPRESS
Placement:	Accepted anywhere, but to suppress the compiler listing without altering the source text, put SUPPRESS on the command line
Scope:	Applies until its opposite overrides it
Dependencies:	SUPPRESS overrides LIST (and therefore, it also overrides CODE, CROSSREF, ICODE, LMAP, MAP, SHOWCOPY, and SQL).
References:	• LIST and NOLIST • MAP and NOMAP • SHOWCOPY and NOSHOWCOPY • SQL and NOSQL

SYMBOLS and NOSYMBOLS



SYMBOLS

includes a symbol table in the object file used by a symbolic debugger.

NOSYMBOLS

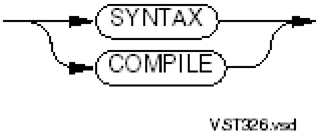
excludes the symbol table from the object file used by a symbolic debugger.

Default:	NOSYMBOLS
Placement:	Accepted anywhere, but to affect a program other than the first program in a compilation unit, SYMBOLS must follow either an ENDUNIT directive or an END PROGRAM statement, which signals the end of the preceding program’s Procedure Division. If you want a symbol table for a given separately compiled program, put SYMBOLS before its Identification Division header. If the compiler encounters a subsequent SYMBOLS or NOSYMBOLS directive within the text of that separately compiled program, it issues a warning and ignores that directive.
Scope:	Applies until its opposite overrides it
Dependencies:	SYMBOLS does not work when SYNTAX is active.
References:	• ENDUNIT • SYNTAX

You can include or exclude the symbol table from the object file on a program-unit-by-program-unit basis.

If the SYMBOLS directive is active, the compiler does not perform certain object code optimizations, so the object program might execute slightly more slowly and be slightly larger in size.

SYNTAX



SYNTAX

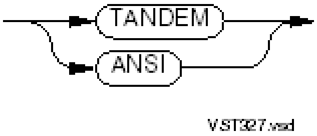
checks the syntax of the source text. No target file is produced.

COMPILE

compiles the program unit and included its code and data blocks in the target file being created.

Default:	COMPILE
Placement:	Outside the boundaries of a separately compiled program; that is, not between the Identification Division header of a separately compiled program and its end, which is marked by one of: • The corresponding END PROGRAM statement • ENDUNIT • The end of the source file
Scope:	The last COMPILE or SYNTAX in the compilation unit applies to the entire compilation unit.
Dependencies:	SYNTAX overrides RUNNABLE.
References:	• ENDUNIT • RUNNABLE

TANDEM



TANDEM

specifies that subsequent source text is in Tandem reference format, which is described in Reference Format for Source Program Lines (page 54).

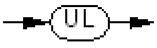
ANSI

specifies that subsequent source text is in ANSI reference format, which is described in Chapter 17: ANSI Reference Format (page 711).

Default:	TANDEM
Placement:	Anywhere
Scope:	TANDEM or ANSI within a section of text obtained from a copy library or source library is effective only for the length of that text section. When the compiler reverts to the source file where it found the COPY verb or SOURCE directive, the previously active reference format applies.
Dependencies:	None
References:	ANSI

UL

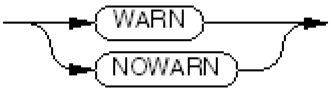
UL directive tells the compiler that the resulting object code will be in a DLL (that is, it has the same effect as the SHARED directive).



VST735.vsd

Default:	None
Placement:	Before the first IDENTIFICATION DIVISION header of the first program in the compilation unit
Scope:	Applies to the compilation unit
Dependencies:	• If RUNNABLE is active, UL uses the linker to create a PIC library file (DLL); otherwise, UL creates a PIC linkfile. • Do not use with CALL-SHARED or SHARED.
References:	• CALL-SHARED • SHARED (page 576)

WARN and NOWARN



VST390.vsd

WARN

reports compilation and Binder warnings in the listing.

NOWARN

suppresses compilation and Binder warnings in the listing.

Default:	WARN
Placement:	Anywhere
Scope:	Applies to the compilation unit
Dependencies:	If LIST is not enabled, the last line of source text processed is also listed to provide a point of reference for each warning message.
References:	LIST and NOLIST

13 Program Execution

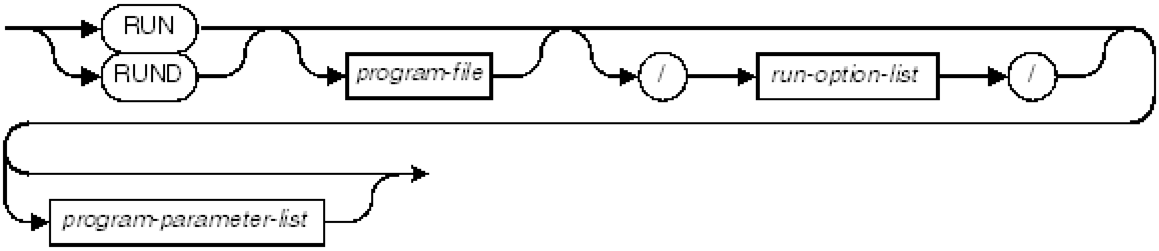
This section applies to the Guardian environment. If you are executing HP COBOL programs in the OSS environment, see Running HP COBOL Programs (page 722).

For detailed information about the TACL commands, see the *TACL Reference Manual*. For more information about running COBOL programs in the Guardian environment, see Chapter 26: Executing and Debugging HP COBOL Programs (page 835).

When a system executes a loadfile, the running program is called a process.

RUN or RUND Command

Both the RUN and RUND commands run a loadfile as a process. The RUND command calls the selected debugger before beginning execution.



VST331.vsd

RUN

runs the program without the debugger.

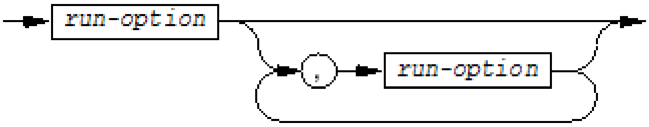
RUND

runs the program with the debugger.

program-file

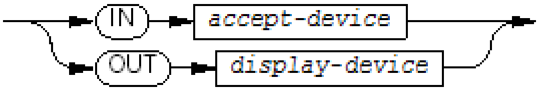
is the name of the loadfile to run. *program-file* can be represented as a file-system file name or (in the Guardian environment) a DEFINE name.

run-option-list



VST332.vsd

run-option



VST333.vsd

is an option of the RUN command, described in the *TACL Reference Manual*. For COBOL programs, two of the run options, IN and OUT, have special significance:

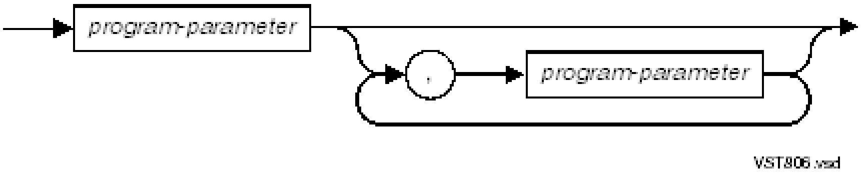
accept-device

has a value that designates a terminal or a process and can be a file-system file name or (in the Guardian environment) a DEFINE name. *accept-device* specifies the device from which an unqualified ACCEPT statement retrieves input. If you omit this option, unqualified ACCEPT statements retrieve input from the home terminal (unless PARAM EXECUTION-LOG specifies a different device).

display-device

can be a file-system file name or (in the Guardian environment) a DEFINE name. Its value designates a terminal, line printer, process, existing entry-sequenced disk file, or operator console to which unqualified DISPLAY statements are to deliver output. If you omit *display-device*, unqualified DISPLAY statements deliver output to the home terminal (unless PARAM EXECUTION-LOG specifies a different device).

program-parameter-list



program-parameter

is a parameter of the program that is to be run.

Example 13-1 RUN Commands

```
RUN PROG1
RUN RUNUNIT /IN $TERM1/
RUN MYPROG /OUT =CHECKS-PRINTER/
```

Usage Considerations:

- Effect of Commands on Run-Time Environment

  Certain run-time commands establish an environment in which the running process operates. For example, the ASSIGN command stores logical file assignments, the PARAM command stores string values and associates them with parameter names, the CLEAR command clears assignments and parameter values, and a group of commands handle DEFINEs.

  When a process begins executing, it obtains the current ASSIGN and PARAM values from the Guardian environment, and can act on the information they contain. See ASSIGN Command and PARAM Command. For additional information on TACL run-time commands, see the *TACL Reference Manual* or the *Guardian User's Guide*.

  If no run-time commands are specified, the file-control entries in the program are used to associate the COBOL file names with file-system file names, and all external switches but one are set to OFF. The only exception is the NONSTOP parameter, which defaults to ON if the program was compiled with the NONSTOP directive in the Guardian environment.

- When Commands Take Effect

  Commands take effect when encountered and can be overridden by subsequent commands. Keywords, file names, and numbers cannot be continued from one line to the next.

- COMMENT Run-Time Command

  A COMMENT run-time command is available to include documentation in the TACL command stream.

ASSIGN Command

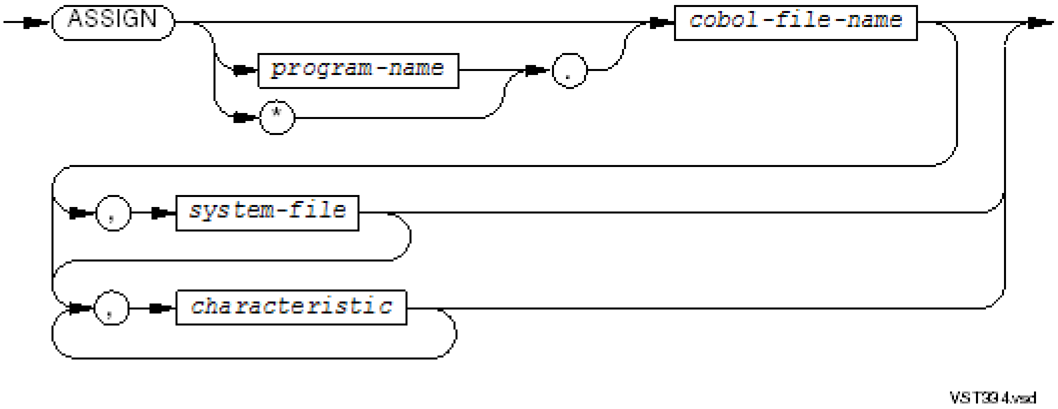
This section applies to programs that were not compiled with the ENV LIBRARY or UL directive. Programs compiled with the ENV LIBRARY or UL directive ignore ASSIGN commands.

NOTE: ASSIGN commands for files in a DLL are ignored.

The ASSIGN command creates an ASSIGN message in your current run-time environment. A process can use the information in an ASSIGN message to override the file-system file name, or other attributes, of a file specified in the source program. This command is fully discussed in the *TACL Reference Manual*.

When a process begins to execute, it can request its ASSIGN messages and use their contents to modify the information it possesses about its files before opening them. During the initialization of any process compiled from a COBOL source, the run-time routines automatically handle any ASSIGN messages made available by the Guardian environment.

This partial syntax diagram of the ASSIGN command describes features of greatest interest to the COBOL programmer:



program-name

is the *program-name* in the PROGRAM-ID paragraph in the Identification Division, for which the file assignment is to take effect. This is distinct from the name of the loadfile. Do not specify *program-name* for an external file.

*

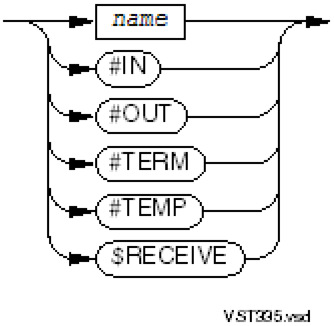
means that the file assignment applies to all programs in the loadfile. You must not specify * for an external file.

If neither *program-name* nor * is present, then *cobol-file-name* must be unique among all programs in the loadfile.

cobol-file-name

is the COBOL file name in a SELECT clause, also called the *fd-name*.

system-file



name

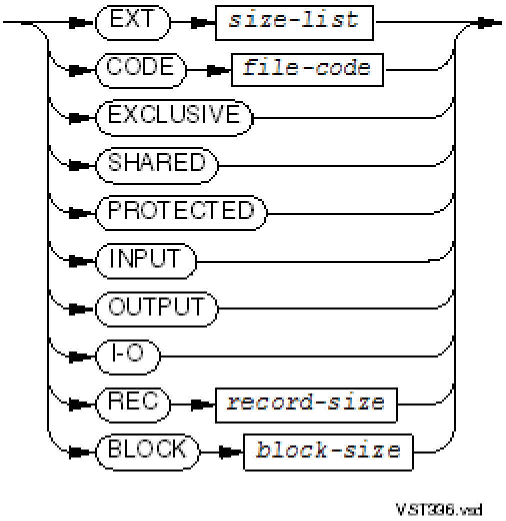
is a spooler location or the name of a disk file, disk volume, tape device, DEFINE, process, or terminal.

#IN, #OUT, #TERM, #TEMP, $RECEIVE

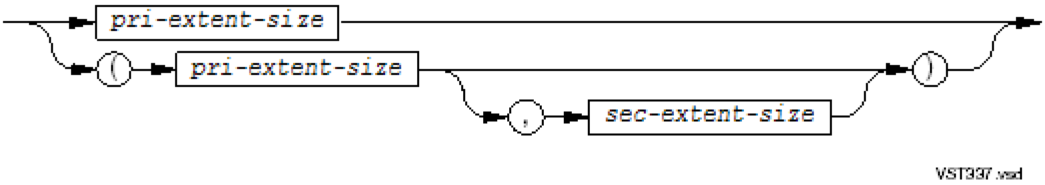
are explained in the usage considerations. They must be uppercase.

When only *system-file* is present, and all other clauses are omitted, the current file for *cobol-file-name* is assigned.

characteristic



size-list



specifies the extent size for disk file creation. Extent sizes are specified as a number of file pages to be allocated for one extent of the file. A file page is 2048 bytes of storage.

pri-extent-size

is an unsigned integer specifying the primary extent size. If EXT is omitted, *pri-extent-size* defaults to 4.

sec-extent-size

is an unsigned integer specifying the secondary extent size. If EXT is omitted, *sec-extent-size* defaults to 20. If *sec-extent-size* is omitted, the secondary extent size defaults to the primary extent size.

file-code

is 101 (for an EDIT file) or an unsigned integer in the range 0 through 99 or in the range 1000 through 32767, which specifies a file code for disk file creation. If the CODE clause is omitted, a *file-code* of 0 is used.

EXCLUSIVE, SHARED, PROTECTED

specify an exclusion mode for disk file reading or writing. This can override an implied exclusion mode, but not an explicit exclusion mode specification in the OPEN statement.

I-O, INPUT, OUTPUT

specify an access mode for the disk file; however, the run-time routines ignore the value. The COBOL OPEN statement controls the access mode entirely.

record-size

is an integer accepted by TACL as specifying record length, but the run-time routines ignore the value. The record declaration in the COBOL program controls the record size entirely.

block-size

is an integer accepted by TACL as specifying block size. *block-size* determines the block size.

NOTE: To see what ASSIGN commands are currently in force, enter just ASSIGN followed by a carriage return.

Usage Considerations:

- Places for ASSIGN Commands

  You can give ASSIGN commands individually at the TACL prompt before you enter the RUN command, or you can place the ASSIGN commands in an EDIT-format file and use the OBEY command to execute all the commands in the file. You could also include the RUN command in the OBEY command file.

- Special COBOL Names for Operating System Files

  See Operating system file names and Special names for operating system files.

  ### Example 13-2 #TEMP

  ```
  17>ASSIGN temp-file,#TEMP
  18>RUN program
  ...
  * File created on run-time default volume
  OPEN I-O temp-file
  ...
  * File purged
  CLOSE temp-file
  ...
  * New empty file created
  OPEN I-O temp-file
  ...
  * File automatically closed and purged
  STOP RUN.
  ...
  ```

- Interaction Between ASSIGN Command and OPEN Statement

  The interaction between the *exclusion-specification* of the ASSIGN command and the *exclusion-mode* of the OPEN statement is:

  — If an *exclusion-mode* is given in the OPEN statement, that value is used regardless of any ASSIGN command settings.

  — If the OPEN statement has no *exclusion-mode* but the ASSIGN command for the file does give an *exclusion-specification*, the ASSIGN command value is used.

  — If neither the *exclusion-mode* of the OPEN statement nor the *exclusion-specification* of the ASSIGN command is present, the COBOL default mode is active (see OPEN (page 385)).

The temporary file in Example 13-3 is purged when it is closed.

Example 13-3 ASSIGN Command With Special COBOL Names

```
11>ASSIGN REPORT-OUT, $TERM4
12>RUN MYPROG

13>ASSIGN IN-MASTER,$DATA.ZZ.INFILE
14>RUN MYPROG1

15>                                == FD INPUT-FILE is the
16>ASSIGN INPUT-FILE, #IN          == file named in the
17>                                == IN option.

18>                                == FD WORK-FILE is to be
19>ASSIGN WORK-FILE, #TEMP         == a temporary disk file
```

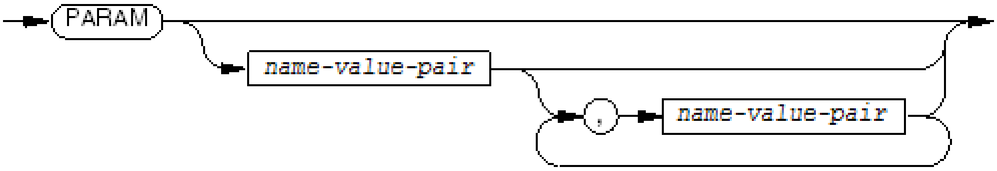
```
20>                               == on the default volume.

21>                                                == FD OUTPUT-FILE is the
22>ASSIGN OUTPUT-FILE, #OUT        == file named in the
23>                               == OUT option.

24>ASSIGN RECEIVE-FILE, $RECEIVE

25>                                  == Run the loadfile,
26>                                  == designating NEWINP as
27>                                  == the IN file (thus

28>RUN PROG3 /IN NEWINP,OUT NEWOUT/ == equating it with the
29>                                  == fd-name INPUT-FILE)
30>                                  == and NEWOUT as the OUT
31>                                  == file (thus equating it
32>                                  == with the fd-name
33>                                  == OUTPUT-FILE).
```

PARAM Command

The PARAM command creates a PARAM message in your current environment.

NOTE: PARAM commands for files in a DLL are ignored.

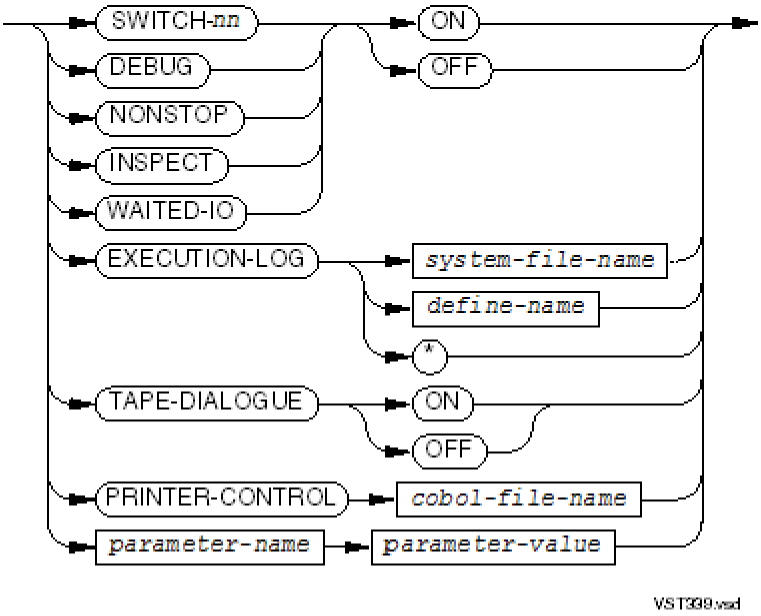
When a process compiled by the compiler begins to execute, the run-time routines automatically request the PARAM message and use its contents to set the options defined for HP COBOL. Application code within the program can also use the SMU procedures to copy the PARAM message into the Working-Storage area and examine it for additional parameters that apply to the program. Further, the program can manipulate the saved parameters and cause them to be passed to any processes it initiates using the CREATEPROCESS or CLU_PROCESS_CREATE_ routine.



VST339.vsd

name-value-pair

is one of the name-value pairs recognized by the run-time routines, or a user-defined parameter name and value:



SWITCH-*nn*

sets (ON) or clears (OFF) the specified switch. In the Guardian environment, the value of *nn* can be a one-digit integer between 1 and 9 or a two-digit integer between 01 and 15. In the OSS environment, the value of *nn* can be a one-digit integer between 1 and 9 or a two-digit integer between 10 and 15. (For an explanation of these switches, see SPECIAL-NAMES Paragraph (page 118). In the absence of a PARAM SWITCH-*nn* ON command, SWITCH-*nn* is off.

DEBUG

is used only if the program was compiled with the DEBUGGING clause and contains USE DEBUGGING sections. When set to ON, the program executes debug code (declaratives). In the absence of a PARAM DEBUG ON command, the debugging declaratives are not executed.

NONSTOP

controls whether or not a process runs as a process pair, provided that the NONSTOP directive was included in the compilation of the program. For information on using process pairs, see Chapter 33: Fault-Tolerant Processes (page 963).

If the program was compiled with the NONSTOP directive, the process runs as a process pair unless you use the PARAM NONSTOP OFF command.

If this switch is set OFF, no checkpointing or backup creation occurs, and PROGRAM-STATUS is always set to “0000.”

INSPECT

determines what the run unit does after reporting fatal errors. If INSPECT is OFF, the run unit calls ABEND. If INSPECT is ON, the run unit calls the selected debugger.

WAITED-IO

is ignored.

EXECUTION-LOG

specifies the destination of messages issued by the HP COBOL run-time library routines, which are:

- Diagnostic messages
- DISPLAY messages that do not specify a mnemonic-name when the default OUT device is the home terminal

The default EXECUTION-LOG is the home terminal.

`system-file-name`

is the name of the file to which run-time messages are directed. It must be an existing entry-sequenced (TYPE E) file (that is, a file-system file name). If `system-file-name` does not begin with a dollar sign ($), backward slash (\), or number sign (#), then it must be enclosed in quotation marks unless it forms a COBOL word. For more information about operating system file names, see the *Guardian Procedure Calls Reference Manual*.

`define-name`

is the name of a DEFINE associated with the file to which run-time messages are directed. It is not enclosed in quotation marks.

*

specifies that all run-time messages are to be discarded.

TAPE-DIALOGUE

is ignored.

PRINTER-CONTROL

specifies that the operating environment must return control to the program when the printer specified by the `cobol-file-name` is not ready or is out of paper.

`cobol-file-name`

is the FD name of a file in a COBOL program (not the operating system file name or a DEFINE name).

When the PRINTER-CONTROL parameter is absent, the HP COBOL run-time routines provide automatic handling for printer fault conditions, without control returning to the program until the printer is again made ready.

`parameter-name`

is a parameter to be handled through the SMU. For information on the SMU, see Saved Message Utility (SMU) Overview (page 619).

`parameter-value`

is the value to be given to `parameter-name`.

NOTE: To see what parameters are currently defined, enter just PARAM followed by a carriage return.

Usage Considerations:

- WAITED-IO and HP COBOL Fast I-O

  If a program for which WAITED-IO was specified attempts to open a file for HP COBOL Fast I-O, sequential block buffering of buffered cache is used instead, degrading performance by a factor of five to ten.

- EXECUTION-LOG Parameter

  There are three circumstances in which a COBOL program can interact with the home terminal without your explicitly instructing it to do so:

  — A DISPLAY statement does not include an UPON phrase, so it defaults to the OUT file, which itself is defaulted to the home terminal.

  — An ACCEPT statement does not include a FROM phrase, so it defaults to the IN file, which itself is defaulted to the home terminal.

  — A run-time routine needs to report an error or to engage in dialog with the system operator, and normally does so through the home terminal.

  The EXECUTION-LOG parameter provides a means of redirecting such unintentional input-output activity from the home terminal to another device, called the execution log file.

  The EXECUTION-LOG parameter can also redirect unintentional input-output activity from #IN and #OUT to the execution log file.

  In the absence of an EXECUTION-LOG parameter, the execution log file is the home terminal.

  If the EXECUTION-LOG parameter specifies asterisk (*), such output is discarded. In this case, whenever the program attempts to engage the operator in dialog, such as for:

  — Tape mount

  — Device not ready

  — No write ring

  — Printer is out of paper

  The run-time routines return I-O status code “30” to the program. The special register GUARDIAN-ERR then contains a code indicating what action is expected.

  If a program has its EXECUTION-LOG parameter set to asterisk (*), and the program attempts to execute an ACCEPT statement that defaults to the home terminal, the run unit terminates abnormally.

  If the EXECUTION-LOG parameter specifies a file name, the processing of the designated file depends on the device-type assigned to the file:

Device Type	Exclusion	Access
0 (process)	Exclusive	Read/write
1 (operator console)	Shared	Write
3 (disk)	Exclusive	Write
4 (magnetic tape)	Exclusive	Write
5 (line printer)	Exclusive	Write
6 (terminal)	Shared	Read/write

The file is opened the first time the run unit attempts to write a message. This means that it will not be opened at all for most jobs. If the open operation fails, no messages can be written anywhere and run unit execution terminates abnormally. Once opened, the file remains open for the duration of the run unit.

If any error occurs during a write or read operation on the execution log file, no messages can report this problem and the run unit terminates abnormally.

If the access mode is read/write, the run-time routines use the WRITEREAD system routine to send messages requiring a response and WRITE for messages not requiring a response.

If the access mode is write, the run-time routines use a WRITE request for all messages. Those requiring a response are assumed to have a response of end of file.

If the device is a process, the process is responsible for analyzing the messages to determine if an operator response is needed. If the process is also a COBOL run unit, it can do this by using the MESSAGE-SOURCE option in its (the process's) RECEIVE-CONTROL paragraph to get the appropriate information to use in an ENTER RECEIVEINFO call. If RECEIVEINFO returns a nonzero value for the read-count, a reply is expected. In this case, the process can return an appropriate string value (such as would be supplied by an operator) indicating what action to take.

If a reply is null or the access mode of the execution log is write, the run-time routines assume that no action is desired and, if the operation is a file processing statement, return I-O Status 30 to the program (GUARDIAN-ERR contains the code indicating what action is expected). If the operation is an ACCEPT statement, the default values are returned to the receiving item. In the case of tape mount messages, the run unit terminates abnormally.

- PRINTER-CONTROL Parameter

  When an application program is writing to a printer, and the printer leaves the ready state (perhaps it is out of paper or it is off line), the program does not automatically regain control. The HP COBOL run-time library issues a message to the home terminal to report the printer fault, and waits for the printer to be restored to the ready state.

  In certain circumstances, the application program needs to regain control and take some action when an active printer leaves the ready state. For example, the program might be printing checks and keeping record of the preprinted check numbers issued to each payee. The program asks the operator for the first check number in the sequence, then processes checks depending on the number to increase by 1 each time. When the printer comes to the end of a box of check forms (or suffers a paper jam and destroys some checks), the program needs to be informed that a particular write operation was not completed routinely so that it can ask the operator for the number of the next check it will be printing on.

  The PRINTER-CONTROL parameter provides a way for the program to respond to printer faults. Suppose a PRINTER-CONTROL parameter specifies the file name CHEX. This causes the run-time library to keep track of each elementary control function or write request that it sends to the operating environment for the file CHEX. If one of those operations fails because the printer to which CHEX is assigned is not ready, the WRITE statement completes with I-O status code “30” (permanent error condition) and a GUARDIAN-ERR value of 100 (not ready) or 102 (out of paper).

  To make use of the PRINTER-CONTROL parameter, the application program must use a declarative procedure, preferably one of the form

  ```
  USE AFTER STANDARD ERROR PROCEDURE ON CHEX
  ```

  that responds only to errors on the printer file.

  The procedure receives control when the operating environment detects the printer problem. The procedure determines that the printer's I-O status code is “30” and handles the GUARDIAN-ERR values of 100 and 102. It might notify the operator of which problem has occurred.

Although the record area still contains the record that was being written, there is no way for the procedure to determine whether or not the record actually has been written to the printer. For example, if “WRITE a-rec BEFORE ADVANCING PAGE” was specified, perhaps the printer was off line before the write operation began, or perhaps the record was written but the printer reached the end of the paper before it could position to the top of a new sheet.

There is, however, a way to assure that the record does get written only once and that any related control operations occur only once. The program must reissue exactly the same form of the WRITE statement (same record name, same ADVANCING clause, and so on). The run-time library does not repeat any control function or system write request that successfully completed the first time, and re-issues the control function or system write request that caused the status to occur, plus any subsequent operations that were not completed.

One easy way to use this mechanism is to have the declarative respond to the combination of I-O status code “30” and GUARDIAN-ERR value of 100 or 102 by moving a FALSE value to a condition variable (associated to the condition-name PRINT-OK) and conducting the dialog with the operator, then instead of coding just a WRITE statement, code the sequence:

```
PERFORM UNTIL PRINT-OK
  SET PRINT-OK TO TRUE
  WRITE CHEX BEFORE ADVANCING 10
END-PERFORM
```

- Additional Considerations
  - A parameter value that contains any embedded commas or leading or trailing spaces must be enclosed in quotation marks. Between the delimiting quotation marks, two consecutive quotation marks ("") represent any one quotation mark (") that is part of the value. The delimiting quotation marks are not stored.
  - TACL provides internal storage for 1024 bytes of parameters. Each parameter occupies a number of bytes equal to:

    ```
    2 +
    number-of-characters-in-parameter-name +
    number-of-characters-in-parameter-value
    ```

  - Multiple PARAM name and value pairs must be delimited by commas.
  - The compiler makes use of certain parameters and their values in performing the compilation (see PARAM Commands (page 536)).

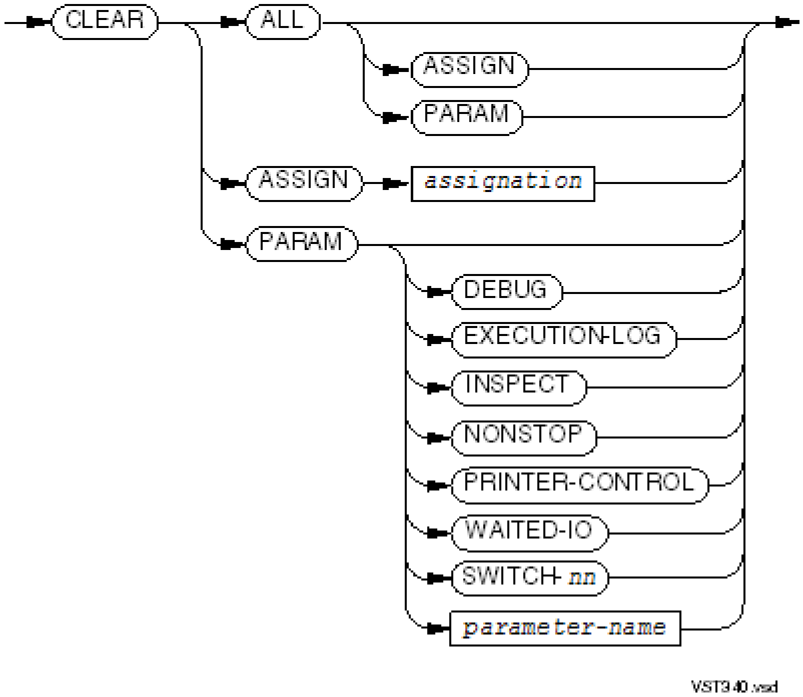
For more information on the PARAM command, see the *TACL Reference Manual*.

Example 13-4 PARAM Command

```
85> ==     Set the DEBUG switch, and set SWITCH-02
86> PARAM DEBUG ON, SWITCH-02 ON
87> ==     Set user parameter THEDATE
88> PARAM THEDATE 1991DEC02
89> ==     Route home terminal messages to a file
90> PARAM EXECUTION-LOG $ARK.GKH.D0289
91> ==     Return control when the file that "myprog"
92> ==     calls PRINT-FL leaves the ready state
93> PARAM PRINTER-CONTROL PRINT-FL
94> RUN myprog
...
99> CLEAR ALL PARAM
```

CLEAR Command

Use the CLEAR command to revoke a specific ASSIGN or PARAM command, all of the ASSIGN commands, all of the PARAM commands, or all of both commands.



ALL

means clear all currently active ASSIGN and PARAM references.

ALL ASSIGN

means clear all currently active ASSIGN references.

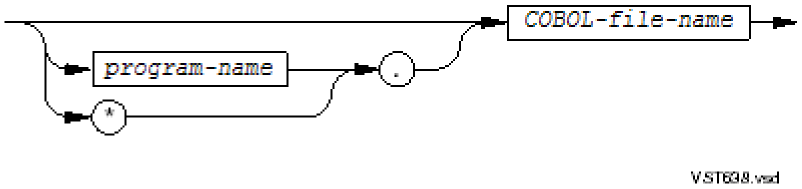
ALL PARAM

means clear all currently active PARAM references.

ASSIGN

means clear the reference to the specified COBOL file name.

assignation



specifies what is to be cleared.

COBOL-file-name

is the COBOL name of a file.

program-name

is the name of the COBOL program that contains *COBOL-file-name*.

*

means all programs in the object file.

PARAM

means clear the reference to the specified parameter.

DEBUG, EXECUTION-LOG, INSPECT, NONSTOP, PRINTER-CONTROL, SWITCH-nn, WAITED-IO, *parameter-name*

are parameters of the PARAM command (see PARAM Command).

Example 13-5 CLEAR Command

```
43>CLEAR ALL              { Clears all ASSIGN and PARAM values }
44>CLEAR ALL ASSIGN       { Clears all ASSIGN values }
45>CLEAR ALL PARAM        { Clears all PARAM values }
46>CLEAR ASSIGN in-file { Clears assignment for in-file only }
47>CLEAR PARAM DEBUG      { Clears the DEBUG switch }
```

DEFINEs

A DEFINE is a named set of attribute/value pairs. DEFINEs are similar to ASSIGN messages, but more versatile. DEFINEs involve too many commands to be described completely here.

Topics:

- DEFINE and ASSIGN
- Controlling the Propagation of DEFINEs
- DEFINE Names
- DEFINE Attributes
- ADD DEFINE Command

For more information on DEFINEs:

Topics	Sources
Full documentation	*TACL Reference Manual*
Programmatic use	*Guardian Programmer's Guide*
Further information	*Guardian User's Guide*

DEFINE and ASSIGN

Both DEFINEs and ASSIGN messages allow you to specify information about a file before you start the process that uses that file.

Table 13-1 Differences Between DEFINE and ASSIGN

	DEFINE	ASSIGN
Processed by	Operating environment	HP COBOL program
Propagated to	Any process that your HP COBOL program creates, unless the new process specifies DEFMODE OFF	Processes that your HP COBOL program creates and to which it explicitly passes ASSIGNs
Accepted for compiler input and output files	Yes	No
Not accepted by	FastSort	SQL/MP or SQL/MX
Available from OSS environment	Yes, but it can be used only to access a Guardian disk file, tape device, or spooler location	No

The statements of a COBOL program refer to files through a COBOL file name. The ASSIGN clause in each file-control entry associates the COBOL file name with a *system-file-name*, the name by which a file is known to the operating environment. In the Guardian environment, the TACL command ASSIGN can override that association and DEFINEs provide a different form of name redirection.

Suppose a program PROG1 includes a file-control entry for an fd-name of MAJORACCT, and associates it with a *system-file-name*:

```
SELECT MAJORACCT
       ASSIGN TO "\AKRON.$SLB.MAJ.ACC"
```

If you execute PROG1, when it opens MAJORACCT, the file it actually opens is \AKRON.$SLB.MAJ.ACC.

If, when you execute the program, you want to redirect the assignment to a different file, you can issue a command interpreter ASSIGN command such as

```
ASSIGN PROG1.MAJORACCT,\NICE.$FRNC.SIGNIF.CUST
```

and then run PROG1. When the COBOL program opens MAJORACCT, it opens the file \NICE.$FRNC.SIGNIF.CUST.

If, however, the program had associated MAJORACCT with a define-name with a file-control entry of the form

```
SELECT MAJORACCT
       ASSIGN TO "=BIGGY"
```

then you would need to have a DEFINE named “=BIGGY” established at execution time; otherwise, any OPEN statement would fail.

Controlling the Propagation of DEFINEs

You control whether or not the Guardian environment allows the creation and processing of DEFINEs. Use the DEFMODE ON/OFF command to enable or disable DEFINE creation and processing. Similarly, when one process starts another process, the creator can specify a DEFMODE setting for the new process; if it does not specify one, the new process inherits the DEFMODE setting of the creator.

DEFINE Names

Every DEFINE has a name. A name you give to a DEFINE must:

- Consist of at least 2 and no more than 24 characters
- Begin with an equals sign (=) followed by a letter
- Continue with any combination of letters, digits, hyphens (-), underscores (_), and carets (^).

HP has reserved the set of DEFINE names beginning with equals sign followed by underscore (_) for future use.

Example 13-6 DEFINE Names

```
=A
=The_chosen_file
=Long--but-not-too-long
=The-File-of-The-Week
=X_-^-_-_-^-_-_-^
```

Uppercase and lowercase letters are equivalent in DEFINE names.

Wherever a DEFINE name can appear in the text of a COBOL source program, it must appear in quotation marks. Wherever a DEFINE name appears in commands to the command interpreter, such as on the command line that initiates a COBOL compilation, the DEFINE name must appear without quotation marks.

DEFINE Attributes

The CLASS attribute determines which other attributes a DEFINE can have.

Table 13-2 DEFINE Attributes

	Other Attributes of DEFINE	
CLASS Attribute Value	**Required**	**Optional**
CATALOG	SUBVOL	
DEFAULTS	VOLUME	SWAP
MAP (default)	FILE	
SEARCH	SUBVOL *n*	
SORT		BLOCK CPU CPUS MODE NOTCPUS PRI PROGRAM SCRATCH SEGMENT SUBSORTS SWAP
SPOOL	LOC	BATCHID BATCHNAME COPIES FORM HOLDAFTER MAXPRINTLINES MAXPRINTPAGES OWNER PAGESIZE REPORT SELPRI
SUBSORT	SCRATCH	BLOCK CPU PRI PROGRAM SEGMENT SWAP
TAPE	VOLUME*	BLOCKLEN DENSITY, DEVICE EBCDIC, EXPIRATION FILEID, FILESECT, FILESEQ GEN LABELS OWNER MOUNTMSG RECFORM, RECLEN REELS, RETENTION SYSTEM TAPEMODE USE VERSION

* Required only if you specify the optional USE attribute with IN.

To create a DEFINE, you must:

1. Set the value of its CLASS attribute (use the SET DEFINE CLASS command in your command interpreter)
2. Give values to its other attributes

For details, see the *TACL Reference Manual*.

Usage Considerations:

- TAPE DEFINE Provides the Only Way to Use Labeled Tapes

  For details, see Chapter 26: Executing and Debugging HP COBOL Programs (page 835), and the *Guardian User's Guide*.

- You Can Use SPOOL DEFINES instead of COBOLSPOOLOPEN
- Special DEFAULTS DEFINE: =_DEFAULTS DEFINE

  Processes can use the =_DEFAULTS DEFINE without referring to it explicitly. The file system uses this DEFINE whenever it needs the values stored in the DEFINE's attributes. You cannot delete the =_DEFAULTS DEFINE or change its name.

ADD DEFINE Command

Your COBOL program can, in the ASSIGN clause of a file-control entry, associate a DEFINE name with a COBOL file name, just as it would associate a file-system file name with a COBOL file name. When the process compiled from that source attempts to open that file, the file system looks for the stored set of attribute/value pairs and uses them to complete the specification of the file before making it available to the process. If it cannot make a file available (based on what it finds, or fails to find), the open operation fails.

To create and name a DEFINE, you use the ADD DEFINE command. Suppose that your current CLASS attribute has the value MAP. If your COBOL program includes this file-control entry:

```
SELECT MARKET-ANALYSIS
       ASSIGN TO "=MARKT"
       ACCESS MODE IS DYNAMIC
       RECORD KEY IS PRODUCT-CODE
       FILE STATUS IS M-STAT.
```

Before executing your program, you must issue a command of this form:

```
>ADD DEFINE =MARKT, FILE \MAG.$EAST.APRIL.SALES
```

to associate the file SALES in subvolume APRIL on volume $EAST on system \MAG with the name =MARKT.

Initial State

Every program in a run unit has an initial state in which a process places it at certain times while the run unit executes.

When a program is in its initial state:

- Each data item in the Linkage Section has the value specified by the calling program.
- Each Working-Storage or Extended-Storage data item that is described with a VALUE clause (except level-88 condition-names) has the specified value.
- The value of each data item in the Working-Storage Section or the Extended-Storage Section that is not described with a VALUE clause depends on whether the NOBLANK directive was active during compilation. If not, each data item has a space character (octal 40) in each character position or (for a COMPUTATIONAL, COMPUTATIONAL-3, or COMPUTATIONAL-5 data item) a value consisting of a series of space characters interpreted as a binary number. If the NOBLANK directive was active during compilation, the value of each data item is unpredictable.
- The open mode of each internal file connector is either Closed or Locked. The other dynamic attributes of the file connector (such as the file position indicator setting) and the contents of the record area for the file associated with the file connector are undefined.

- The control mechanisms for all PERFORM statements in the program are set to their initial state (inactive).
- GO TO statements whose destinations can be modified by ALTER statements are restored to the forms specified in the source program.

A program is placed in its initial state at these times:

- The first time the program is executed in a run unit. For a main program, this is at the beginning of each separate execution of the run unit. For a called program, this is the first time it is called by a CALL statement during each execution of the run unit.
- The first time the program is called after a CANCEL statement cancels it.
- For an initial program, each time it is called.

Status of Internal Entities During Program Execution

During the execution of a program, each internal entity retains the status in which it was left by the execution of the last statement that affected its status:

- Each internal data item retains the last value assigned to it. If no new value has been assigned since the last time the program was placed in its initial state, its “last value” is its initial value.
- Each internal file connector retains the open-mode state and other dynamic attributes, such as the file position indicator, set or assigned by execution of the last statement that affects them. The record area for a file associated with an internal file connector retains whatever contents were last assigned to it. If the file has not been opened since the last time the program was placed in its initial state, the file is closed (and perhaps locked), and the other dynamic attributes are undefined. In this case, the record area contents are also undefined.
- Each GO TO statement that was modified by execution of an ALTER statement retains the last destination procedure specified.
- A PERFORM statement is active if it has been executed during the current execution of the program it is in, and the termination conditions for its control mechanism have not yet occurred (see PERFORM (page 399)); otherwise, the statement is inactive, and its control mechanism is in the initial state.

Status of External Entities During Program Execution

The values of external data items and the statuses of external file connectors are not affected by the actions performed for individual programs. When a program is called, placed in its initial state, terminated by its own execution logic, or cancelled, the status of each external entity remains unchanged. Of course, the execution of a statement that makes reference to an external data item or an external file connector changes its status as the topic on that statement in Chapter 10: Procedure Division Verbs (page 289), describes.

Called Program Termination

A called program terminates its own execution either by executing an EXIT PROGRAM statement or by allowing control to pass beyond the end of its Procedure Division. The completion of the program’s execution implicitly completes the execution of all of its statements; therefore, if any PERFORM statement contained in the program is still active, it is rendered inactive by the program’s resetting its control mechanism to the initial state. The final status of any other internal entity depends on whether or not the program has the initial attribute.

When a called program that has the initial attribute terminates its own execution, an implicit CANCEL statement referencing the program is executed just after control passes from the called program. This causes the value of each internal data item to become undefined and performs an implicit close operation for each internal file connector that is not closed (with or without lock). The implicit close operations proceed as if a CLOSE statement without any of the optional phrases were executed for each of the affected files. If the program is called again, it is placed in its initial state.

When a called program that does not have the initial attribute terminates its own execution, each internal entity (except a data item described in the Linkage Section) retains its status:

- Each internal data item (except those described in the Linkage Section) retains its current value.
- Each internal file connector retains all of its current dynamic attributes; therefore, if it is open in some mode, it remains open in that mode and the record area for the file associated with the internal file connector retains its current contents.
- Each GO TO statement modified by execution of an ALTER statement retains the last destination procedure specified.

If the program is called again during this execution of the run unit, its state upon entry is normally unchanged from the state in which its termination logic left it, except that those data items described in the Linkage Section have whatever values are supplied by the calling program; however, if a CANCEL statement designating the program is executed after the program terminates, it causes the value of each internal data item to become undefined and performs an implicit close operation for each internal file connector that is neither closed nor locked. The implicit close operations proceed as if a CLOSE statement without any of the optional phrases were executed for each of the affected files. In this case, the program is placed in its initial state the next time it is called.

14 Libraries and Utility Routines

Utility routines are in dynamic-link libraries (DLLs) named ZCOBDLL and ZCREDLL. (You can also create a user library for an HP COBOL program.) If a program calls a utility routine, the ECOBOL compiler automatically searches the DLLs for the routine, but the routine is not bound into the program's target file.Utility routines

Programs can call dynamic-link libraries (DLLs), which you can build.

NOTE: In this section, references to $SYSTEM apply only to the NonStop system. For the locations of files on the PC, see NonStop COBOL for TNS/E (ETK) (page 983)).

Table 14-1 All TNS/E Utility Routines

Utility Routines	Library
CLU_PROCESS_CREATE_	ZCREDLL
COBOL_ASSIGN_	ZCOBDLL
COBOL_COMPLETION_	ZCOBDLL
COBOL_CONTROL_	ZCOBDLL
COBOL_FILE_INFO_	ZCOBDLL
COBOL_GETENV_	ZCOBDLL
COBOL_PUTENV_	ZCOBDLL
COBOL_RETURN_SORT_ERRORS_	ZCOBDLL
COBOL_REWIND_SEQUENTIAL_	ZCOBDLL
COBOL_SET_SORT_PARAM_TEXT_	ZCOBDLL
COBOL_SET_SORT_PARAM_VALUE_	ZCOBDLL
COBOL_SETMODE_	ZCOBDLL
COBOL_SPECIAL_OPEN_	ZCOBDLL
COBOLFILEINFO (Guardian only)	ZCOBDLL
SMU_Assign_CheckName_	ZCREDLL
SMU_Assign_Delete_	ZCREDLL
SMU_Assign_GetText_	ZCREDLL
SMU_Assign_GetValue_	ZCREDLL
SMU_Assign_PutText_	ZCREDLL
SMU_Assign_PutValue_	ZCREDLL
SMU_Message_CheckNumber_	ZCREDLL
SMU_Param_Delete_	ZCREDLL
SMU_Param_GetText_	ZCREDLL
SMU_Param_PutText_	ZCREDLL
SMU_Startup_Delete_	ZCREDLL
SMU_Startup_GetText_	ZCREDLL
SMU_Startup_PutText_	ZCREDLL

For information about individual routines:

Library	Source
ZCREDLL	*CRE Programmer's Guide*
ZCOBDLL	ZCOBDLL Routines

Memory Areas

The memory a COBOL process uses is divided into distinct areas, each holding code blocks or data blocks for the COBOL run unit or for Guardian environment routines that serve the run unit.

Table 14-2 Memory Area Characteristics

Memory Area	Purpose	Size	Comments
User code space	Holds code blocks produced by the compilation	256 megabytes	Routines from loadfiles execute here.
User data space	Holds data blocks produced by the compilation	1.5 GB	The maximum size of one data item is 128 MB.
System code space	Holds code for the system routines that service the run unit in the user code space		System library routines execute here.
System data space	Holds data for the system routines that service the run unit in the user code space		

System Library

System library content is determined at system generation (SYSGEN). System library routines run in the system code space.

User Library

A user library contains routines that the operating environment links to the loadfile (run unit) at run time. Run-time linking does not include copying the routines into the loadfile. A loadfile can have only one user library associated with it.

A user library is a DLL. See Dynamic-Link Libraries (DLLs).

Dynamic-Link Libraries (DLLs)

A DLL is a library of shared code [PIC] that contains functions or data for other PIC loadfiles. (For more information about PIC and DLLs, see the *DLL Programmer's Guide for TNS/E Systems*.)

You can put a program in a DLL if:

- It is not a main program.
- It does not contain embedded SQL/MP or SQL/MX statements.
- It does not contain external objects (data or files that are to be shared with code that is outside the library).

NOTE: ASSIGN and PARAM commands for files in a DLL are ignored.

If you put an HP COBOL program in a DLL, you must export the program name when you build the DLL; otherwise, other programs cannot call the HP COBOL program. To export the HP COBOL program name, use either the ELD directive (Guardian) or the `-Weld` option (OSS and PC), with one of the linker options in Table 14-3.

Table 14-3 Linker Options That Export Program Names

	Option exports ...	
Environment	All program names in the source file	Only the specified program name
Guardian	`ELD (-export_all)`	`ELD (-export`*`program-name`*`)`
OSS and PC	`-Weld="-export_all"`	`-Weld="-export`*`program-name`*`"`

Topics:

- Building a DLL From a Single Source File
- Building a DLL from Multiple Source Files
- Specifying a DLL

Building a DLL From a Single Source File

The single source file from which you build the DLL can contain one or more source programs.

You can build a DLL from a single source file in either one step or two: either compile the source file and have the compiler call the linker, or compile the source file and call the linker yourself. (For linker instructions, see the *eld Manual*.)

Topics:

- Guardian Environment: One-Step Method
- Guardian Environment: Two-Step Method
- OSS Environment: One-Step Method
- OSS Environment: Two-Step Method
- PC Environment: One-Step Method
- PC Environment: Two-Step Method

Guardian Environment: One-Step Method

Compile the source file with these directives:

- SHARED (page 576)
- RUNNABLE (page 570)
- ELD (page 557) with either of these options:
  - — `-export_all`, which exports all program names in the source file
  - — `-export` *`program-name`*, which exports only the specified program name. You must repeat this option for every program name that you want to export.

The resulting loadfile is a DLL.

The command in Example 14-1 compiles the source file LIB1 and links the resulting object file, creating a DLL named MYDLL.

Example 14-1 Building a DLL From a Single Source File in One Step (Guardian)

```
ECOBOL /IN LIB1/ MYDLL; SHARED; RUNNABLE; &
&ELD(-export_all)
```

Guardian Environment: Two-Step Method

1. Compile the source file with the directive SHARED (page 576).

   NOTE: Do not use the RUNNABLE directive.

   The result is a linkfile.

2. Link the linkfile, using the `eld` utility with these options:
   - Either:
     - — `-export_all`, which exports all program names in the source file
     - — `-export` *`program-name`*, which exports only the specified program name. You must repeat this option for every program name that you want to export.
   - `-optional_lib -l ZCOBDLL -l ZCREDLL`, which links your file with the SRLs named ZCOBDLL and ZCREDLL.

The resulting loadfile is a DLL.

In Example 14-2, the first command compiles the source file LIB1, creating the linkable object file LIB1O. The second command links LIB1O with the DLLs named ZCOBDLL and ZCREDLL, creating a DLL named MYDLL.

Example 14-2 Building a DLL From a Single Source File in Two Steps (Guardian)

```
ECOBOL /IN LIB1/ LIB1O; SHARED
ELD LIB1O -SHARED -EXPORT_ALL -optional_lib -l ZCOBDLL
-l ZCREDLL -o MYDLL
```

OSS Environment: One-Step Method

Compile the source file with these options:

- `-Wshared`

  NOTE: Do not use `-c`.

- `-Weld` with either of these arguments:
  - — `-export_all`, which exports all program names in the source file
  - — `-export` *`program-name`*, which exports only the specified program name. You must repeat this option for every program name that you want to export.

The resulting loadfile is a DLL.

The command in Example 14-3 compiles the source file `lib1.cob` and links the resulting object file, creating a DLL named `mydll`.

Example 14-3 Building a DLL From a Single Source File in One Step (OSS)

```
ecobol lib1.cob -Wshared -Weld="-export_all
```

OSS Environment: Two-Step Method

1. Compile the source file with these options:
   - `-Wshared`
   - `-c`

   The result is a linkfile.
2. Link the linkfile, using the `eld` utility with these options:
   - Either:
     - `-export_all`, which exports all program names in the source file
     - `-export` *`program-name`*, which exports only the specified program name. You must repeat this option for every program name that you want to export.
   - `-optional_lib -l ZCOBDLL -l ZCREDLL`, which links your file with the DLLs named ZCOBDLL and ZCREDLL.

The resulting loadfile is a DLL.

In Example 14-4, the first command compiles the source file `lib1.cob`, creating the linkable object file `lib1.o`. The second command links `lib1.o` with the DLLs named ZCOBDLL and ZCREDLL, creating a DLL named `mydll`.

Example 14-4 Building a DLL From a Single Source File in Two Steps (OSS)

```
ecobol lib1.cob -Wshared -c

eld lib1.o -shared -export_all -optional_lib -l ZCOBDLL
-l ZCREDLL -o mydll
```

PC Environment: One-Step Method

Compile the source file with these options:

- `-Wshared`

NOTE: Do not use `-c`.

- `-Weld` with either of these arguments:
  - `-export_all`, which exports all program names in the source file
  - `-export` *`program-name`*, which exports only the specified program name. You must repeat this option for every program name that you want to export.

The resulting loadfile is a DLL.

The command in Example 14-5 compiles the source file `lib1.cob` and links the resulting object file, creating a DLL named `mydll`.

Example 14-5 Building a DLL From a Single Source File in One Step (PC)

```
ecobol lib1.cob -Wshared -Weld="-export_all" -o mydll
```

PC Environment: Two-Step Method

1. Compile the source file with these options:
   - `-Wshared`
   - `-c`

   The result is a linkfile.
2. Link the linkfile, using the `eld` utility with these options:
   - Either:
     - `-export_all`, which exports all program names in the source file
     - `-export` *`program-name`*, which exports only the specified program name. You must repeat this option for every program name that you want to export.
   - `-optional_lib` followed by:
     - `-L` *`lib-directory`* `-lcob`, which links your file with the DLL named ZCOBDLL
     - `-obey` *`lib-directory`*`\libc.obey`, which links your file with the DLL named ZCREDLL

     `lib-directory` is the name of the directory of libraries.

The resulting loadfile is a DLL.

In Example 14-6, the first command compiles the source file `lib1.cob`, creating the linkable object file `lib1.o`. The second command links `lib1.o` with the DLLs named ZCOBDLL and ZCREDLL, creating a DLL named `mydll`.

Example 14-6 Building a DLL From a Single Source File in Two Steps (PC)

```
ecobol lib1.cob -Wshared -c
eld lib1.o -shared -export_all -optional_lib -L lib-directory
-lcob -obey lib-directory\libc.obey -o mydll
```

Building a DLL from Multiple Source Files

Each source file from which you build the DLL can contain one or more source programs.

Building a DLL from multiple source files takes two steps:

1. Compile the source files.
2. Link the resulting linkfiles into a single loadfile.

Topics:

- Guardian Environment
- OSS Environment
- PC Environment

Guardian Environment

1. Compile each source file with the directive SHARED (page 576).

   NOTE: Do not use the RUNNABLE directive.

   The result of each compilation is a linkfile.

2. Link the linkfiles into a single loadfile, using the `eld` utility with these options:
   - Either:
     — `-export_all`, which exports all program names in the source file
     — `-export` *`program-name`*, which exports only the specified program name. You must repeat this option for every program name that you want to export.
   - `-shared`, which links the linkfiles into a single loadfile that is a DLL
   - `-optional_lib -l ZCOBDLL -l ZCREDLL`, which links your file with the DLLs named ZCOBDLL and ZCREDLL

In Example 14-7, the first four commands compile four source files (LIB1, LIB2, LIB3, and LIB4), creating four linkable object files (LIB1O, LIB2O, LIB3O, and LIB4O). The last command links the four linkable object files with the DLLs named ZCOBDLL and ZCREDLL, creating the DLL named MYDLL.

Example 14-7 Building a DLL From Multiple Source Files (Guardian)

```
ECOBOL /IN LIB1/ LIB1O;SHARED
ECOBOL /IN LIB2/ LIB2O;SHARED
ECOBOL /IN LIB3/ LIB3O;SHARED
ECOBOL /IN LIB4/ LIB4O;SHARED
ELD LIB1O LIB2O LIB3O LIB4O -shared -export_all -optional_lib
-l ZCOBDLL -l ZCREDLL -o MYDLL
```

OSS Environment

1. Compile each source file with these options:
   - `-Wshared`
   - `-c`

   The result of each compilation is a linkfile.
2. Link the linkfiles into a single loadfile, using the `eld` utility with these options:
   - Either:
     — `-export_all`, which exports all program names in the source file
     — `-export` *`program-name`*, which exports only the specified program name. You must repeat this option for every program name that you want to export.
   - `-shared`, which links the linkfiles into a single loadfile that is a DLL
   - `-optional_lib -l ZCOBDLL -l ZCREDLL`, which links your file with the SRLs named ZCOBDLL and ZCREDLL

In Example 14-8, the first four commands compile four source files (`lib1.cob`, `lib2.cob`, `lib3.cob`, and `lib4.cob`), creating four linkable object files (`lib1.o`, `lib2.o`, `lib3.o`, and `lib4.o`). The last command links the four linkable object files with the SRLs named ZCOBDLL and ZCREDLL, creating a DLL named `mydll`.

Example 14-8 Building a DLL From Multiple Source Files (OSS)

```
ecobol lib1.cob -Wshared -c
ecobol lib2.cob -Wshared -c
ecobol lib3.cob -Wshared -c
ecobol lib4.cob -Wshared -c
eld lib1.o lib2.o lib3.o lib4.o -shared -export_all
-optional_lib -l ZCOBDLL -l ZCREDLL -o mydll
```

PC Environment

1. Compile each source file with these options:
   - `-Wshared`
   - `-c`

   The result of each compilation is a linkfile.
2. Link the linkfiles into a single loadfile, using the `eld` utility with these options:
   - Either:
     - `-export_all`, which exports all program names in the source file
     - `-export` *`program-name`*, which exports only the specified program name. You must repeat this option for every program name that you want to export.
   - `-shared`, which links the linkfiles into a single loadfile that is a DLL
   - `-optional_lib` followed by:
     - `-L` *`lib-directory`* `-lcob`, which links your file with the DLL named ZCOBDLL
     - `-obey` *`lib-directory`*`\libc.obey`, which links your file with the DLL named ZCREDLL

   `lib-directory` is the name of the directory of libraries.

In Example 14-9, the first four commands compile four source files (`lib1.cob`, `lib2.cob`, `lib3.cob`, and `lib4.cob`), creating four linkable object files (`lib1.o`, `lib2.o`, `lib3.o`, and `lib4.o`). The last command links the four linkable object files with the DLLs named ZCOBDLL and ZCREDLL, creating a DLL named `mydll`.

Example 14-9 Building a DLL From Multiple Source Files (PC)

```
ecobol lib1.cob -Wshared -c
ecobol lib2.cob -Wshared -c
ecobol lib3.cob -Wshared -c
ecobol lib4.cob -Wshared -c
eld lib1.o lib2.o lib3.o lib4.o -shared -export_all
-optional_lib -L lib-directory -lcob
-obey lib-directory\libc.obey -o mydll
```

Specifying a DLL

To use an existing DLL, you must specify it to both the compiler and the linker. You can do this in either one step or two.

Topics:

- Guardian Environment: Compile and Link in One Step
- Guardian Environment: Compile and Link in Two Steps
- OSS Environment: Compile and Link in One Step
- OSS Environment: Compile and Link in Two Steps
- PC Environment: Compile and Link in One Step
- PC Environment: Compile and Link in Two Steps

Guardian Environment: Compile and Link in One Step

Compile the program that uses the DLL with these directives:

- CALL-SHARED (page 549) (default)
- CONSULT with the name of the DLL as an *object-name* (see CONSULT and NOCONSULT (page 552))
- ELD (page 557) with these options:
  - — -L (which is similar to the ecobol option -L)
  - — -l or -lib (which is similar to the ecobol option -l)

  For more information about the eld options -l or -lib, see the *eld Manual*.

The result is a loadfile.

The command in Example 14-10 compiles the source file MAIN and links the resulting object file with the DLL named MYDLL, creating a loadfile named MAINO.

Example 14-10 Specifying a DLL When the Compiler Calls the Linker (Guardian)

```
ECOBOL /IN MAIN/ MAINO; CALL-SHARED; CONSULT MYDLL; RUNNABLE;&
&ELD(-L $VOL.SUBVOL -l MYDLL)
```

Guardian Environment: Compile and Link in Two Steps

1. Compile the program that uses the DLL with these directives:
   - CALL-SHARED (page 549)
   - CONSULT with the name of the DLL as an *object-name* (see CONSULT and NOCONSULT (page 552))

   The result is a linkfile.
2. Link the linkfile, using the eld utility with these eld options:
   - -L (which is similar to the ecobol option -L)
   - -l or -lib (which is similar to the ecobol option -l)
   - -optional_lib -l ZCOBDLL -l ZCREDLL, which links your file with the DLLs named ZCOBDLL and ZCREDLL

   For more information about the eld options -l or -lib, see the *eld Manual*.

In Example 14-11, the first command compiles the source file MAIN, specifying the DLL named MYDLL and creating the linkable object file MAINO. The second command links MAINO with MYDLL and the DLLs named ZCOBDLL and ZCREDLL, creating a loadfile named MYPROG.

Example 14-11 Specifying a DLL When You Compile and Then Link (Guardian)

```
ECOBOL /IN MAIN/ MAINO; CALL-SHARED; CONSULT MYDLL

ELD MAINO -CALL_SHARED -L $VOL.SUBVOL -l MYDLL -optional_lib
-l ZCOBDLL -l ZCREDLL -O MYPROG
```

OSS Environment: Compile and Link in One Step

Compile the program that uses the DLL with these options:

- -Wcall_shared
- -L
- -l

NOTE: Do not use -c.

The result is a loadfile.

The command in Example 14-12 compiles the source file main.cob and links the resulting object file with the DLL named mydll, creating the loadfile main.exe.

Example 14-12 Specifying a DLL When the Compiler Calls the Linker (OSS)

```
ecobol main.cob -Wcall_shared -Wcobol="CONSULT MYDLL" -L /usr/xxx/yyy
-l mydll -o main.exe
```

OSS Environment: Compile and Link in Two Steps

1. Compile the program that uses the DLL with these options:
   - -Wcall_shared
   - -c

   The result is a linkfile.
2. Link the linkfile, using the eld utility with these eld options:
   - -L (which is similar to the ecobol option -L)
   - -l or -lib (which is similar to the ecobol option -l)
   - -optional_lib -l ZCOBDLL -l ZCREDLL, which links your file with the SRLs named ZCOBSRL and ZCRESRL

   For more information about the eld options -l or -lib, see the *eld Manual*.

In Example 14-13, the first command compiles the source file main.cob, specifying the DLL named mydll and creating the linkable object file main.o. The second command links main.o with mydll and the DLLs named ZCOBDLL and ZCREDLL, creating the loadfile main.exe.

Example 14-13 Specifying a DLL When You Compile and Then Link (OSS)

```
ecobol main.cob -Wcall_shared -Wcobol="CONSULT MYDLL" -c

eld main.o -call_shared -L /usr/xxx/yyy -l mydll -optional_lib
-l ZCOBDLL -l ZCREDLL -o main.exe
```

PC Environment: Compile and Link in One Step

Compile the program that uses the DLL with these ecobol flags:

- -Wcall_shared
- -Wcobol with the argument “CONSULT MYDLL”
- -L, where *directory* is the name of the directory of libraries
- -l, where *library* is the name of the DLL

NOTE: Do not use -c.

The result is a loadfile.

The command in Example 14-14 compiles the source file main.cob and links the resulting object file with the DLLs named ZCOBDLL and ZCREDLL and the DLL named mydll, creating the loadfile main.exe. MYDLL is in *dll-location*. (An example of *dll-location* is C:\xxx\yyy.)

Example 14-14 Specifying a DLL When the Compiler Calls the Linker (PC)

```
ecobol main.cob -Wcall_shared -Wcobol="CONSULT MYDLL" -L dll-location
-l mydll  -optional_lib -L lib-directory  -l ZCOBDLL -l ZCREDLL
-o main.exe
```

PC Environment: Compile and Link in Two Steps

1. Compile the program that uses the DLL with these options:
   - `-Wcall_shared`
   - `-c`
   - `-Wcobol`, with the argument "CONSULT MYDLL"

   The result is a linkfile.
2. Link the linkfile, using the `eld` utility with these `eld` options:
   - `-l` or `-lib` (which is similar to the `ecobol` option `-l`), where *`library`* is the name of the DLL
   - `-optional_lib` followed by:
     - — `-L` *`lib-directory`* `-lcob`, which links your file with the DLL named ZCOBDLL
     - — `-obey` *`lib-directory`*`\libc.obey`, which links your file with the DLL named ZCREDLL

       `lib-directory` is the name of the directory of libraries.

   For more information about the `eld` options `-l` or `-lib`, see the *eld Manual*.

In Example 14-15, the first command compiles the source file `main.cob`, specifying the DLL named MYDLL and creating the linkable object file `main.o`. The second command links `main.o` with `mydll` and the DLLs named ZCOBDLL and ZCREDLL, creating the loadfile `main.exe`. MYDLL is in *`dll-location`*. (An example of *`dll-location`* is `C:\xxx\yyy`.)

Example 14-15 Specifying a DLL When You Compile and Then Link (PC)

```
ecobol main.cob -Wcall_shared -c -Wcobol="CONSULT MYDLL"

ld main.o -call_shared -L dll-location  -l mydll -optional_lib
-L lib-directory -lcob -obey lib-directory\libc.obey -o main.exe
```

Files of Dummy Routines

Files of dummy routines enable the compiler to accept ENTER statements that are to be resolved at program load time rather than by the linker.

Table 14-4 Files of Dummy Routines

File Name	Description of File (as released by HP)
ECOBEX0	Contains one dummy routine for each operating system routine in the EXTDECS0 file. Represents the latest RVU.
ECOBEX1	Contains one dummy routine for each operating system routine in the EXTDECS1 file. Represents the next-to-latest RVU.
ECOBEXT	Contains one dummy routine for each operating system routine in the EXTDECS file. Represents the second-next-to-latest RVU.

When you install the compiler, the files of dummy routines are stored on the subvolume $SYSTEM.SYSTEM. If you move any of them to another subvolume, use the CONSULT directive to tell the compiler where to find the files you moved.

FastSort Interface Overview

The FastSort interface routines (in the ZCOBDLL file) give your HP COBOL program access to the FastSort utility program, which provides more extensive sorting features than HP COBOL has (such as parallel sorting).

Use the FastSort interface routines only with FastSort. If you use them with SORT, the sort process might terminate abnormally.

To use FastSort routines, you must establish a set of parameters and then execute a statement that uses those parameters. You can establish the set of parameters with either FastSort interface routines or the =_SORT_DEFAULTS DEFINE.

Topics:

- Establishing Parameters With FastSort Interface Routines
- Establishing Parameters With =_SORT_DEFAULTS DEFINE

For descriptions of individual FastSort interface routines, see ZCOBDLL Routines.

Establishing Parameters With FastSort Interface Routines

One way to establish FastSort parameters is with the FastSort interface routines.

Table 14-5 FastSort Interface Routines*

Routine	Purpose
COBOL_RETURN_SORT_ERRORS_	Specifies the data item to which FastSort will report error messages
COBOL_SET_SORT_PARAM_TEXT_	Establishes an alphanumeric parameter (which must be a data item described in any section of the Data Division)
COBOL_SET_SORT_PARAM_VALUE_	Establishes a numeric parameter, specifying that it is a literal, an identifier, or an arithmetic expression

* In the ZCOBDLL file.

At run time, HP COBOL routines collect these parameters into a storage area called the option block. When a SORT or MERGE verb is executed, other HP COBOL run-time routines pass these parameters to FastSort.

If you do not establish FastSort parameters with FastSort interface routines (and certain other conditions are true), FastSort uses the attributes of the =_SORT_DEFAULTS DEFINE). For details, see Establishing Parameters With =_SORT_DEFAULTS DEFINE.

For explanations of individual FastSort interface routines, see ZCOBDLL Routines.

Establishing Parameters With =_SORT_DEFAULTS DEFINE

Another way to establish FastSort parameters is with the default SORT DEFINE =_SORT_DEFAULTS. This method is recommended when it would be difficult to modify existing applications.

When an HP COBOL program calls FastSort, FastSort uses the attributes of =_SORT_DEFAULTS if all of these conditions are true:

- No user DEFINEs have been specified
- No FastSort parameters are set
- DEFMODE is on

You can create a =_SORT_DEFAULTS DEFINE by using the ALTER DEFINE statement in your command interpreter. You can use the command

```
22> INFO DEFINE =_SORT_DEFAULTS, DETAIL
```

to determine whether the DEFINE exists and what its current settings are.

You cannot use =_SORT_DEFAULTS to alter the FastSort parameters SCRATCH and MODE, because the compiler sets these itself whenever it executes a COBOL SORT statement. (It sets SCRATCH to the file that the SD entry specifies. It sets MODE to AUTOMATIC.)

If you specify any FastSort options with FastSort interface routines, you can only use =_SORT_DEFAULTS to alter the parameters CPU, NOTCPUS, and SUBSORTS; however, if you specify CPU-MASK, NO-CPU-MASK, or SUBSORT-COUNT with a FastSort interface, routine, it overrides the respective parameter that you specified with =_SORT_DEFAULTS.

For more information on the =_SORT_DEFAULTS DEFINE, see the *FastSort Manual*.

ZCOBDLL Overview

ZCOBDLL, a DLL, is the native HP COBOL run-time library. It contains the HP COBOL run-time routines. ZCOBDLL routines are not bound into a user object file.

The compiler consults ZCOBDLL automatically. ZCOBDLLs resides on the subvolume $SYSTEM.SYS*nn*.

Table 14-6 ZCOBDLL Routines

Utility Routine
COBOL_ASSIGN_
COBOL_COMPLETION_
COBOL_CONTROL_
COBOL_FILE_INFO_
COBOL_GETENV_
COBOL_PUTENV_
COBOL_RETURN_SORT_ERRORS_
COBOL_REWIND_SEQUENTIAL_
COBOL_SETMODE_
COBOL_SET_SORT_PARAM_TEXT_
COBOL_SET_SORT_PARAM_VALUE_
COBOL_SPECIAL_OPEN_[1]
COBOLFILEINFO[2]

[1] Replaces COBOLSPOOLOPEN routine (for level 2 spooling) and Guardian environment routines (for level 3 spooling).

[2] Guardian environment only.

For descriptions of individual ZCOBDLL routines, see Non-SMU Routines.

Saved Message Utility (SMU) Overview

The Saved Message Utility (SMU) consists of several routines that manipulate saved ASSIGN, PARAM, and startup messages. SMU routines can be used only in the Guardian environment. The Open System Services environment does not have ASSIGN, PARAM, or startup messages.

The internal data structures of the saved ASSIGN, PARAM, and startup messages differ from COBOL data structures. SMU routines retrieve, replace, alter, or delete components of saved messages and reformat them either to or from a specific external representation. (For descriptions of the ASSIGN, PARAM, and startup messages, see the *Guardian Programmer's Guide*.)

Figure 14-1 Effect of Saved Message Utility (SMU) Routines

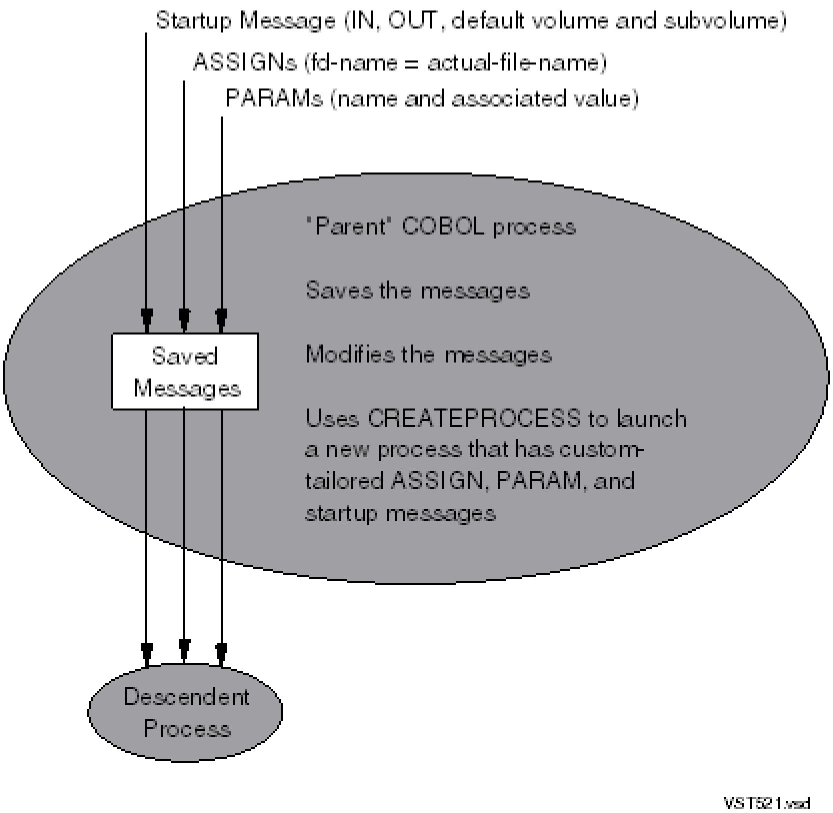

Startup Message (IN, OUT, default volume and subvolume)
ASSIGNs (fd-name = actual-file-name)
PARAMs (name and associated value)
"Parent" COBOL process
Saves the messages
Modifies the messages
Uses CREATEPROCESS to launch a new process that has custom-tailored ASSIGN, PARAM, and startup messages
Saved Messages
Descendent Process
VST521.vsd

Table 14-7 Message Operated Upon by Saved Message Utility (SMU) Routines

	Messages Operated Upon		
Function	**STARTUP**	**ASSIGN**	**PARAM**
CHECK		SMU_Assign_CheckName_ SMU_Message_CheckNumber_	
GET	SMU_Startup_GetText_	SMU_Assign_GetText_ SMU_Assign_GetValue_	SMU_Param_GetText_
PUT	SMU_Startup_PutText_	SMU_Assign_PutText_ SMU_Assign_PutValue_	SMU_Param_PutText_
DELETE	SMU_Startup_Delete_	SMU_Assign_Delete_	SMU_Param_Delete_

The SMU consists of these routines. For complete descriptions of these routines, see the *CRE Programmer’s Guide*.

Routine	Description
SMU_Assign_CheckName_	Checks whether a saved ASSIGN message with a given logical file name exists
SMU_Message_CheckNumber_	Checks whether a specific saved message exists
SMU_Assign_Delete_	Deletes either a portion or all of a saved ASSIGN message
SMU_Param_Delete_	Deletes either a portion or all of the saved PARAM message
SMU_Startup_Delete_	Deletes the entire saved startup message
SMU_Assign_GetText_	Retrieves a portion of a saved ASSIGN message as text and assigns it to a string variable

Routine	Description
SMU_Assign_GetValue_	Retrieves a portion of a saved ASSIGN message as an integer and assigns it to an integer variable
SMU_Param_GetText_	Retrieves a portion of the saved PARAM message as text and assigns it to a string variable
SMU_Startup_GetText_	Retrieves a portion of the saved startup message as text and assigns it to a string variable
SMU_Assign_PutText_	Creates or replaces a portion of a saved ASSIGN message with text from a string variable
SMU_Assign_PutValue_	Creates or replaces a portion of a saved ASSIGN message with a value from an integer variable
SMU_Param_PutText_	Creates or replaces a portion of the saved PARAM message with text from a string variable
SMU_Startup_PutText_	Creates or replaces a portion of the saved startup message with text from a string variable

These routines operate upon copies of the process-creation messages that establish the execution environment of the program. Copies of these messages are not saved automatically and must be requested by a SAVE directive in the main program. For information on the SAVE directive, see SAVE (page 572).

The SMU routines operate on the ASSIGN, PARAM, and startup messages, which have standard sets of defined portions.

Table 14-8 Portions of the ASSIGN Message

Portion	What It Identifies	Portion Type
LOGICALNAME	Logical unit name	Text message portion
TANDEMNAME	File-system file name	Text message portion
PRIEXT	Primary extent size	Integer message portion
SECEXT	Secondary extent size	Integer message portion
FILECODE	File code	Integer message portion
EXCLUSION	Exclusion mode	Integer message portion
ACCESS	Access mode	Integer message portion
RECSIZE	Record size	Integer message portion
BLKSIZE	Block size	Integer message portion

Table 14-9 Portions of the Startup Message

Portion	What It Identifies	Portion Type
VOLUME	Default volume and subvolume names	Text message portion
IN	Input file name	Text message portion
OUT	Output file name	Text message portion
STRING	Message’s parameter string	Text message portion

The PARAM message portions consist of the set of parameter names and their values. A legal parameter name identifier is any combination of letters, digits, hyphens, and circumflexes. It can contain a maximum of 30 characters.

Table 14-10 Portions of the PARAM Message

Portion	What It Identifies	Portion Type
Parameter name	Associated parameter value	Text message portion

ZCREDLL Overview

ZCREDLL is the native CRE run-time library. It contains the SMU routines. ZCREDLL routines are not bound into a user object file. The compiler consults ZCREDLL automatically.

ZCREDLL resides on the subvolume $SYSTEM.SYS*nn*.

The ZCREDLL routines are:

- CLU_PROCESS_CREATE_
- SMU_Assign_CheckName_
- SMU_Assign_Delete_
- SMU_Assign_GetText_
- SMU_Assign_GetValue_
- SMU_Assign_PutText_
- SMU_Assign_PutValue_
- SMU_Message_CheckNumber_
- SMU_Param_Delete_
- SMU_Param_GetText_
- SMU_Param_PutText_
- SMU_Startup_Delete_
- SMU_Startup_GetText_
- SMU_Startup_PutText_

CLU_PROCESS_CREATE_ is available in the Guardian and OSS environments, but the SMU routines are available only in the Guardian environment.

For information about individual ZCREDLL routines, see the *CRE Programmer's Guide*.

ZCOBDLL Routines

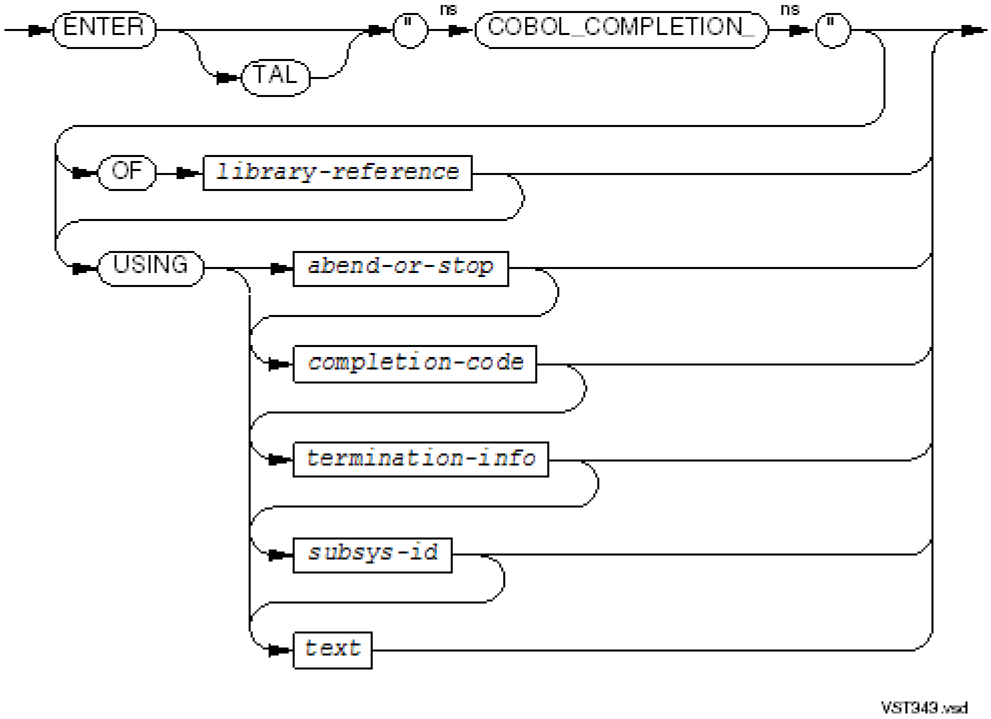
If you omit an optional parameter when you call a ZCOBDLL routine, you must put the keyword OMITTED in its position if you specify subsequent parameters. Trailing OMITTEDs are not required. (The syntax diagrams do not reflect this, because it would make them much harder to read.)

Topics:

- COBOL_COMPLETION_
- COBOL_CONTROL_
- COBOL_GETENV_
- COBOL_PUTENV_
- COBOL_RETURN_SORT_ERRORS_
- COBOL_REWIND_SEQUENTIAL_
- COBOL_SET_SORT_PARAM_TEXT_
- COBOL_SET_SORT_PARAM_VALUE_
- COBOL_SET_MAX_RECORD_
- COBOL_SETMODE_
- COBOL_SPECIAL_OPEN_

COBOL_COMPLETION_

The COBOL_COMPLETION_ routine passes completion parameters to either the STOP or ABEND routine in the operating environment (depending on the *abend-or-stop* parameter).



library-reference

is a mnemonic-name in the SPECIAL-NAMES paragraph. This mnemonic-name is associated with the DLL containing an object copy of COBOL_COMPLETION_.

abend-or-stop

is a numeric operand that specifies either the STOP or ABEND routine. *abend-or-stop* is evaluated and truncated to produce an integer result with a value in the range of -32,768 through 32,767. The value 0 specifies the STOP routine; all other values specify the ABEND routine. The default is 0.

completion-code

is a numeric operand, the evaluation of which includes truncation as necessary to an integer result with a value in the range of -32,768 through 32,767. If this parameter is present, the value is passed to STOP or ABEND as the *completion-code* parameter for that routine.

termination-info

is a numeric operand, the evaluation of which includes truncation as necessary to an integer result with a value in the range of -32,768 through 32,767. If this parameter is present, the value is passed to STOP or ABEND as the *termination-info* parameter for that routine.

subsys-id

is a data item occupying at least twelve character positions. If this parameter is present, a reference to the data item is passed to STOP or ABEND as the *subsys-id* parameter for that routine.

text

is an alphanumeric data item containing at least as many characters as specified by the value of *text-length* (alphabetic and alphanumeric-edited data items are also permissible). If this parameter is present, a reference to the data item is passed to STOP or ABEND as the *text* parameter for that routine.

Usage Considerations:

- Parameters Are Not Validated

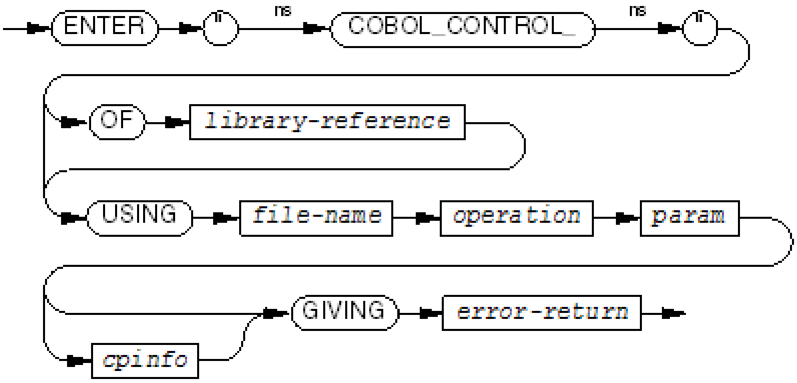
  When the run unit calls COBOL_COMPLETION_, the HP COBOL run-time routines terminate execution of the run unit in an orderly fashion. The final step in the termination logic is to call either the STOP or the ABEND routine of the operating environment (as specified by the implicit or explicit abend-or-stop parameter value). All other parameters are made available to the STOP or ABEND routine as described previously. It is the application's responsibility to verify that the combination of supplied parameters and their values meet the expectations of the STOP or ABEND routine. The COBOL_COMPLETION_ routine does not validate the parameters.

COBOL_CONTROL_

The COBOL_CONTROL_ routine controls device-dependent I-O operations through a call to the Guardian environment routine CONTROL.

If a no-waited, input-output request is active when the COBOL_CONTROL_ routine is called, the request is completed before the CONTROL routine is called. If the file is closed when the call is made, the request is queued until the next OPEN request for the file is issued.

CAUTION: The HP COBOL run-time library does not attempt to validate calls to CONTROL. To avoid interfering with the operation of the HP COBOL run-time library, use COBOL_CONTROL_ only if absolutely necessary.



VST610.vsd

`library-reference`

is a mnemonic-name in the SPECIAL-NAMES paragraph. This mnemonic-name is associated with the DLL containing an object copy of COBOL_CONTROL_.

`file-name`

is a COBOL file name associated with a file that is not $RECEIVE and that is not open for HP COBOL Fast I-O. If the file is not open, the call to the Guardian environment routine CONTROL is made during the next successful open request for the file.

`operation`

is a numeric data item or arithmetic expression that is the `operation` parameter of the Guardian environment routine CONTROL. For more information, see the *Guardian Procedure Calls Reference Manual*.

`param`

is a numeric data item or arithmetic expression that is the `param` parameter of the Guardian environment routine CONTROL. For more information, see the *Guardian Procedure Calls Reference Manual*.

cpinfo

is a checkpoint list in which the routine records the changes to the message storage data space, for example:

```
01   CP-LIST-1.
     05   MAX-COUNT          PIC 9999   COMP   VALUE IS 100.
     05   CURRENT-COUNT      PIC 9999   COMP   VALUE IS 0.
     05   ELEMENT            PIC 9(9)   COMP   OCCURS 100 TIMES.
```

The initial value of MAX-COUNT must be the same as the number of occurrences of ELEMENT.

The maximum number of elements that *cpinfo* can contain is the initial value of MAX-COUNT divided by 2.5. The *cpinfo* in the preceding example can contain 40 elements.

The required number of table elements depends on the number of operations the list must record. The worst-case situation uses six elements.

A complete checkpoint list is required only when a program has a backup that must be kept current. When a record of changes is not required, a null checkpoint list can be used. This is an example description:

```
03 CP-LIST-1   PIC 9(9)   COMP   VALUE IS 0.
```

error-return

is a numeric data item that has one of these values upon completion:

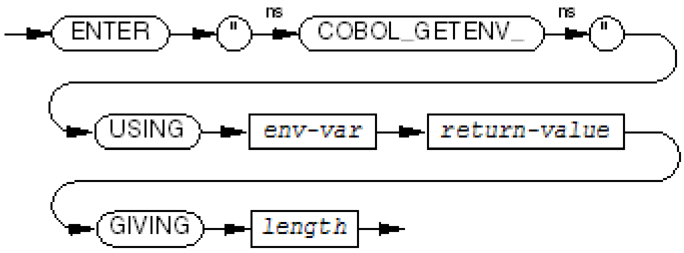
Value	Meaning
n	The call to CONTROL caused Guardian file error *n*.
0*	No error occurred.
1	A required parameter is missing.
2	*file-name* is not a COBOL file name.
5	One of: • *file-name* specifies an open file that is either $RECEIVE or is open for Fast I-O. • *file-name* specifies a closed file and there is not enough control space to allocate the queued information for the next open request.

**error-return* is 0 for a deferred call (that is, one that is held until the file is open). If an error occurs at that time, a run-time diagnostic is issued.

Usage Consideration: In a fault-tolerant program, put the CHECKPOINT statement that specifies *cpinfo* immediately before the call to COBOL_CONTROL_; otherwise, the queued information can be lost if a backup process takes over.

COBOL_GETENV_

The COBOL_GETENV_ routine returns the value of a specified environment variable.



VST722.vsd

env-var

is the name of the environment variable whose value is to be returned. It cannot have trailing spaces or a zero-byte terminator.

return-value

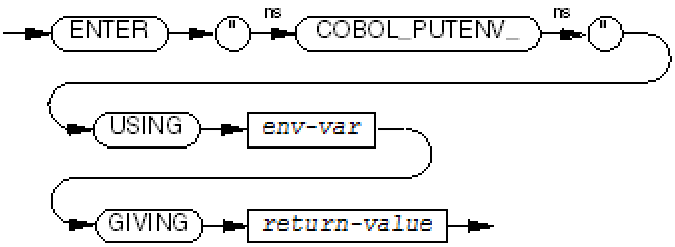
is the name of the variable that will hold the returned value of *env-var*. It must be large enough to hold the maximum possible value of *env-var*.

length

is the number of characters that *return-value* contains. If *length* is zero, *env-var* was not found.

COBOL_PUTENV_

The COBOL_PUTENV_ routine sets a specified environment variable to a specified value.



VST723.vsd

env-var

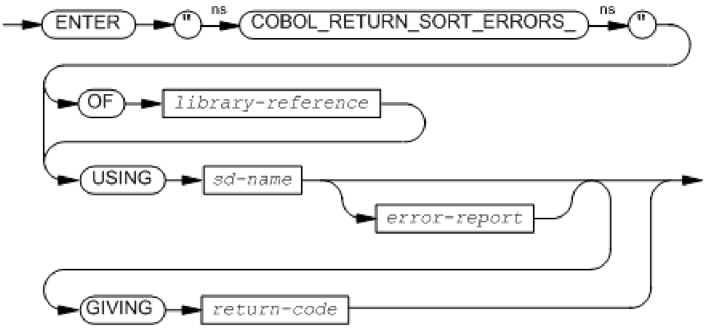
is the name of the environment variable whose value is to be set. It cannot have trailing spaces or a zero-byte terminator.

return-value

is zero if the value of *env-var* was successfully set, nonzero if the value of *env-var* was not set because of insufficient heap space.

COBOL_RETURN_SORT_ERRORS_

The COBOL_RETURN_SORT_ERRORS_ routine returns any error message from the FastSort utility program.



VST809.vsd

library-reference

is a mnemonic-name in the SPECIAL-NAMES paragraph. This mnemonic-name is associated with the DLL containing an object copy of COBOL_RETURN_SORT_ERRORS_.

sd-name

is the COBOL name of a file defined in an SD entry. The specification of the remaining parameter is applied to any subsequent SORT or MERGE statements that refer to this file name. If a subsequent call to the same routine refers to the same *sd-name*, that call resets the specifications for any subsequent SORT or MERGE statements that refer to the same file name.

error-report

is an 01 level item with this structure:

```
01  error-info
   02  error-message-text   PIC X(64).
   02  error-message-length NATIVE-2.
   02  error-code.
       03  file-system-code NATIVE-2.
       03  input-file-index NATIVE-2.
       03  sort-code        NATIVE-2.
   02  error-source         NATIVE-2.
   02  subsort-index        NATIVE-2.
   02  subsort-id           NATIVE-2
```

For the meaning of these fields, see the *FastSort Manual* discussion of SORTERRORSUM. The *input-file-index* and *sort-code* are partial word fields in that section, but are full word fields in the above record. If this parameter is omitted, any previously set error reporting is cleared.

return-code

is an identifier of a numeric data item in which an error value will be returned. The expected values and their respective meanings are:

Value	Meaning
0	The call was correct.
1	*file-name* does not name a sort-merge file.
2	*error-report* is not the correct length.
5	Space for the options block cannot be allocated.

Usage Considerations:

- Error Detection and Reporting

  If the process calls COBOL_RETURN_SORT_ERRORS_ before it executes a SORT or MERGE statement, the HP COBOL run-time library returns the appropriate values in *error-info* after execution of a a RELEASE, RETURN, SORT, or MERGE statement.

  If the sort/merge routines do not detect errors during the execution of one of the above statements, *sort-code* is set to 0 and the contents of the other data items is undefined (unpredictable).

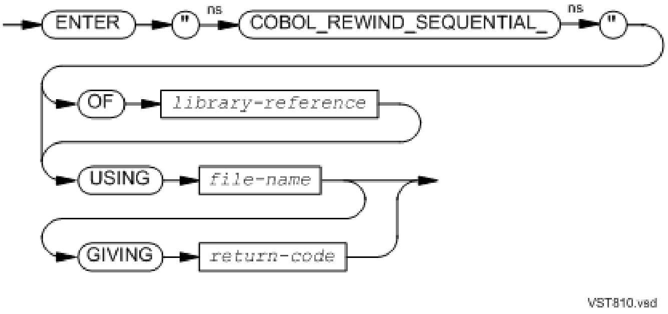
  If the sort/merge routines do detect an error, the HP COBOL run-time routines terminate execution of the sort process and return to the program with the fields of *error-info* set to the values returned by the sort/merge routines. The HP COBOL run-time routines do not produce error messages of their own (they typically go to the home terminal), and no recovery is possible.

  It is your responsibility to determine whether the operation is successful, and to determine how to proceed. If the error happened during execution of a RELEASE or RETURN statement, execution should proceed to the end of the input or output procedure, at which point it continues after the SORT or MERGE statement.

  If an error occurs without the process ever having made a call to COBOL_RETURN_SORT_ERRORS_ (or if the error-report parameter was omitted, cancelling further error reporting), the HP COBOL run-time library processes sort errors, prints a message on the error file (normally the home terminal) that has all of the above information, and abnormally terminates the run unit.

COBOL_REWIND_SEQUENTIAL_

The COBOL_REWIND_SEQUENTIAL_ routine enables you to reposition a sequential disk or single-file tape file to its beginning without closing and reopening it.



library-reference

is a mnemonic-name in the SPECIAL-NAMES paragraph. This mnemonic-name is associated with the DLL containing an object copy of COBOL_REWIND_SEQUENTIAL_.

file-name

is the name of a COBOL data file; that is, a file described in a File Description entry. It must be an FD name associated with a file name that specifies an unstructured or entry sequenced file.

return-code

is an identifier of a numeric data item in which an error value will be returned. The expected values and their respective meanings are:

Value	Meaning
0	Successful execution

1	*file-name* is missing or is invalid.
5	An operation error occurred (for example, the file could not be rewound).

Usage Considerations:

- File Criteria

  file-name must be a valid COBOL file name, the file associated with *file-name* must be open at the time of the call and the file-control entry for *file-name* must specify ORGANIZATION SEQUENTIAL.

  If these criteria are not met, *return-code* (if specified) is set to 1, I-O Status for *file-name* is set to “30” (if *file-name* is a COBOL file name), GUARDIAN-ERR is set to 0, and no action is taken on the file (the rewind operation is considered to be unsuccessful, but execution of the run unit continues).

- Successful Rewind Operation

  If the rewind operation is successful, the run-time routine rewinds the file if rewinding or the concept of rewinding applies to the file, I-O Status is set to “00,” and GUARDIAN-ERR is set to 0.

- If Rewind Does Not Apply to the File

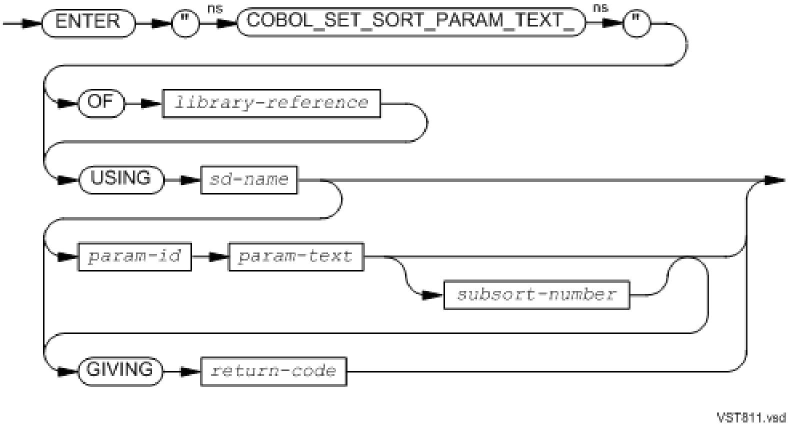
  If rewinding does not apply to the file (a printer for example), I-O Status is set to “07,” GUARDIAN-ERR is set to 0, and the operation is a “no-op.”

- File Position Indicator

  For all files, the file position indicator is set as if the file were newly opened. If alternate keys are specified for the file, the key of reference is set to the prime key.

COBOL_SET_SORT_PARAM_TEXT_

The COBOL_SET_SORT_PARAM_TEXT_ routine establishes alphanumeric parameters in the option block (the local list of parameters that HP COBOL passes to FastSort for the next sort or merge operation).



library-reference

is a mnemonic-name in the SPECIAL-NAMES paragraph. This mnemonic-name is associated with the DLL containing an object copy of COBOL_SET_SORT_PARAM_TEXT_.

sd-name

is the COBOL name of a file defined in an SD entry. The specification of the remaining parameter is applied to any subsequent SORT or MERGE statements that refer to this file name. If a subsequent call to a FastSort routine refers to the same sd-name and to the same parameter, that call resets the specifications for any subsequent SORT or MERGE statements that refer to the same file name. Certain parameters can be reset to their default values, as indicated in the discussion of the parameters below.

`param-id`

is one of:

```
CPU-MASK
SCRATCH-FILE
SWAP-FILE
NO-CPU-MASK
SORT-PROGRAM
```

`param-text`

is the identifier of an alphanumeric data item whose contents is used for the parameter specified by *`param-id`*. If this parameter is omitted, any text previously set for this value of *`param-id`* is reset to its default value.

The value of *`param-text`* depends on the value of *`param-id`* and sometimes on the value of *`subsort-number`* (see Table 14-11).

`subsort-number`

is a numeric operand whose value specifies the sort process to which the parameter pertains. The number of subsorts must have been set previously, and the specified *`subsort-number`* must be in the range 0 to that number.

The value of *`subsort-number`* depends on the value of *`param-id`* (seeTable 14-11).

`return-code`

is an identifier of a numeric data item in which an error value will be returned. The expected values and their respective meanings are:

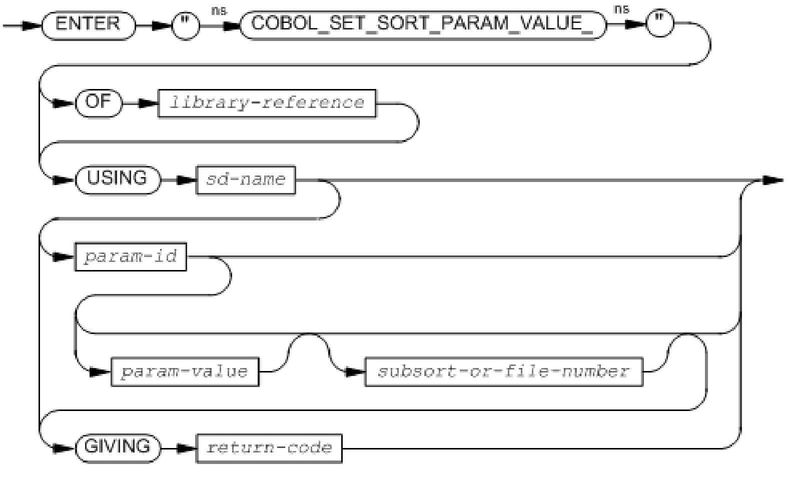
Value	Meaning
0	The call was correct.
1	*`file-name`* does not name a sort-merge file.
2	*`param-id`* is not one of the allowed values.
3	*`param-text`* does not conform to the expected values.
4	*`subsort-number`* was not within the range of the specified number of subsorts.
5	Space for the options block cannot be allocated.

Table 14-11 Relationship Between param-id, subsort-number, and param-text

param-id	subsort-number	param-text
CPU-MASK	Unspecified or 0	Contains a string of zeros and ones, with a maximum of 16 characters. The value of each character indicates whether a processor is to be used for subsort processes for a parallel sort. The leftmost character corresponds to processor 0, the next to 1, and so on, up to processor 15. If there are fewer than 16 characters, the unspecified ones are assumed to be 0. If the value of the data item is "0" (ASCII zero), that processor will not be used. If the value is "1," that processor is a candidate for use. Any other value is an error. Default: All processors on the system are candidates for use.
NO-CPU-MASK	Unspecified or 0	Contains a string of zeros and ones, with a maximum of 16 characters. The value of each character indicates whether a processor is to be used for subsort processes for a parallel sort. The leftmost character corresponds to processor 0, the next to 1, and so on, up to processor 15. If there are fewer than 16 characters, the unspecified ones are assumed to be 0. If the value of the data item is "0" (ASCII zero), that processor will not be used. If the value is "1," that processor is a candidate for use. Any other value is an error. NO-CPU-MASK overrides CPU-MASK.
SCRATCH-FILE		File system name of file to be used by sort as a scratch file. If only a volume name is specified, FastSort assigns a scratch file on that volume; otherwise, a legal file name must be specified. ASSIGN commands that specify that file name are ignored.
	Unspecified or 0	File in the ASSIGN clause for the SD used is used.
	Greater than 0	Scratch file is assigned on $SYSTEM for that subsort.
SORT-PROGRAM	Unspecified or 0	File-system file name of the file that contains the SORTPROG program. If SORTPROG is not specified or is specified with a value of all spaces, FastSort uses $SYSTEM.SYSTEM.SORTPROG.
SWAP-FILE		File-system file name of the file to be used for the extended memory swap file (must be on the local system). If the name is all spaces (the default value) or is not specified, the swap file is created on the same volume as the scratch file. If the scratch file is not local, the swap file is assigned on $SYSTEM. If *param-id* is omitted (*param-text* and *subsort-number* must also be omitted in this case), all values and text previously set for *sd-name* will be reset to their default values. This option releases the option block (about 700 characters long), so you might want to use it if the SD will not be referenced again in this program.
	Unspecified or 0	If *subsort-number* is not specified or is 0, the scratch file for the distributor/collector is implied.If SUBSORT-COUNT is 0, the swap file for the normal sort process in assigned.
	Greater than 0	If SUBSORT-COUNT is greater than 0, the subsort-number indicates the subsort to which the swap file applies.

COBOL_SET_SORT_PARAM_VALUE_

The COBOL_SET_SORT_PARAM_VALUE_ routine establishes numeric parameters in the option block. These parameters affect any subsequent SORT statement execution.



VST811.vsd

`library-reference`

is a mnemonic-name in the SPECIAL-NAMES paragraph. This mnemonic-name is associated with the DLL containing an object copy of COBOL_SET_SORT_PARAM_VALUE_.

`sd-name`

is the COBOL name of a file defined in an SD entry. The specification of the remaining parameter is applied to any subsequent SORT statements that refer to this file name. If a subsequent call to a FastSort routine refers to the same file name and to the same parameter, that call resets the specifications for any subsequent sort operations that refer to the same file name. Certain parameters can be reset to their default values, as indicated in the discussion of the parameters below.

`param-id`

is an alphanumeric data item whose contents identify the parameter whose value is to be set. The values `param-id` might contain are:

BUFFER-SIZE
CPU
CREATE-NEW-OUTPUT
CREATE-NEW-SCRATCH
DATA-SLACK
IN-FILE-COUNT
IN-FILE-EXCL
INDEX-SLACK
MINSPACE
MINTIME
NOPURGE-OUTPUT
NOWAIT-IO
OUT-FILE-EXCL
OUT-FILE-FORMAT
PRIORITY
REMOVE-DUPLICATES
SAVE-SCRATCH
SCRATCH-BLOCK
SCRATCH-CHECK
SECOND-BUFFER
SEGMENT
SUBSORT-COUNT
OUT-FILE-EXCL
SYSTEM

Any letter in the value can be either uppercase or lowercase.

If `param-id`, `param-value`, and `subsort-or-file-number` are all omitted, the run-time routines reset all values and texts previously set for `sd-name` to their default values. This option releases the option block (about 700 characters long), so you may wish to use it if the SD will not be referenced again in this program.

param-value

is a numeric operand whose value gives the value to be used for the parameter specified by *param-id*. If this parameter is omitted, any value previously set for this value of *param-id* will be reset to its default value.

In the case where a value may be zero or nonzero, the nonzero value must be greater than -2,147,483,648 and less than 2,147,483,648. If it is outside this range, an arithmetic overflow will result when the ENTER statement executes.

The value of *param-value* depends on the value of *param-id*. See Usage Considerations.

subsort-or-file-number

is a numeric operand whose value is the sort process to which the parameter pertains or, if the parameter applies to a file, it is a file ordinal.

If a *subsort-number* is being supplied, the number of subsorts must have been set previously (with the SUBSORT-COUNT parameter) and *subsort-number* must be in the range 1 to that number. Subsort-number must be 0 if the parameter applies to the distributor/collector process or if the number of subsorts has been set to zero (a normal sort is implied). In the latter case, the parameter applies to the normal sort.

return-code

is an identifier of a numeric data item in which an error value will be returned. The expected values and their respective meanings are:

Value	Meaning
0	The call was correct.
1	The *file-name* does not name a sort-merge file.
2	The *param-id* does not have one of the allowed values.
3	The *param-value* is outside the allowed range.
4	If *subsort-or-file-number* is used to specify *subsort-number*, one of these is true: • *subsort-number* was not within the range of the specified number of subsorts. • *subsort-number* was required but not specified. • *subsort-number* was specified, was nonzero and a *subsort-number* was not legal for this parameter. If *subsort-or-file-number* is used to specify *file-number*, then *file-number* is not within the range 1 through 32.
5	Space for the options block cannot be allocated.

Usage Considerations:

- Value of *param-value* Depends on Value of *param-id*

  The default for each *param-value* is the same as the default for its *param-id*. For example, if you do not specify BUFFER-SIZE, you get the same result as you do if you specify BUFFER-SIZE with no *param-value* : you get 8192.

  — BUFFER-SIZE

  specifies the size of the buffer used for processing records when an input procedure or an output procedure is specified.

  The value of *param-value* must be greater than 4095 and less than 8193. The default is 8192.

  subsort-or-file-number must not be specified or must be 0.

  — CPU

  specifies a processor in which the sort or subsort is to occur.

Allowable Value for param-value	Meaning
-1	If the *param-value* associated with SUBSORT-COUNT is 0 or has not been specified, the sort process selects a processor.If the *param-value* associated with SUBSORT-COUNT is greater than 1, the selection of a processor depends on subsort-or-file-number: ◦ If *subsort-or-file-number* is 0 or not specified, the processor for the distributor/collector process will be the same as that in which the run unit is executing. ◦ If *subsort-or-file-number* is greater than 0, the sort process will select a processor for the specified subsort depending on CPU-MASK and NO-CPU-MASK.
0 - maximum processor number of the system	Specifies a specific processor. If the *param-value* associated with SUBSORT-COUNT is 0 or not specified the processor selected will be used for the distributor/collector process. If *subsort-or-file-number* is specified, it indicates which subsort will use the specified processor.

  — CREATE-NEW-OUTPUT

  specifies whether the output (GIVING) file is to be created anew or the existing output file is to be used.

Allowable Value for param-value	Meaning
0 (default)	If the output file exists and is large enough for the sort, the data is purged from the file. If the file is too small, the file is purged and a new one is created.
nonzero	The file is purged and a new one is created.

  subsort-or-file-number must not be specified or must be 0.

  — CREATE-NEW-SCRATCH

  specifies whether the scratch file is to be created anew or the existing scratch file is to be used.

Allowable Value for param-value	Meaning
0 (default)	If the scratch file exists and is large enough for the sort, the data is purged from the file. If the file is too small, the file is purged and a new one is created.

Allowable Value for param-value	Meaning
nonzero	The file is purged and a new one is created.

subsort-or-file-number must not be specified or must be 0.

— DATA-SLACK

specifies a percentage of slack space in each data block for an indexed output file. It is applicable only if one GIVING file is specified and that file is an indexed file.

The value of *param-value* is in the range 0 through 99. The default is 0.

subsort-or-file-number must not be specified or must be 0.

— IN-FILE-COUNT

specifies the number of records that will be sorted from each input file or that will be supplied by an input procedure. FastSort uses it to determine the size of the scratch file.

Allowable Value for param-value	Meaning
-1 (default)	FastSort uses a default based on the file assigned. For disk files, FastSort calculates a number based on the file size. For tape files or if an input procedure is used, the default is 50K.
1 - 2,147,483,647	Number of records that will be sorted from each input file or that will be supplied by an input procedure.

Large sorts (greater than 50K records) may terminate abnormally with the scratch file being too small if *record-count* is not used for sorts using tapes or sorts using an input procedure. Small sorts may use more system resources than necessary.

If *subsort-or-file-number* specifies a file number, it must be in the range 1 through 32. The number is the ordinal number of a USING file (from left to right in the USING phrase of the next SORT statement that is executed). If an input procedure is to be used, *subsort-or-file-number* should be omitted or should be one.

— IN-FILE-EXCL

specifies the exclusion mode with which the sort opens a USING file.

Allowable Value for param-value	Meaning
-1	Default (exclusive for disk or tape, shared for any other file type)
0	Shared
1	Exclusive
3	Protected

If *subsort-or-file-number* specifies a file-number, it must be in the range 1 through 32. The number is the ordinal number of a USING file (from left to right in the USING phrase of the next SORT statement that is executed).

If *subsort-or-file-number* is omitted, the first file is assumed.

- — INDEX-SLACK

  specifies a percentage of slack space in each index block for an indexed output file. It is applicable only if one GIVING file is specified and that file is an indexed file.

  The value of `param-value` is in the range 0 through 99. The default is 0.

  `subsort-or-file-number` must not be specified or must be 0.

- — MINSPACE

  specifies whether FastSort is to operate in the MINSPACE mode; that is, whether FastSort should attempt to restrict the use of physical memory at the expense of sort time.

Allowable Value for param-value	Meaning
0 (default)	FastSort uses up to 50% of the available physical memory. This may result in loss of overall system performance if the system is busy.
nonzero	FastSort uses a 64-page extended memory area.

  See the *FastSort Manual* for details.

  `subsort-or-file-number` must not be specified or must be 0.

- — MINTIME

  specifies whether FastSort is to operate in the MINTIME mode; that is, whether FastSort should attempt to reduce execution time at the expense of physical memory.

Allowable Value for param-value	Meaning
0 (default)	Operation is similar to MINSPACE mode.
nonzero	Up to 70% of the available memory will be used.

  See the *FastSort Manual* for details.

  `subsort-or-file-number` must not be specified or must be 0.

- — NOPURGE-OUTPUT

  specifies whether FastSort is to purge an existing output (GIVING) file if it seems too small.

Allowable Value for param-value	Meaning
0 (default)	FastSort purges an existing GIVING file that seems too small before writing the new data to it.
nonzero	FastSort does not purge an existing GIVING file that seems too small unless it has the wrong file type or wrong record length.

  `subsort-or-file-number` must not be specified or must be 0.

- NOWAIT-IO

  specifies whether FastSort performs NOWAIT I-O in passing records to and from the HP COBOL run-time interface (for input and output procedures, for example).

Allowable Value for param-value	**Meaning**
0 (default)	FastSort must use waited I-O.
nonzero	FastSort must use NOWAIT I-O.

  `subsort-or-file-number` must not be specified or must be 0. If *`param-value`* is nonzero, you must also specify SECOND-BUFFER.

  NOWAIT-IO is ignored if the file if being read or written by FastSort. If SECOND-BUFFER is not specified, FastSort error 74 occurs. If SECOND-BUFFER cannot be allocated, run-time error 123 occurs, followed by FastSort error 74.

- OUT-FILE-EXCL

  specifies the exclusion mode with which the sort opens a GIVING file. Allowable values are:

Value	Exclusion Mode Specified
-1	Default (exclusive for disk or tape, shared for any other file type)
0	Shared
1	Exclusive
3	Protected

  `subsort-or-file-number` must not be specified or must be 0.

- OUT-FILE-FORMAT

  specifies the format of the output file.

  Allowable values are 0, 1, 2, and 3. The default value is 0. See the *FastSort Manual* for details.

  This parameter may cause problems if the record area is not large enough or too large. Use it with caution.

  `subsort-or-file-number` must not be specified or must be 0.

- PRIORITY

  specifies the desired execution priority of the sort or subsort process.

Allowable Value for param-value	**Meaning**
-1 (default)	The process is to have the same priority as the calling process.
1 - 199	Specifies the desired priority.

  If the *`param-value`* associated with SUBSORT-COUNT is 0 or not specified, PRIORITY applies to the normal process and *`subsort-or-file-number`* must be omitted or 0.

  If the *`param-value`* associated with SUBSORT-COUNT is greater than 0, *`subsort-or-file-number`* specifies the subsort to which the priority applies; 0 refers to the distributor/collector process.

— REMOVE-DUPLICATES

specifies whether FastSort should remove records with duplicate keys during the sort operation.

Allowable Value for param-value	Meaning
0 (default)	Duplicates are not to be removed.
nonzero	FastSort must remove every record whose keys duplicate those of a previous record.

`subsort-or-file-number` must not be specified or must be 0.

— SAVE-SCRATCH

specifies whether FastSort should save the scratch file after a sort run.

Allowable Value for param-value	Meaning
0 (default)	FastSort must purge the scratch file after the sort run.
nonzero	FastSort must save the scratch file, if it has an explicit name (provided by a call to COBOL_SET_SORT_PARAM_TEXT_ with the SCRATCH-FILE parameter).

`subsort-or-file-number` must not be specified or must be 0.

— SCRATCH-BLOCK

specifies the size of the input and output blocks for the scratch file.

Allowable Value for param-value	Meaning
-1 (default)	FastSort chooses a default block size depending on the type of disk on which the scratch file resides.
Any multiple of 512 in the range 512 through 32768	Size of the input and output blocks for the scratch file

See the *FastSort Manual* for a description of this parameter.

`subsort-or-file-number` must not be specified or must be 0.

— SCRATCH-CHECK

specifies whether FastSort must check the size of each named scratch file to see whether it is large enough.

Allowable Value for param-value	Meaning
0	FastSort must check scratch file size.
nonzero (default)	FastSort must not check scratch file size.

Nonzero is the default for HP COBOL—the default for the FastSort product by itself is 0).

subsort-or-file-number must not be specified or must be 0.

When the *param-value* associated with SUBSORT-COUNT is nonzero, each subsort requires a scratch file. You must first call COBOL_SET_SORT_PARAM_TEXT_ once for each subsort, specifying a *param-id* of SCRATCH-FILE, a *param-text* giving a name for the scratch file (usually just a volume name), and a *subsort-number*.

— SECOND-BUFFER

specifies whether FastSort must use a second buffer.

Allowable Value for param-value	Meaning
0 (default)	FastSort does not use a second buffer.
nonzero	FastSort uses a second buffer, which is allocated in user data space.SECOND-BUFFER is ignored under any of these conditions: ◦ There is not enough space for a second buffer. ◦ NOWAIT-IO is not specified. ◦ The operation is a merge and none of the USING files are magnetic tape files.

subsort-or-file-number must not be specified or must be 0.

— SEGMENT

specifies the size of the extended memory area, in pages.

Allowable Value for param-value	Meaning
-1 (default)	Segment size is controlled by the MINSPACE and MINTIME parameters. The size cannot be more than 90% of the processor's physical memory not locked down by the NonStop operating system.
64 - 8192(16 MB)	Size of the extended memory area, in pages.

If *subsort-or-file-number* is specified, SEGMENT applies to the indicated subsort.

If *subsort-or-file-number* is 0, SEGMENT applies to the distributor/collector process.

If the *param-value* associated with SUBSORT-COUNT is 0 or not specified, SEGMENT applies to the normal sort, and *subsort-or-file-number* must be omitted or 0.

— SUBSORT-COUNT

specifies the number of parallel sorts that FastSort is to use to improve performance.

Allowable Value for param-value	Meaning
0 (default)	FastSort uses normal, non-parallel sort.
2 - 8	FastSort uses parallel sort, and the value of `param-value` is the number of processes to be used. (A parallel sort might be faster, but it uses more system resources. In a busy system, this might cause an overall loss in system throughput.)

subsort-or-file-number must not be specified or must be 0. An error occurs if SUBSORT-COUNT has the value 1.

— SYSTEM

specifies the system on which the sorting is to occur.

A value other than -1 specifies that a particular other system will be used (which may result in very poor performance).

Allowable Value for param-value	Meaning
-1 (default)	Sorting occurs on the current system.
valid system number	Sorting occurs on the system specified by `param-value`.

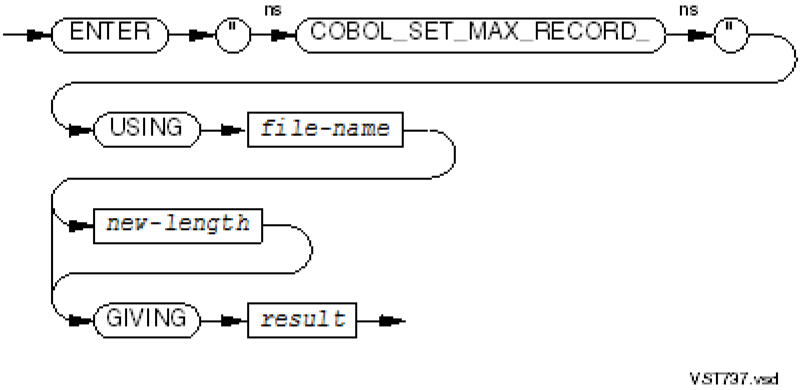
subsort-or-file-number must not be specified or must be 0.

If the *param-value* associated with SUBSORT-COUNT is greater than 1, SYSTEM is ignored (the current system is used).

COBOL_SET_MAX_RECORD_

The COBOL_SET_MAX_RECORD_ routine sets the maximum record size in the COBOL internal file block.

This routine is typically used to allow one file description (FD) to be used to write and read files whose maximum record lengths are less than or equal to that specified in the COBOL program. Without the use of this routine, if the file has a maximum record length that is shorter than that specified in the program, you can open the file in INPUT mode, but not in EXTEND or I-O mode.



file-name

is the COBOL file name of the file to be affected. It must be the name of a file defined by an FD, not a sort-merge file description (SD). At the time the ENTER statement is executed, the file connector referenced by *file-name* must not be open.

new-length

is the new maximum record size. It must be less than 4096 and must not be larger than the length specified in the RECORD clause in the FD; if no RECORD clause is specified, *new-length* must not be larger than the largest record description associated with the FD. The compiler cannot determine if the new size is too large; improper use can cause corruption when reading the file. It is your responsibility to ensure this does not happen.

If you omit *new-length*, the run-time system queries the file that will be assigned when the file is opened and uses the maximum record length from that file.

return

is a variable into which the routine places a value indicating the success of the operation:

Value	Meaning
0	The operation was successful.
1	The first parameter is missing.
2	One of these occurred: • The first parameter is not a *file-name* of an FD. • The second parameter is greater than 4095.
3	The second parameter has been omitted and no file exists from which to obtain the record length.
5	The file is not closed.

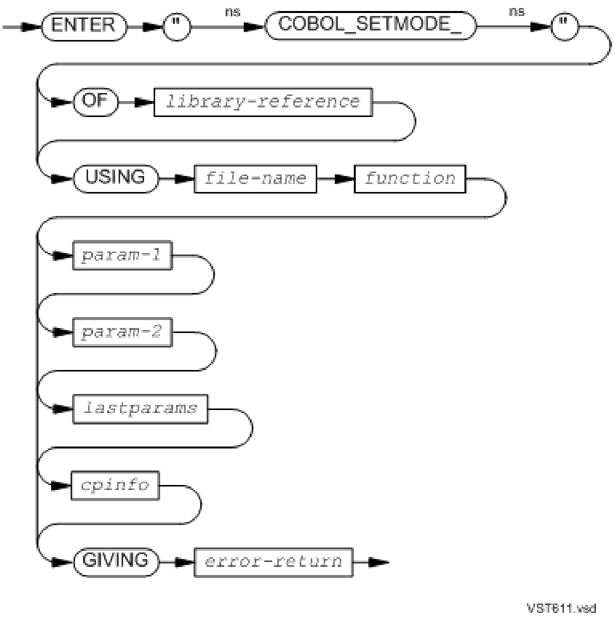
Usage Consideration: If you use COBOL_ASSIGN_ or another method to assign the physical file before the OPEN occurs, that assignment must take place before you call COBOL_SET_MAX_RECORD_.

COBOL_SETMODE_

The COBOL_SETMODE_ routine sets device-dependent functions using a call to the Guardian environment routine SETMODE.

If a NOWAIT input-output request is active when the COBOL_SETMODE_ routine is called, the request is completed before the SETMODE routine is called. If the file is closed when the call is made, the request is queued until the next OPEN request for the file is issued.

CAUTION: The HP COBOL run-time library does not attempt to validate calls to SETMODE. To avoid interfering with the operation of the HP COBOL run-time library, use COBOL_SETMODE_ only if absolutely necessary.



`library-reference`

is a mnemonic-name associated in the SPECIAL-NAMES paragraph with the DLL containing an object copy of COBOL_SETMODE_.

`file-name`

is a COBOL file name associated with a file that is not $RECEIVE and is not open for HP COBOL Fast I-O. If the file is not open, the call to the Guardian environment routine SETMODE is made during the next successful open request for the file.

`function`

is a numeric data item or arithmetic expression that is the `function` parameter of the Guardian environment routine SETMODE. For more information, see the *Guardian Procedure Calls Reference Manual*.

`param-1`, `param-2`

are numeric data items or arithmetic expressions that are the `param1` and `param2` parameters of the Guardian environment routine SETMODE. For more information, see the *Guardian Procedure Calls Reference Manual*.

`lastparams`

is a data item at least four characters long that corresponds to the `last-params` parameter of the Guardian environment routine SETMODE. For best results, define it as a group with two NATIVE-2 data items immediately subordinate to it; for example:

```
01 LASTPARAMS
   02 PARAM1    NATIVE-2
   02 PARAM2    NATIVE-2
```

`cpinfo`

is a checkpoint list in which the routine records the changes to the message storage data space. Example:

```
01  CP-LIST-1.
    05  MAX-COUNT         PIC 9999  COMP  VALUE IS 100.
    05  CURRENT-COUNT     PIC 9999  COMP  VALUE IS 0.
    05  ELEMENT           PIC 9(9)  COMP  OCCURS 100 TIMES.
```

The initial value of MAX-COUNT must be the same as the number of occurrences of ELEMENT.

The maximum number of elements that `cpinfo` can contain is the initial value of MAX-COUNT divided by 2.5. The `cpinfo` in the preceding example can contain 40 elements.

The required number of table elements depends on the number of operations the list must record. The worst-case situation uses six elements.

A complete checkpoint list is required only when a program has a backup that must be kept current. When a record of changes is not required, a null checkpoint list can be used. This is an example description:

```
03 CP-LIST-1  PIC 9(9)  COMP  VALUE IS 0.
```

`error-return`

is a numeric data item that has one of these values upon completion:

Value	Meaning
n	The call to SETMODE caused Guardian file error *n*.
0*	No error occurred.
1	A required parameter is missing.
2	*`file-name`* is not a COBOL file name.
5	One of: • *`file-name`* specifies an open file that is either $RECEIVE or is open for HP COBOL Fast I-O. • *`file-name`* specifies a closed file and there is not enough control space to allocate the queued information for the next open request.

**`error-return`* is 0 for a deferred call (that is, one that is held until the file is open). If an error occurs at that time, a run-time diagnostic is issued.

Usage Consideration: In a fault-tolerant program, put the CHECKPOINT statement that specifies `cpinfo` immediately before the call to COBOL_SETMODE_; otherwise, the queued information can be lost if a backup process takes over.

COBOL_SPECIAL_OPEN_

The COBOL_SPECIAL_OPEN_ routine opens these:

- Printers or Spoolers
- System Log Files
- Partitioned Disk Files
- Tape Files

The COBOL file-control entry, file description entry, and OPEN statement do not provide a way to specify these attributes.

Usage Considerations:

- Acceptable Values for the Parameter open-type

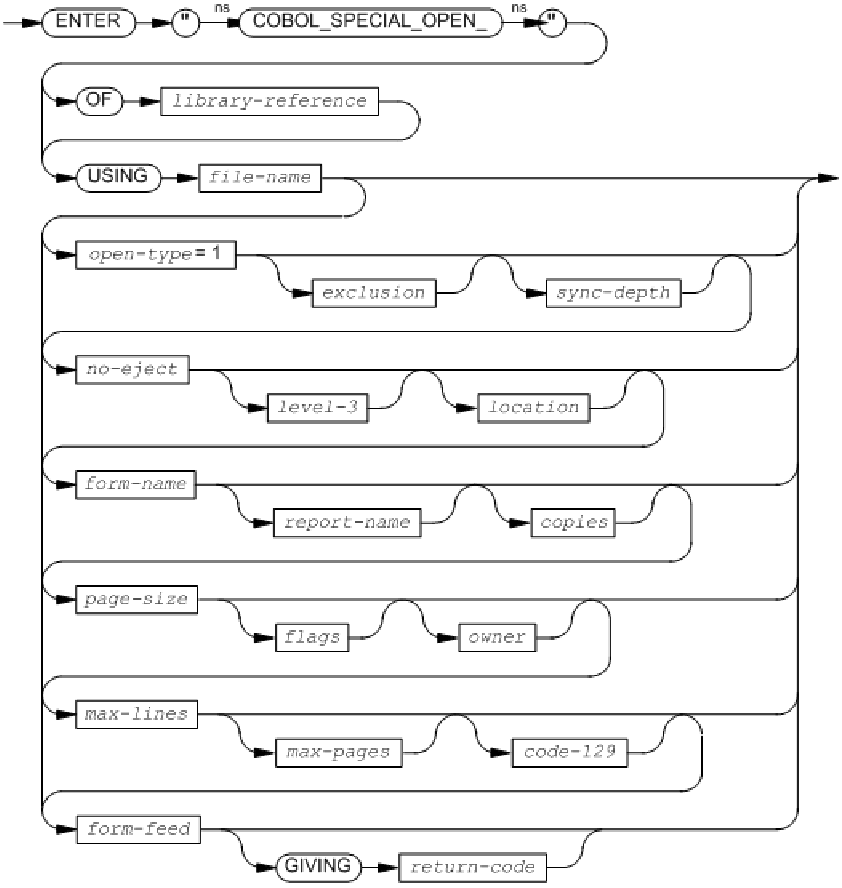
  The value of the `open-type` parameter determines what type of device the COBOL_SPECIAL_OPEN_ routine opens. Acceptable values for `open-type` (after truncation to an integer value, if necessary) and their meanings are:

Value	Meaning
1	Open a printer or spooler.
2	Open a system log file.
3	Open a partitioned disk file.
4	Open a tape file in the CRE.

  Any other value for `open-type` causes an error.

Printers or Spoolers

When the parameter `open-type` has the value 1, the COBOL_SPECIAL_OPEN_ routine opens a line printer or a print spooler collector. Unlike the COBOLSPOOLOPEN routine, COBOL_SPECIAL_OPEN_ allows parameters to be defined in the Extended-Storage Section. Furthermore, COBOL_SPECIAL_OPEN_ provides the only mechanism for suppressing the initial page ejection that COBOL always provides for jobs destined for printing.



VST813.vsd

`library-reference`

is a mnemonic-name in the SPECIAL-NAMES paragraph. This mnemonic-name is associated with the DLL containing an object copy of COBOL_SPECIAL_OPEN_.

`file-name`

is the name of a COBOL data file; that is, a file described in a File Description entry. It must be an FD name associated with a file-system file name that specifies a line printer (device type 5) or a spooler collector (device type 0, subdevice type 31).

`open-type`

is a numeric operand whose value (after truncation to an integer value, if necessary) is 1. The value 1 (the default) signifies the opening of a line printer or a spooler collector.

`exclusion`

is a numeric operand that specifies the exclusion attribute for the open operation. The evaluation of this operand includes truncation to an integer value if necessary. If `exclusion` is present, its value must be 0 (shared), 1 (exclusive), or 3 (protected). If it is omitted, the routine determines the exclusion attribute from a command interpreter ASSIGN command if one applies to the specified FD-name; otherwise, it assumes the value 1 (exclusive).

`sync-depth`

is a numeric operand that specifies the sync-depth attribute for the open operation. The evaluation of this operand includes, if necessary, truncation to an integer value. If this parameter is present, it must be in the range 0 through 255; if it is omitted, the value assumed depends on the nature of the file. For a line printer, the routine uses a value of 1; for a spooler collector, it uses a value of 3.

`no-eject`

is a numeric operand that specifies whether a page ejection must precede the first record delivered to the file. The evaluation of this operand includes, if necessary, truncation to an integer value. A value of 0 requests that a page ejection be done; a nonzero value requests that there be no initial ejection. If no value is specified, the routine uses a value of zero.

`no-eject` does not apply if `file-name` has LINAGE specified.

`level-3`

is a numeric operand that specifies whether level-2 or level-3 spooling is to be used. The evaluation of this operand includes, if necessary, truncation to an integer value. Any nonzero value requests level-3 spooling; the value zero requests level-2 spooling. If no value is specified, the routine uses a value of zero. This parameter applies only to spooler collectors.

`location`

is a 16-character alphanumeric, PIC X(16), that identifies the string passed to the spooler as its `location` parameter. The data item must be at least 16 characters. If longer, only the first 16 characters are passed to the spooler. The spooler expects a two-part location name in the format

```
#gggggggdddddddd
```

where *#ggggggg* is the group name and *dddddddd* is the destination name. The pound sign (#) is required. All remaining characters can be letters, digits, or spaces.

The access mode (see Linkage Section (page 191)) of `location` must be STANDARD.

`form-name`

is an alphanumeric or equivalent data item that specifies the spooler job form name.

`form-name` must not:

- Start with a number or a space character
- Contain any special characters or embedded blanks

Any value provided is adjusted to a length of 16 characters (by extending with spaces or truncating). If no value is specified, the routine uses a value of all spaces. This parameter applies only to spooler collectors.

report-name

is an alphanumeric or equivalent data item that specifies the spooler job report name. Any value provided is adjusted to a length of 16 characters (by extending with spaces or truncating). The first character cannot be a space. No character can be a hyphen (-). If no value is specified, the routine uses the user name associated with the calling process. This parameter applies only to spooler collectors.

copies

is a numeric operand that specifies the number of copies of the job the spooler must print. The evaluation of this operand includes, if necessary, truncation to an integer value. If no value is specified, the routine uses a value of 1. This parameter applies only to spooler collectors.

page-size

is a numeric operand that specifies the number of lines the spooler interface program is to consider as a page. The evaluation of this operand includes, if necessary, truncation to an integer value. If no value is specified, the routine uses the value 60. This parameter applies only to spooler collectors.

flags

is a numeric operand that specifies certain attributes of the job. The evaluation of this operand includes, if necessary, truncation to an integer value. The value is represented as a 16-bit integer:

Bits	Contents
0	Reserved
1	ASCII compression flag (0 = OFF, 1 = ON). Ignored if *code-129* is omitted. See the description of *code-129* for more information.
2-8	Reserved
9	HOLD flag (0 = OFF, 1 = ON)
10	HOLDAFTER flag (0 = OFF, 1 = ON)
11	Specifies whether the output routines of the interface should exit before writing a level-3 buffer to the collector process, so that you can checkpoint (0 = not exit, 1 = exit). The run-time library always sets bit 11 to 0 because you cannot control checkpointing of the level-3 buffer.
12	Purge existing data flag for spooler job files (0 = append new data to existing data, 1 = purge existing data). Ignored if *code-129* is omitted. See the description of *code-129* for more information.
13-15	Job priority

If no value is specified, the routine uses a value of 4 (job priority = 4; all other bits off).

This parameter applies only to spooler collectors.

owner

is an alphanumeric or equivalent data item that specifies a user name; the job is created in the spooler as belonging to that user. Any value provided is adjusted to a length of 16 characters (by extending with spaces or truncating). If no value is specified, the routine uses the user name associated with the calling process. This parameter applies only to spooler collectors.

max-lines

is a numeric operand in the range 0 through 65534 that specifies the maximum number of lines that can be printed. If *max-lines* is 0 or omitted, no maximum number is enforced.

max-pages

is a numeric operand in the range 0 through 65534 that specifies the maximum number of pages that can be printed. If *max-lines* is 0 or omitted, no maximum number is enforced.

code-129

is a numeric operand that specifies whether a code 129 file (spooler job file) is to be created or a regular spooler file is to be created. A nonzero value indicates that the run-time library is to call SPOOLSTART with *file-name* reflecting the name specified in the ASSIGN phrase of the SELECT clause for the file or by an ASSIGN TACL command. See the description of SPOOLSTART in the *Guardian Procedure Calls Reference Manual* for details on spooler job files. The resulting file must be a disk file and must be a code 129 spooler job file if it exists. If the actual file is incorrect, an error is reported and the program terminates abnormally. If the value of *code-129* is 0, it is assumed that the actual file assigned will be a normal spooler process. You can use *flags* to control whether the file is compressed and whether old data in the file is deleted or new data is appended. If *code-129* is specified, these parameters are ignored:

Parameter	**Value Used**
exclusion	Protected
sync-depth	1
level-3	Level 3
location	The file specified in the SELECT clause

form-feed

is a numeric operand that specifies whether a form feed (CONTROL (file, 1, 0)) is to be used to position to the top of a new page rather than using spacing when LINAGE is specified for the file. A value of 0 (the default) indicates that spacing is to be used; a nonzero value indicates that a form feed is to be used. If *form-feed* is nonzero, the sizes of the pages may not conform to that specified in the LINAGE clause for the file. The size depends on the forms control mechanism of the printer.

return-code

is an identifier of a numeric data item in which an error value will be returned. The expected values and their respective meanings are:

Value	Meaning
0	Successful execution
1	*file-name* is missing or is invalid, or an extraneous parameter is present.
2	The value of the *open-type*, *exclusion*, or *sync-depth* parameter is invalid.
3	The value of the *owner* parameter is invalid.
4	The *device-type* of the file specified is not compatible with the value specified for *open-type*.
5	An operation error occurred (for example, the file could not be opened).

In the Guardian environment, all the functions of this form of COBOL_SPECIAL_OPEN_ except those provided by the parameters *no-eject*, *code-129*, and *form-feed* can also be provided by a DEFINE of class SPOOL, but COBOL_SPECIAL_OPEN_ brings the functionality into the program, while the DEFINE solution leaves it out in the run-time environment.

A file assigned to a printer or to a spooler collector process (identified at run time by its subdevice type of 31) is handled as a printer, in that calls to Guardian environment routines CONTROL and SETMODE are used to initialize the file's state, to suppress automatic forms skipping, and to handle ADVANCING clauses, and the writing of a given record is deferred until the next record is ready to be written (or the file is to be closed) to minimize the number of Guardian procedure calls involved in writing.

When a file is assigned to a process that is not a spooler collector, the records are written immediately, with no optimization attempted and no SETMODE or CONTROL procedure involvement.

System Log Files

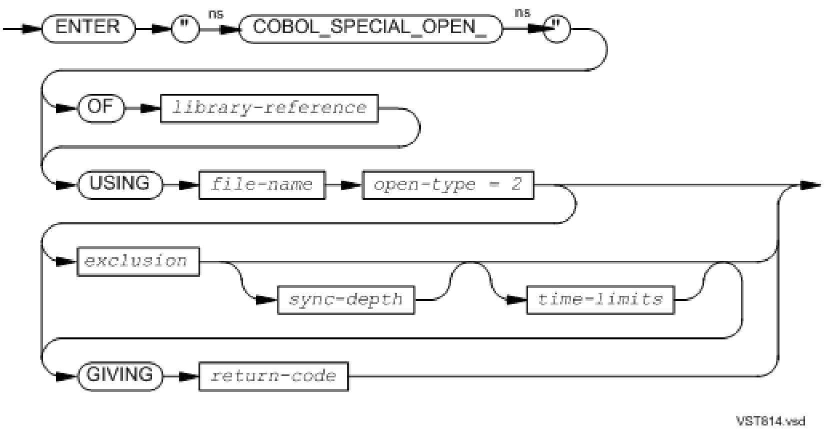
When the parameter *open-type* has the value 2, the COBOL_SPECIAL_OPEN_ routine does one of:

- Opens the EMS device ($0) or an alternate collector so that it can receive EMS events

NOTE: To send an ASCII text message (rather than an EMS event) to a collector, open the collector with an OPEN statement, not with the COBOL_SPECIAL_OPEN_ routine.

- Opens the SPI device ($0.#ZSPI) so that it can receive SPI commands.

NOTE: To send SPI commands to an alternate collector, open the collector with an OPEN statement, not with the COBOL_SPECIAL_OPEN_ routine.



library-reference

is a mnemonic-name in the SPECIAL-NAMES paragraph. This mnemonic-name is associated with the DLL containing an object copy of COBOL_SPECIAL_OPEN_.

file-name

is the name of a COBOL data file; that is, a file described in a File Description entry. It must be an FD name associated with a file-system file name that specifies a system log (device type 1, subdevice type 0 or 1).

open-type

is a numeric operand whose value (after truncation to an integer value, if necessary) is 2. The value 2 signifies the opening of a system log file.

exclusion

is a numeric operand that specifies the exclusion attribute for the open operation. The evaluation of this operand includes, if necessary, truncation to an integer value. If the parameter is present, its value must be 0 (shared), 1 (exclusive), or 3 (protected); if it is omitted, the routine determines the exclusion attribute from a command interpreter ASSIGN command (when one applies to the specified FD-name, or assumes the value 0 (shared) for a system log file otherwise.

sync-depth

is a numeric operand that specifies the sync-depth attribute for the open operation. The evaluation of this operand includes, if necessary, truncation to an integer value. If this parameter is present, it must be in the range 0 through 255; if it is omitted, the value assumed depends on the nature of the file. For a system log file, the routine uses a value of 1.

time-limits

is a numeric operand that specifies the TIME-LIMITS attribute for a system log file only when the subdevice type is other than 0; if the subdevice type is 0, this parameter is ignored. The evaluation of this operand includes, if necessary, truncation to an integer value. If this parameter is present and its value is not zero, LOCKFILE and READ statements that refer to the system log file can include the TIME LIMIT wait-time phrase, indicating the maximum time a process waits for a request to complete. If the parameter is omitted, the routine uses a value of zero.

return-code

is an identifier of a numeric data item in which an error value will be returned. The expected values and their respective meanings are:

Value	Meaning
0	Successful execution
1	*file-name* is missing or is invalid, or an extraneous parameter is present.
2	The value of the *open-type*, *exclusion*, or *sync-depth* parameter is invalid.
4	The *device-type* of the file specified is not compatible with the value specified for *open-type*.
5	An operation error occurred (for example, the file could not be opened).

Usage Considerations:

- Sending a Message to $0

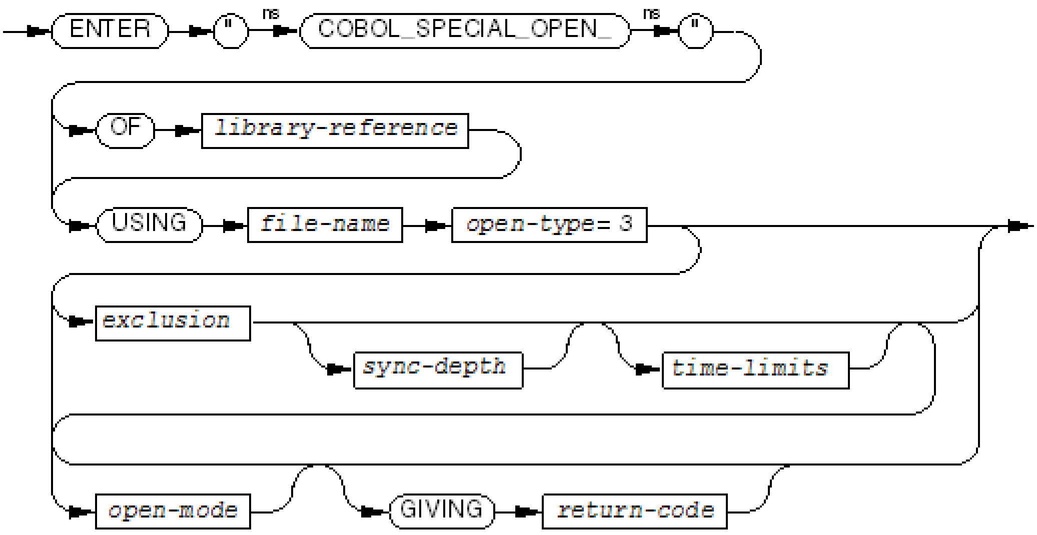
  To send an EMS event or ASCII text message to the EMS Event Management Service (EMS) device ($0) or an alternate collector, use the WRITE statement.

- Sending an SPI command to $0.#ZSPI

  To send an SPI command to the SPI device ($0.#ZSPI), use the READ statement with a PROMPT phrase.

Partitioned Disk Files

When the parameter *open-type* has the value 3, the COBOL_SPECIAL_OPEN_ routine opens a partitioned disk file even though some partitions cannot be accessed. (The OPEN statement requires all partitions to be accessible.)



VST815.vsd

`library-reference`

is a mnemonic-name in the SPECIAL-NAMES paragraph. This mnemonic-name is associated with the DLL containing an object copy of COBOL_SPECIAL_OPEN_.

`file-name`

is the name of a COBOL data file; that is, a file described in a File Description entry. It must be an FD name associated with a file-system file name that specifies a disk file (device type 3).

`open-type`

is a numeric operand whose value (after truncation to an integer value, if necessary) is 3. The value 3 signifies the opening of a partitioned disk file, even though some partitions cannot be accessed.

`exclusion`

is a numeric operand that specifies the exclusion attribute for the open operation. The evaluation of this operand includes, if necessary, truncation to an integer value. If the parameter is present, its value must be 0 (shared), 1 (exclusive), or 3 (protected); if it is omitted, the routine determines the exclusion attribute from a command interpreter ASSIGN command (when one applies to the specified FD-name); otherwise the routine assumes the value 3 (protected) when the open mode is Input or assumes the value 1 (exclusive) for any other open mode.

`sync-depth`

is a numeric operand that specifies the sync-depth attribute for the open operation. The evaluation of this operand includes, if necessary, truncation to an integer value. If this parameter is present, it must be in the range 0 through 255; if it is omitted, the value assumed depends on the nature of the file. For a disk file, the routine uses a value of 1.

`time-limits`

is a numeric operand that specifies the TIME-LIMITS attribute for the open operation. The evaluation of this operand includes, if necessary, truncation to an integer value. The presence of a nonzero value for this parameter is equivalent to the presence of a TIME LIMITS phrase with a nonzero value in an ordinary OPEN statement. If the parameter is omitted, the routine uses a value of zero, which has no effect.

`open-mode`

is a numeric operand that specifies the open mode attribute for the open operation. The evaluation of this operand includes, if necessary, truncation to an integer value. If this parameter is supplied, its value must be 0 (open mode is Input), 1 (I-O), 2 (Output), or 3 (Extend). If the parameter is omitted, the routine uses a value of 0 (Input).

return-code

is an identifier of a numeric data item in which an error value will be returned. The expected values and their respective meanings are:

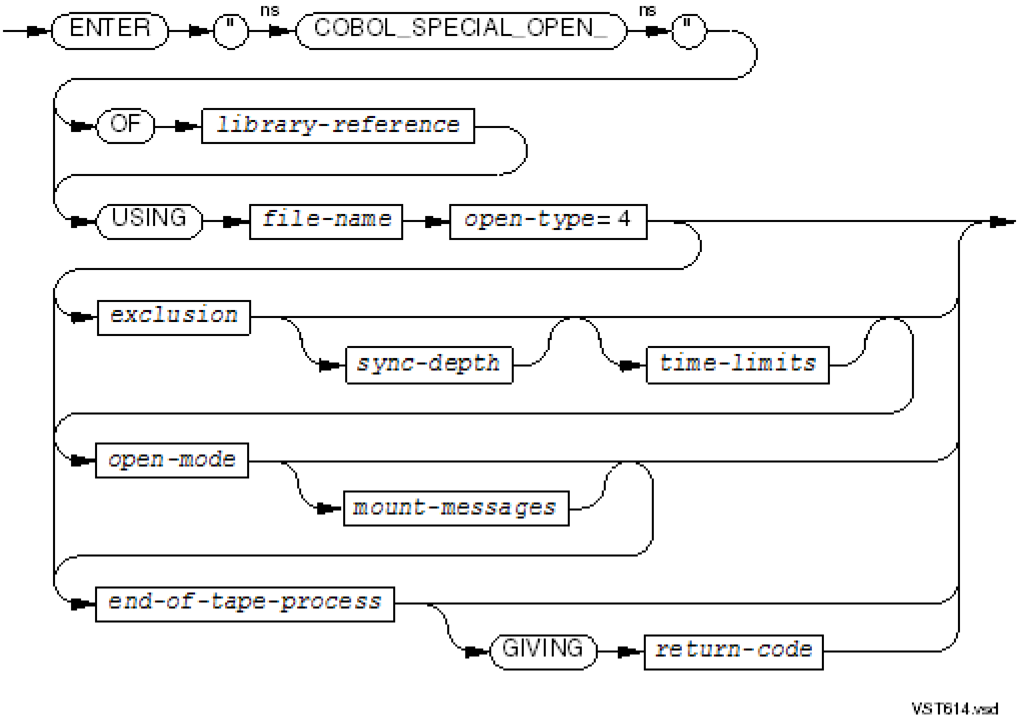
Value	Meaning
0	Successful execution
1	*file-name* is missing or is invalid, or an extraneous parameter is present.
2	The value of the *open-type*, *exclusion*, or *sync-depth* parameter is invalid.
4	The *device-type* of the file specified is not compatible with the value specified for *open-type*.
5	An operation error occurred (for example, the file could not be opened).

When the COBOL85_SPECIAL_OPEN_ routine with an open-type of 3 has performed a successful open operation, the special register GUARDIAN-ERR contains the value 3 (failure to open a partition) if the operating environment reports that file code. In this case, the I-O status code is “00.”

If a subsequent input-output statement attempts to make reference to a logical record which resides in an unavailable partition of the file, the HP COBOL run-time routines assign a value of 72 (attempt to access unmounted partition) to the special register GUARDIAN-ERR, because that is the file code reported by the operating environment. The I-O status code in this case is “30” (COBOL permanent error). This, in turn, causes the run-time routines to abnormally terminate the execution of the run unit, unless an applicable Declarative procedure is specified in the Procedure Division of the program unit that contains the statement whose execution failed.

Tape Files

When the parameter *open-type* has the value 4, the COBOL_SPECIAL_OPEN_ routine opens a tape file.



`library-reference`

is a mnemonic-name associated in the SPECIAL-NAMES paragraph with the DLL containing an object copy of COBOL_SPECIAL_OPEN_.

`file-name`

is the name of a COBOL data file; that is, a file described in a File Description entry. It must be an FD name associated with a file-system file name that specifies a tape file.

`open-type`

is a numeric operand whose value (after truncation to an integer value, if necessary) is 4. The value 4 signifies the opening of a tape file.

`exclusion`

is a numeric operand that specifies the exclusion attribute for the open operation. The evaluation of this operand includes, if necessary, truncation to an integer value. If the parameter is present, its value must be 0 (shared), 1 (exclusive), or 3 (protected). If it is omitted, the routine determines the exclusion attribute from a command interpreter ASSIGN command (when one applies to the specified FD-name); otherwise the routine assumes the value 3 (protected) when the open mode is Input or assumes the value 1 (exclusive) for any other open mode.

`sync-depth`

is a numeric operand that specifies the sync-depth attribute for the open operation. The evaluation of this operand includes, if necessary, truncation to an integer value. If this parameter is present, it must be in the range 0 through 255. If it is omitted, the value assumed depends on the nature of the file.

`time-limits`

is a numeric operand that specifies the TIME-LIMITS attribute for the open operation. The evaluation of this operand includes, if necessary, truncation to an integer value. The presence of a nonzero value for this parameter is equivalent to the presence of a TIME LIMITS phrase with a nonzero value in an ordinary OPEN statement. If the parameter is omitted, the routine uses a value of zero, which has no effect.

`open-mode`

is a numeric operand that specifies the open mode attribute for the open operation. The evaluation of this operand includes, if necessary, truncation to an integer value. If this parameter is supplied, its value must be one of:

Value	Open Mode
0	Input (default)
1	Input-Output
2	Output
3	Extend

`mount-messages`

is the name of a data item that describes the name of the file to be used for tape mount messages. The file name must be left-justified and in Guardian external format with no trailing spaces. Reference modification can give the length of the name. The file must be a process or terminal.

If `mount-messages` is omitted, tape mount messages are written to the execution log file.

end-of-tape-process

is an alphanumeric data item of at least seven characters, whose value specifies the action to be taken if the file operation causes a reel swap.

If the value is "RETURN " (in any combination of uppercase and lowercase letters and with a space after *N*), a read or write operation returns an I-O status code of "97" and a GUARDIAN-ERR of 150 when a reel swap occurs. If the reel swap occurs during a WRITE statement, the record that was just written and several of the records previously written might or might not be on the tape just completed.

If the value is not "RETURN " or *end-of-tape-process* is omitted, a read or write operation does not return anything to the program when a reel swap occurs.

return-code

is an identifier of a numeric data item in which an error value will be returned. The expected values and their respective meanings are:

Value	Meaning
0	Successful execution
1	*file-name* is missing or is invalid, or an extraneous parameter is present.
2	The value of the *open-type*, *exclusion*, *sync-depth*, or *mount-messages* parameter is invalid.
4	The *device-type* of the file specified is not compatible with the value specified for *open-type*.
5	An operation error occurred (for example, the file could not be opened).

Non-SMU Routines

This topic explains the individual routines in the ZCOBDLL file that are not Saved Message Utility (SMU) routines.

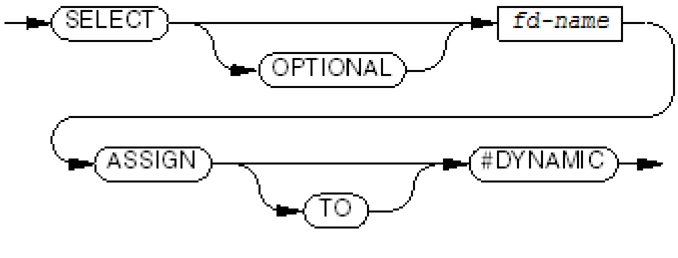
Table 14-12 Non-SMU ZCOBDLL Routines

Routine	Comment
COBOL_SPECIAL_OPEN_ (page 644)	
COBOL_ASSIGN_	For the Guardian environment
COBOLFILEINFO	For the Guardian environment
COBOL_FILE_INFO_	For the OSS environment
CREATEPROCESS	Cannot create a high-PIN process. To create a high-PIN process, use CLU_PROCESS_CREATE

COBOL_ASSIGN_

The COBOL_ASSIGN_ routine enables an HP COBOL program to perform an ASSIGN-like statement while the program is executing. It associates a COBOL file name (*fd-name*) with a file-system file name.

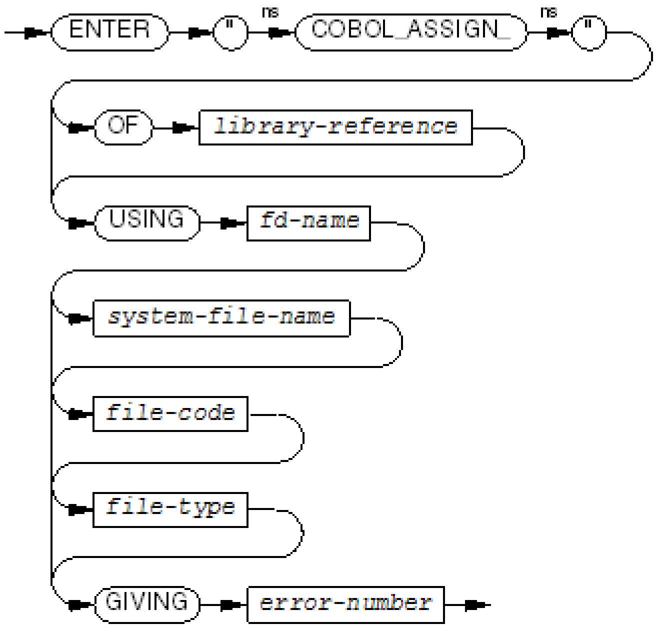
You must declare the COBOL file name as dynamically assignable in a file-control entry in the Input-Output Section. The syntax for this declaration is:



VST355.vsd

This form of the ASSIGN clause specifies only that *fd-name* is dynamically assignable; unlike other forms of the ASSIGN clause (or of the command interpeter ASSIGN command), it does not associate *fd-name* with any file-system file name.

After *fd-name* is declared as dynamically assignable, the HP COBOL program can use ENTER to call COBOL_ASSIGN_. The general form of the ENTER statement to use COBOL_ASSIGN_ is:



VST356.vsd

library-reference

is a mnemonic-name associated, in the SPECIAL-NAMES paragraph, with either COBOLLIB or some other object file containing an object copy of COBOL_ASSIGN_. See Files of Dummy Routines.

fd-name

is the *fd-name* of a file connector. The file specified in the ASSIGN phrase of the SELECT clause that references *fd-name* will be replaced by *system-file-name*. If the program was compiled in the OSS environment, the file will be an OSS file, unless it was defined as a Guardian file by an ASSIGN such as:

```
ASSIGN TO "GUARDIAN #DYNAMIC"
```

system-file-name

is an alphanumeric data item holding the file-system file name of the file to be assigned. *system-file-name* is in external form. It must be left-justified in the data item and any unused portion of the name must contain spaces.

If the program identifies *system-file-name* as an OSS file system name, then the program assumes that *fd-name* is an OSS file; otherwise, the program assumes that *fd-name* is a Guardian file.

If system, volume, or subvolume of a Guardian file is not specified, the respective default value is used ($*volume*.*file* is no longer allowed.)

If *system-file-name* is omitted, COBOL_ASSIGN_ does not change the original file-system file name.

For more information on OSS file system names, see Files in the OSS Environment (page 723).

file-code

can be specified in the Guardian environment only. It is a numeric data item or a numeric literal whose value is a file code acceptable to the file system. For example, file code 101 indicates that the file is an EDIT file and file code 180 indicates that the file is an OSS ASCII text file.

If *file-code* is omitted, COBOL_ASSIGN_ does not change the original file code.

file-type

can be specified in the Guardian environment only. It is a numeric data item whose content, or a numeric literal whose value, is either 0 (for an unstructured file) or 2 (for an entry-sequenced file). You can specify *file-type* only for a file whose organization is sequential. For other file organizations, the organization determines the file type (relative for ORGANIZATION RELATIVE and key sequenced for ORGANIZATION INDEXED).

If *file-type* is omitted, COBOL_ASSIGN_ and does not change the original file type.

error-number

is a numeric data item in which COBOL_ASSIGN_ returns an error code. Possible values for *error-number* and their respective meanings are:

Value	Meaning
0	No error occurred.
1	*system-file-name* has invalid syntax or the name has no length (such as a group containing an OCCURS DEPENDING whose length is 0).
2	*system-file-name* is open and cannot be changed.
4	There is not enough system space to add the new name.
5	*file-type* is not 0 or 2, *file-type* is specified for a file whose organization is not sequential, or *file-type* or *file-code* is specified for an OSS file.

COBOL_ASSIGN_ does not determine whether the assigned file actually exists or is accessible. This checking is done by the OPEN statement. COBOL_ASSIGN_ can be called repeatedly for one *fd-name*.

An attempt to open a file using a dynamically assignable *fd-name* before COBOL_ASSIGN_ has been used on it is an attempt to open $*volume*.#DYNAMIC (where $*volume* is the default volume). This causes a run-time error.

A CLOSE LOCK on a file using a dynamic *fd-name* closes the file but does not lock it. If locking were allowed, it would only prevent the reuse of the *fd-name* for the remainder of the run unit's existence; the file would still be accessible through another dynamic *fd-name*.

A command interpreter ASSIGN command cannot designate an *fd-name* as dynamically assignable—you must do this in the source program. ASSIGN can, however, assign a default value to the previously designated dynamic file name. The effect is the same as having an ENTER COBOL_ASSIGN_ statement as the first statement in the program. See ASSIGN Command (page 590).

A CRE HP COBOL program can call either COBOL_ASSIGN_ (the COBOL-environment routine) or COBOL_ASSIGN_ (the equivalent CRE routine). If it calls COBOL_ASSIGN_, it actually calls COBOL_ASSIGN_ indirectly. There are advantages to calling COBOL_ASSIGN_ directly (see COBOL_ASSIGN_).

Example 14-16 COBOL_ASSIGN_ Routine

```
IDENTIFICATION DIVISION.
    ...
ENVIRONMENT DIVISION.
    ...
SPECIAL-NAMES.
   FILE "$SYSTEM.SYSTEM.COBOLLIB" IS COBOLLIB.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
    SELECT PARTNO   ASSIGN #DYNAMIC.
DATA DIVISION.
    ...
WORKING-STORAGE SECTION.
01  PART-FILE-NAME           PICTURE X(35).
01  ASSIGN-ERROR      COMP   PICTURE 9999.
PROCEDURE DIVISION.
SETUP.
    DISPLAY "ENTER CHOSEN PART FILE NAME"
    ACCEPT PART-FILE-NAME
    ENTER "COBOL_ASSIGN_" OF COBOLLIB USING PARTNO
                                          PART-FILE-NAME
                                  GIVING ASSIGN-ERROR
    IF ASSIGN-ERROR = 0
         OPEN INPUT PARTNO
    ELSE IF ASSIGN-ERROR = 1
         ...
```

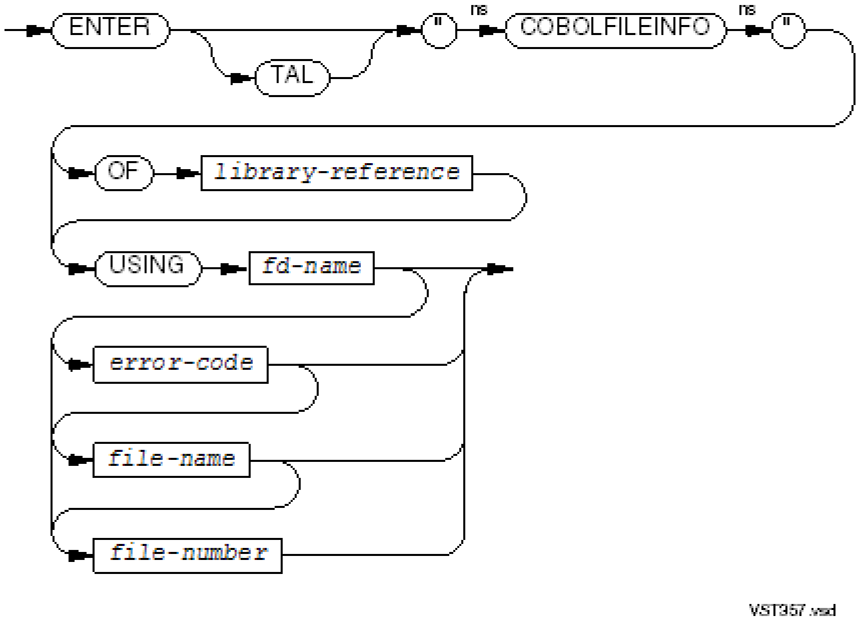
COBOLFILEINFO

The COBOLFILEINFO routine is for the Guardian environment. In the OSS environment, use the COBOL_FILE_INFO_ routine.

The COBOLFILEINFO routine is for these situations:

- An application needs to perform some operation not defined in HP COBOL on a file normally managed by the HP COBOL run-time routines. This application must obtain the file system file number assigned when the file was opened.
- An application can put the file-system file name of a file into an application-generated error message.

As with any ENTER statement, you must either qualify the name COBOLFILEINFO with the *library-reference* ZCOBDLL or include a SEARCH directive specifying ZCOBDLL to provide automatic resolution of the reference. The general form of the ENTER statement to use COBOLFILEINFO follows.



library-reference

is a mnemonic-name associated, in the SPECIAL-NAMES paragraph, with either ZCOBDLL or some other object file containing an object copy of COBOLFILEINFO.

fd-name

is the COBOL name of the file in the FD entry.

error-code

is a numeric data item described as USAGE COMPUTATIONAL PICTURE 9999 (or S9999)or USAGE NATIVE-2. The COBOLFILEINFO routine returns the last file system file-system error code to this data item. The value returned is always positive.

The access mode (see Linkage Section (page 191)) of *error-code* must be STANDARD.

file-name

is a DISPLAY data item of at least 24 characters where the actual 24-byte internal file name is returned. The internal form of the file name is three 8-character fields containing the three portions of the file name, with trailing spaces inserted and with no period (.) separators.

If *file-name* is composed entirely of spaces or has an 8-character volume name, it cannot be converted to internal form. To convert it to internal form, use the COBOL_FILE_INFO_ routine.

The access mode (see Linkage Section (page 191)) of *file-name* must be STANDARD.

file-number

is a numeric data item described as USAGE COMPUTATIONAL PICTURE S99. The COBOLFILEINFO routine returns the file's open number to this data item.

file-number has the value -1 in each of these cases:

- The file is not open.
- The file is OPTIONAL and not present (even if it is open).
- After the file was opened, the I-O status code was "05," rather than "00" (see Nonexistent Files).

The access mode (see Linkage Section (page 191)) of *file-number* must be STANDARD.

Usage Consideration: Do not use COBOLFILEINFO to obtain the operating environment error number associated with the failure of any input-output operation that is performed simply with COBOL input-output verbs (such as READ). The HP COBOL special register GUARDIAN-ERR was created for that purpose. Certain operations, such as a READ statement operating on the $RECEIVE file, deliver meaningful values to GUARDIAN-ERR, but a call to COBOLFILEINFO would return an inappropriate value of zero or an arbitrary, undefined value.

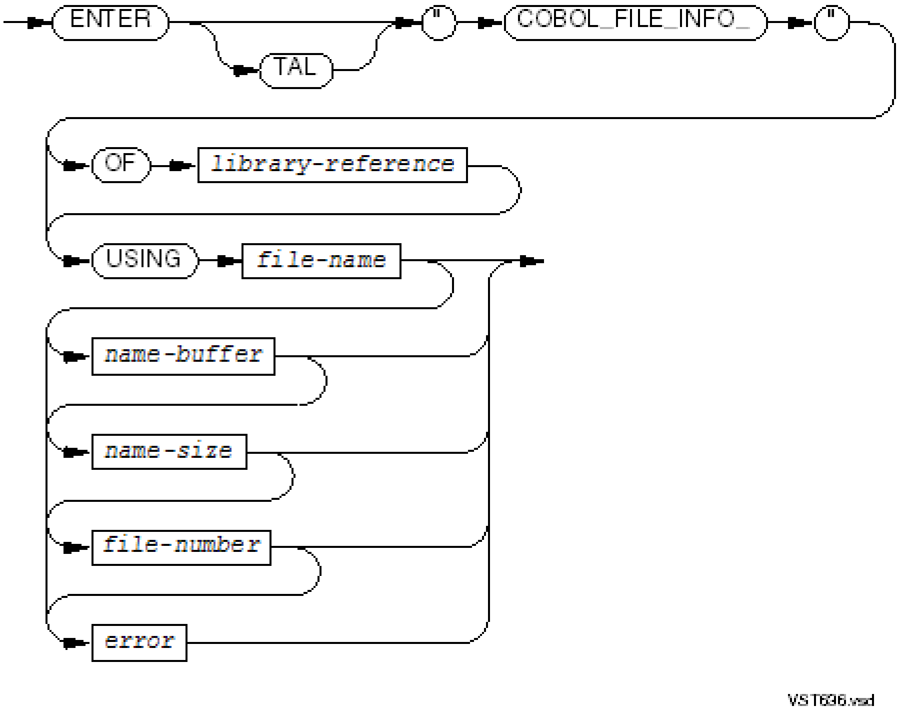
COBOL_FILE_INFO_

The COBOL_FILE_INFO_ routine is for both the OSS and Guardian environments.

The COBOL_FILE_INFO_ routine is for these situations:

- An application needs to perform some operation not defined in HP COBOL on a file normally managed by the HP COBOL run-time routines. This application must obtain the file system file number assigned when the file was opened.
- An application can put the file-system file name of a file into an application-generated error message.

As with any ENTER statement, you must either qualify the name COBOL_FILE_INFO_ with the `library-reference` ZCOBDLL or include a SEARCH directive specifying ZCOBDLL to provide automatic resolution of the reference. The general form of the ENTER statement to use COBOL_FILE_INFO_ follows.



`library-reference`

is a mnemonic-name associated, in the SPECIAL-NAMES paragraph, with either ZCOBDLL or some other object file containing an object copy of COBOL_FILE_INFO_.

`file-name`

is the COBOL name of the file in the FD entry.

`name-buffer`

is the buffer in which the file name is returned. To accommodate the maximum file name size, `name-buffer` must hold at least 1023 characters. To accommodate the largest Guardian file name, `name-buffer` must be at least 35 characters. If `name-buffer` is too short for the file name, the file name is truncated. If the file name is an OSS file, the terminating zero byte is included in the name and in the name size.

`name-size`

is the data item in which the size of the file name is returned. If the file name was truncated, `name-size` contains the actual size, not the truncated size.

file-number

is the Guardian or OSS file number. If the file is not open, *file-number* is 0.

If, after the file was opened, the I-O status code was “05,” rather than “00” (see Nonexistent Files), *file-number* is -1.

error

is the last Guardian error number. If the file is an OSS file, *error* is 0.

Usage Consideration: Do not use COBOL_FILE_INFO_ to obtain the operating environment error number associated with the failure of any input-output operation that is performed simply with COBOL input-output verbs (such as READ). The HP COBOL special register GUARDIAN-ERR was created for that purpose. Certain operations, such as a READ statement operating on the $RECEIVE file, deliver meaningful values to GUARDIAN-ERR, but a call to COBOL_FILE_INFO_ would return an inappropriate value of zero or an arbitrary, undefined value.

Non-Local Jumps in HP COBOL Applications

Sometimes an HP COBOL application may need to bypass the normal program call and return mechanism to handle exceptions or errors encountered during execution. The standard HP C functions `setjmp()` and `longjmp()` provide the functionality for non-local jumps but cannot be called directly from HP COBOL programs using ENTER statements and have more restricted behavior in mixed language applications. If a `longjmp()` call from an HP C function causes an active HP COBOL program to be discarded, the next call to the same HP COBOL program will not work as expected because the context information maintained by HP COBOL is not restored by the `longjmp()` call.

The `cobsetjmp.h` (`cobsetjh` in the Guardian environment) header file defines the `cobsetjmp()` macro and declares the `coblongjmp()` function and `cobjmp_buf` structure for non-local jumps in mixed language HP COBOL programs.

`cobsetjmp()` and `coblongjmp()` can be called only from HP C functions, but unlike `setjmp()` and `longjmp()`, `cobsetjmp()` and `coblongjmp()` can be safely used in mixed language applications. All other restrictions on the use of `setjmp()` and `longjmp()` apply to `cobsetjmp()` and `coblongjmp()` also. In particular, if a C function uses `cobsetjmp()` and then subsequently returns to its caller, you cannot later use `coblongjmp()` with the buffer that was passed to `cobsetjmp()`.

Applications that use `cobsetjmp()` and `coblongjmp()` must be linked with ZCOBDLL.

cobsetjmp macro

NAME

`cobsetjmp()` — saves the current execution context in mixed language applications

SYNOPSIS

```
#include <cobsetjmp.h>
int cobsetjmp (struct cobjmp_buf *buf);
```

LIBRARY

H-series and J-series native Guardian processes: $SYSTEM.ZDLL*nnn*.ZCOBDLL

H-series and J-series OSS processes: /G/system/zdll*nnn*/zcobdll

PARAMETERS

buf

Specifies a buffer in which the current execution environment is saved. The results of calling the `coblongjmp()` function are undefined if the caller changes the contents of *buf*.

DESCRIPTION

The `cobsetjmp()` macro is used with the `coblongjmp()` function for handling errors and interrupts encountered in low-level functions of a mixed-language program. The `cobsetjmp()` macro provides functionality similar to the standard HP C `setjmp()` function.

The `cobsetjmp()` macro cannot be called directly from an HP COBOL program. Applications must call `cobsetjmp()` from an HP C module.

NOTES

All restrictions on the use of `setjmp()` apply to `cobsetjmp()` also.

EXAMPLES

The following example illustrates the use of `cobsetjmp()`:

```
#include <cobsetjmp.h>
struct cobjmp_buf  buf;
void
c_prog_1(void)
{


    if (!cobsetjmp(&buf))
    {
      ..............

      /* can call C or COBOL programs here */
      ..............
    }
    else
      printf("Returned from coblongjmp\n");



}
```

RELATED INFORMATION

cobsetjmp( ), longjmp(3), setjmp(3)

NOTE: The reference pages for longjmp(3) and setjmp(3) are located in the *Open System Services Library Calls Reference Manual*.

coblongjmp function

NAME

`coblongjmp()` — performs a non-local jump in mixed language applications

SYNOPSIS

```
#include <cobsetjmp.h>
void coblongjmp (struct cobjmp_buf *buf);
```

LIBRARY

H-series and J-series native Guardian processes: $SYSTEM.ZDLL*nnn*.ZCOBDLL

H-series and J-series OSS processes: /G/system/zdll*nnn*/zcobdll

PARAMETERS

`buf`

Specifies the buffer in which the execution environment was saved by a call to `cobsetjmp()`.

DESCRIPTION

The `coblongjmp()` function restores the environment preserved in the *`buf`* parameter by a previous call to the `cobsetjmp()` macro. The `coblongjmp()` function provides functionality similar to the standard HP C `longjmp()` function.

The `coblongjmp()` function returns control to the HP C function that called `cobsetjmp()`. The HP C function that called `cobsetjmp()` must still be active.

The `coblongjmp()` function cannot be called directly from an HP COBOL program. Applications must call `coblongjmp()` from an HP C function.

NOTES

All restrictions on the use of `longjmp()` apply to `coblongjmp()` also.

EXAMPLES

The following example illustrates the use of `coblongjmp()`:

```
#include <cobsetjmp.h>
extern struct cobjmp_buf  buf;

void  c_prog_2(void)
{
    ...........
    /* can call C or COBOL programs */
    ...........
    coblongjmp(&buf); /* transfers control back to c_prog_1 */
}
```

RELATED INFORMATION

cobsetjmp( ), longjmp(3), setjmp(3)

NOTE: The reference pages for longjmp(3) and setjmp(3) are located in the *Open System Services Library Calls Reference Manual*.

15 Intrinsic Functions

An intrinsic function is a function that your program can use, but does not need to declare. It returns a value that is computed at the time of reference during the execution of the object program.

Like an identifier, an intrinsic function has a class and category, and you can use an intrinsic function wherever you can use an identifier of the same class and category as a sending data item. An intrinsic function call, which has the format

```
FUNCTION function_name (arguments)
```

is considered an identifier. For example, this is allowed:

```
MOVE FUNCTION NUMVAL (STRING-ITEM) TO NUM-ITEM.
```

You cannot use an intrinsic function as a receiving data item (in a CALL statement, for example). (For explanations of class and category, see Data Levels, Classes, and Categories (page 87).)

Intrinsic Function Types

The type of an intrinsic function is the type of the value it returns. An intrinsic function can be:

- Alphanumeric Intrinsic Functions
- Numeric Intrinsic Functions
- Integer Intrinsic Functions
- Variable-Type Intrinsic Functions

You can use an integer function wherever you can use a numeric operand, but you cannot use a numeric function where an integer operand is required, even if a particular reference to the numeric function returns an integer value.

You can nest intrinsic functions; that is, you can use one intrinsic function as the argument of another. The function that is used as the argument must be of the type required for that argument.

For example,

```
FUNCTION MAX ( FUNCTION INTEGER(X) FUNCTION INTEGER-PART(X) )
```

is allowed because the functions INTEGER and INTEGER-PART are integer functions and the function MAX accepts integer arguments.

```
FUNCTION UPPER-CASE ( FUNCTION FACTORIAL(5) )
```

is not allowed because the function FACTORIAL is an integer function and the function UPPER-CASE requires an alphanumeric argument.

Alphanumeric Intrinsic Functions

An alphanumeric intrinsic function returns an alphanumeric value (a string of one or more alphanumeric characters).

Table 15-1 Alphanumeric Intrinsic Functions

Function Name	Value Returned
CHAR Function	Character in specified position of program collating sequence
CURRENT-DATE Function	Current date and time and difference from Greenwich mean time
LOWER-CASE Function	Its argument with all letters set to lowercase
REVERSE Function	Its argument with characters in reverse order

Table 15-1 Alphanumeric Intrinsic Functions *(continued)*

Function Name	Value Returned
UPPER-CASE Function	Its argument with all letters set to uppercase
WHEN-COMPILED Function	Date and time when program was compiled

Numeric Intrinsic Functions

A numeric intrinsic function returns a numeric value.

Table 15-2 Numeric Intrinsic Functions

Function Name	Value Returned
ACOS Function	Arccosine of argument
ANNUITY Function	Ratio of annuity paid for a specified number of periods at a specified interest rate to initial investment of one
ASIN Function	Arcsine of argument
ATAN Function	Arctangent of argument
COS Function	Cosine of argument
LOG Function	Natural logarithm of argument
LOG10 Function	Logarithm to base 10 of argument
MEAN Function	Arithmetic mean of arguments
MEDIAN Function	Median of arguments
MIDRANGE Function	Midrange of arguments
NUMVAL Function	Numeric value of simple numeric string
NUMVAL-C Function	Numeric value of numeric string with optional commas and currency sign
PRESENT-VALUE Function	Present value of a specified series of future period-end amounts at a specified discount rate
RANDOM Function	Pseudorandom number
REM Function	Remainder when one argument is divided by the other
SIN Function	Sine of argument
SQRT Function	Square root of argument
STANDARD-DEVIATION Function	Standard deviation of arguments
TAN Function	Tangent of argument
VARIANCE Function	Variance of arguments

Integer Intrinsic Functions

An integer intrinsic function returns an integer value.

Table 15-3 Integer Intrinsic Functions

Function Name	Value Returned
DATE-OF-INTEGER Function	Standard date equivalent of integer date (*YYYYMMDD*)*
DAY-OF-INTEGER Function	Julian date equivalent of integer date (*YYYYDDD*)*
FACTORIAL Function	Factorial of argument

Table 15-3 Integer Intrinsic Functions *(continued)*

Function Name	Value Returned
INTEGER Function	Greatest integer not greater than argument
INTEGER-OF-DATE Function	Integer date equivalent of standard date (*YYYYMMDD*)*
INTEGER-OF-DAY Function	Integer date equivalent of Julian date (*YYYYDDD*)*
INTEGER-PART Function	Integer part of argument
LENGTH Function	Length of argument
MOD Function	First argument modulo second argument
ORD Function	Ordinal position of argument in program collating sequence
ORD-MAX Function	Ordinal position of maximum argument
ORD-MIN Function	Ordinal position of minimum argument
TEST-NUMVAL Function	Zero or character position of first invalid character
TEST-NUMVAL-C Function	Zero or character position of first invalid character

* Date conversion functions use the Gregorian calendar.

Variable-Type Intrinsic Functions

The type of some intrinsic functions depends on the type(s) of its arguments.

Table 15-4 Variable-Type Intrinsic Functions

Function Name	Value Returned
MAX Function	Value of maximum argument
MIN Function	Value of minimum argument
RANGE Function	Value of maximum argument minus value of minimum argument
SUM Function	Sum of arguments

Table 15-5 How Argument Type Determines Variable-Type Intrinsic Function Type

Argument Type	Function Type
Alphabetic	Alphanumeric
Alphanumeric	Alphanumeric
Integer	Integer
Numeric (some can be integer)	Numeric

NOTE: Not every function in Table 15-4 can have arguments of every type in Table 15-5. For valid argument types for a particular function, see the description of that function later in this section.

Argument Types

Table 15-6 lists the types of arguments that intrinsic functions can have, tells you what you must specify for an argument of each type, and tells you what is used in determining the function’s value.

NOTE: Not every intrinsic function can have arguments of every type in Table 15-6. For valid argument types for a particular function, see the description of that function later in this section.

Table 15-6 Argument Types

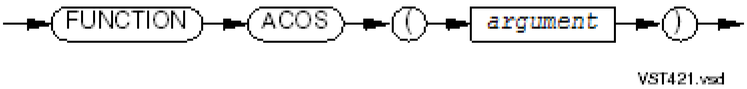
Argument Type	What You Must Specify	What is Used in Determining the Function's Value
Numeric	Arithmetic expression	Value of arithmetic expression, including operational sign
Integer	Arithmetic expression that always results in integer value	Value of arithmetic expression, including operational sign
Alphabetic	Elementary data item of the class alphabetic, or nonnumeric literal containing only alphabetic characters	Value of argument and possibly its size
Alphanumeric	Data item of the class alphabetic or alphanumeric, or nonnumeric literal	Value of argument and possibly its size

An arithmetic expression used as an argument can include subscripts, and any subscript can be the keyword ALL. When ALL is a subscript, the effect is as if each table element associated with that subscript position were specified. The order of the implicit specification of each occurrence is from left to right. The leftmost specification is the identifier with each subscript specified by the word ALL replaced by one. The next specification is the same identifier with the rightmost subscript specified by the word ALL incremented by one. This process continues with the rightmost ALL subscript being incremented by one for each implicit specification until the rightmost ALL subscript has been incremented through its range of values. If there are additional ALL subscripts, the ALL subscript immediately to the left of the rightmost ALL subscript is incremented by one, the rightmost ALL subscript is set to one, and the process of varying the rightmost ALL subscript is repeated. The ALL subscript to the left of the rightmost ALL subscript is incremented by one through its range of values. For each additional ALL subscript, this process is repeated until the leftmost ALL subscript has been incremented by one through its range of values. If the ALL subscript is associated with an OCCURS DEPENDING clause, the object of the OCCURS DEPENDING clause determines the range of values. The evaluation of an ALL subscript must result in at least one argument, or the result of the reference to the function-identifier is undefined.

An invalid argument causes a run-time error. In the non-CRE environment, it is error 130 ("Invalid function parameter"). In the CRE, it is error 40 ("Invalid function parameter"). If you are executing an old object file on a newer system, the error might be "Corrupted environment."

ACOS Function

ACOS, a numeric function, returns a value in radians that approximates the arccosine of its argument.



`argument`

is a numeric argument whose value is in the range -1 to +1.

The returned value approximates the arccosine of the value of *argument*. It is a floating-point number in the range zero to pi. It has no implied decimal point and is precise to approximately 14 digits.

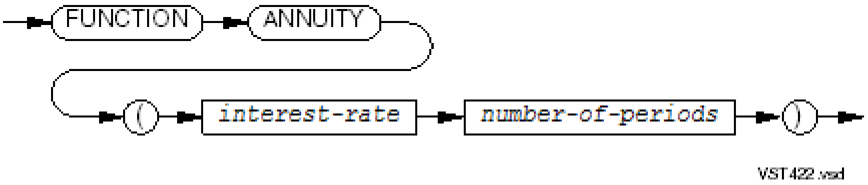
Example 15-1 ACOS Function

Code:

```
DATA DIVISION.
  WORKING-STORAGE SECTION.
    01 A PICTURE S9V99.
PROCEDURE DIVISION.
  MOVE FUNCTION ACOS (-1)  TO A.
  DISPLAY A.
  MOVE FUNCTION ACOS (.25) TO A.
  DISPLAY A.
  MOVE FUNCTION ACOS (.5)  TO A.
  DISPLAY A.
  MOVE FUNCTION ACOS (.75) TO A.
  DISPLAY A.
  MOVE FUNCTION ACOS (1)   TO A.
  DISPLAY A.
```

ANNUITY Function

ANNUITY, a numeric function, returns a value that approximates the ratio of an annuity paid at the end of each period for a specified number of periods at a specified interest rate to an initial investment of one. Interest is applied at the end of the period, before the payment (annuity immediate).



interest-rate

is a numeric argument whose value is greater than or equal to zero.

number-of-periods

is a positive integer.

When the value of *interest-rate* is zero, the returned value is approximately:

$$\frac{1}{number\text{-}of\text{-}periods}$$

When the value of *interest-rate* is not zero, the returned value is approximately:

$$\frac{interest\text{-}rate}{+\ interest\text{-}rate)^{-number\text{-}of\text{-}periods}}$$

Example 15-2 ANNUITY Function

Code:

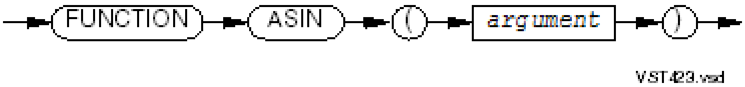
```
DATA DIVISION.
   WORKING-STORAGE SECTION.
     01 A PICTURE 9V99.
PROCEDURE DIVISION.
   MOVE FUNCTION ANNUITY (1 12) TO A.
   DISPLAY A.
   MOVE FUNCTION ANNUITY (1  6) TO A.
   DISPLAY A.
   MOVE FUNCTION ANNUITY (2  6) TO A.
   DISPLAY A.
   MOVE FUNCTION ANNUITY (5  2) TO A.
   DISPLAY A.   MOVE FUNCTION ANNUITY (0  5) TO A.
   DISPLAY A.
```

Output:

```
 1.00 1.01 2.00 5.14 0.20
```

ASIN Function

ASIN, a numeric function, returns a value in radians that approximates the arcsine of its argument.



argument

is a numeric argument whose value is in the range -1 to +1.

The returned value approximates the arcsine of the value of *argument*. It is a floating-point number in the range -pi/2 to +pi/2. It has no implied decimal point and is precise to approximately 14 digits.

Example 15-3 ASIN Function

Code:

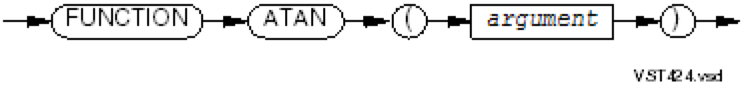
```
DATA DIVISION.
  WORKING-STORAGE SECTION.
    01 A PICTURE S9V99.

PROCEDURE DIVISION.
  MOVE FUNCTION ASIN (-1)   TO A.
  DISPLAY A.
  MOVE FUNCTION ASIN (.25) TO A.
  DISPLAY A.
  MOVE FUNCTION ASIN (.5)   TO A.
  DISPLAY A.
  MOVE FUNCTION ASIN (.75) TO A.
  DISPLAY A.
  MOVE FUNCTION ASIN (1)    TO A.
  DISPLAY A.
```

Output:

```
-1.57
0.25
0.52
0.84
1.57
```

ATAN Function

ATAN, a numeric function, returns a value in radians that approximates the arctangent of its argument.



argument

is a numeric argument.

The returned value approximates the arctangent of the value of *argument*. It is a floating-point number in the range -pi/2 to +pi/2, excluding the endpoints. It has no implied decimal point and is precise to approximately 14 digits.

Example 15-4 ATAN Function

Code:

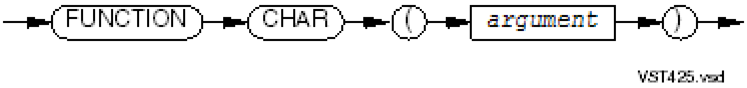
```
DATA DIVISION.
  WORKING-STORAGE SECTION.
    01 A PICTURE S9V99.
PROCEDURE DIVISION.
  MOVE FUNCTION ATAN (-10)   TO A.
  DISPLAY A.
  MOVE FUNCTION ATAN (0)     TO A.
  DISPLAY A.
  MOVE FUNCTION ATAN (.25)   TO A.
  DISPLAY A.
  MOVE FUNCTION ATAN (15.5)  TO A.
  DISPLAY A.
  MOVE FUNCTION ATAN (45)    TO A.
  DISPLAY A.
```

Output:

```
-1.47
0.00
0.24
1.50
1.54
```

CHAR Function

CHAR, an alphanumeric function, returns a character whose ordinal number in the program collating sequence is the value of its argument.



argument

is an integer greater than zero and less than or equal to *n*, where *n* is the number of positions in the program collating sequence.

If more than one character has the same position in the program collating sequence, the returned character is the first character that the ALPHABET clause specifies for that character position. If the current program collating sequence was not specified by an ALPHABET clause, HP COBOL uses the ASCII program collating sequence.

Example 15-5 CHAR Function Without ALPHABET Clause

Code:

```
DISPLAY FUNCTION CHAR (36)
DISPLAY FUNCTION CHAR (43)
DISPLAY FUNCTION CHAR (54)
DISPLAY FUNCTION CHAR (69)
DISPLAY FUNCTION CHAR (99)
```

Output:

```
#
*
5
D
b
```

Example 15-6 CHAR Function With ALPHABET Clause

Code:

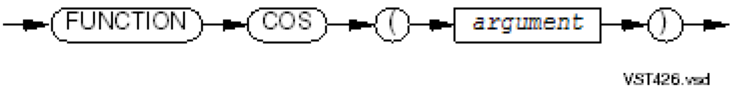
```
ENVIRONMENT DIVISION.
  CONFIGURATION SECTION.
    SPECIAL-NAMES.
      ALPHABET CAPITAL-LETTERS IS 66 THRU 91.
  SOURCE-COMPUTER.  ABD.
  OBJECT-COMPUTER.  ABD
    PROGRAM COLLATING SEQUENCE IS CAPITAL-LETTERS.
PROCEDURE DIVISION.
  Startt.
  DISPLAY FUNCTION CHAR (1)
  DISPLAY FUNCTION CHAR (2)
  DISPLAY FUNCTION CHAR (13)
  DISPLAY FUNCTION CHAR (20)
  DISPLAY FUNCTION CHAR (25)
```

Output:

```
A
B
M
T
Y
```

COS Function

COS, a numeric function, returns a value that approximates the cosine of the angle or arc (in radians) specified by its argument.



argument

is a numeric argument.

The returned value approximates the cosine of the value of *argument*. It is a floating-point number in the range -1 to +1. It has no implied decimal point and is precise to approximately 14 digits.

Example 15-7 COS Function

Code:

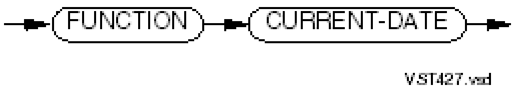
```
DATA DIVISION.
  WORKING-STORAGE SECTION.
    01 A PICTURE S9V99.
PROCEDURE DIVISION.
  MOVE FUNCTION COS (-10)   TO A.
  DISPLAY A.
  MOVE FUNCTION COS (0)     TO A.
  DISPLAY A.
  MOVE FUNCTION COS (.25)   TO A.
  DISPLAY A.
  MOVE FUNCTION COS (15.5) TO A.
  DISPLAY A.
  MOVE FUNCTION COS (45)    TO A.
  DISPLAY A.
```

Output:

```
-0.83
1.00
0.96
-0.97
0.52
```

CURRENT-DATE Function

CURRENT-DATE, an alphanumeric function, returns a 21-character string that represents the calendar date, time of day, and local time differential factor provided by the operating environment.



The returned value represents the current date this way (where character position 1 is the leftmost character position):

Character Positions	Date Part Represented	Represented by
1-4	Year in Gregorian calendar	Four digits
5-6	Month of the year	Two digits in the range 01 through 12
7-8	Day of the month	Two digits in the range 01 through 31
9-10	Hours past midnight	Two digits in the range 00 through 23
11-12	Minutes past the hour	Two digits in the range 00 through 59
13-14	Seconds past the minute	Two digits in the range 00 through 59
15-16	Hundredths of seconds past the second	Two digits in the range 00 through 99
17	Relationship to Greenwich mean time	Minus (-) if the reported time is behind Greenwich mean time, plus (+) if it is the same as or ahead of Greenwich mean time
18-19	Hours behind or ahead of Greenwich mean time	If character position 17 is minus (-), two digits in the range 00 through 12; if character position 17 is plus (+), two digits in the range 00 through 13
20-21	Additional minutes from Greenwich mean time	Two digits in the range 00 through 59

Example 15-8 CURRENT-DATE Function

Code:

```
DISPLAY FUNCTION CURRENT-DATE
```

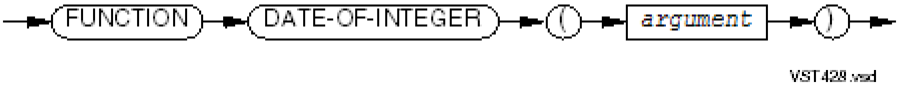
Output:

```
1997041116522224-0700
```

The date in Example 15-8 is April 11, 1997 (19970411). The time is 16:52:22.24 or 4:52:22.24 PM (16522224), which is 7 hours behind Greenwich mean time (-0700).

DATE-OF-INTEGER Function

DATE-OF-INTEGER, an integer function, converts a date in the Gregorian calendar from integer date form to standard date form (*YYYYMMDD*).



argument

is a positive integer that represents a number of days since December 31, 1600, in the Gregorian calendar.

The returned value is of the form *YYYYMMDD*, where *YYYY* represents the year in the Gregorian calendar, *MM* represents the month of that year, and *DD* represents the day of that month.

Example 15-9 DATE-OF-INTEGER Function

Code:

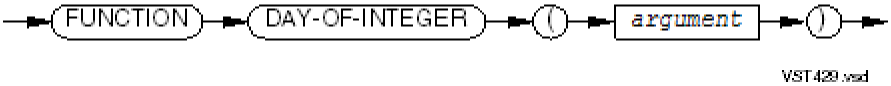
```
DISPLAY FUNCTION DATE-OF-INTEGER (1)
DISPLAY FUNCTION DATE-OF-INTEGER (365)
DISPLAY FUNCTION DATE-OF-INTEGER (36500)
DISPLAY FUNCTION DATE-OF-INTEGER (12345)
DISPLAY FUNCTION DATE-OF-INTEGER (100000)
```

Output:

```
16010101
16011231
17001207
16341019
18741016
```

DAY-OF-INTEGER Function

DAY-OF-INTEGER, an integer function, converts a date in the Gregorian calendar from integer date form to Julian date form (*YYYYDDD*).



argument

is a positive integer that represents a number of days since December 31, 1600, in the Gregorian calendar.

The returned value is of the form *YYYYDDD*, where *YYYY* represents the year in the Gregorian calendar and *DDD* represents the day of that year.

Example 15-10 DAY-OF-INTEGER Function

Code:

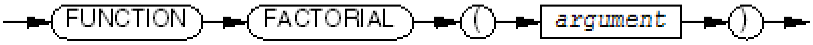
```
DISPLAY FUNCTION DAY-OF-INTEGER (1)
DISPLAY FUNCTION DAY-OF-INTEGER (365)
DISPLAY FUNCTION DAY-OF-INTEGER (36500)
DISPLAY FUNCTION DAY-OF-INTEGER (12345)
DISPLAY FUNCTION DAY-OF-INTEGER (100000)
```

Output:

```
1601001
1601365
1700341
1634292
1874289
```

FACTORIAL Function

FACTORIAL, an integer function, returns the factorial of its argument.



VST430.vsd

argument

is an integer greater than or equal to zero.

When *argument* is zero, the returned value is 1; when *argument* is positive, the returned value is its factorial.

Example 15-11 FACTORIAL Function

Code:

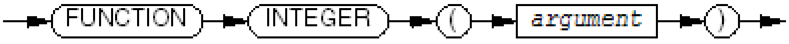
```
DISPLAY FUNCTION FACTORIAL (0)
DISPLAY FUNCTION FACTORIAL (1)
DISPLAY FUNCTION FACTORIAL (4)
DISPLAY FUNCTION FACTORIAL (6)
DISPLAY FUNCTION FACTORIAL (10)
```

Output:

```
000000000000000001
000000000000000001
000000000000000024
000000000000000720
000000000003628800
```

INTEGER Function

INTEGER, an integer function, returns the greatest integer that is less than or equal to its argument.



VST431.vsd

argument

is a numeric argument.

The returned value is the greatest integer that is less than or equal to the value of *argument*.

Example 15-12 INTEGER Function

Code:

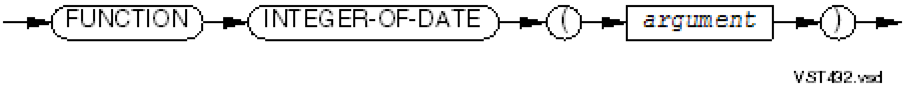
```
DISPLAY FUNCTION INTEGER (-3.5)
DISPLAY FUNCTION INTEGER (-3)
DISPLAY FUNCTION INTEGER (3)
DISPLAY FUNCTION INTEGER (3.5)
```

Output:

```
-4
-3
3
3
```

INTEGER-OF-DATE Function

INTEGER-OF-DATE, an integer function, converts a date in the Gregorian calendar from standard date form (*YYYYMMDD*) to integer date form.



argument

is an integer of the form YYYYMMDD, whose value is calculated from the equation (YYYY * 10,000) + (MM * 100) + DD where:

Factor	Date Part Represented	Represented by
YYYY	Year in Gregorian calendar	Integer in the range 1,601 through 9,999
MM	Month of the year	Integer in the range 1 through 12
DD	Day of the month	Integer in the range 1 through 31 that is a valid day for the month specified by *MM*

The returned value is an integer that is the number of days that the date represented by *argument* succeeds December 31, 1600, in the Gregorian calendar.

Example 15-13 INTEGER-OF-DATE Function

Code:

```
DISPLAY FUNCTION INTEGER-OF-DATE (16010101)
DISPLAY FUNCTION INTEGER-OF-DATE (16011231)
DISPLAY FUNCTION INTEGER-OF-DATE (17001207)
DISPLAY FUNCTION INTEGER-OF-DATE (16341019)
DISPLAY FUNCTION INTEGER-OF-DATE (18741016)
```

Output:

```
00000001
00000365
00036500
00012345
00100000
```

Example 15-14 uses the INTEGER-OF-DATE and REM functions to find the day of the week for dates in standard date form (*YYYYMMDD*). For a description of the REM function, see REM Function.

Example 15-14 INTEGER-OF-DATE Function

Code:

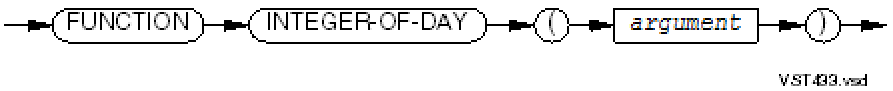
```
?ENV COMMON
 IDENTIFICATION DIVISION.
 PROGRAM-ID. TESTT.
 ENVIRONMENT DIVISION.
   CONFIGURATION SECTION.
     SOURCE-COMPUTER.  ABD.
     OBJECT-COMPUTER.  ABD.
 DATA DIVISION.
   WORKING-STORAGE SECTION.
     01 D PICTURE 9 USAGE COMPUTATIONAL.
     01 STD-DATE PICTURE 9(8) USAGE COMPUTATIONAL.
 PROCEDURE DIVISION.
 STARTT.
   PERFORM VARYING STD-DATE FROM 19970618 BY 1
   UNTIL STD-DATE > 19970624
     DISPLAY STD-DATE
     MOVE FUNCTION REM (FUNCTION INTEGER-OF-DATE (STD-DATE) 7)
     TO D
     IF D = 0 THEN DISPLAY "  SUNDAY" ELSE
       IF D = 1 THEN DISPLAY "  MONDAY" ELSE
         IF D = 2 THEN DISPLAY "  TUESDAY" ELSE
           IF D = 3 THEN DISPLAY "  WEDNESDAY" ELSE
             IF D = 4 THEN DISPLAY "  THURSDAY" ELSE
               IF D = 5 THEN DISPLAY "  FRIDAY" ELSE
                 IF D = 6 THEN DISPLAY "  SATURDAY" END-IF
               END-IF
             END-IF
           END-IF
         END-IF
       END-IF
     END-IF
   END-PERFORM.
 STOP RUN.
```

Output:

```
19970618
  WEDNESDAY
19970619
  THURSDAY
19970620
  FRIDAY
19970621
  SATURDAY
19970622
  SUNDAY
19970623
  MONDAY
19970624
  TUESDAY
```

INTEGER-OF-DAY Function

INTEGER-OF-DAY, an integer function, converts a date in the Gregorian calendar from Julian date form (*YYYYDDD*) to integer date form.



argument

is an integer of the form *YYYYDDD*, whose value is calculated from the equation (*YYYY* * 1000) + *DD* where:

Factor	Date Part Represented	Represented by
YYYY	Year in Gregorian calendar	Integer in the range 1,601 through 9,999
DD	Day of the year	Integer in the range 1 through 366 that is a valid day for the year specified by *YYYY*

The returned value is an integer that is the number of days that the date represented by *argument* succeeds December 31, 1600, in the Gregorian calendar.

Example 15-15 INTEGER-OF-DAY Function

Code:

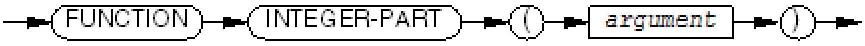
```
DISPLAY FUNCTION INTEGER-OF-DAY (1601001)
DISPLAY FUNCTION INTEGER-OF-DAY (1601365)
DISPLAY FUNCTION INTEGER-OF-DAY (1700341)
DISPLAY FUNCTION INTEGER-OF-DAY (1634292)
DISPLAY FUNCTION INTEGER-OF-DAY (1874289)
```

Output:

```
00000001
00000365
00036500
00012345
00100000
```

INTEGER-PART Function

INTEGER-PART, an integer function, returns the integer part of its argument.



VST494.vsd

argument

is a numeric argument.

The returned value is determined as shown in this table:

Value of argument	Returned Value
Zero	Zero
Positive	Greatest integer less than or equal to the value of *argument*
Negative	Least integer greater than or equal to the value of *argument*

Example 15-16 INTEGER-PART Function

Code:

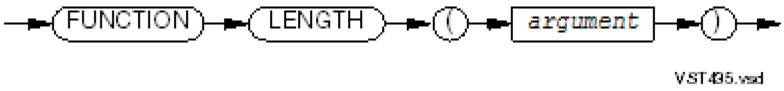
```
DISPLAY FUNCTION INTEGER-PART (-1.5)
DISPLAY FUNCTION INTEGER-PART (-1)
DISPLAY FUNCTION INTEGER-PART (0)
DISPLAY FUNCTION INTEGER-PART (1)
DISPLAY FUNCTION INTEGER-PART (1.5)
```

Output:

```
-1
-1
0
1
1
```

LENGTH Function

LENGTH, an integer function, returns the length of its argument (the number of character positions it has, regardless of its current value).



argument

is a nonnumeric literal or data item of any class or category.

The returned value is determined as shown in this table. It includes any implicit FILLER characters.

argument	Returned Value
nonnumeric literal or elementary data item or data structure that does not contain a variable occurrence data item	Length of the value of *argument*.
data structure containing a variable occurrence data item	Length of the current value of the data item specified in the DEPENDING phrase of the OCCURS clause for the variable occurrence data item. The current value is determined as if the data item were a sending data item.

Example 15-17 LENGTH Function

Code:

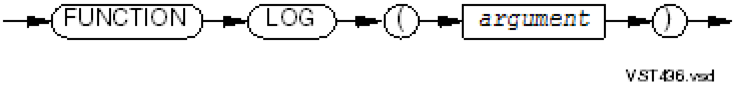
```
01 STUFF-1 PIC XXXX.
01 STUFF-2 PIC 9999.
MOVE "AB" TO STUFF-1.
MOVE 12 TO STUFF-2.
DISPLAY FUNCTION LENGTH(STUFF-1).
DISPLAY STUFF-2.
DISPLAY FUNCTION LENGTH(STUFF-2).
DISPLAY FUNCTION LENGTH("ABC").
```

Output:

```
4
0012
4
3
```

LOG Function

LOG, a numeric function, returns a value that approximates the logarithm to the base e (natural logarithm) of its argument.



argument

is a numeric argument greater than zero.

The returned value is approximately the logarithm to the base e of the value of `argument`.

Example 15-18 LOG Function

Code:

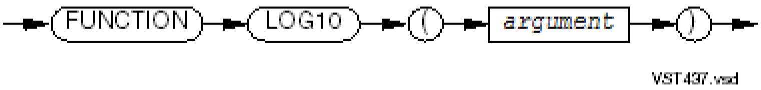
```
 DATA DIVISION.
   WORKING-STORAGE SECTION.
     01 A PICTURE S9V99.
 PROCEDURE DIVISION.
  MOVE FUNCTION LOG (1)     TO A.
  DISPLAY A.
  MOVE FUNCTION LOG (10)    TO A.
  DISPLAY A.
  MOVE FUNCTION LOG (100)   TO A.
  DISPLAY A.
  MOVE FUNCTION LOG (1000)  TO A.
  DISPLAY A.
  MOVE FUNCTION LOG (10000) TO A.
  DISPLAY A.
```

Output:

```
0.00
2.30
4.60
6.90
9.21
```

LOG10 Function

LOG10, a numeric function, returns a value that approximates the logarithm to the base 10 of its argument.



argument

  is a numeric argument greater than zero.

The returned value is approximately the logarithm to the base 10 of the value of *argument*.

Example 15-19 LOG10 Function

Code:

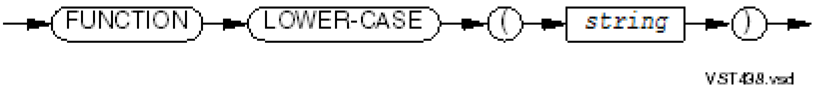
```
DATA DIVISION.
  WORKING-STORAGE SECTION.
    01 A PICTURE S9V99.
PROCEDURE DIVISION.
  MOVE FUNCTION LOG10 (1)     TO A.
  DISPLAY A.
  MOVE FUNCTION LOG10 (10)    TO A.
  DISPLAY A.
  MOVE FUNCTION LOG10 (100)   TO A.
  DISPLAY A.
  MOVE FUNCTION LOG10 (1000)  TO A.
  DISPLAY A.
  MOVE FUNCTION LOG10 (10000) TO A.
  DISPLAY A.
```

Output:

```
0.00
1.00
2.00
3.00
4.00
```

LOWER-CASE Function

LOWER-CASE, an alphanumeric function, returns a character-string that is the same as its argument, except that each uppercase letter is replaced by the corresponding lowercase letter.



string

  is an alphabetic or alphanumeric string at least one character in length.

The returned value is the same as the value of *string*, except that each uppercase letter is replaced by the corresponding lowercase letter. If the value of *string* contains no uppercase letters, or if an ALPHABET clause specifies a collating sequence that contains no lowercase letters, the returned value is the same as the value of *string*.

Example 15-20 LOWER-CASE Function

Code:

```
DISPLAY FUNCTION LOWER-CASE ("HEWLETT-PACKARD COMPANY 2003")
DISPLAY FUNCTION LOWER-CASE ("Hewlett-packard Company 2003")
DISPLAY FUNCTION LOWER-CASE ("hewlett-packard company 2003")
```

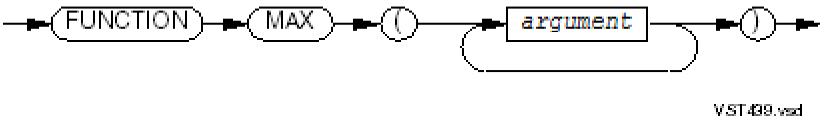
Output:

```
hewlett-packard company 2003
hewlett-packard company 2003
hewlett-packard company 2003
```

MAX Function

MAX is a function that returns the value of its maximum argument. Its type depends on the type of its arguments, as this table shows:

Argument Type	Function Type
Alphabetic	Alphanumeric
Alphanumeric	Alphanumeric
Integer	Integer
Numeric (some can be integer)	Numeric



argument

is an alphabetic, alphanumeric, integer, or numeric argument. If you specify more than one argument, they must all be of the same class. Integer and numeric arguments can be mixed, because integers are of the class numeric.

argument can be an array; for example,

```
FUNCTION MAX (ARRAY1(ALL))
```

returns the largest element of the array ARRAY1.

The returned value is the *argument* with the greatest value (according to the rules for evaluating simple conditions). If the function type is alphanumeric, the size of the returned value is the same as that of the *argument* with the greatest value. If two arguments have the same greatest value, the value of the leftmost *argument* is returned.

Example 15-21 MAX Function

Code:

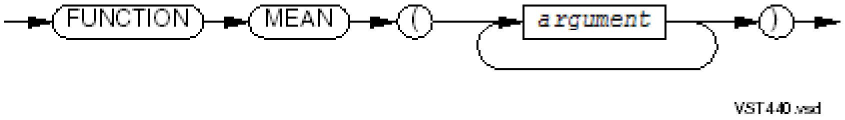
```
DATA DIVISION.
  WORKING-STORAGE SECTION.
    01 NUMERIC-ARRAY.
       05 PICTURE 9(1) VALUE 4.
       05 PICTURE 9(1) VALUE 9.
       05 PICTURE 9(1) VALUE 3.
       05 PICTURE 9(1) VALUE 7.
       05 PICTURE 9(1) VALUE 5
    01 NUM-ARRAY REDEFINES NUMERIC-ARRAY.
       05 NUM OCCURS 5 TIMES PICTURE 9(1).
    01 ALPHABETIC-ARRAY.
       05 PICTURE X(5) VALUE "dog".
       05 PICTURE X(5) VALUE "cat".
       05 PICTURE X(5) VALUE "horse".
       05 PICTURE X(5) VALUE "sheep".
       05 PICTURE X(5) VALUE "goat".
    01 ALPHA-ARRAY REDEFINES ALPHABETIC-ARRAY.
       05 ALPHA OCCURS 5 TIMES PICTURE X(5).
PROCEDURE DIVISION.
  ...
  DISPLAY FUNCTION MAX (NUM(ALL))
  DISPLAY FUNCTION MAX (ALPHA(ALL))
  DISPLAY FUNCTION MAX (NUM(1) NUM(4))
  DISPLAY FUNCTION MAX (ALPHA(3) "bird" "fish")
  DISPLAY FUNCTION MAX (3.4 5 6.2 9)
```

Output:

```
9
sheep
7
horse
9.0
```

MEAN Function

MEAN, a numeric function, returns the arithmetic mean (average) of its arguments.



argument

is a numeric argument.

argument can be an array; for example,

```
FUNCTION MEAN (ARRAY1(ALL))
```

returns the arithmetic mean of the elements of the array ARRAY1.

The returned value is the arithmetic mean (average) of the *argument* series; that is

$$\frac{\sum_{i=1}^{n} argument_i}{n}$$

where n is the number of arguments.

Example 15-22 MEAN Function

Code:

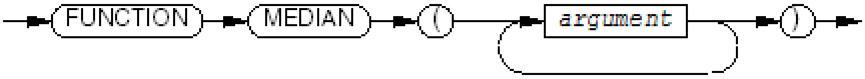
```
DATA DIVISION.
  WORKING-STORAGE SECTION.
    01 NUMERIC-ARRAY.
       05 PICTURE 9(1) VALUE 4.
       05 PICTURE 9(1) VALUE 9.
       05 PICTURE 9(1) VALUE 3.
       05 PICTURE 9(1) VALUE 7.
       05 PICTURE 9(1) VALUE 5
    01 NUM-ARRAY REDEFINES NUMERIC-ARRAY.
       05 NUM OCCURS 5 TIMES PICTURE 9(1).
    01 A PICTURE S9V9.
PROCEDURE DIVISION.
  MOVE FUNCTION MEAN (NUM(ALL)) TO A.
  DISPLAY A.
  MOVE FUNCTION MEAN (NUM(1) NUM(3) NUM(5)) TO A.
  DISPLAY A.
  MOVE FUNCTION MEAN (3.4  5  6.2  9) TO A.
  DISPLAY A.
```

Output:

```
5.6
4.0
5.9
```

MEDIAN Function

MEDIAN, a numeric function, returns the value that would be the middle value in a list formed by sorting its arguments numerically.



VST441.vsd

argument

is a numeric argument.

argument can be an array; for example,

```
FUNCTION MEDIAN (ARRAY1(ALL))
```

returns the median element of the array ARRAY1.

The returned value is the value that would be in the middle of the list if the arguments were sorted numerically (according to the rules for comparing simple conditions). If there are an odd number of arguments, at least half of them are less than or equal to the returned value, and at least half of them are greater than or equal to the returned value. If there are an even number of arguments, the returned value is the arithmetic mean of the two middle arguments. (For the definition of arithmetic mean, see MEAN Function).

Example 15-23 MEDIAN Function

Code:

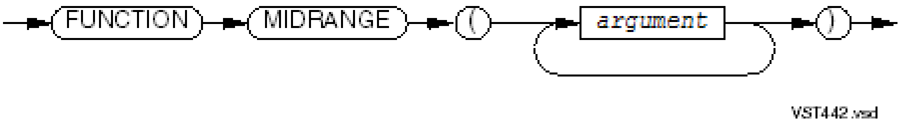
```
DATA DIVISION.
  WORKING-STORAGE SECTION.
    01 NUMERIC-ARRAY.
       05 PICTURE 9(1) VALUE 4.
       05 PICTURE 9(1) VALUE 9.
       05 PICTURE 9(1) VALUE 3.
       05 PICTURE 9(1) VALUE 7.
       05 PICTURE 9(1) VALUE 5
    01 NUM-ARRAY REDEFINES NUMERIC-ARRAY.
       05 NUM OCCURS 5 TIMES PICTURE 9(1).
    01 A PICTURE S9V9.
PROCEDURE DIVISION.
  MOVE FUNCTION MEDIAN (NUM(ALL)) TO A.
  DISPLAY A.
  MOVE FUNCTION MEDIAN (NUM(1) NUM(3) NUM(5)) TO A.
  DISPLAY A.
  MOVE FUNCTION MEDIAN (4 1 7 5 2 3 6) TO A.
  DISPLAY A.
  MOVE FUNCTION MEDIAN (4 1 6 5 2 3) TO A.
  DISPLAY A.
```

Output:

```
5.0
4.0
4.0
3.5
```

MIDRANGE Function

MIDRANGE, a numeric function, returns the arithmetic mean (average) of its minimum and maximum arguments.



argument

is a numeric argument.

argument can be an array; for example,

```
FUNCTION MIDRANGE (ARRAY1(ALL))
```

returns the arithmetic mean of the minimum and maximum elements of the array ARRAY1.

The returned value is the arithmetic mean (average) of the least *argument* and the greatest *argument* (according to the rules for comparing simple conditions).

Example 15-24 MIDRANGE Function

Code:

```
DATA DIVISION.
  WORKING-STORAGE SECTION.
    01 NUMERIC-ARRAY.
       05 PICTURE 9(1) VALUE 4.
       05 PICTURE 9(1) VALUE 9.
       05 PICTURE 9(1) VALUE 3.
       05 PICTURE 9(1) VALUE 7.
       05 PICTURE 9(1) VALUE 5
    01 NUM-ARRAY REDEFINES NUMERIC-ARRAY.
       05 NUM OCCURS 5 TIMES PICTURE 9(1).
    01 A PICTURE S9V9.
PROCEDURE DIVISION.
  MOVE FUNCTION MIDRANGE (NUM(ALL)) TO A.
  DISPLAY A.
  MOVE FUNCTION MIDRANGE (NUM(1) NUM(3) NUM(5)) TO A.
  DISPLAY A.
  MOVE FUNCTION MIDRANGE (8 1 3 6) TO A.
  DISPLAY A.
```

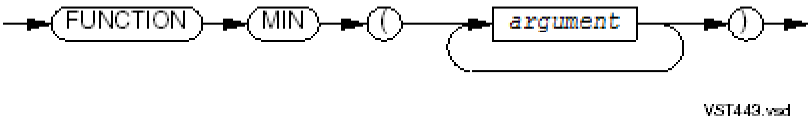
Output:

```
6.0
4.0
4.5
```

MIN Function

MIN is a function that returns the value of its minimum argument. Its type depends on the type of its arguments, as this table shows:

Argument Type	Function Type
Alphabetic	Alphanumeric
Alphanumeric	Alphanumeric
Integer	Integer
Numeric (some can be integer)	Numeric



argument

is an alphabetic, alphanumeric, integer, or numeric argument. If you specify more than one argument, they must all be of the same class. Integer and numeric arguments can be mixed, because integers are of the class numeric.

argument can be an array; for example,

```
FUNCTION MIN (ARRAY1(ALL))
```

returns the smallest element of the array ARRAY1.

The returned value is the *argument* with the least value (according to the rules for evaluating simple conditions). If the function type is alphanumeric, the size of the returned value is the same as that of the *argument* with the least value. If two arguments have the same least value, the value of the leftmost *argument* is returned.

Example 15-25 MIN Function

Code:

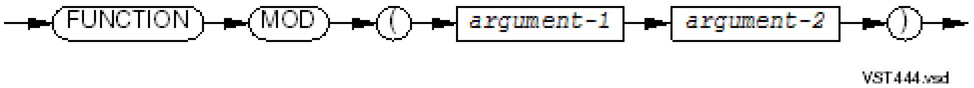
```
DATA DIVISION.
  WORKING-STORAGE SECTION.
    01 NUMERIC-ARRAY.
       05 PICTURE 9(1) VALUE 4.
       05 PICTURE 9(1) VALUE 9.
       05 PICTURE 9(1) VALUE 3.
       05 PICTURE 9(1) VALUE 7.
       05 PICTURE 9(1) VALUE 5
    01 NUM-ARRAY REDEFINES NUMERIC-ARRAY.
       05 NUM OCCURS 5 TIMES PICTURE 9(1).
    01 ALPHABETIC-ARRAY.
       05 PICTURE X(5) VALUE "dog".
       05 PICTURE X(5) VALUE "cat".
       05 PICTURE X(5) VALUE "horse".
       05 PICTURE X(5) VALUE "sheep".
       05 PICTURE X(5) VALUE "goat".
    01 ALPHA-ARRAY REDEFINES ALPHABETIC-ARRAY.
       05 ALPHA OCCURS 5 TIMES PICTURE X(5).
PROCEDURE DIVISION.
  DISPLAY FUNCTION MIN (NUM(ALL))
  DISPLAY FUNCTION MIN (ALPHA(ALL))
  DISPLAY FUNCTION MIN (NUM(1) NUM(4))
  DISPLAY FUNCTION MIN (ALPHA(3) "bird" "fish")
  DISPLAY FUNCTION MIN (3.4 5 6.2 9)
```

Output:

```
3
cat
4
bird
3.4
```

MOD Function

MOD, an integer function, returns the value that is *argument-1* modulo *argument-2*.



argument-1
is an integer.

argument-2
is a nonzero integer.

The returned value is *argument-1* modulo *argument-2*, which is defined:

$$argument\text{-}1 - \left(argument\text{-}2 \times \text{FUNCTION INTEGER}\left(\frac{argument\text{-}1}{argument\text{-}2}\right)\right)$$

Example 15-26 MOD Function

Code:

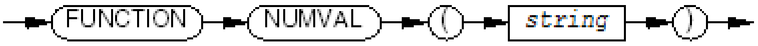
```
DISPLAY FUNCTION MOD (11 5)
DISPLAY FUNCTION MOD (-11 5)
DISPLAY FUNCTION MOD (11 -5)
DISPLAY FUNCTION MOD (-11 -5)
```

Output:

```
00001
00004
-00004
-00001
```

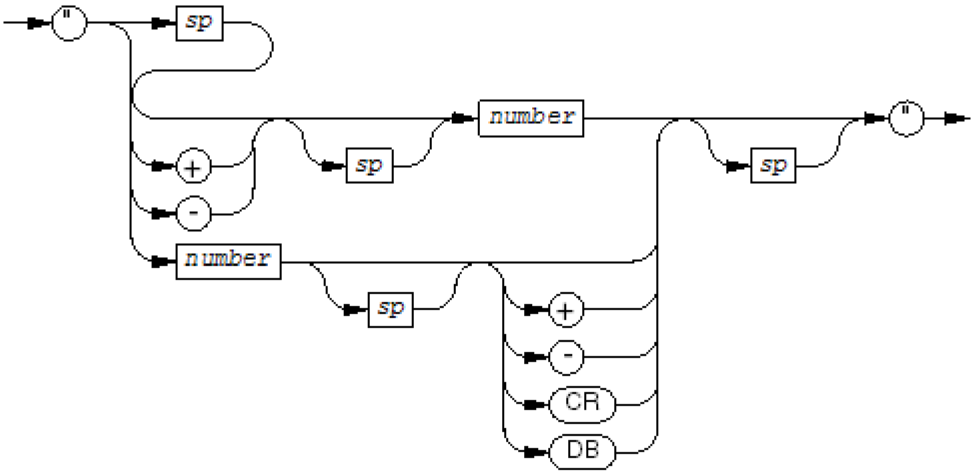
NUMVAL Function

NUMVAL, a numeric function, returns the numeric value represented by its argument, which is a character-string. NUMVAL ignores leading and trailing spaces.



VST445.vsd

`string`

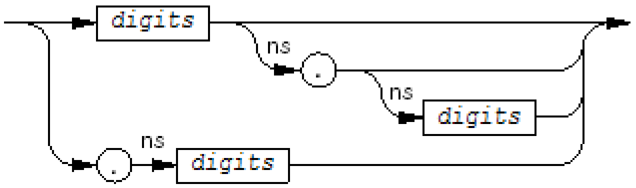


VST446.vsd

`sp`

is a string of zero or more space characters.

`number`



VST447.vsd

If the program contains the DECIMAL POINT COMMA phrase in the SPECIAL-NAMES paragraph, use a comma instead of a decimal point in `number`.

`digits`

is a string of one to 18 digits. The total number of digits in `string` cannot exceed 18.

The returned value is the numeric value represented by *string*. The format of the value depends on the size of *string*. If the length of *string* is fewer than 10 characters, the equivalent PICTURE is `S9(n)V9(n-1)`, where *n* is the number of characters. Any size greater than 9 characters results in an equivalent PICTURE of `S9(19)V9(18)`. This is a special internal item that cannot be expressed by the user. Using it is inefficient, so it is best to avoid using a string that is longer than 9 characters.

If the NUMVAL function is moved to an alphanumeric item, the results might not be what is expected because of the implied PICTURE described previously. For string sizes greater than 10, the result is changed to `S9(18)`.

Example 15-27 NUMVAL Function

Code:

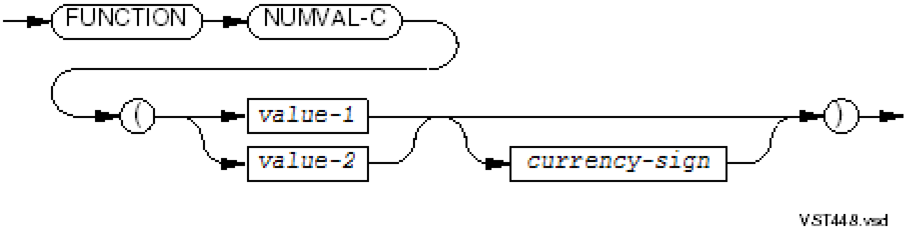
```
DATA DIVISION
  WORKING-STORAGE SECTION.
    01 A PICTURE S99V99.
PROCEDURE DIVISION.
  MOVE FUNCTION NUMVAL ("35") TO A.
  DISPLAY A.
  MOVE FUNCTION NUMVAL ("35.") TO A.
  DISPLAY A.
  MOVE FUNCTION NUMVAL ("35.75") TO A.
  DISPLAY A.
  MOVE FUNCTION NUMVAL (".35") TO A.
  DISPLAY A.
  MOVE FUNCTION NUMVAL ("     - 35.75     ") TO A.
  DISPLAY A.
  MOVE FUNCTION NUMVAL ("  35.75  CR") TO A.
  DISPLAY A.
```

Output:

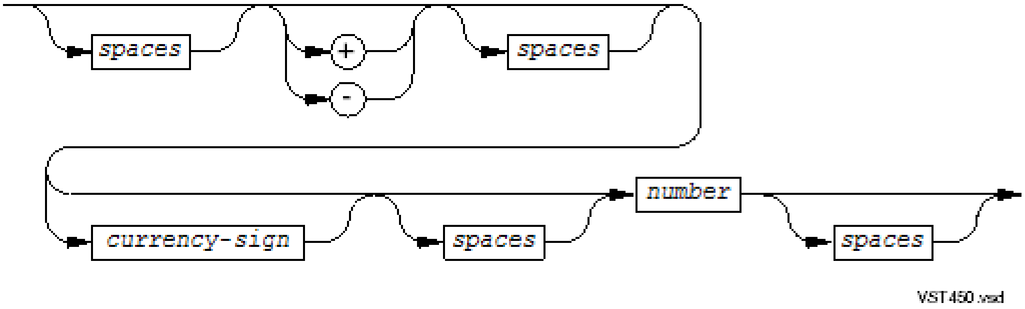
```
35.00
35.00
35.75
00.35
-35.75
-35.75
```

NUMVAL-C Function

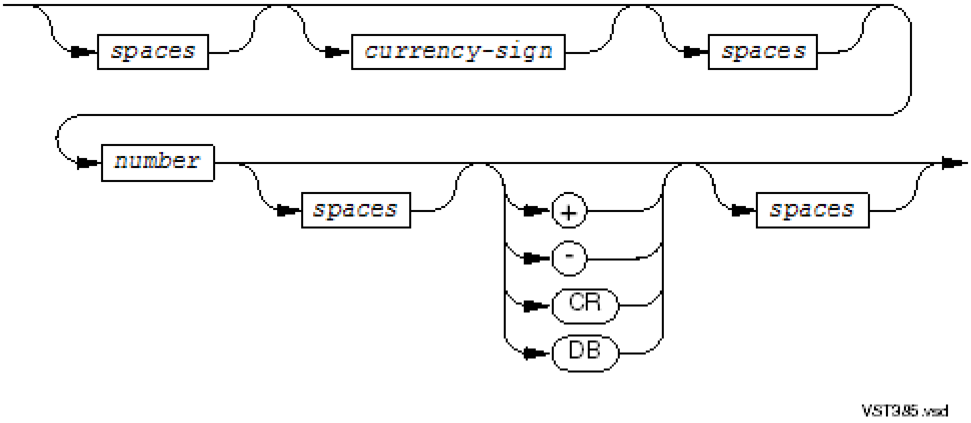
NUMVAL-C, a numeric function, returns the numeric value of a specified character-string, ignoring any currency sign or commas preceding the decimal point.



`value-1`



`value-2`



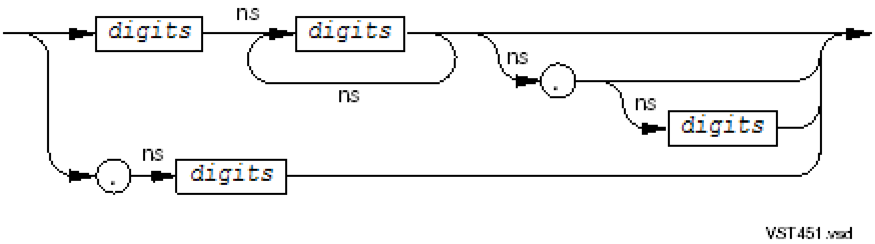
`spaces`

is a string of zero or more space characters.

`currency-sign`

is a nonnumeric literal or alphanumeric data item; a string of one or two characters that specifies the currency sign. The default is the currency sign specified for the program.

`number`



If the program contains the DECIMAL POINT COMMA phrase in the SPECIAL-NAMES paragraph, use a comma instead of a decimal point in *number*.

digits

is a string of one to 18 digits. The total number of digits in *string* cannot exceed 18.

The returned value is the numeric value represented by *string*. The format of the value depends on the size of *string*. If the length of *string* is fewer than 10 characters, the equivalent PICTURE is `S9(`*n*`)V9(`*n*`-1)`, where *n* is the number of characters. Any size greater than 9 characters results in an equivalent PICTURE of `S9(19)V9(18)`. This is a special internal item that cannot be expressed by the user. Using it is inefficient, so it is best to avoid using a string that is longer than 9 characters.

If the NUMVAL-C function is moved to an alphanumeric item, the results might not be what is expected because of the implied PICTURE described previously.

Example 15-28 NUMVAL-C Function

Code:

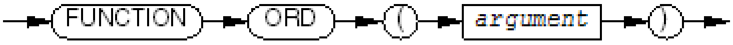
```
DATA DIVISION
  WORKING-STORAGE SECTION.
    01 A PICTURE S99V99.
PROCEDURE DIVISION.
  MOVE FUNCTION NUMVAL-C ("35") TO A.
  DISPLAY A.
  MOVE FUNCTION NUMVAL-C ("$35") TO A.
  DISPLAY A.
  MOVE FUNCTION NUMVAL-C ("35$") TO A.
  DISPLAY A.
  MOVE FUNCTION NUMVAL-C ("35.75") TO A.
  DISPLAY A.
  MOVE FUNCTION NUMVAL-C (".35") TO A.
  DISPLAY A.
  MOVE FUNCTION NUMVAL-C ("     - 35.75     ") TO A.
  DISPLAY A.
  MOVE FUNCTION NUMVAL-C ("  35.75  CR") TO A.
  DISPLAY A.
```

Output:

```
35.00
35.00
35.00
35.75
00.35
-35.75
-35.75
```

ORD Function

ORD, an integer function, returns the ordinal number of its argument in the program collating sequence.



VST452.vsd

argument

is a one-character alphabetic or alphanumeric argument.

The returned value is the ordinal number of *argument* in the program collating sequence. It is at least one.

Example 15-29 ORD Function

Code:

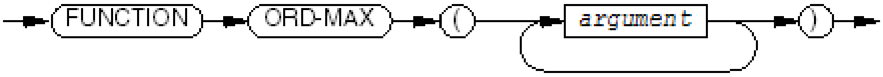
```
DISPLAY FUNCTION ORD ("A")
DISPLAY FUNCTION ORD ("b")
DISPLAY FUNCTION ORD ("3")
DISPLAY FUNCTION ORD ("%")
```

Output:

```
066
099
052
038
```

ORD-MAX Function

ORD-MAX, an integer function, returns the ordinal number of its maximum argument.



VST435.vsd

argument

is an alphabetic, alphanumeric, integer, or numeric argument. If you specify more than one argument, they must all be of the same class. Integer and numeric arguments can be mixed, because integers are of the class numeric.

argument can be an array; for example,

```
FUNCTION ORD-MAX (ARRAY1(ALL))
```

returns the ordinal number of the largest element of the array ARRAY1.

The returned value is the ordinal number of the *argument* with the greatest value (according to the rules for evaluating simple conditions). If more than one *argument* has the same greatest value, the returned value is the ordinal number of the leftmost *argument* with that value. The returned value is at least one.

Example 15-30 ORD-MAX Function

Code:

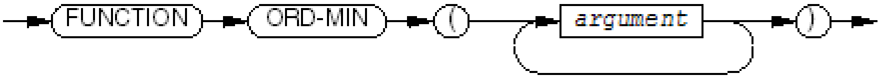
```
DATA DIVISION.
  WORKING-STORAGE SECTION.
    01 NUMERIC-ARRAY.
       05 PICTURE 9(1) VALUE 4.
       05 PICTURE 9(1) VALUE 9.
       05 PICTURE 9(1) VALUE 3.
       05 PICTURE 9(1) VALUE 7.
       05 PICTURE 9(1) VALUE 5
    01 NUM-ARRAY REDEFINES NUMERIC-ARRAY.
       05 NUM OCCURS 5 TIMES PICTURE 9(1).
    01 ALPHABETIC-ARRAY.
       05 PICTURE X(5) VALUE "dog".
       05 PICTURE X(5) VALUE "cat".
       05 PICTURE X(5) VALUE "horse".
       05 PICTURE X(5) VALUE "sheep".
       05 PICTURE X(5) VALUE "goat".
    01 ALPHA-ARRAY REDEFINES ALPHABETIC-ARRAY.
       05 ALPHA OCCURS 5 TIMES PICTURE X(5).
PROCEDURE DIVISION.
  DISPLAY FUNCTION ORD-MAX (NUM(ALL))
  DISPLAY FUNCTION ORD-MAX (ALPHA(ALL))
  DISPLAY FUNCTION ORD-MAX (NUM(1) NUM(4))
  DISPLAY FUNCTION ORD-MAX (ALPHA(3) "bird" "fish")
  DISPLAY FUNCTION ORD-MAX (3.4 5 6.2 9)
```

Output:

```
0000000002
0000000004
0000000002
0000000001
0000000004
```

ORD-MIN Function

ORD-MIN, an integer function, returns the ordinal number of its minimum argument.



VST454.vsd

argument

is an alphabetic, alphanumeric, integer, or numeric argument. If you specify more than one argument, they must all be of the same class. Integer and numeric arguments can be mixed, because integers are of the class numeric.

argument can be an array; for example,

```
FUNCTION ORD-MIN (ARRAY1(ALL))
```

returns the ordinal number of the smallest element of the array ARRAY1.

The returned value is the ordinal number of the *argument* with the least value (according to the rules for evaluating simple conditions). If more than one *argument* has the same least value, the returned value is the ordinal number of the leftmost *argument* with that value. The returned value is at least one.

Example 15-31 ORD-MIN Function

Code:

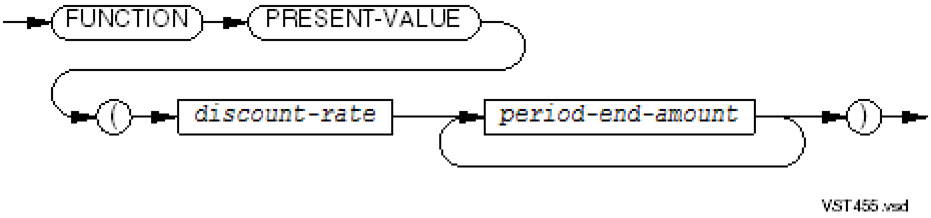
```
DATA DIVISION.
  WORKING-STORAGE SECTION.
    01 NUMERIC-ARRAY.
       05 PICTURE 9(1) VALUE 4.
       05 PICTURE 9(1) VALUE 9.
       05 PICTURE 9(1) VALUE 3.
       05 PICTURE 9(1) VALUE 7.
       05 PICTURE 9(1) VALUE 5
    01 NUM-ARRAY REDEFINES NUMERIC-ARRAY.
       05 NUM OCCURS 5 TIMES PICTURE 9(1).
    01 ALPHABETIC-ARRAY.
       05 PICTURE X(5) VALUE "dog".
       05 PICTURE X(5) VALUE "cat".
       05 PICTURE X(5) VALUE "horse".
       05 PICTURE X(5) VALUE "sheep".
       05 PICTURE X(5) VALUE "goat".
    01 ALPHA-ARRAY REDEFINES ALPHABETIC-ARRAY.
       05 ALPHA OCCURS 5 TIMES PICTURE X(5).
PROCEDURE DIVISION.
  DISPLAY FUNCTION ORD-MIN (NUM(ALL))
  DISPLAY FUNCTION ORD-MIN (ALPHA(ALL))
  DISPLAY FUNCTION ORD-MIN (NUM(1) NUM(4))
  DISPLAY FUNCTION ORD-MIN (ALPHA(3) "bird" "fish")
  DISPLAY FUNCTION ORD-MIN (3.4 5 6.2 9)
```

Output:

```
0000000003
0000000002
0000000001
0000000002
0000000001
```

PRESENT-VALUE Function

PRESENT-VALUE, a numeric function, returns a value that approximates the present value of a specified series of future period-end amounts at a specified discount rate.



discount-rate

is a numeric argument whose value is greater than -1.

period-end-amount

is a numeric argument.

period-end-amount can be an array; for example,

```
FUNCTION PRESENT-VALUE (DISCOUNT-RATE ARRAY1(ALL))
```

uses the elements of the array ARRAY1 as a series of numeric arguments.

The returned value is approximately

$$\sum_{i=1}^{n} \frac{\mathit{period\text{-}end\text{-}amount}_i}{(1 + \mathit{discount\text{-}rate})^i}$$

where n is the number of *period-end-amount* s.

Example 15-32 PRESENT-VALUE Function

Code:

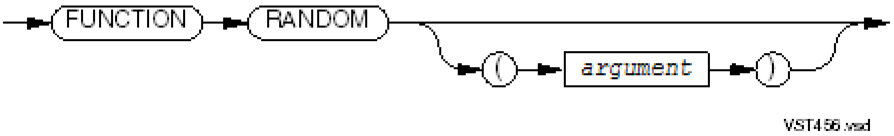
```
DATA DIVISION.
  WORKING-STORAGE SECTION.
    01 NUMERIC-ARRAY.
       05 PICTURE 9V99 VALUE 0.05.
       05 PICTURE 9V99 VALUE 1.15.
       05 PICTURE 9V99 VALUE 2.50.
       05 PICTURE 9V99 VALUE 3.75.
       05 PICTURE 9V99 VALUE 4.55.
    01 NUM-ARRAY REDEFINES NUMERIC-ARRAY.
       05 N OCCURS 5 TIMES PICTURE 9V99.
    01 A PICTURE S99V99.
PROCEDURE DIVISION.
  MOVE FUNCTION PRESENT-VALUE (0 NUM(ALL) TO A) TO A
  DISPLAY A.
  MOVE FUNCTION PRESENT-VALUE (1 NUM(ALL) TO A) TO A
  DISPLAY A.
  MOVE FUNCTION PRESENT-VALUE (2.5 NUM(ALL) TO A) TO A
  DISPLAY A.
  MOVE FUNCTION PRESENT-VALUE (0 3) TO A
  DISPLAY A.
  MOVE FUNCTION PRESENT-VALUE (1.5 3.6 7.8 4.9) TO A
  DISPLAY A.
```

Output:

```
12.00
01.00
00.20
03.00
03.00
```

RANDOM Function

RANDOM, a numeric function, returns a pseudorandom number from a rectangular distribution.



argument

is zero or a positive integer.

The first time a run unit calls RANDOM, RANDOM generates a series of pseudorandom numbers and returns the first number in the series. If *argument* is specified, RANDOM uses the value of *argument* as the seed; if not, it uses the default seed, one.

Each subsequent time that the run unit calls RANDOM without specifying *argument*, RANDOM returns the next pseudorandom number in the series. If the run unit calls RANDOM again and specifies *argument*, RANDOM generates a new series of pseudorandom numbers and returns the first number in the new series.

The same value of *argument* always generates the same sequence of pseudorandom numbers. The returned value is always greater than or equal to 0 and less than 1.

Example 15-33 RANDOM Function

Code:

```
DISPLAY FUNCTION RANDOM (0)
DISPLAY FUNCTION RANDOM (15)
DISPLAY FUNCTION RANDOM (253)
DISPLAY FUNCTION RANDOM (4067)
DISPLAY FUNCTION RANDOM (32767)
```

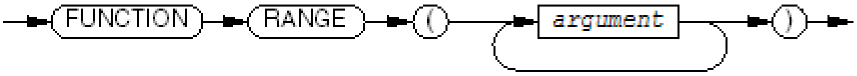
Output:

```
0.000000000
0.707956564
0.007660018
0.886242720
0.792491424
```

RANGE Function

RANGE is a function that returns the difference between its maximum and minimum arguments. Its type depends on the type of its arguments, as this table shows:

Argument Type	Function Type
Integer	Integer
Numeric (some can be integer)	Numeric



VST457.vsd

argument

is an integer or numeric argument. Integer and numeric arguments can be mixed, because integers are of the class numeric.

argument can be an array; for example,

```
FUNCTION RANGE (ARRAY1(ALL))
```

returns the difference between the largest and smallest elements of the array ARRAY1.

The returned value is the value of the greatest *argument* minus the value of the least *argument* (where the greatest and least arguments are determined according to the rules for evaluating simple conditions). If two arguments have the same greatest value, the leftmost of the two is considered the greatest. If two arguments have the same least value, the leftmost of the two is considered the least.

Example 15-34 RANGE Function

Code:

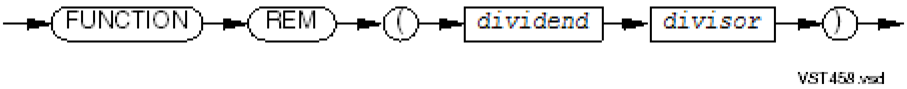
```
DATA DIVISION.
  WORKING-STORAGE SECTION.
    01 NUMERIC-ARRAY.
       05 PICTURE 9V99 VALUE 0.05.
       05 PICTURE 9V99 VALUE 1.15.
       05 PICTURE 9V99 VALUE 2.50.
       05 PICTURE 9V99 VALUE 3.75.
       05 PICTURE 9V99 VALUE 4.55.
    01 NUM-ARRAY REDEFINES NUMERIC-ARRAY.
       05 NUM OCCURS 5 TIMES PICTURE 9V99.
PROCEDURE DIVISION.
  DISPLAY FUNCTION RANGE (NUM(ALL))
  DISPLAY FUNCTION RANGE (9 2 5 3.5)
  DISPLAY FUNCTION RANGE (1.5 3.6 7.8 4.9)
```

Output:

```
4.50
7.0
6.3
```

REM Function

REM, a numeric function, returns the remainder that results from dividing its first argument by its second argument.



dividend

is a numeric argument.

divisor

is a nonzero numeric argument.

If ARRAY1 has only two elements, the first of which satisfies the requirements for *dividend* and the second of which satisfies the requirements of *divisor*, then

```
FUNCTION REM (ARRAY1(ALL))
```

returns the remainder that results from dividing ARRAY1(1) by ARRAY1(2).

The returned value is the remainder of *dividend* divided by *divisor*, which is defined:

```
dividend - (divisor * FUNCTION INTEGER-PART (dividend / divisor))
```

Example 15-35 REM Function

Code:

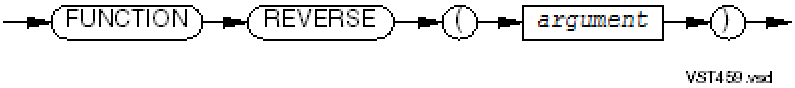
```
DISPLAY FUNCTION REM (11 5)
DISPLAY FUNCTION REM (-11 5)
DISPLAY FUNCTION REM (11 -5)
DISPLAY FUNCTION REM (-11 -5)
```

Output:

```
01
-01
01
-01
```

REVERSE Function

REVERSE, an alphanumeric function, returns a string that is the same as its argument, except that the characters are in reverse order.



argument

is an alphabetic or alphanumeric argument at least one character in length.

The returned value is the same as *argument*, except that the characters are in reverse order. That is, if *argument* has *n* characters, then the *i* th character of *argument* is the (*n* - *i* +1)th character of the returned value.

Example 15-36 REVERSE Function

Code:

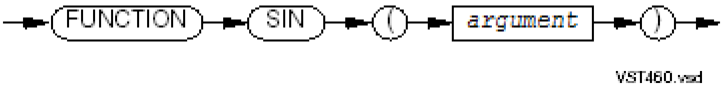
```
DISPLAY FUNCTION REVERSE ("A")
DISPLAY FUNCTION REVERSE ("1234567")
DISPLAY FUNCTION REVERSE ("stressed")
DISPLAY FUNCTION REVERSE ("ABC 123")
```

Output:

```
A
7654321
desserts
321 CBA
```

SIN Function

SIN, a numeric function, returns a value that approximates the sine of the angle or arc (in radians) specified by its argument.



argument

is a numeric argument.

The returned value approximates the sine of the value of *argument*. It is a floating-point number in the range -1 to +1. It has no implied decimal point and is precise to approximately 14 digits.

Example 15-37 SIN Function

Code:

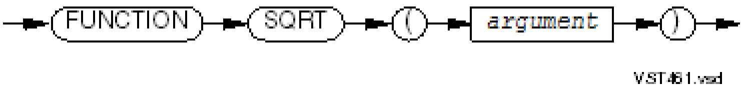
```
DATA DIVISION.
  WORKING-STORAGE SECTION.
    01 A PICTURE S9V99.
PROCEDURE DIVISION.
  MOVE FUNCTION SIN (-10)  TO A.
  DISPLAY A.
  MOVE FUNCTION SIN (0)    TO A.
  DISPLAY A.
  MOVE FUNCTION SIN (.25)  TO A.
  DISPLAY A.
  MOVE FUNCTION SIN (15.5) TO A.
  DISPLAY A.
  MOVE FUNCTION SIN (45)   TO A.
  DISPLAY A.
```

Output:

```
0.54
0.00
0.24
0.20
0.85
```

SQRT Function

SQRT, a numeric function, returns a value that approximates the square root of its argument.



argument

is a numeric argument whose value is zero or positive.

The returned value is approximately:

$$\left|\sqrt{argument}\right|$$

Example 15-38 SQRT Function

Code:

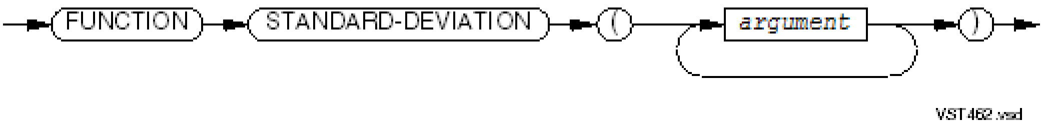
```
DATA DIVISION.
  WORKING-STORAGE SECTION.
    01 A PICTURE S9V99.
PROCEDURE DIVISION.
  MOVE FUNCTION SQRT (0)     TO A.
  DISPLAY A.
  MOVE FUNCTION SQRT (25)    TO A.
  DISPLAY A.
  MOVE FUNCTION SQRT (100)   TO A.
  DISPLAY A.
  MOVE FUNCTION SQRT (47)    TO A.
  DISPLAY A.
  MOVE FUNCTION SQRT (7.923) TO A.
  DISPLAY A.
```

Output:

```
00.000
05.000
10.000
06.855
02.814
```

STANDARD-DEVIATION Function

STANDARD-DEVIATION, a numeric function, returns a value that approximates the standard deviation of its arguments.



argument

is a numeric argument.

argument can be an array; for example,

```
FUNCTION STANDARD-DEVIATION (ARRAY1(ALL))
```

returns a value that approximates the standard deviation of the elements of the array ARRAY1.

The returned value is the approximation of the standard deviation of the *argument* series.

If there is only one *argument*, or if every *argument* is a variable occurrence data item and the total number of occurrences for all of them is one, the returned value is zero; otherwise, the returned value is approximately

$$\left| \sqrt{\frac{\sum_{i=1}^{n} (argument_i - \text{FUNCTION MEAN } (argument_1,.., argument_n))^2}{n}} \right|$$

where n is the number of *arguments*.

Example 15-39 STANDARD-DEVIATION Function

Code:

```
DATA DIVISION.
  WORKING-STORAGE SECTION.
    01 NUMERIC-ARRAY
       05 PICTURE 9V99 VALUE 0.05.
       05 PICTURE 9V99 VALUE 1.15.
       05 PICTURE 9V99 VALUE 2.50.
       05 PICTURE 9V99 VALUE 3.75.
       05 PICTURE 9V99 VALUE 4.55.
    01 NUM-ARRAY REDEFINES NUMERIC-ARRAY.
       05 NUM OCCURS 5 TIMES PICTURE 9V99.
    01 A PICTURE S9V99.
PROCEDURE DIVISION.
  MOVE FUNCTION STANDARD-DEVIATION (NUM(ALL)) TO A.
  DISPLAY A.
  MOVE FUNCTION STANDARD-DEVIATION (9 2 5 3.5) TO A.
  DISPLAY A.
  MOVE FUNCTION STANDARD-DEVIATION (1.5 3.6 7.8 4.9) TO A.
  DISPLAY A.
```

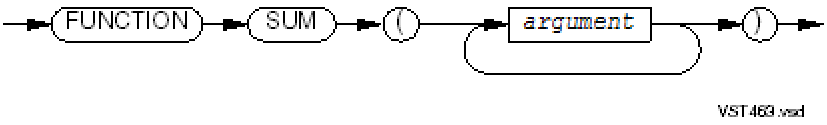
Output:

```
1.64
2.60
2.28
```

SUM Function

SUM is a function that returns the sum of its arguments. Its type depends on the type of its arguments, as this table shows:

Argument Type	Function Type
Integer	Integer
Numeric (some can be integer)	Numeric



argument

is an integer or numeric argument.

argument can be an array; for example,

```
FUNCTION SUM (ARRAY1(ALL))
```

returns the sum of the elements of the array ARRAY1.

The returned value is

$$\sum_{i=1}^{n} \mathit{argument}_i$$

where *n* is the number of arguments.

Example 15-40 SUM Function

Code:

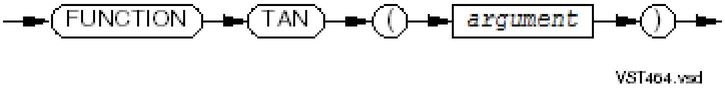
```
DATA DIVISION.
  WORKING-STORAGE SECTION.
    01 NUMERIC-ARRAY
       05 PICTURE 9V99 VALUE 0.05.
       05 PICTURE 9V99 VALUE 1.15.
       05 PICTURE 9V99 VALUE 2.50.
       05 PICTURE 9V99 VALUE 3.75.
       05 PICTURE 9V99 VALUE 4.55.
    01 NUM-ARRAY REDEFINES NUMERIC-ARRAY.
       05 NUM OCCURS 5 TIMES PICTURE 9V99.
PROCEDURE DIVISION.
  DISPLAY FUNCTION SUM (NUM(ALL))
  DISPLAY FUNCTION SUM (9 2 5 3)
  DISPLAY FUNCTION SUM (1.5 3.6 7.8 4.9)
```

Output:

```
012.00
19
17.8
```

TAN Function

TAN, a numeric function, returns a value that approximates the tangent of the angle or arc (in radians) specified by its argument.



argument

is a numeric argument.

The returned value approximates the tangent of the value of *argument*. It is a floating-point number with no implied decimal point, precise to approximately 14 digits.

Example 15-41 TAN Function

Code:

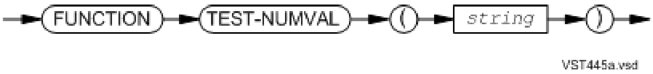
```
DATA DIVISION.
  WORKING-STORAGE SECTION.
    01 A PICTURE S9V99.
PROCEDURE DIVISION.
  MOVE FUNCTION ATAN (-10)   TO A.
  DISPLAY A.
  MOVE FUNCTION ATAN (0)     TO A.
  DISPLAY A.
  MOVE FUNCTION ATAN (.25)   TO A.
  DISPLAY A.
  MOVE FUNCTION ATAN (15.5)  TO A.
  DISPLAY A.
  MOVE FUNCTION ATAN (45)    TO A.
  DISPLAY A.
```

Output:

```
-0.64
0.00
0.25
-0.21
1.61
```

TEST-NUMVAL Function

TEST-NUMVAL, an integer function, returns a value indicating whether the specified character-string is valid input to the NUMVAL Function (page 687).



string

is a nonnumeric literal or an alphanumeric data item.

The returned value is zero if *string* conforms to the format described for the argument of the NUMVAL Function (page 687), or gives the character position of the first invalid character. If *string* is valid but incomplete (for example, it contains no digits), the returned value is the length of the string plus one (indicating the character position immediately following *string*).

Example 15-42 TEST-NUMVAL Function

Code:

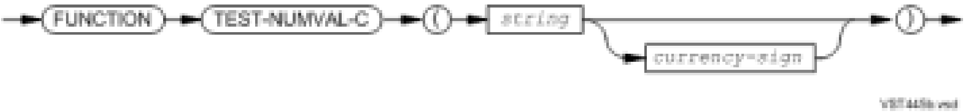
```
IDENTIFICATION DIVISION.
    PROGRAM-ID. TEST-NUMVAL.
 DATA DIVISION.
 WORKING-STORAGE SECTION.
  01 WS-DATA.
     03 A PIC S9(4).

 PROCEDURE DIVISION.
 MAIN-LOGIC SECTION.
    MOVE FUNCTION TEST-NUMVAL("35") to A.
    DISPLAY A.
    MOVE FUNCTION TEST-NUMVAL("35.") to A.
    DISPLAY A.
    MOVE FUNCTION TEST-NUMVAL("35.75") to A.
    DISPLAY A.
    MOVE FUNCTION TEST-NUMVAL(".35") to A.
    DISPLAY A.
    MOVE FUNCTION TEST-NUMVAL("=35.75") to A.
    DISPLAY A.
    MOVE FUNCTION TEST-NUMVAL("35.75 CR") to A.
    DISPLAY A.

    STOP RUN.
```

Output:

```
0000
0000
0000
0000
0001
0000
```

TEST-NUMVAL-C Function

TEST-NUMVAL-C, an integer function, returns a value indicating whether the specified character-string is valid input to the NUMVAL-C Function (page 688).



`string`

is a nonnumeric literal or an alphanumeric data item.

`currency-sign`

the optional `currency-sign` is a nonnumeric literal or alphanumeric data item; a string of one or two characters that specifies the currency sign. The default is the currency sign specified for the program. For more information, refer to the NUMVAL-C Function (page 688).

The returned value is zero if `string` conforms to the format described for the argument of the NUMVAL-C Function (page 688), or gives the character position of the first invalid character. If `string` is valid but incomplete (for example, it contains no digits), the returned value is the length of the string plus one (indicating the character position immediately following `string`).

Example 15-43 TEST-NUMVAL-C Function

Code:

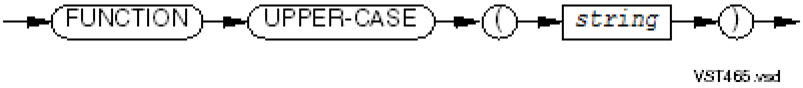
```
IDENTIFICATION DIVISION.
    PROGRAM-ID. TEST-NUMVAL-C.
 DATA DIVISION.
 WORKING-STORAGE SECTION.
  01 WS-DATA.
     03 A PIC S9(4).

 PROCEDURE DIVISION.
 MAIN-LOGIC SECTION.
    MOVE FUNCTION TEST-NUMVAL-c("35") to A.
    DISPLAY A.
    MOVE FUNCTION TEST-NUMVAL-c("$35") to A.
    DISPLAY A.
    MOVE FUNCTION TEST-NUMVAL-c("35$") to A.
    DISPLAY A.
    MOVE FUNCTION TEST-NUMVAL-c("35.75") to A.
    DISPLAY A.
    MOVE FUNCTION TEST-NUMVAL-c(".35") to A.
    DISPLAY A.
    MOVE FUNCTION TEST-NUMVAL-c("=35.75") to A.
    DISPLAY A.
    MOVE FUNCTION TEST-NUMVAL-c("35.75 CR") to A.
    DISPLAY A.

    STOP RUN.
```

Output:

```
0000
0000
0000
0000
0000
0001
0000
```

UPPER-CASE Function

UPPER-CASE, an alphanumeric function, returns a character-string that is the same as its argument, except that each lowercase letter is replaced by the corresponding uppercase letter.



string

is an alphabetic or alphanumeric string at least one character in length.

The returned value is the same as the value of *string*, except that each lowercase letter is replaced by the corresponding uppercase letter. If the value of *string* contains no lowercase letters, or if the ALPHABET clause specifies a collating sequence that contains no uppercase letters, the returned value is the same as the value of *string*.

Example 15-44 UPPER-CASE Function

Code:

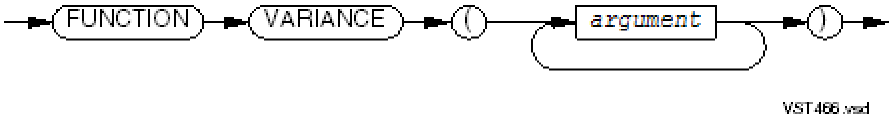
```
DISPLAY FUNCTION UPPER-CASE ("HEWLETT-PACKARD COMPANY 2003")
DISPLAY FUNCTION UPPER-CASE ("Hewlett-packard Company 2003")
DISPLAY FUNCTION UPPER-CASE ("hewlett-packard company 2003")
```

Output:

```
HEWLETT-PACKARD COMPANY 2003
HEWLETT-PACKARD COMPANY 2003
HEWLETT-PACKARD COMPANY 2003
```

VARIANCE Function

VARIANCE, a numeric function, returns a value that approximates the variance of its arguments.



argument

is a numeric argument.

argument can be an array; for example,

```
FUNCTION VARIANCE (ARRAY1(ALL))
```

returns a value that approximates the variance of the elements of the array ARRAY1.

The returned value is the approximation of the variance of the *argument* series.

If there is only one *argument*, or if every *argument* is a variable occurrence data item and the total number of occurrences for all of them is one, the returned value is zero; otherwise, the returned value is the square of the standard deviation of the *argument* series; that is:

$$\left| \sqrt{\frac{\sum_{i=1}^{n} \left(\mathit{argument}_i - \mathtt{FUNCTION\ MEAN}\ (\mathit{argument}_1, \ldots, \mathit{argument}_n)\right)^2}{n}} \right|^2$$

Example 15-45 VARIANCE Function

Code:

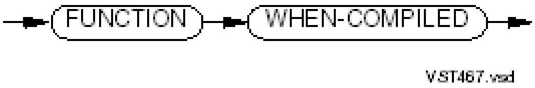
```
DATA DIVISION.
  WORKING-STORAGE SECTION.
    01 NUMERIC-ARRAY
       05 PICTURE 9V99 VALUE 0.05.
       05 PICTURE 9V99 VALUE 1.15.
       05 PICTURE 9V99 VALUE 2.50.
       05 PICTURE 9V99 VALUE 3.75.
       05 PICTURE 9V99 VALUE 4.55.
    01 NUM-ARRAY REDEFINES NUMERIC-ARRAY.
       05 NUM OCCURS 5 TIMES PICTURE 9V99.
    01 A PICTURE 9V99.
PROCEDURE DIVISION.
  MOVE FUNCTION VARIANCE (NUM(ALL)) TO A
  DISPLAY A..
  MOVE FUNCTION VARIANCE (9 2 5 3) TO A.
  DISPLAY A.
  MOVE FUNCTION VARIANCE (1.5 3.6 7.8 4.9) TO A.
  DISPLAY A.
  MOVE FUNCTION VARIANCE (5) TO A.
  DISPLAY A.
```

Output:

```
2.70
7.18
5.21
0.00
```

WHEN-COMPILED Function

WHEN-COMPILED, an alphanumeric function, returns a 21-character string that represents the date and time the program was compiled, according to the system on which the program was compiled.



The returned value is the date and time of compilation of the source program that calls the WHEN-COMPILED function. If the program is a contained program, the returned value is the date and time of the separately compiled program that contains it.

The returned value represents the current date this way (where character position 1 is the leftmost character position):

Character Positions	Date Part Represented	Represented by
1-4	Year in Gregorian calendar	Four digits
5-6	Month of the year	Two digits in the range 01 through 12
7-8	Day of the month	Two digits in the range 01 through 31
9-10	Hours past midnight	Two digits in the range 00 through 23
11-12	Minutes past the hour	Two digits in the range 00 through 59
13-14	Seconds past the minute	Two digits in the range 00 through 59
15-16	Hundredths of seconds past the second	Two digits in the range 00 through 99

Character Positions	Date Part Represented	Represented by
17	Relationship to Greenwich mean time	Minus (-) if the reported time is behind Greenwich mean time, plus (+) if it is the same as or ahead of Greenwich mean time
18-19	Hours behind or ahead of Greenwich mean time	If character position 17 is minus (-), two digits in the range 00 through 12; if character position 17 is plus (+), two digits in the range 00 through 13
20-21	Additional minutes from Greenwich mean time	Two digits in the range 00 through 59

The returned value matches the compilation time and date on the listing and object code files (if they have it), but the representation and precision may differ.

Example 15-46 WHEN-COMPILED Function

Code:

```
DISPLAY FUNCTION WHEN-COMPILED
```

Output:

```
1997041516104953-0700
```

The date in Example 15-46 is April 15, 1997 (19970415). The time is 16:10:49.53 or 4:10:49.53 PM (16104953), which is 7 hours behind Greenwich mean time (-0700).

16 Debugging Tools

This section briefly describes the HP debugging tools and refers you to appropriate sources for more information.

CAUTION: If you use the CODECOV (page 551) compiler directive to direct your compiler to generate instrumented object code, the Code Coverage Utility connects to the application program as a debugger to read its memory and so on.

This causes all the debug requests to wait until the Code Coverage Utility (the active debugger) detaches from the application. The Code Coverage Utility only detaches after the instrumented application stops.

Therefore, if you must debug an instrumented application, it should be started in debug mode by using the TACL RUND or RUNV commands.

For more information, see Generating Instrumented Object Code for Use With the Code Coverage Tool (page 534).

Debugging Lines

Debugging lines, a batch-oriented COBOL debugging tool, apply to individual source program lines.

Debugging lines are compiled only if the source program includes a DEBUGGING MODE clause (see SOURCE-COMPUTER Paragraph (page 114)); otherwise, they are handled as comments.

NOTE: The code that the compiler generates when it compiles debugging lines and debugging declaratives is not the same as the code that it generates when it ignores them. The additional code that the compiler generates for the debugging lines and debugging declaratives can alter the alignment of data items and cause the program to behave differently.

You can put debugging lines anywhere in a source program. You usually use them to report values of data items at strategic points in the program.

A debugging line has the letter D or d in the indicator area (column 1 in Tandem reference format, column 7 in ANSI reference format). If a debugging line contains a COPY statement, every line that the COPY statement introduces inherits the D or d in its indicator column.

Run-Time Debuggers

The HP run-time debuggers are Native Inspect and Visual Inspect. Native Inspect is a command line debugger that runs on the NonStop server. Visual Inspect runs on the PC and provides a graphical user interface (GUI). Both Native Inspect and Visual Inspect provide an interactive, symbolic debugging capability.

Native Inspect replaces Inspect and Debug in the TNS/E programming environment. It provides most of the same capabilities, including maching-level debugging, as Inspect and Debug, although the Native Inspect command syntax differs from Inspect and Debug. Native Inspect is based on the industry-standard GDB debugger.

If you compile your program with the SYMBOLS directive (described in SYMBOLS and NOSYMBOLS (page 586)), the symbolic debuggers allow you to debug the program using the names you assigned to the procedures and data items of the source program, rather than the addresses that the compiler and linker ultimately assigned to them. If you do not compile your program with the SYMBOLS directive, you can use only the low-level commands of a symbolic debugger.

NOTE: When you have debugged the program, recompile it without the SYMBOLS directive to decrease the size of the object file and decrease program compilation time.

Any mechanism that calls the debugging facility calls the selected run-time debugger. Examples of such mechanisms are:

- The RUND command
- An explicit call to the Debug debugger by the process
- A command interpreter DEBUG command

For more information, see:

- Debugging (page 987)
- Visual Inspect's online help
- *Native Inspect Manual*

Debugger Selection

The INSPECT attribute setting determines which debugger is selected for a process:

- If INSPECT is ON, Visual Inspect is selected, provided that a client connection exists.
- If INSPECT is ON and a Visual Inspect client connection does not exist, Native Inspect is selected.
- If INSPECT is OFF, Native Inspect is selected.

You set the INSPECT attribute in any of these ways:

- Compile the program without a NOINSPECT directive (described in INSPECT and NOINSPECT (page 564)).
- Compile the program with a SAVEABEND directive (described in SAVEABEND and NOSAVEABEND (page 573)).
- Set the INSPECT attribute in the linker.
- Use the PARAM INSPECT ON command in the command interpreter environment before running the process
- Specify INSPECT ON as a run option in the RUN command that initiates the process.

Visual Inspect is available only if you established a connection to it before the process entered debugging mode.

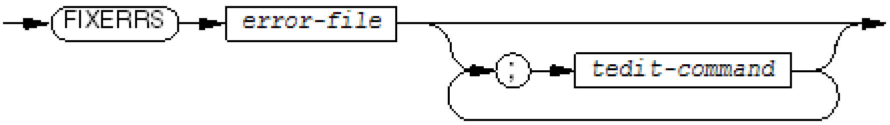
FIXERRS Macro

FIXERRS is a TACL macro that helps you find and fix syntax errors in your source file.

To use the FIXERRS TACL macro to fix errors in a source file, you must compile your program with the directive ERRORFILE (page 558). You can then use FIXERRS with the error-logging file produced by that directive.

The FIXERRS TACL macro starts a two-window TEDIT session and displays an error message on the top line of the screen with the corresponding source text in the remaining lines. When you start FIXERRS, the cursor is positioned at the first error in the source text.

To start FIXERRS enter:



VST372.vsd

`error-file`

is the name of the error file produced by the compiler. This file has a file code of 106.

`tedit-command`

is the TEDIT command that you want to execute at the start of the FIXERRS session.

You can move from error to error using the NEXTERR and PREVERR commands on the TEDIT command line. NEXTERR displays the error following the currently displayed error; PREVERR displays the error preceding the currently displayed error.

When you enter PREVERR at the beginning of the error file or NEXTERR at the end of the error file, FIXERRS displays the message:

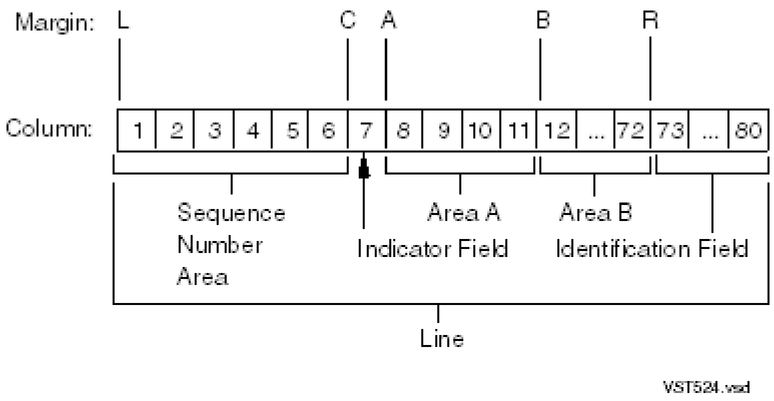
```
There are no more errors
```

When you enter PREVERR after you have corrected errors in the file, the compiler positions you at the line and column on which the error was originally reported. This line and column number might be invalid after you correct the error.

17 ANSI Reference Format

In ANSI reference format, each line has 80 characters (columns). Five margins divide each line into five areas.

Figure 17-1 ANSI Reference Format



To ensure that each line has 80 characters, the COBOL compiler truncates lines that are too long and space-pads lines that are too short. The topics of this section explain the five areas that Figure 17-1 shows, which are:

- Sequence Number Area
- Indicator Area
- Area A
- Record Description Entries
- Identification Field

Sequence Number Area

The Sequence Number Area begins at Margin L and uses columns 1 through 6. It can be empty, or it can contain a line number (up to six digits) or a line label (a combination of letters and numbers).

No sequence number area exists for compiler directive lines.

Indicator Area

The indicator area begins at Margin C and uses only column 7. It can be empty, or it can contain a single character that describes the type of information on the line.

Table 17-1 Valid Indicator Area Characters (ANSI Reference Format)

Valid Character	Character Name	Indicates that the line is a ...
?	Question mark	Compiler Directive
*	Asterisk	Ordinary Comment
/	Slash	Comment for Top of Page
D	Uppercase *D*	Debugging Line
d	Lowercase *d*	Debugging Line
-	Hyphen	Continuation Line

Compiler Directive

A compiler directive has a question mark (?) in the indicator area. If the question mark is in column 1 instead of column 7, the compiler treats the line as if it begins with the indicator area.

A compiler directive is an instruction to the COBOL compiler, and a compiler directive line has no sequence number area. (For more information on compiler directives, see Compiler Directives (page 542).)

Ordinary Comment

An ordinary comment has an asterisk (*) in the indicator area. A comment can appear anywhere in a program. The compiler ignores it.

Comment for Top of Page

A comment to be printed at the top of the next page has a slash (/) in the indicator area. Like an ordinary comment, this comment can appear anywhere in a program. The compiler advances to the top of the next page and prints the comment at the top of that page.

Debugging Line

A debugging line has `D` or `d` in the indicator area. If you run the program with DEBUGGING MODE, the line is part of the program; otherwise, the line is a comment. A debugging line cannot contain embedded SQL/MP or SQL/MX statements. For information on DEBUGGING MODE, see SOURCE-COMPUTER Paragraph (page 114).

Continuation Line

A continuation line has a hyphen (-) in the indicator area. It is a continuation of the previous line. Always leave area A of a continuation line blank.

You can continue any word or literal. If you continue a numeric literal, a reserved word, or a user-defined word, the compiler ignores the trailing spaces of the previous line and initial spaces of the continuation line.

The ANSI rules for continuing a nonnumeric literal are:

- The nonnumeric literal contains all trailing spaces present in area B (through column 72) in the previous line.
- The first character in area B of the continuation line that is not a space must be a quotation mark. The continuation begins with the character immediately following that quotation mark.

Example 17-1 Continuation of Nonnumeric Literal in ANSI Format

```
....+....1....+....2....+....3....+....4....+....5....+....6....+....7....+....8

           DISPLAY "THIS IS AN EXAMPLE OF CONTINUING A LITERAL IN ANSI S

      -            "TANDARD FORMAT."
```

The ANSI rules for continuing a national literal are:

- The national literal contains all trailing spaces present in area B in the previous line.
- The first character in area B of the continuation line that is not a space must be an *N* or *n* followed by a quotation mark. The *N* or *n* must be in an odd-numbered column. The continuation begins with the character immediately following the quotation mark.
- The literal string must end on the previous line before the internal character count (the number of bytes) reaches 73. If you are counting the national characters as you type them, remember that each national character represents two bytes internally. The number of characters allowed on the line depends on the column in which the character-string begins.

It might take a few compilations before you have the literal string divided properly over multiple lines.

Area A

Area A begins at Margin A and uses columns 8 through 11. These must begin in area A:

- Division names
- Section headers
- Paragraph headers
- These level indicators and level numbers for the Data Division:
  - FD (file descriptions)
  - SD (sort-merge file descriptions)
  - Level-numbers 01 and 77

Level-numbers 02 through 49, 66, and 88 can begin in either area A or area B.

Area B

Area B begins at Margin B and uses columns 12 through 72. Sentences within a paragraph must begin in area B. Level-numbers 02 through 49, 66, and 88 can begin in either area A or area B.

Identification Field

The identification field begins at Margin R and uses columns 73 through 80. The identification field can contain any ASCII characters. The compiler treats it as a comment.

18 HP Extensions to ISO COBOL

This section lists the HP extensions to ISO/ANSI COBOL, grouping them according to the sections of this manual that explain them; that is:

- Source Program Organization and Format
- Language Elements
- Data Fundamentals
- Environment Division
- Data Division
- Procedure Division Verbs

These HP extensions are explained in other sections, and are not repeated in this section:

- HP Reserved Words (page 755)
- Chapter 19: HP COBOL CRE Support (page 719)
- Chapter 49: Run-Time Diagnostic Messages (page 1193)

Compiler Directives are HP extensions to ISO COBOL.

HP extensions for using SQL/MP with HP COBOL are explained in the *SQL/MP Programming Manual for COBOL*.

HP extensions for using SQL/MX with HP COBOL are explained in the *SQL/MX Programming Manual for C and COBOL*.

Source Program Organization and Format

The only HP extension to ISO/ANSI COBOL that affects source program organization or format is Tandem reference format, which features:

- 132 columns
- Continuation of a nonnumeric literal without including trailing spaces
- No sequence number area or identification field (comment lines instead)

For more information on Tandem reference format, see Reference Format for Source Program Lines (page 54).

Language Elements

The HP extensions to ISO/ANSI COBOL that are language elements are:

- HP COBOL requires at least one character in a COBOL word to be either a letter or a hyphen; COBOL requires a letter.
- In HP COBOL, using a COBOL word as a library-name does not interfere with its concurrent use as the name of an entity in another category; however, library-names must be unique among themselves.
- COBOL limits the forms of system-names to those permitted for user-defined names. HP COBOL relaxes this restriction. The use of each particular system-name is limited to contexts appropriate for its category. The few minor restrictions on duplicate usages of system-names are discussed in SPECIAL-NAMES Paragraph (page 118).

- HP COBOL allows you to qualify status condition-names.
- COBOL does not permit the decimal point as the rightmost (last) character of a numeric literal; HP COBOL relaxes this restriction somewhat:
  - If the apparent last character of a numeric literal qualifies as a decimal point and the immediately following character is not a space (but is a semicolon, comma, or right parenthesis), the compiler interprets that last character as the decimal point.
  - If the apparent last character qualifies as a decimal point but the immediately following character is a space, the compiler interprets that last character and the space together as the separator that follows the literal.

Data Fundamentals

The only HP extension to ISO/ANSI COBOL that affects data fundamentals is byte alignment of binary data items (see PORT and NOPORT (page 569)).

Environment Division

These HP extensions to ISO/ANSI COBOL are elements of the Environment Division:

- CHARACTER-SET Clause (page 116)
- Special system-names:
  - CONSOLE for the operator’s console
  - MYTERM for the home terminal of a process
  - CHANNEL-1 through CHANNEL-12 for carriage-control tape channels

  See System-Name Clause (page 118).
- File-Control Entries in General (page 127)
- RECEIVE-CONTROL Paragraph (page 155)

Data Division

These HP extensions to ISO/ANSI COBOL are elements of the Data Division:

- Extended-Storage Section (page 190)
- ACCESS MODE Clause (page 191)
- These additional data formats (see USAGE Clause (page 214)):
  - COMPUTATIONAL-3 (COMP-3)
  - COMPUTATIONAL-5 (COMP-5)
  - NATIVE-2
  - NATIVE-4
  - NATIVE-8
  - PACKED-DECIMAL
  - POINTER

Procedure Division Verbs

These HP extensions to ISO/ANSI COBOL involve Procedure Division verbs:

- For developing fault-tolerant programs to run as process pairs:
  - CHECKPOINT (page 314)
  - STARTBACKUP (page 467)
  - SYNCDEPTH phrase of OPEN (page 385)
- To mark the beginning of a parameter list, the USING phrase of ENTER (page 330)
- To specify a data item that stores the value a function returns, the GIVING phrase of ENTER (page 330)

- To control file access and maintain the integrity of data bases in transaction-processing applications or other applications that allow multiple access to files:
  - LOCKFILE (page 369)
  - UNLOCKFILE (page 479)
  - UNLOCKRECORD (page 480)
  - TIME LIMITS phrase of these statements:
    - OPEN (page 385)
    - READ (page 414)READ
    - START (page 457)START
  - EXCLUSIVE, SHARED, and PROTECTED phrases of OPEN (page 385)
  - LOCK phrase of READ (page 414)
  - UNLOCK phrase of REWRITE (page 431)
- To display a prompt before a read operation, PROMPT phrase of READ (page 414)
- REVERSED phrase of READ (page 414)
- To specify an object file that contains a program unit, the OF or IN phrase of these statements:
  - CALL (page 303)
  - CANCEL (page 312)
  - ENTER (page 330)
- Parameters of CALL (page 303) can be at levels other than 01 or 77 and, under some conditions, can be subscripted
- Specification of any type of conditional expression in the WHEN phrase of SEARCH ALL (page 440) (at the cost of not being able to use binary searching in that case)
- To enable different methods of positioning files, the POSITION, GENERIC, and APPROXIMATE phrases of START (page 457)
- PARAGRAPH, SECTION, and PERFORM CYCLE options of EXIT (page 341), which transfer control to the end of the paragraph, section, and PERFORM cycle, respectively
- The GUARDIAN-ERR special register is updated each time a file-manipulating statement is executed. The value of GUARDIAN-ERR usually provides more specific information about the cause of an unsuccessful completion signaled by the file-status data item. For example, if the file status is 30 (permanent error), GUARDIAN-ERR contains the file system error number identifying the cause. See Diagnosing Input-Output Errors (page 261).
- Additional special registers:
  - PROGRAM-STATUS
  - PROGRAM-STATUS-1
  - PROGRAM-STATUS-2
  - RETURN-CODE
- If the program is compiled with the PORT directive, CALL (page 303) behaves like the X/OPEN CALL statement, which is not an element of COBOL (see PORT and NOPORT (page 569)).
- SET TO statement for pointers

  In COBOL, `identifier-3` is a data item in the Linkage Section of any level except 66 or 88; in HP COBOL, it is any data item in the Linkage Section or Data Division of any level except 88.
- MOVE TO statement with sender with decimal digits

  The 1985 COBOL standard does not allow decimal positions in a numeric item used as `sender` in a MOVE TO statement. HP COBOL does allow decimal positions in such an item. The decimal point is ignored and `sender` is handled as if it were an alphanumeric item whose size is equal to the number of 9s in the PICTURE clause. If its USAGE is not DISPLAY, it is converted to DISPLAY for the move operation.

19 HP COBOL CRE Support

All native programs run in the CRE. Native HP COBOL programs can call and be called by programs written in native HP C, native HP C++, and pTAL, even if the main program is not written in HP COBOL.

For more information about writing HP COBOL programs for the CRE, see the *CRE Programmer's Guide*.

Introducing the CRE

The Common Run-Time Environment (CRE) is a set of services that supports mixed-language programs. The CRE library is a collection of routines that implements the CRE. The CRE library enables the language-specific run-time libraries to coexist peacefully with each other. User routines and run-time libraries call CRE library routines to access shared resources managed by the CRE, such as the standard files (input, output, and log) and the user heap, regardless of language.

The CRE does not support all possible operations. For example, the CRE supports file sharing only for the three standard files: standard input, standard output, and standard log. The language-specific run-time libraries access all other files by calling operating environment procedures directly, whether or not a program uses the CRE.

Each routine in the program appears to be running in its own language-specific run-time environment, regardless of the language of the main routine. For example, if the main routine of a mixed-language program is written in HP COBOL, an HP C routine has complete access to the HP C run-time library.

Refer to the *CRE Programmer's Guide* for more information on writing programs that use the services provided by the CRE.

Shared File Operations

A CRE HP COBOL program shares certain file operations with other programs in its run unit. The shared file operations are associated with these files. Other file operations are not shared.

- Standard Output File

  For DISPLAY statements, it is the default file (no UPON phrase is specified).

- Standard Input File

  For ACCEPT statements, it is the default file (no FROM phrase is specified). This file can be a disk file, including an EDIT file.

- Execution Log File

  The HP COBOL run-time diagnostic messages are written to this file. It is normally the home terminal, but can be selected by PARAM EXECUTION-LOG.

- $RECEIVE File

  It is specified in the RECEIVE-CONTROL paragraph or by a TACL ASSIGN command.

Any program in the run unit can read from or write to the preceding files and there is only one shared file connector. In the case of the $RECEIVE File, HP COBOL has high-level interface to it, but programs written in other languages must make special calls to CRE routines to share it. In the case of the other files listed previously, sharing is done by concurrent opens of the file.

Arithmetic Overflow Processing

When an arithmetic overflow condition occurs that is not covered by ON SIZE ERROR, the CRE traps the error. Then, if the program specifies PARAM INSPECT ON, the CRE causes the program to enter the selected debugger (see Debugger Selection). If the program specifies PARAM INSPECT OFF, the CRE causes the program to terminate abnormally with a run-time diagnostic message and a trace to the offending statement.

20 Using HP COBOL in the OSS Environment

The NonStop operating system offers two operating environments:

- Guardian environment
- OSS environment

Open System Services provides industry-standard application program interfaces (APIs) and utilities to enable you to port existing applications quickly and easily to NonStop systems. The NonStop operating system continues to support Guardian services. HP has enhanced many tools from the Guardian environment so that they also operate in the OSS environment. For example, the linker now identifies OSS object files as well as Guardian object files.

Most features of the HP COBOL language and library are available in the OSS environment, and most of them operate as they do in the Guardian environment. This section explains the few exceptions. It also describes line sequential files (also called OSS ASCII files), which are available in the OSS environment. For complete information about the OSS environment, see the *Open System Services Programmer's Guide*.

To terminate a compilation in the OSS environment, press Control-C (that is, press Control and C—not c—simultaneously).

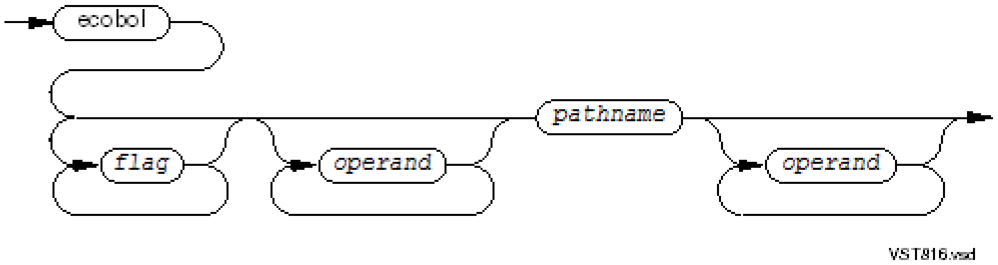
Running the ECOBOL Compiler

In the OSS environment, the OSS utility ecobol calls the compiler (`ecobol`), optionally followed by the linker and SQLCOMP compiler (for SQL/MP) or MXCMP compiler (for SQL/MX). The flags and the types of files in the operands determine which processes operate on the files in the operands.

The `ecobol` utility generates programs that run in the OSS environment. To generate programs that run in the Guardian environment, use either the -Wsystype=guardian flag in the OSS environment or the ECOBOL command in the Guardian environment (see Starting a Compilation (page 538)).

Text file inputs to the compiler can be OSS ASCII text files (code 180) or Guardian EDIT files (code 101).

The command to run the ECOBOL compiler in the OSS environment



is:

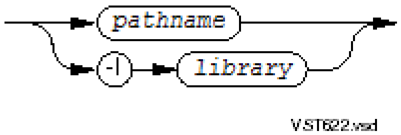
ecobol

  must be typed as shown, in lowercase letters.

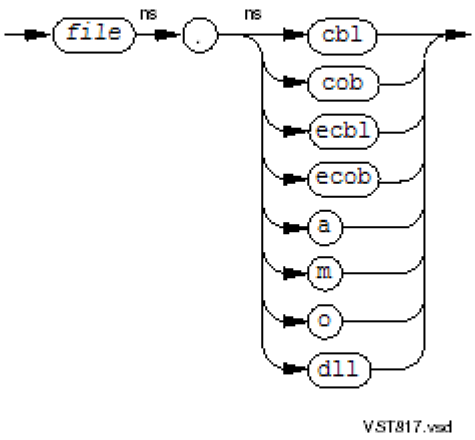
flag

  is as described in the *Open System Services Shell and Utilities Reference Manual*.

operand



pathname



NOTE: The file suffixes (`cbl`, `cob`, and so on) are not case-sensitive.

Running HP COBOL Programs

After successfully compiling your HP COBOL program in the OSS environment, you have a loadfile. Its name is either `a.out` (by default) or the name you gave it with the `-o` flag. If the current directory is in your search path, you can run your program by typing the name of the loadfile and pressing Return. For example, if the name of the loadfile is `a.out`, enter:

```
a.out
```

To run the program with the selected debugger (see Debugger Selection), enter:

```
run -debug a.out
```

If the current directory is not in your search path, add it with this command:

```
export PATH=$PATH:.
```

Calling OSS Functions From HP COBOL Programs

An HP COBOL program can call an OSS function with either the X/Open CALL statement or the ENTER statement. The ENTER statement is easier.

Mixed-Language Programs

In the OSS environment, you can write mixed-language programs.

Native HP COBOL programs can call programs written in native HP COBOL, native HP C, native HP C++, or pTAL.

To produce a native HP COBOL program that contains native HP COBOL and native HP C modules, follow these steps:

1. Using the `c89` utility, compile the HP C modules, but do not link them.
2. Using the `ecobol` utility, compile the HP COBOL modules, specifying any necessary linking or SQL-compiling flags.

For example, suppose that you want to produce a loadfile from the native HP COBOL modules `cobol1.cbl` and `cobol2.cbl` and the native HP C modules `c1.c` and `c2.c`.

First you use the `c89` utility to compile the HP C modules without linking them:

```
c89 -c -o cprog.o c1.c c2.c
```

The resulting object file is `cprog.o`.

Then you use the nmecobol utility to do this:

- Compile the HP COBOL modules
- Link the ecobol compiler output, the object file cprog.o, and the native HP C library.
- Produce the loadfile myprog:

  ```
  ecobol -o myprog cprog.o cobol1.cbl cobol2.cbl
  ```

  The ecobol utility automatically links the HP C DLLs to myprog.

Changing Default Pathnames and Disk Volume

The “Default” column of Table 20-1 shows the default pathnames of the programs that the ecobol commands call and the default disk volume on which they create temporary files.

To change one or more of these defaults before executing the ecobol command, use the export command.

To execute an ecobol command with a specified set of environment variables, use the OSS env function with the environment variables that Table 20-1 lists and explains. For the syntax of the env function, see the env(1) command reference page either online or in the *Open System Services System Calls Reference Manual*.

The effect of the export command lasts until you explicitly change it. The effect of the env function applies only to the ecobol command with which you use it.

Table 20-1 Environment Variables

Variable	Effect	Default
ECOBOL	Determines the pathname of the ECOBOL compiler that the ecobol utility calls	/G/system/system/ecobfe
SQLCOMP	Determines the pathname of the SQL/MP compiler that the ecobol utility calls	/G/system/system/sqlcomp
MXSQLCO	Determines the pathname of the alternate COBOL SQL/MX preprocessor that the ecobol utility calls	/usr/tandem/sqlmx/bin/
MXCMP	Determines the pathname of the alternate SQL/MX compiler that the ecobol utility calls	/G/system/system/
SQLCLIO	Determines the pathname of the object file that describes the SQL/MX API to the ecobol utility	/usr/tandem/sqlmx/lib/ esqlcli.o

Files in the OSS Environment

In the OSS environment, you can use both OSS and Guardian files. OSS files are either line sequential files (see Line Sequential Files) or sequential files with fixed-length records and no alternate keys. Guardian files are all types of files that HP COBOL programs can access in the Guardian environment. In the OSS environment, both OSS and Guardian files must have OSS pathnames.

Topics:

- OSS Pathnames
- OSS Pathnames in HP COBOL Source Programs
- OSS Files in HP COBOL Source Programs
- #IN and #OUT

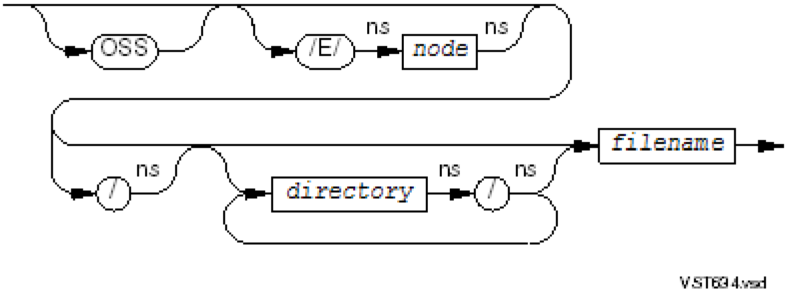
OSS Pathnames

In the OSS environment, both OSS files and Guardian files must have OSS pathnames. These two topics briefly describe these pathnames. For further information, see the *Open System Services*

Programmer's Guide and the `filename(5)` reference page either online or in the *Open System Services System Calls Reference Manual*.

OSS Pathnames for OSS Files

The OSS pathname of an OSS file has this format:



OSS

specifies that the file is an OSS file (the default). If OSS is specified, the entire pathname must be enclosed in quotation marks.

node

specifies the name of a remote node. The operating system on the node must be version D40.00 or later, and the node must have a compatible OSS name server.

/

specifies the root directory when it appears at the beginning of a pathname; elsewhere, it separates directory names and file names.

directory

is the name of a directory. A directory name can contain any characters except slash (/) and the NULL character (zero). Its first character cannot be a hyphen (-). Its maximum length is `NAME_MAX` characters, as defined by the `limitsh` header file. If *directory* contains special characters, the entire pathname must be enclosed in quotation marks.

These directory names have special meanings:

Directory Name	Meaning
.	OSS current working directory
..	Parent directory of the OSS current working directory

file-name

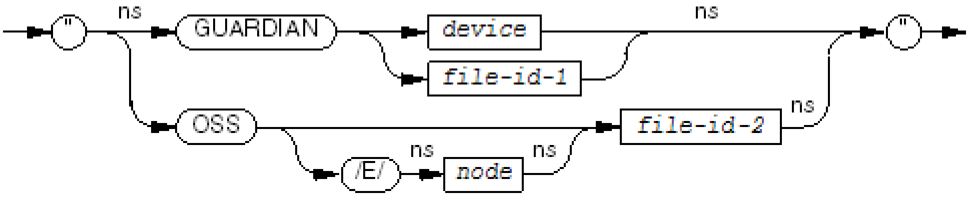
is the name of a file. A file name can contain any characters except slash (/) and the NULL character (zero). Its first character cannot be a hyphen (-). Its maximum length is `NAME_MAX` characters, as defined by the `limitsh` header file. If *file-name* contains special characters, the entire pathname must be enclosed in quotation marks.

Example 20-1 OSS Pathnames for OSS Files

```
file2
vol1/file2
usr/vol1/file2
/usr/vol1/file2
/usr/vol1/part3/file2
"OSS /usr/vol1/file2"
/E/qa2/usr/vol1/file2
"OSS /E/qa2/usr/vol1/file2"
```

OSS Pathnames for Guardian Files

The OSS pathname of a Guardian file has this format:



VST734.vsd

device

specifies a Guardian device or either of the special names #DYNAMIC or #TERM.

file-id-1

node.volume.subvolume.file_id

node

specifies the name of a remote node. The operating system on the node must be version D40.00 or later, and the node must have a compatible OSS name server.

file-id-2

/G/volume/subvolume/file_id

Example 20-2 OSS Pathnames for Guardian Files

```
"GUARDIAN \qa.tests.cobol85.release4"
"GUARDIAN S.#PRNT2"
"OSS /G/tests.cobol85.release4"
"OSS /E/qa/G/$tests.cobol85.release4"
```

OSS Pathnames in HP COBOL Source Programs

Within an HP COBOL source program, OSS pathnames are allowed as *system-file-name* parameters in these contexts:

- In the File-Mnemonic clause of the SPECIAL-NAMES paragraph, for example:

```
SPECIAL-NAMES.
  FILE "OSS /usr/test/fileID" IS A-FILE.
  FILE "GUARDIAN $MYVOL.SUBVOL.FILEID" IS B-FILE.
  FILE "OSS /G/MYVOL/SUBVOL/FILEID" IS C-FILE.
```

NOTE: If *system-file-name* is an OSS file, it cannot be a DEFINE name. If *system-file-name* is a Guardian file, it can be DEFINE name.

- In the ASSIGN clause of a file-control entry, for example:

```
FILE-CONTROL.
  SELECT A-FILE ASSIGN TO "OSS /usr/test/fileID".
  SELECT B-FILE ASSIGN TO "GUARDIAN $MYVOL.SUBVOL.FILEID".
  SELECT C-FILE ASSIGN TO "OSS /G/MYVOL/SUBVOL/FILEID".
```

Within an HP COBOL program, the maximum length of an OSS pathname is the maximum length of a literal (160 characters).

OSS Files in HP COBOL Source Programs

The only OSS files that an HP COBOL program can use are line sequential files (see Line Sequential Files) and sequential files with fixed-length records and no alternate keys. Relative files, indexed files, and sequential files with keys or variable-length records must be Guardian files (Enscribe files), and their OSS pathnames must include "/G" or "GUARDIAN" (see OSS Pathnames for Guardian Files). If the OSS pathname of a sequential file does not include "/G" or "GUARDIAN," then that file is an OSS unstructured file. It has fixed-length records of the maximum record size. In contrast, the default Guardian sequential file is entry-sequenced and can have variable-length records.

#IN and #OUT

In the OSS environment, #IN and #OUT are the default input device (FD 0) and the default output device (FD 1), respectively. You cannot use #IN and #OUT in SELECT clauses or the SPECIAL-NAMES paragraph as you can in the Guardian environment.

Line Sequential Files

Line sequential files (code 180) are available only in the OSS environment. Their organization is line sequential. They are compatible with the system text editor of the OSS environment; therefore, they can also be called OSS ASCII text files. (The X/Open CAE specification defines an OSS ASCII text file as one that is compatible with the system text editor.)

Line sequential files have these characteristics:

- Every character in a record is printable.

NOTE: The operating system does not check for this.

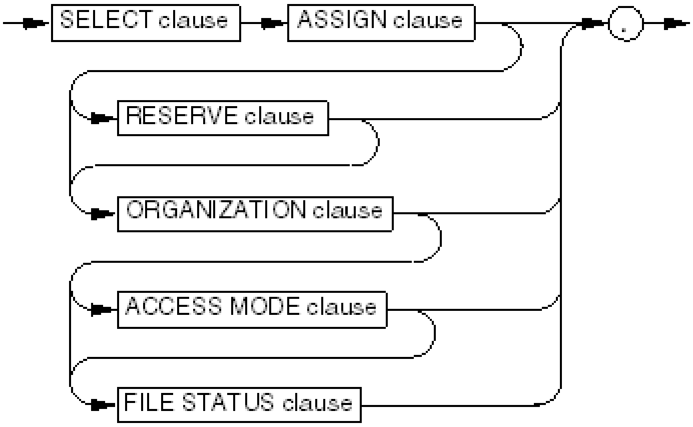
- Every record ends with a line-feed character, which is appended by a write operation and removed by a read operation. (This characteristic distinguishes line sequential files from sequential files.)
- After a read operation, the record area from the last character of the actual record to the end of the record area is filled with space characters (ASCII character code SP).
- If the record area is shorter than the record in the file, the next read operation starts after the last character moved to the record area. (The length of the record area is determined by the RECORD clause if it exists; otherwise it is determined by the largest of the record descriptions.)

These topics explain these items with respect to line sequential files:

- File-Control Entries
- File Description Entries
- I-O-CONTROL Paragraph
- OPEN Statement
- READ Statement
- WRITE Statement

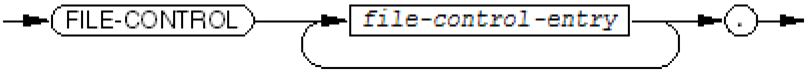
File-Control Entries

The syntax for a file-control entry for a line sequential file is:



VST624.vsd

SELECT clause



VST096.vsd

OPTIONAL

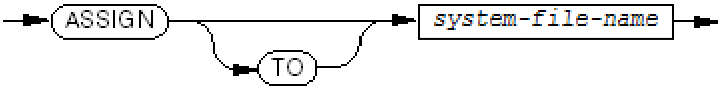
makes the file optional, which means that an OPEN statement with an INPUT or EXTEND phrase can open the file regardless of whether the file exists. If the file exists, its I-O status code is “00”; if not, its I-O status code is “05.” OPTIONAL does not affect the OPEN statement with an OUTPUT phrase.

When you open a nonexistent optional file for input, the first READ statement for that file calls the AT END option (or USE procedure if the READ statement has no AT END phrase).

`file-name`

is the COBOL file name (the *`file-name`* in a file description entry).

ASSIGN clause



VST640.vsd

associates *`file-name`* with *`system-file-name`*. Only the first *`system-file-name`* has meaning. The compiler ignores subsequent names and literals and issues a warning.

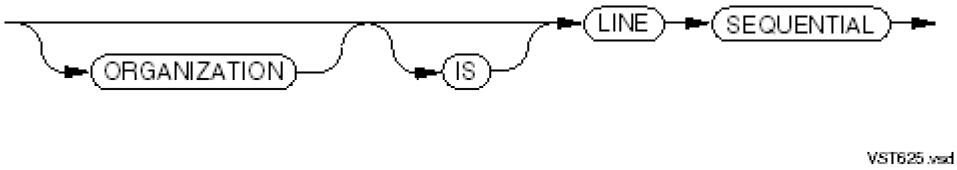
`system-file-name`

is the name of a file as it is known to the file system. It must be enclosed in quotation marks unless it forms a COBOL word. For more information about Guardian file names, see the *Guardian Procedure Calls Reference Manual*. For more information about OSS file names, see the `filename(5)` reference page either online or in the *Open System Services System Calls Reference Manual*.

RESERVE clause

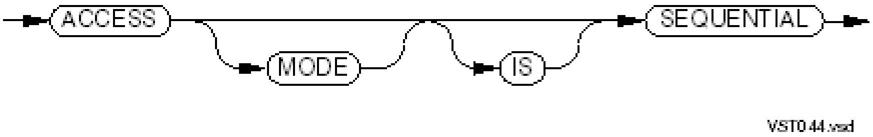
is ignored.

ORGANIZATION clause



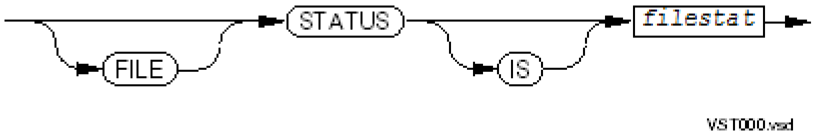
makes the organization of the file line sequential.

ACCESS MODE clause



makes the access mode of the file sequential (the default).

FILE STATUS clause



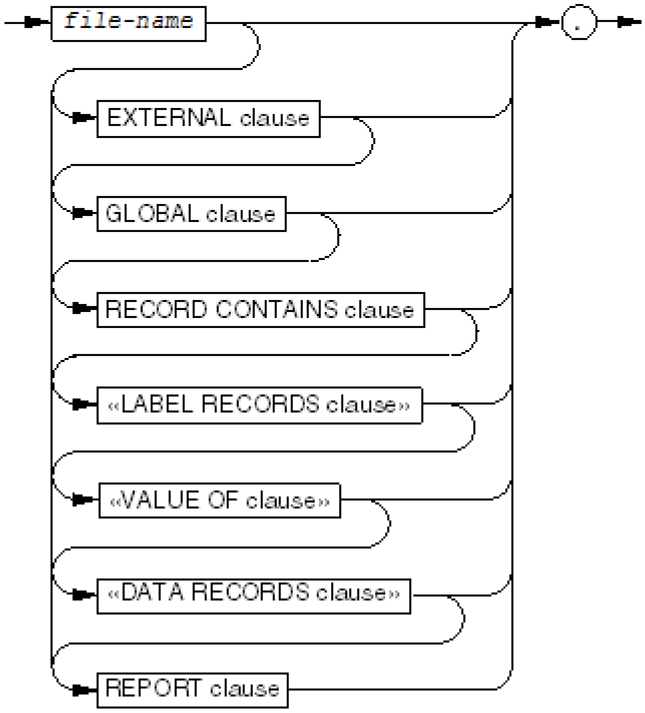
defines `filestat` as the file-status data item for the file. When a COBOL run-time I-O routine completes an operation on the file, it stores the status code in `filestat` before returning control to your program (see I-O Status Code (page 257)).

`filestat`

is an alphanumeric, nonnational data item defined in the Working-Storage Section, Extended-Storage Section, or Linkage Section.

File Description Entries

The syntax for a file description entry for a line sequential file is:



VST633.vsd

file-name

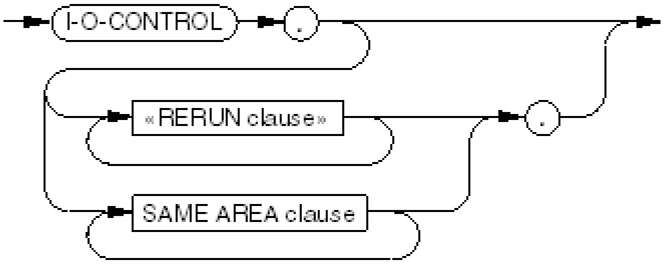
is the highest-level qualifier for both a file description entry and its data descriptions; therefore, the name must be unique within a program.

EXTERNAL clause, GLOBAL clause, RECORD CONTAINS clause, «LABEL RECORDS clause», «VALUE OF clause», «DATA RECORDS clause», REPORT clause

are described in File Description Entries (page 167).

I-O-CONTROL Paragraph

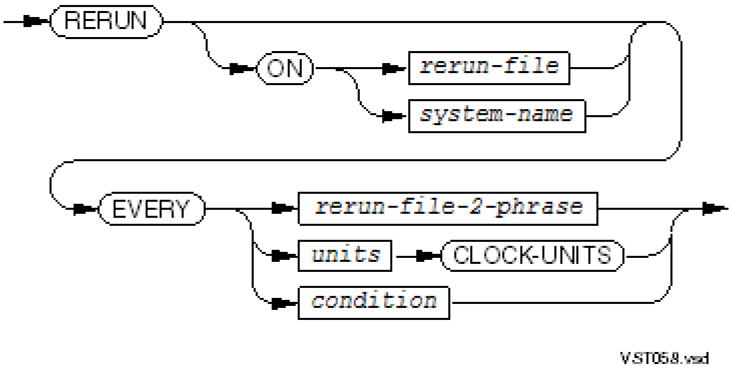
The syntax for an I-O-CONTROL paragraph for a line sequential file is:



VST632.vsd

«RERUN clause»

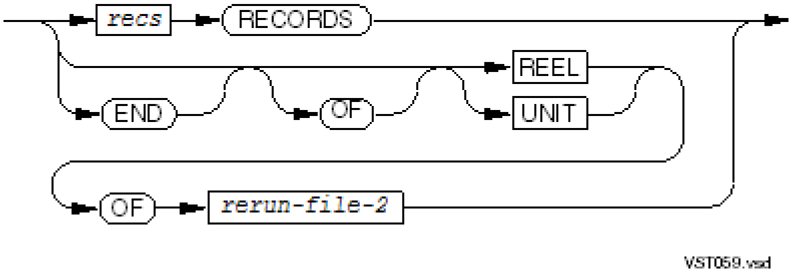
NOTE: The 1985 COBOL standard classifies RERUN as obsolete, so you are advised not to use it.



rerun-file, *system-name*

are handled as comments.

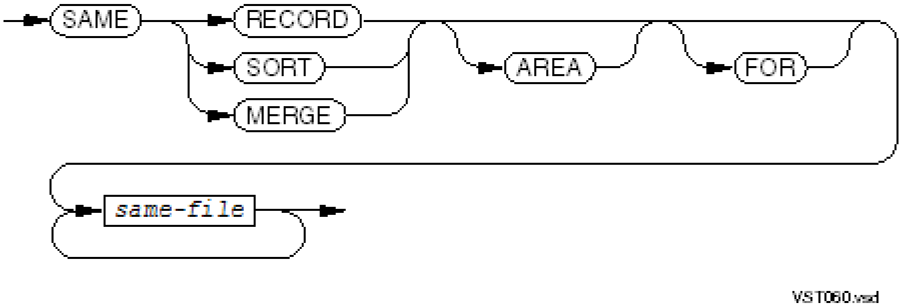
rerun-file-2-phrase



units, *condition*

are handled as comments.

SAME AREA clause



specifies the files that share the same memory during program execution. These files do not share disk space or tape space.

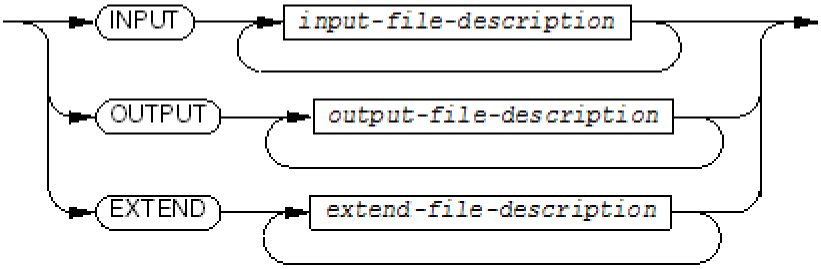
The SAME AREA clause has different meanings for I-O files (sequential, relative, indexed) than for sort-merge files.

same-file

is a file name.

OPEN Statement

The OPEN statement has this `file-specification`:

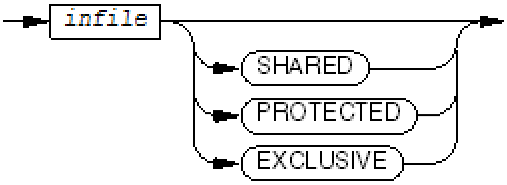


VST830.vsd

INPUT

specifies that the file or files in `input-file-description` are being opened for reading only.

input-file-description



VST627.vsd

infile

is the file description file name of a file to open in INPUT mode, for read operations only.

SHARED

allows other processes to read or write the file while this process is open. SHARED is the default for terminals.

PROTECTED

allows other processes to read but not write the file while this process is open. PROTECTED is the default for input files that are not terminals.

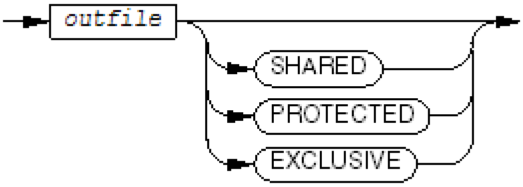
EXCLUSIVE

prevents other processes from reading or writing the file while this process is open. EXCLUSIVE is the default for all other files.

OUTPUT

specifies that the file or files in `output-file-description` are being opened for writing only.

output-file-description



VST628.vsd

outfile

is the file description file name of a file to open in OUTPUT mode, for write operations only.

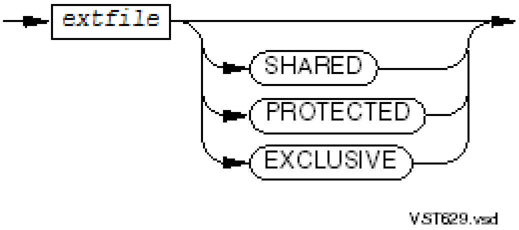
SHARED, PROTECTED, EXCLUSIVE

are the same as described earlier for `infile`.

EXTEND

specifies that the file or files in `extend-file-description` are being opened for writing additional data following any existing data.

`extend-file-description`



`extfile`

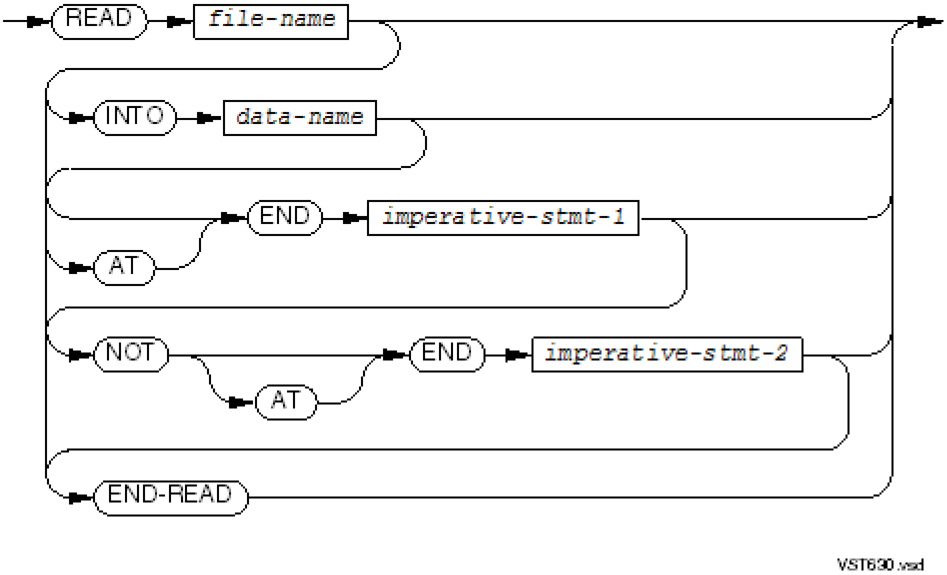
is the file description file name of a sequential file to open in EXTEND mode, for write operations that append records to the file. The file is positioned after the last logical record when opened. All operations on the file must be write operations, as if the file had been opened in OUTPUT mode. If the file is an Enscribe unstructured file, its size must be a multiple of the record size.

SHARED, PROTECTED, EXCLUSIVE

are the same as described earlier for `infile`.

READ Statement

The READ statement for a line sequential file has this syntax:



`file-name`

is the file description name of the file to retrieve a record from.

`data-name`

is the identifier of the data area defined by your program to which the contents of the record area are transferred after the read operation is complete.

`data-name` cannot be an index-name or the identifier of an index data item. The transfer is conducted as if it were a move from an alphanumeric item to an alphanumeric item.

The INTO phrase is allowed only when either:

- All records associated with `file-name` are defined as data structures or as elementary alphanumeric items and `data-name` is either a data structure or an elementary alphanumeric item.
- Only one record is associated with `file-name`.

`imperative-stmt-1`

is an imperative statement to be performed when the end of the file is encountered at the beginning of the read operation. This phrase is required if no USE statement is applicable for the file. If both a USE statement and an AT END phrase are present, only the AT END phrase is used.

`imperative-stmt-2`

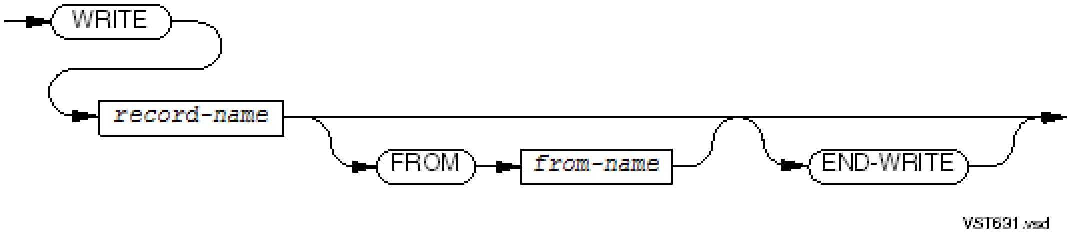
is an imperative statement to be performed when the end of the file is not encountered at the beginning of the read operation.

END-READ

ends the scope of the READ statement, causing the READ to be a delimited-scope statement. If the READ statement does not end with an END-READ phrase, the presence of the AT END or the NOT AT END phrase causes the READ statement to be a conditional statement, which ends at the next period separator.

WRITE Statement

The WRITE statement for a line sequential file has this syntax:



`record-name`

is a logical record described in the File Section of the Data Division. The `record-name` can be qualified by the name of the file with which the record is associated. The data written is the current contents of `record-name`.

`from-name`

is the identifier of a data area whose contents are to be moved to the record specified by `record-name` before the WRITE occurs. `from-name` must specify a data area other than that specified by `record-name`. It cannot specify an index data item.

END-WRITE

ends the scope of the WRITE statement, causing the WRITE to be a delimited-scope statement.

Features Unavailable in the OSS Environment

Features that are available in the Guardian environment but not in the OSS environment fall into these categories:

- ASSIGN Commands
- PARAM Commands
- Compiler Directives
- Process Pairs

ASSIGN Commands

ASSIGN commands are not available in the OSS environment, You can use DEFINEs for some ASSIGN functions, but because DEFINEs do not accept OSS pathnames, they cannot perform the ASSIGN function of overriding file assignments made at compilation time.

PARAM Commands

PARAM commands are not directly available in the OSS environment, but analogous OSS environment variables exist for the PARAM commands in Table 20-2. A mixed-language COBOL program can access these environment variables by using C routines (see the *Open System Services Library Calls Reference Manual* and the *Guardian Native C Library Calls Reference Manual*).

Table 20-2 Analogous PARAM Commands and OSS Environment Variables

PARAM Command	**OSS Environment Variable**[1]
PARAM DEBUG ON	export DEBUG=ON
PARAM DEBUG OFF	export DEBUG=OFF
PARAM EXECUTION-LOG	None—STDERR is used
PARAM INSPECT ON	export INSPECT=ON
PARAM INSPECT OFF	export INSPECT=OFF
PARAM NONSTOP ON	export NONSTOP=ON
PARAM NONSTOP OFF	export NONSTOP=OFF
PARAM PRINTER-CONTROL *name*	export PRINTER_CONTROL=*name*
PARAM SWITCH-*n* ON[2]	export SWITCH_*n* =ON[2]
PARAM SWITCH-*n* OFF[2]	export SWITCH_*n* =OFF[2]

[1] Type nonvariables with the cases and spacing shown.

[2] *n* is “1” through “9” or “10” through “15” (without quotes).

Compiler Directives

These compiler directives are not allowed in the OSS environment:

Directive	**Reason**
NONSTOP	The OSS environment does not support HP COBOL process pairs.
OBJEXTENT	The OSS environment does not support the concept of file extents
SAVE	The OSS environment does not have initialization messages.
SUBTYPE	The OSS environment does not support the concept of subtypes.

Process Pairs

The OSS environment does not support HP COBOL process pairs; therefore, the CHECKPOINT and STARTBACKUP statements return error 7000 in the special register PROGRAM-STATUS.

Features That Operate Differently in the OSS Environment

Features that operate differently in the Guardian and OSS environments fall into these categories:

- ACCEPT and DISPLAY Statements
- Utility Routines that require or return file names

ACCEPT and DISPLAY Statements

These are true in the OSS environment but not in the Guardian environment:

- No prompt is given for an ACCEPT statement.
- If a DISPLAY statement includes *mnemonic-name*, it must be the OSS pathname of either a Guardian file or a pipe to a terminal or device.
- If an ACCEPT statement includes *mnemonic-name*, it must be the OSS pathname of a Guardian process or terminal. If *mnemonic-name* is an OSS device, a diagnostic is issued and the default input device (#IN) is used instead.

Utility Routines

These utility routines accept only Guardian file names:

- COBOL_RETURN_SORT_ERRORS_
- COBOL_SET_SORT_PARAM_TEXT_
- COBOL_SPECIAL_OPEN_

If a program passes a file name to the utility routine COBOL_ASSIGN_, then COBOL_ASSIGN_ assumes that the file name is a Guardian file name unless the calling program identified the file name as an OSS file name.

The utility routine COBOLFILEINFO does not return useful information in the OSS environment. Use the routine COBOL_FILE_INFO_ instead.

When using #DYNAMIC in the OSS environment to dynamically assign Guardian file names or Guardian spooler process names, you must use "GUARDIAN #DYNAMIC" in the file-control entry. The file name passed by the COBOL_ASSIGN_ utility is a standard Guardian name.

Example 20-3 “GUARDIAN #DYNAMIC” in a File-Control Entry

```
SELECT ASGN ASSIGN TO "GUARDIAN #DYNAMIC"
FILE STATUS IS FILE-STATUS.
...
MOVE "$S.#TEST" TO FILENAM
ENTER "COBOL_ASSIGN_" USING ASGN, FILENAM GIVING asn-error
```

21 HP COBOL Limits

The HP architecture and the method of implementation of the HP COBOL compiler ECOBOL, impose these kinds of limits on HP COBOL programs:

- Size (page 737)
- Number (page 738)
- Callability (page 740)
- External Names (page 740)
- Ranges of Values (page 740)
- Organization (page 91)

Size

- **Block size of record in a data file**

  The maximum block size of a record in a data file is 32,767 characters.

- **Buffer size for reading (HP COBOL Fast I-O)**

  2 * `number` * `dbs`, where `number` is specified in the RESERVE clause for the file being read, and `dbs` is the data block size of the file being read.

- **Buffer size for writing (buffered cache or HP COBOL Fast I-O)**

  29K characters per block, regardless of the `number` in the RESERVE clause.

- **Buffer space needed for files**

  The sum of the buffer space needed for files cannot exceed 62 KB.

- **Code**

  — **Single HP COBOL program**

  The code space limit for any single HP COBOL program is 16 MB. This limit does not include any contained programs, each of which has its own 16 MB limit.

  — **HP COBOL run unit**

  The combined object code space for the program file and ordinary DLLs is 256 MB. Ordinary DLLs do not include the public DLLs such as ZCOBDLL and ZCREDLL.

- **Composite of Operands**

  The composite of operands for an arithmetic statement other than COMPUTE must not exceed a size of 18 decimal digits. For arithmetic expressions or COMPUTE statements, the composite of operands does not apply.

- **Data in a run unit**

  The combined size of static data, heap, and flat segments is limited to approximately 1.5 GB.

  Data allocated dynamically (that is, on the stack) is limited to less than 32 MB (see Storage Allocation For an Initial Program). Stack data includes all data items declared in an INITIAL program (other than the main program) and some temporary data items generated by the compiler, but stack data counts toward the limit only if the program is active (has been called and has not exited).

- **File-control entry with multiple keys**

  In a file-control entry with multiple keys, the sum of the lengths of the keys cannot exceed 253 characters.

- **Index**

  An index (index-name or index data item) occupies four bytes, in any section: File, Working-Storage, or Extended-Storage.

- **Length of a numeric literal**

  A numeric literal has a maximum length of 18 digits.

- **Length of a nonnumeric literal or OSS path name**

  A nonnumeric literal or OSS path name has a maximum length of 160 characters.

- **Logical page**

  When a file definition includes a LINAGE clause, the size of its logical page is limited to 9,999 lines.

- **Record or table**

Location of Record or Table	Maximum Record or Table Size
Extended-Storage Section	No explicit limit—one record of 134,217,726 characters is possible
File assigned to $RECEIVE	2 MB
File assigned to another device	Operating environment limits for the device apply
File Section	2 MB for files assigned to $RECEIVE; 32,767 for other files
Linkage Section	134 megabytes
Working-Storage Section	No explicit limit—one record of 134,217,726 characters is possible

- **Relative record number**

  The maximum relative record number for a relative file is 64 bits.

- **Tables allocated by the RECEIVE-CONTROL paragraph**

  The maximum size of the tables allocated by the RECEIVE-CONTROL paragraph is 62 KB (65,400 characters). Anything larger causes a compilation error.

Number

These items, whose number in a program is limited, are in alphabetic order.

- **AFTER phrases in a PERFORM statement**

  The maximum number of AFTER phrases in a PERFORM statement is 6.

- **Characters written to an Edit-format file**

  An HP COBOL program can write at most 239 characters to a record of an Edit-format file

- **Digits in a hexadecimal numeric literal**

  The minimum number of hexadecimal digits in a hexadecimal numeric literal is 2 (1 pair); the maximum is 16 (8 pairs).

- **File names**
  - **In a MULTIPLE FILE TAPE clause**

    HP COBOL supports 31 file names within a single MULTIPLE FILE TAPE clause.
  - **In the GIVING or USING phrase of a MERGE or SORT statement**

    HP COBOL supports a maximum of 31 file names within the USING or GIVING phrase of a SORT or MERGE statement.
- **Keys**

  HP COBOL supports a maximum of 31 alternate record keys for a file, 31 key references in an OCCURS clause, and 31 keys in a SORT or MERGE statement.
- **Maximum number of concurrent statement names in SQL/MP**

  Without Extended Dynamic SQL (EDS), the maximum number of concurrent statement names in SQL/MP is 20. With EDS, you need not prespecify the maximum number of concurrent statement names.
- **MULTIPLE FILE TAPE clauses**

  HP COBOL supports a maximum of 8 MULTIPLE FILE TAPE clauses within a source program.
- **Nesting depth**
  - **OCCURS clauses**

    The maximum depth of OCCURS clauses in a table is 7.
  - **Programs**

    Programs can be nested within other programs to a depth of 7.
  - **PERFORM stack**

    The maximum depth is 50.
- **Parameters**

  The maximum number of parameters that can be passed by an HP COBOL program or to an HP COBOL program is 126.
- **Prime key offset**

  For indexed files, the maximum offset of a prime key from the beginning of a record is 2,034, which is an Enscribe record manager limitation.
- **Records**

  There is no explicit limit on the number of internal or external records in a main program, a called program, or a run unit.
- **Receivers**
  - **In an INITIALIZE Statement**

    The maximum number of receivers in an INITIALIZE statement is 127.
  - **In a MOVE statement**

    The maximum number of receivers in a MOVE statement is 255.
- **References to level-88 condition-names in a program**

  A condition-name can be defined in the Environment Division (associated with an external switch) or in the Data Division (as a level-88 item). The Procedure Division of a program can have a maximum of 32,767 references to level-88 condition-names. There is no limit to the number of references to condition-names for external switches.

- **Subjects in an EVALUATE statement**

  The maximum number of receivers in an EVALUATE statement is 127.

- **Subscripts**

  See Nesting Depth.

Callability

Native HP COBOL programs can call programs written in:

- Native HP C
- Native HP C++
- Native HP COBOL
- pTAL

External Names

Within a run unit, no external data item can have the same name as any external file connector or any program.

Ranges of Values

The ranges of values for types NATIVE-2, NATIVE-4, and NATIVE-8 are:

Type	Lower Bound	Upper Bound
NATIVE-2	-32768	+32767
NATIVE-4	-2147483648	+2147483647
NATIVE-8	-9223372036854775808	+9223372036854775807

The ranges of values for the type COMPUTATIONAL-5 are. Bracketed items are optional.

		Range	
PICTURE Clause	**NATIVE-n Equivalent**	**Signed**	**Unsigned**
PIC [S]9(1) - PIC [S]9(4)	NATIVE-2	-32,768 through 32,767	0 through 65,535
PIC [S]9(5) - PIC [S]9(9)	NATIVE-4	-2,147,483,648 through 2,147,483,647	0 through 4,294,967,295
PIC [S]9(10) - PIC [S]9(18)	NATIVE-8	-9,223,372,036,854,775,808 through 9,223,372,036,854,775,807	0 through 18,446,744,073,709,551,615

Ignored Elements

HP COBOL compilers ignore these elements of COBOL because they are unnecessary in the HP implementation:

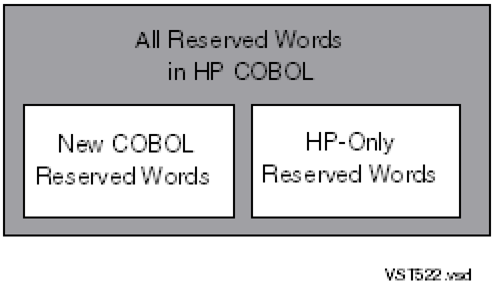
- In the entire source program: In ANSI format, columns 1 through 6 and beyond 72
- In the Environment Division:
  - In the SOURCE-COMPUTER paragraph: Source computer name
  - In the OBJECT-COMPUTER paragraph:
    - Object computer name
    - MEMORY SIZE clause
    - SEGMENT-LIMIT clause
  - In the INPUT-OUTPUT section:
    - In the FILE-CONTROL paragraph: in the `file-control-entry`:
      - In the ASSIGN clause: multiple file names
      - In the SELECT clause: RECORD DELIMITER and PADDING CHARACTER clauses
    - In the I-O-CONTROL paragraph:
      - RERUN clause
      - SAME AREA clause
      - SAME SORT AREA clause
- Procedure Division
  - In independent segments: SEGMENT-LIMIT
  - ENTER COBOL statement

22 Reserved Words

This section contains these alphabetized lists:

- All Reserved Words
- HP Reserved Words

Figure 22-1 How Lists of Reserved Words Are Related



All Reserved Words

This is an alphabetic list of all reserved words in HP COBOL. It includes new HP COBOL reserved words (additions to COBOL 74), words reserved by the COBOL 2002 standard (indicated with an asterisk), and reserved words found only in HP COBOL (HP additions to COBOL). COBOL 2002 words are reserved by the ECOBOL compiler only when the STANDARD 2002 directive is specified. The MIGRATION-CHECK directive causes the compiler to flag these words.

A B C D E F G H I J K L M N O P Q R S T U V W Z Special Characters

A

ACCEPT

ACCESS

*ACTIVE-CLASS

ADD

ADDRESS

ADVANCING

AFTER

*ALIGNED

ALL

ALLOCATE

ALPHABET

ALPHABETIC

ALPHABETIC-LOWER

ALPHABETIC-UPPER

ALPHANUMERIC

ALPHANUMERIC-EDITED

ALSO

ALTER

ALTERNATE

AND

ANY

*ANYCASE

APPROXIMATE

AREA

AREAS

*AS

ASCENDING

ASSIGN

AT

AUTHOR

B

*B-AND

*B-NOT

*B-OR

*B-XOR

*BASED

BEFORE

BINARY

*BINARY-CHAR

*BINARY-DOUBLE

*BINARY-LONG

*BINARY-SHORT

*BIT

BLANK

BLOCK

*BOOLEAN

BOTTOM

BY

C

CALL

CANCEL

CD

CF

CHARACTER

CHARACTERS

CHARACTER-SET

CHECKPOINT

CLASS

*CLASS-ID

CLOCK-UNITS

CLOSE
COBOL
CODE
CODE-SET
*COL
COLLATING
*COLS
COLUMN
COMMA
COMMON
COMMUNICATION
COMP
COMP-3
COMP-5
COMPUTATIONAL
COMPUTATIONAL-3
COMPUTATIONAL-5
COMPUTE
*CONDITION
CONFIGURATION
*CONSTANT
CONTAINS
CONTENT
CONTINUE
CONTROL
CONTROLS
CONVERTING
COPY
CORR
CORRESPONDING
COUNT
*CRT
CURRENCY
*CURSOR

D

DATA
*DATA-POINTER
DATE
DATE-COMPILED
DATE-WRITTEN
DAY

DAY-OF-WEEK

DE

DEBUG-CONTENTS

DEBUG-ITEM

DEBUG-LINE

DEBUG-NAME

DEBUG-SUB-1

DEBUG-SUB-2

DEBUG-SUB-3

DEBUGGING

DECIMAL-POINT

DECLARATIVES

*DEFAULT

DELETE

DELIMITED

DELIMITER

DEPENDING

DESCENDING

DESTINATION

DETAIL

DISABLE

DISPLAY

DIVIDE

DIVISION

DOWN

DUPLICATES

DYNAMIC

E

*EC

EGI

ELSE

EMI

ENABLE

END

*END-ACCEPT

END-ADD

END-COMPUTE

END-DELETE

*END-DISPLAY

END-DIVIDE

END-EVALUATE

END-IF

END-MULTIPLY

END-OF-PAGE

END-PERFORM

END-READ

END-RECEIVE

END-RETURN

END-REWRITE

END-SEARCH

END-START

END-STRING

END-SUBTRACT

END-UNSTRING

END-WRITE

ENTER

*EO

EOP

EQUAL

ERROR

ESI

EVALUATE

EVERY

EXCEPTION

*EXCEPTION-OBJECT

EXCLUSIVE

EXIT

EXTEND

EXTENDED-STORAGE

EXTERNAL

F

*FACTORY

FALSE

FD

FILE

FILE-CONTROL

FILLER

FINAL

FIRST

*FLOAT-EXTENDED

*FLOAT-LONG

*FLOAT-SHORT

FOOTING

FOR

*FORMAT

*FREE

FROM

FUNCTION

*FUNCTION-ID

G

GENERATE

GENERIC

*GET

GIVING

GLOBAL

GO

*GO BACK

GREATER

GROUP

GUARDIAN-ERR

H

HEADING

HIGH-VALUE

HIGH-VALUES

I

I-O

I-O-CONTROL

IDENTIFICATION

IF

IN

INDEX

INDEXED

INDICATE

*INHERITS

INITIAL

INITIALIZE

INITIATE

INPUT

INPUT-OUTPUT

INSPECT

INSTALLATION

*INTERFACE

*INTERFACE-ID

INTO

INVALID

*INVOKE

IS

J

JUST

JUSTIFIED

K

KEY

L

LABEL

LAST

LEADING

LEFT

LENGTH

LESS

LIMIT

LIMITS

LINAGE

LINAGE-COUNTER

LINE

LINE-COUNTER

LINKAGE

*LOCAL-STORAGE

*LOCALE

LOCK

LOCKFILE

LOW-VALUE

LOW-VALUES

M

MEMORY

MERGE

MESSAGE

*METHOD

*METHOD-ID

*MINUS

MODE

MODULES

MOVE

MULTIPLE

MULTIPLY

N

*NATIONAL

*NATIONAL-EDITED

NATIVE

*NESTED

NEGATIVE

NEXT

NO

NOT

NULL

NULLS

NUMBER

NUMERIC

NUMERIC-EDITED

O

*OBJECT

OBJECT-COMPUTER

*OBJECT-REFERENCE

OCCURS

OF

OFF

OMITTED

ON

OPEN

OPTIONAL

*OPTIONS

OR

ORDER

ORGANIZATION

OTHER

OUTPUT

OVERFLOW

*OVERRIDE

P

PACKED-DECIMAL

PADDING

PAGE

PAGE-COUNTER

PERFORM

PF

PH

PIC

PICTURE

PLUS

POINTER

POSITION

POSITIVE

*PRESENT

PRINTING

PROCEDURE

PROCEDURES

PROCEED

PROGRAM

PROGRAM-ID

*PROGRAM-POINTER

PROGRAM-STATUS

PROGRAM-STATUS-1

PROGRAM-STATUS-2

PROMPT

*PROPERTY

PROTECTED

PURGE

Q

QUEUE

QUOTE

QUOTES

R

*RAISE

*RAISING

RANDOM

RD

READ

RECEIVE

RECEIVE-CONTROL

RECORD

RECORDS

REDEFINES

REEL

REFERENCE

REFERENCES

RELATIVE

RELEASE

REMAINDER

REMOVAL

RENAMES

REPLACE

REPLACING

REPLY

REPORT

REPORTING

REPORTS

*REPOSITORY

RERUN

RESERVE

RESET

*RESUME

*RETRY

RETURN

*RETURNING

REVERSED

REWIND

REWRITE

RF

RH

RIGHT

ROUNDED

RUN

S

SAME

*SCREEN

SD

SEARCH

SECTION

SECURITY

SEGMENT

SEGMENT-LIMIT

SELECT

*SELF

SEND

SENTENCE

SEPARATE

SEQUENCE

SEQUENTIAL

SET

SHARED

*SHARING

SIGN

SORT

SORT-MERGE

SOURCE

*SOURCES

SOURCE-COMPUTER

SPACE

SPACES

SPECIAL-NAMES

STANDARD

STANDARD-1

STANDARD-2

START

STARTBACKUP

STATUS

STOP

STRING

SUB-QUEUE-1

SUB-QUEUE-2

SUB-QUEUE-3

SUBTRACT

SUM

*SUPER

SUPPRESS

SYMBOLIC

SYNC

SYNCDEPTH

SYNCHRONIZED

*SYSTEM-DEFAULT

T

TABLE

TAL

TALLYING

TAPE

TERMINAL

TERMINATE

TEST

TEXT

THAN

THEN

THROUGH

THRU

TIME

TIMES

TO

TOP

TRAILING

TRUE

TYPE

*TYPEDEF

U

UNIT

*UNIVERSAL

UNLOCK

UNLOCKFILE

UNLOCKRECORD

UNSTRING

UNTIL

UP

UPON

USAGE

USE

*USER-DEFAULT

USING

V

*VAL-STATUS

*VALID

*VALIDATE

*VALIDATE-STATUS

VALUE

VALUES

VARYING

W

WHEN

WITH
WORDS
WORKING-STORAGE
WRITE

Z

ZERO
ZEROES

Special Characters

+
-
*
/
**
<
>
=

HP Reserved Words

This is an alphabetic list of reserved words found only in HP COBOL (HP additions to COBOL).

A C E G L N P R S T U

A

ADDRESS
APPROXIMATE

C

CHARACTER-SET
CHECKPOINT
COMP-3
COMP-5
COMPUTATIONAL-3
COMPUTATIONAL-5

E

EXCLUSIVE
EXTENDED-STORAGE

G

GENERIC
GUARDIAN-ERR

L

LOCKFILE

N

NULL
NULLS

P

PROGRAM-STATUS
PROGRAM-STATUS-1
PROGRAM-STATUS-2
PROMPT
PROTECTED

R

RECEIVE-CONTROL
REPLY

S

SHARED
STARTBACKUP
SYNCDEPTH

T

TAL

U

UNLOCK
UNLOCKFILE
UNLOCKRECORD

23 Creating and Compiling HP COBOL Source Programs

NOTE: This section applies primarily to the Guardian environment. If you are compiling HP COBOL programs in the OSS environment, see Chapter 20: Using HP COBOL in the OSS Environment (page 721).

If you want to include SQL/MP or SQL/MX commands in your HP COBOL program, see the *SQL/MP Programming Manual for COBOL* or the *SQL/MX Programming Manual for C and COBOL*.

Using the TACL Command Log

When you log onto an HP system, the HP Tandem Advanced Command Language (TACL) prompt appears. The TACL prompt includes the command number, which starts at 1:

```
1>
```

Displaying Previous Commands

TACL maintains a chronological log of previously entered commands, which you can display with the HISTORY command. Suppose that you have entered these TACL commands (notice the command number in each of the prompts):

```
1> who
2> fileinfo
3> view myfile
4> history
```

Command 4, `history`, displays:

```
1> who
2> fileinfo
3> view myfile
4> history
5>
```

If you enter a question mark (?), TACL displays the last command:

```
5> ?
5> history
5>
```

If you enter a question mark followed by a number, TACL displays the command with that number:

```
5> ?2
5> fileinfo
5>
```

If you enter a question mark followed by text, TACL displays the most recent command that begins with that text:

```
5> ?v
5> view myfile
5>
```

Reexecuting Previous Commands

If you enter an exclamation mark (!), TACL displays and executes the last command:

```
5> !
5> history
1> who
2> fileinfo
3> view myfile
4> history
```

```
5> history
6>
```

If you enter an exclamation mark followed by a number, TACL displays and executes the command with that number:

```
6> !4
6> history
1> who
2> fileinfo
3> view myfile
4> history
5> history
6> history
7>
```

If you enter an exclamation mark followed by text, TACL displays and executes the most recent command that begins with that text:

```
7> !v
7> view myfile
8>
```

Editing and Reexecuting Previous Commands

If you enter the command FC, TACL displays the last command so that you can edit it:

```
8> fc
8> view myfile
8...
```

You can edit the displayed command (`view myfile`) with the commands *`D`* (delete), *`I`* (insert), and *`R`* (replace). When you press Return, TACL displays the edited command. You can continue to edit the command or press Return. When you press Return without editing the command, TACL executes the command.

```
8> fc
8> view myfile
8..     dd
8> view file
8..     isrc
8> view srcfile
8..           r1
8> view srcfil1
8..           e1
8> view srcfile1
8...
```

The absence of a *`D`*, *`I`*, or *`R`* command is handled as an *`R`* command (last change in the preceding example).

If you enter the FC command followed by a number, TACL displays the command with that number so that you can edit it:

```
9> fc 2
9> fileinfo
9...
```

If you enter the FC command followed by text, TACL displays the most recent command that begins with that text so that you can edit it:

```
10> fc view
10> view srcfile1
10...
```

TACL also allows you to program the function keys on your terminal and write macros. For more information about TACL, see the *TACL Reference Manual*.

Creating or Altering an HP COBOL Source Program

The compiler accepts source lines from any sequential file. On NonStop systems, the most common way to create or alter an HP COBOL source program is to use an HP editor or Starbase Corporation's Codewright smart language editor with HP Extensions for Codewright.

HP offers two editors, EDIT and PS TEXT EDIT (abbreviated "TEDIT"). Both editors create and allow you to alter the same type of file, the EDIT format file (abbreviated "EDIT file").

Both HP editors and Codewright smart language editor allow you to:

- Create a new file
- Enter lines in a file
- Copy lines from one file into another file
- Move lines from one place in a file to another place in the same file
- Duplicate lines from one place in a file to another place in the same file
- Delete lines from a file
- Change one portion of text to another throughout a range of lines in a file
- List the lines that a file contains

Whether or not you use an HP editor, your HP COBOL source file must conform to a reference format that the compiler accepts.

Topics:

- EDIT Files
- Reference Format
- EDIT Editor
- TEDIT Editor
- HP Tandem Extensions for Codewright (TEC)

EDIT Files

An EDIT format file (abbreviated "EDIT file") is a specially formatted disk file. It is the only type of file that the EDIT and TEDIT editors create or alter. The file-code value that the EDIT or TEDIT editor assigns to an EDIT file is 101.

To use the EDIT or TEDIT editor on the contents of a non-EDIT file, create a new EDIT file and copy the text of the non-EDIT file into it.

For more information about EDIT files, see the *EDIT User's Guide and Reference Manual*.

Reference Format

Individual source program lines must follow a reference format that the compiler accepts. There are two choices of reference format: Tandem and ANSI. Tandem format, an HP extension to COBOL, is less restrictive than ANSI. The principal differences between the Tandem and ANSI formats are in:

- Margin locations
- Permitted line lengths
- Absence of sequence number and identification field in Tandem format
- Continuation lines

Figure 23-1 Tandem Reference Format

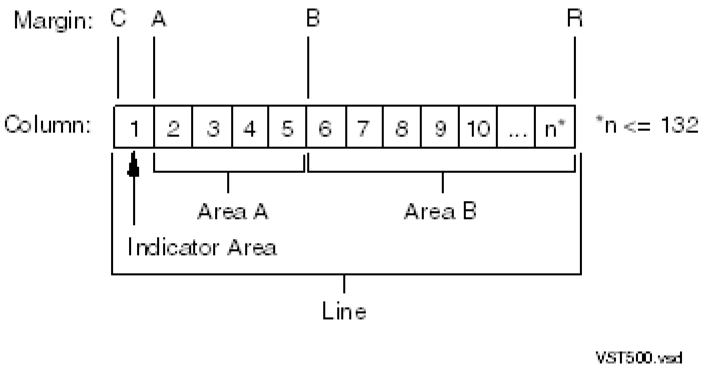
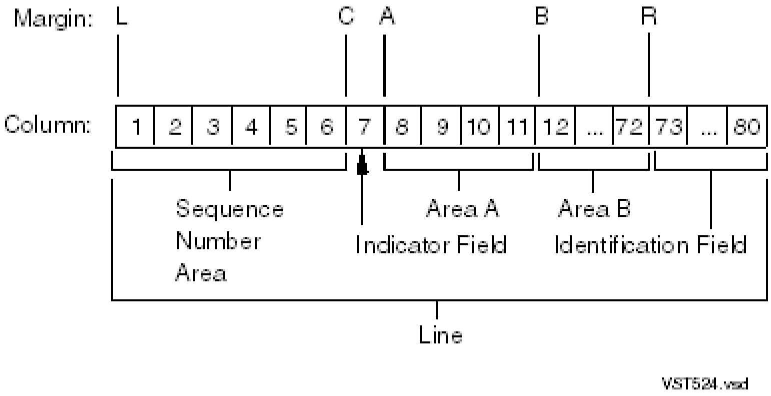


Figure 23-2 ANSI Reference Format



You can write a COBOL source program entirely in either format or in a mixture of both. The default is Tandem format. Unless you direct the compiler to accept ANSI format, the compiler assumes the entire program is in Tandem format.

EDIT Editor

The EDIT editor is not as powerful as the TEDIT editor, but it has some useful abilities that TEDIT either lacks or does not perform as simply. Examples of both cases are in this explanation.

CAUTION: The EDIT editor works on your original file, not on a copy of it. If you want to provide for the possibility of discarding an entire editing session, you must make a backup copy yourself; that is, the EDIT editor does not automatically create a backup file. You can make the backup copy with the EDIT command PUT.

To call the EDIT editor, type its name at the TACL prompt (case is unimportant):

```
93> EDIT
```

The EDIT editor identifies itself (including version and RVU date) and prompts you with an asterisk:

```
TEXT EDITOR - T9601D10 - (08JUN92)
*
```

To retrieve an EDIT file to work on (in this example, PAYROLS), use the GET command:

```
*GET PAYROLS
CURRENT FILE IS \NODE1.$VOL3.SUBVOL2.PAYROLS
```

You can also call the EDIT editor with a file name:

```
94> EDIT PAYROLS
TEXT EDITOR - T9601D10 - (08JUN92)
CURRENT FILE IS \NODE1.$VOL3.SUBVOL2.PAYROLS
```

To retrieve lines from a non-EDIT file (in this example, PAYROLXX) and put them into an EDIT file (in this example, PAYROLS), use the GET and PUT commands together:

```
*GET PAYROLXX PUT PAYROLS
```

(With the TEDIT editor, you cannot combine the GET and PUT commands.)

If the file PAYROLS already exists, the EDIT editor asks you whether it should purge the old PAYROLS (that is, overwrite its current text with the new text from PAYROLXX, keeping the name PAYROLS for the file):

```
*GET PAYROLXX PUT PAYROLS
SHALL I PURGE THE OLD \NODE1.$VOL3.SUBVOL2.PAYROLS?
```

If you tell the EDIT editor not to delete PAYROLS, the EDIT editor asks you for a new name for the EDIT file:

```
SHALL I PURGE THE OLD \NODE1.$VOL3.SUBVOL2.PAYROLS? NO
NAME THE NEW FILE: PAYROLS2
CURRENT FILE IS \NODE1.$VOL3.SUBVOL2.PAYROLS2
```

If you are sure that you want to overwrite PAYROLS, you can avoid the question-and-answer routine by following PAYROLS with an exclamation mark:

```
*GET PAYROLXX PUT PAYROLS!
CURRENT FILE IS \NODE1.$VOL3.SUBVOL2.PAYROLS
```

Another convenience the EDIT editor provides that the TEDIT editor lacks is the ability to list all lines that contain two or more combinations of text characters in any order. Although the TEDIT editor has some powerful pattern-matching abilities, it is much easier to find all lines containing both ABC and XYZ, in any order, using the EDIT editor.

The EDIT editor has a line editor and a full-screen editor. The full-screen editor is named VS. If VS terminates abnormally, it creates a recovery file whose name is of the form ZZVS*nnnn* (where *nnnn* is a four-digit number chosen by VS to make a unique file name). To detect these ZZVS*nnnn* files, use the TACL command FILES. Purge any ZZVS*nnnn* files that you do not need.

To exit the EDIT editor, type “exit” in response to the asterisk prompt (case is unimportant):

```
*EXIT
95>
```

For complete information about the EDIT editor, see the *EDIT User’s Guide and Reference Manual*.

TEDIT Editor

The TEDIT editor is a full-screen editor with a command line. Overall, the TEDIT editor is more powerful than the EDIT editor. The TEDIT commands SEARCH and REPLACE have powerful pattern-matching abilities, with single-character, multicharacter, and set-of-characters wild cards.

CAUTION: The TEDIT editor works on your original file, not on a copy of it. If you want to provide for the possibility of discarding an entire editing session, you must make a backup copy yourself. You can make the backup copy before editing the file, with the File Utility Program (FUP) DUP command, or you can use the WRITE command from within the TEDIT editor to make a copy at any time.

To call the TEDIT editor, type its name at the TACL prompt (case is unimportant):

```
81> TEDIT
```

The TEDIT editor clears the screen and displays a help screen that explains its function. At the bottom of the help screen, it asks you for a file name:

```
(To EXIT press CTRL-Y) File:  PAYROLS
```

You can also call the TEDIT editor with a file name:

```
82> TEDIT PAYROLS
```

If you call the TEDIT editor with a file name, it does not display the help screen. If the file you specified exists, the TEDIT editor clears the screen and displays the first 24 lines of the file and a banner on line 25. The banner contains:

- The window number (you can open a second window)
- The file name, fully qualified with node, volume, and subvolume names
- The range of lines displayed (with "(BOF)" if you are at the beginning of the file and "(EOF)" if you are at the end of the file)
- The version of the TEDIT editor you are using

Example 23-1 TEDIT Banner

```
1) \NODE1.$VOL3.SUBVOL2.PAYROLS 1/24 (BOF) 1:79         D20.
```

If the TEDIT editor cannot find the PAYROLS file, it clears the screen and displays:

```
\NODE1.$VOL3.SUBVOL2.PAYROLS doesn't exist.
OK to create?  Respond Y or N:
```

To exit the TEDIT editor, press Shift and F16 simultaneously.

These topics briefly describe the TEDIT editor's second window, commands, and customizing. For complete information about the TEDIT editor, see the *PS TEXT EDIT Reference Manual* and the *PS TEXT EDIT and PS TEXT FORMAT User's Guide*.

Topics:

- Second Window
- TEDIT Commands
- Customizing the TEDIT Editor

Second Window

The TEDIT editor allows you to open a second window. Some of the tasks that the TEDIT editor allows you to do with two windows are:

- Display two parts of the same file (such as the first 10 lines and the last 10 lines)
- Display two different files (such as the source text and a COPY library)
- Copy text from one window to the other
- Display only window 1, only window 2, or both windows
- Close one window and open it to a third file

TEDIT Commands

Most TEDIT commands consist of a noun (such as LINE, SENTENCE, WINDOW, BALANCED-EXPRESSION, or REGION) and a verb (such as INSERT, MOVE, COPY, DELETE, FORWARD, BACKWARD, SEARCH, REPLACE, or UNDO). Some TEDIT commands also include a repeat count or parameters. Many nouns and verbs are persistent—that is, you can specify a noun (such as LINE) and apply a series of verbs to it (such as COPY, MOVE, DELETE), or you can specify a verb and apply a series of nouns to it; for example, the command

```
DELETE; LINE; LINE; LINE
```

deletes the line the cursor is on and the two lines after it.

You execute TEDIT commands in two ways:

- Press predefined command keys and key combinations (see the TEDIT keyboard template for your terminal).
- Type the commands on the command line and press Return (to access the command line, press Shift and Return simultaneously, or use the key specified by the TEDIT keyboard template for your terminal).

If a command requires parameters and you do not supply them, the TEDIT editor prompts you for them. SEARCH and REPLACE are examples of TEDIT commands that require parameters.

Customizing the TEDIT Editor

Some ways in which you can customize the TEDIT editor are:

- You can write macros that perform complex tasks with few keystrokes.
- With the RECONFIGURE KEYS command, you can redefine the default keys.
- With the RECONFIGURE OPTIONS command, you can do such things as reset your tab stops and define delimiter pairs.

  You can define up to 10 delimiter pairs, such as left and right parentheses and the commonly used HP COBOL keyword pairs, such as IF and END-IF. If you establish IF and END-IF as the delimiters of a balanced expression, you can use FORWARD BALANCE-EXPRESSION to move the cursor from an IF to its corresponding END-IF.
- You can save your redefined option set in your TEDIT user profile and recall it whenever you want it with the TEDIT command USEPROFILE.
- You can define different option sets for distinct programming tasks.

HP Tandem Extensions for Codewright (TEC)

Starbase Corporation’s Codewright is a smart language editor that allows you to edit source code and issue compiler, version-control, and operating-system commands. The base editor facilitates source code creation with keyword chromacoding, search and replace mechanisms, function execution, undo and redo commands, and general editing capabilities. Codewright blends a command shell with its own tools, and you can easily extend it to work with other third-party tools or dynamic link libraries that you build with your own compiler language. HP has taken advantage of this extensibility to create HP Tandem Extensions for Codewright (TEC).

The main features of TEC are:

- Language-Sensitive Programming Support
- Dictionaries and Templates
- Chromacoding
- Project Definition
- Viewing Source Files
- Comment Headers and Parameter Documentation

More information on TEC is available through its online help.

Language-Sensitive Programming Support

TEC provides language-specific menus to support the coding of these HP languages:

- HP C and HP C++
- COBOL
- pTAL and TAL
- TACL

When you open a source file, the appropriate language-specific menu appears on the menu bar.

For all languages, you can search (GREP) commands in the file, keywords are chromacoded (see Chromacoding), and you can save function call parameters in the dictionary for later use (see Dictionaries and Templates).

For HP C and HP C++, you can create or insert modules or work with #PRAGMAS and #INCLUDES. For HP C++, you have additional support for creating classes and class members.

For COBOL, you can view source code by paragraph and search (GREP) by paragraph.

For pTAL and TAL, you can generate HP C prototypes from TAL procedures by pointing and clicking; insert ?PROC, ?SECTION, or ?PAGE; and check references.

Dictionaries and Templates

TEC maintains four project dictionaries: user, parameter, class, and system. Information can be stored and retrieved from these dictionaries and used in conjunction with comment generation (see Comment Headers and Parameter Documentation).

You can define text templates that contain values for function parameters. The values are defined when you insert the template into the dictionary. When you direct TEC to call a function, TEC calls it with the parameter values that you specified.

Chromacoding

Language-dependent chromacoding enables you to select colors to enhance the visibility of comments, keywords, strings, numbers, preprocessor commands, changed lines, and braces. You can set the colors you select in Document Preferences in TEC version 3.0.

Project Definition

In addition to the basic project information supplied by Codewright itself, TEC allows you to define and change a set of dictionary specifications and search directories to facilitate locating source files.

Viewing Source Files

TEC allows you to view or collapse source files by function, #PRAGMA, paragraph, or normal view. This feature helps you to locate code quickly by higher-level code segments and to provide a call map of the selected module. After you have located the code, you can return the source program to its normal expanded view.

Comment Headers and Parameter Documentation

TEC allows you to automatically create and maintain information in comment boxes, particularly for the beginning and end of each module. After you have entered parameters into the parameter/comment dictionary (by pointing and clicking), TEC automatically retrieves their definitions for building other parameter list comment boxes.

Compiling an HP COBOL Source Program

The compiler translates HP COBOL source programs into machine language.

The ECOBOL compiler does not interact with the `eld` utility, but if you specify the CALL-SHARED and RUNNABLE directives, and there are no compilation errors, the ECOBOL compiler calls the `eld` utility, which produces a PIC loadfile. If you specify the SHARED and RUNNABLE directives, and there are no compilation errors, the ECOBOL compiler calls the `eld` utility, which produces a DLL.

From the user's viewpoint, compilation is a single-step process. (If you are interested in the details of the compilation process, see Compilation Details (page 525).

Topics:

- Running the Compiler
- Naming the Object File
- Specifying the Default COPY Library
- Specifying Compiler Directives
- Specifying Subvolumes to Be Searched for Unqualified Files
- Changing the Compilation Environment

- Improving Compilation Speed
- Launching Compilations to Run Unattended

Running the Compiler

To run the compiler, use the ECOBOL command. Specify the name of the file containing the source program or programs that you want to compile; for example:

```
23> ECOBOL /IN XYZ/
```

The preceding ECOBOL command produces a loadfile only if the source file, XYZ, contains a RUNNABLE directive. If it does not, the command that produces a loadfile is:

```
23> ECOBOL /IN XYZ/; RUNNABLE
```

The file XYZ must be an EDIT or entry-sequenced file.

```
24> ECOBOL /IN XYZ/; RUNNABLE
```

The preceding commands deliver the compiler listing to your terminal. To send the compiler listing to spooler collector $SPX for printing instead, you must specify the OUT parameter:

```
25> ECOBOL /IN XYZ, OUT $SPX/; RUNNABLE
```

The preceding commands send the compiler listing to the default location of spooler collector $SPX. To specify another location, such as #LPPR, include the location in the OUT parameter:

```
26> ECOBOL /IN XYZ, OUT $SPX.#LPPR/; RUNNABLE
```

If spooler collector $SPX recognizes location #LPPR, the compiler listing is queued for printing. If you want to examine the compiler listing before printing it, specify a fictitious location, such as #XYZLST:

```
27> ECOBOL /IN XYZ, OUT $SPX.#XYZLST/; RUNNABLE
```

Using the PERUSE utility, list the last page of the compiler listing (the last page is a compilation summary). If the summary reports errors, use an HP editor to fix the source program and then recompile it (remember to delete the compiler listing from the spooler collector). If the summary does not report errors, change its location to an existing printer. You delete the compiler listing from the spooler collector or change its location to an existing printer from within the PERUSE utility. For information about the PERUSE utility, see Using PERUSE (page 924) and the *Guardian User's Guide*.

NOTE: For easier ways to check a program for syntax errors, see Directives for Syntax Checking Only.

The compiler does not accept an empty OUT file.

To specify a file for information about any compilation errors or warnings, use the ERRORFILE directive:

```
29> ECOBOL /IN XYZ, OUT $SPX.#XYZLST/; RUNNABLE; ERRORFILE EFILE
```

The error logging file is not an EDIT file. You can use it only with the FIXERRS TACL macro. For more information about the FIXERRS macro, see FIXERRS Macro (page 708). For more information about the ERRORFILE directive, see ERRORFILE (page 558).

IN and OUT are options of the TACL command RUN. Other RUN command options that you might find useful when compiling HP COBOL programs are:

- **NOWAIT**

  If you specify

  ```
  ECOBOL /IN XYZ, OUT $SPX.#LPPR, NOWAIT/; RUNNABLE
  ```

  the TACL program does not wait while the compiler runs, but returns a TACL prompt after starting the compiler. (The compiler runs in the background.)

- **CPU**

  If you specify

```
ECOBOL /IN XYZ, OUT $SPX.#LPPR, CPU 7/; RUNNABLE
```

the compiler runs in processor seven (processors are numbered from 0 through 15).

For more information about these and additional RUN commands, see the *TACL Reference Manual*.

Using an HP editor, you can create OBEY command files. OBEY command files contain TACL commands, such as the ECOBOL command, which are executed when you specify the name of the OBEY command file in an OBEY command. Here is a sample session:

```
45> EDIT CFILE1
TEXT EDITOR - T9601D10 - (08JUN92)
$VOL3.SUBVOL2.CFILE1 DOES NOT EXIST.  SHALL I CREATE IT? Y
CURRENT FILE IS $VOL3.SUBVOL2.CFILE1
*ADD
    1         ECOBOL     /IN XYZ,OUT $SPX.#HOLD/
    2     //
*EXIT
46> OBEY CFILE1
```

The command OBEY CFILE1 causes execution of the command:

```
ECOBOL  /IN XYZ,OUT $SPX.#HOLD/
```

An OBEY command file can contain more than one command. If another OBEY command file, CFILE2, contains the commands

```
ECOBOL  /IN XYZ,OUT $SPX.#HOLD/
PERUSE
```

the command OBEY CFILE2 is equivalent to the command series

```
47> ECOBOL  /IN XYZ,OUT $SPX.#HOLD/
48> PERUSE
```

For more information about OBEY command files, see the *Guardian User's Guide*.

Naming the Object File

To simplify the compilation examples in the preceding topic, no name is specified for the object file that the compiler produces. In these examples, the object file is given the default name RUNUNIT. To specify a name for the object file, type the name after the final slash, as in this example, which names the object file XYZOBJ:

```
88> ECOBOL /IN XYZ, OUT $SPX/ XYZOBJ; RUNNABLE
```

Although program compilation is a single-step process from the user's viewpoint, producing a loadfile actually involves the linker. The linker resolves the external references in CALL and ENTER statements and the implicit calls to HP COBOL run-time library routines that many HP COBOL statements cause.

If you specify the CALL-SHARED and RUNNABLE directives, and there are no compilation errors, the ECOBOL compiler calls the `eld` utility, which produces a PIC loadfile. If you specify the SHARED and RUNNABLE directives, and there are no compilation errors, the ECOBOL compiler calls the `eld` utility, which produces a DLL.

The ECOBOL compiler:

1. If the object file already exists and it is not a code 800 file, the compiler attempts to create the file named OBJECT. If this file exists or is open by another process, the compiler creates a file named ZZNC*nnnn*, where *nnnn* is a sequence of randomly chosen alphanumeric characters.
2. If the object file already exists and is a code 800 file and is not open by another process, it is purged and a new one is created.
3. If the object file is open by another process, the compiler renames the existing file, giving it the name ZZNC*nnnn* where *nnnn* is a sequence of randomly chosen alphanumeric characters. The specified object file is then created. If the existing object file cannot be renamed, the compilation terminates.

The ECOBOL compiler tries the name ZZNC*nnnn*, where *nnnn* is a sequence of randomly chosen alphanumeric characters. If the ZZNC*nnnn* name that the ECOBOL compiler tries already exists, the compiler tries another one (with a different sequence of randomly chosen alphanumeric characters). The compiler tries several different ZZNC*nnnn* names if necessary, but does not keep trying indefinitely. If the compiler stops trying to name the object file, verify that you have create access to the designated subvolume.

Suppose that you specify the name OBJFILE for the object file, but the operating environment is executing a process from a file named OBJFILE. The ECOBOL compiler cannot replace the executing version of OBJFILE, so the linker renames the old object file ZLDAF*nnn* and names the new object file OBJFILE as you specified. The next user who executes the object program named OBJFILE executes the new object file, even if another user is still executing the old, renamed one.

Use the TACL command FILES to detect ZZBI*nnnn* or ZZNC*nnnn* files and the TACL command PURGE to purge the unwanted ones.

Specifying the Default COPY Library

The default COPY library is the library from which COPY statements read text when no library name is specified in the COPY statement. If you do not specify the default COPY library (as in the compilation examples in the preceding topic), then COPYLIB is the default COPY library. To specify a different default COPY library, such as COPLIB2, include its name in the compilation command:

```
89> ECOBOL /IN XYZ, OUT $SPX/ XYZOBJ, COPLIB2
```

For more information about COPY libraries, see Using COPY and SOURCE Libraries.

Specifying Compiler Directives

Compiler directives can be included in the source program or specified in the compilation command. To specify compiler directives (such as NOFIPS and NOWARN) in the compilation command, put them at the end of the compilation command, as in this example (note the semicolons):

```
90> ECOBOL /IN XYZ, OUT $SPX/ XYZOBJ, COPLIB2;NOFIPS;NOWARN
```

Compiler directives specified in this way are considered to be on line zero of the source program. Most of them stay active until the compiler encounters a conflicting directive later in the source program, but it depends on the directive. For information about specific directives, see Compiler Directives (page 542).

Directives for Syntax Checking Only

If you compile your program with the SYNTAX directive, the compiler only checks its syntax and does not generate object code. Because the compiler does not generate object code, it runs much faster (but only if there are no errors).

```
90> ECOBOL /IN XYZ, OUT $SPX/ XYZOBJ, COPLIB2; SYNTAX
```

Another way to find syntax errors in your program is with the FIXERRS macro, which requires that you compile your program with the ERRORFILE directive. For more information about the FIXERRS macro, see FIXERRS Macro (page 708). For more information about the ERRORFILE directive, see ERRORFILE (page 558).

Specifying Subvolumes to Be Searched for Unqualified Files

In your HP COBOL source program, an unqualified file is a file whose name does not contain a volume and subvolume. It can be a source file (as in a COPY statement) or an object file (as in an ENTER statement).

Two DEFINEs allow you to specify one or more subvolumes for the compiler to search for unqualified files:

- =_SOURCE_SEARCH (for unqualified source files)
- =_OBJECT_SEARCH (for unqualified object files)

These DEFINEs have CLASS attribute SEARCH. Use them by entering ADD DEFINE commands before compiling your program, as in this example. (An ampersand at the end of a line means that the TACL command continues on the next line.)

```
91> ADD DEFINE =_SOURCE_SEARCH, CLASS SEARCH, SUBVOL0&
91> (=_DEFAULTS,$VOL1.SUB2,$VOL1.SUB3)
92> ADD DEFINE =_OBJECT_SEARCH, CLASS SEARCH, SUBVOL0&
92> (=_DEFAULTS,$VOL2.SUB1,$VOL2.SUB2)
93> ECOBOL /IN XYZ, OUT $SPX/ XYZOBJ
```

In the preceding example, the compiler will search for unqualified source files on the default subvolume first, then on $VOL1.SUB2, and finally on $VOL1.SUB3. The compiler will search for unqualified object files on the default subvolume first, then on $VOL2.SUB1, and finally on $VOL2.SUB2.

Suppose that you want to add the subvolume $VOL3.SUB1 to the _SOURCE_SEARCH list. Use an ADD DEFINE command whose SUBVOL*n* parameter has *n* greater than 0 (the value of *n* in the preceding example):

```
94> ADD DEFINE =_SOURCE_SEARCH, CLASS SEARCH, SUBVOL1&
94> ($VOL3.SUB1)
```

Now the compiler will search for unqualified source files on these subvolumes in this order: default subvolume, $VOL1.SUB2, $VOL1.SUB3, $VOL3.SUB1.

Changing the Compilation Environment

You can change these aspects of the compilation environment:

- Volume(s) for Temporary Files
- Compiler Space Allocation

Volume(s) for Temporary Files

By default, the compiler creates its temporary files on the current default volume. If called by the compiler, the supporting processes and the linker create their temporary files on the current default volume.

The PARAM SWAPVOL command specifies the volume on which the compiler and its processes will create temporary files (if possible). It does not determine where the operating system creates the compiler’s own swap file—the Kernel-Managed Swap Facility (KMSF) does that. For more information, see the *Kernel-Managed Swap Facility (KMSF) Manual*.

If you have given the command PARAM SWAPVOL and decide that you do not want it, you can clear it with the CLEAR command before running the compiler; for example:

```
94> PARAM SWAPVOL $COBVOL
95> CLEAR PARAM SWAPVOL
96> ECOBOL /IN XYZ, OUT $SPX/ XYZOBJ
```

Compiler Space Allocation

The PARAM SYMBOL-BLOCKS command specifies how much space the compiler allocates for its symbol dictionary and embedded SQL/MP statements—see PARAM SYMBOL-BLOCKS (page 537).

If you have given the command PARAM SYMBOL-BLOCKS and decide that you do not want it, you can clear it before running the compiler with the CLEAR command; for example:

```
98> PARAM SYMBOL-BLOCKS 1
99> CLEAR PARAM SYMBOL-BLOCKS
100> ECOBOL /IN XYZ, OUT $SPX/ XYZOBJ
```

If you have given several PARAM commands, you can clear all of them before running the compiler with the command CLEAR ALL PARAM; for example:

```
101> PARAM SAMECPU 7
102> PARAM SWAPVOL $COBVOL
103> PARAM SYMBOL-BLOCKS 1
104> CLEAR ALL PARAM
105> ECOBOL /IN XYZ, OUT $SPX/ XYZOBJ
```

Improving Compilation Speed

Under these conditions, these actions can improve compilation speed:

Condition	Action
The compilation summary shows that the maximum symbol table size is less than 256 KB	Use the PARAM SYMBOL-BLOCKS command (see Compiler Space Allocation).
You only want the compiler to check the program's syntax (not generate object code).	Use the SYNTAX directive (see Specifying Compiler Directives).

Launching Compilations to Run Unattended

You can launch a compilation to run unattended with NetBatch, a job management system. For details, see the *NetBatch Manual*.

Using COPY and SOURCE Libraries

A COPY or SOURCE library is an EDIT file that contains one or more blocks of text. Each block is preceded by a SECTION directive that specifies its name and (optionally) its reference format (ANSI or Tandem, the default being Tandem). The section name in the SECTION directive is the one to use in a COPY statement or SOURCE directive.

Example 23-2 COPY or SOURCE Library

```
?SECTION NAMEREC
 01   NAME-REC.
     03 LAST-NAME      PICTURE X(15).
     03 M-I            PICTURE X(1).
     03 FIRST-NAME     PICTURE X(15).
     03 TITLE          PICTURE X(5).
?SECTION STANDARD-LINAGE
     LINAGE IS 55 LINES
       WITH FOOTING AT 45
       LINES AT TOP 5
       LINES AT BOTTOM 6
?SECTION ZOO-FILE
     ASSIGN TO "$A0101.ZOO.ROSTER"
     ORGANIZATION IS INDEXED
     ACCESS MODE IS DYNAMIC
     RECORD KEY IS ANIMAL-NUMBER
     ALTERNATE RECORD KEY IS SPECIES WITH DUPLICATES
     ALTERNATE RECORD KEY IS HABITAT WITH DUPLICATES
     FILE STATUS IS ZOO-STAT
?SECTION EMPLOYEE-NUM-PIC
     PICTURE 9(6).
```

The only difference between a COPY library and a SOURCE library is context—a library that is referenced by a COPY statement is called a COPY library, and a library that is referenced by a

SOURCE directive is called a SOURCE library. The important differences between the COPY statement and the SOURCE directive are:

- The COPY statement can replace specified text-words in the library text with different text-words; the SOURCE directive cannot.
- The SOURCE directive requires the name of the library file; the COPY statement does not, as it can use the default COPY library.
- The SOURCE directive can be in library text (that is, SOURCE directives can be nested); the COPY statement cannot.

To use the library in Example 23-2 as a COPY library, include a line like this in your source program:

```
01 STARTING-EMPLOYEE COPY EMPLOYEE-NUM-PIC.
```

The COPY statement and the period (.) that follows it are replaced. The effect is the same as including this line in your source program:

```
01 STARTING-EMPLOYEE PICTURE 9(6).
```

To use the library in Example 23-2 as a SOURCE library, include a line like this in your source program:

```
?SOURCE file-name (NAMEREC,ZOO-FILE)
```

where *file-name* is the name of the file containing the library text in Example 23-2. The effect is the same as including these lines in your source program:

```
01  NAME-REC.
    03 LAST-NAME      PICTURE X(15).
    03 M-I            PICTURE X(1).
    03 FIRST-NAME     PICTURE X(15).
    03 TITLE          PICTURE X(5).
    ASSIGN TO "$A0101.ZOO.ROSTER"
    ORGANIZATION IS INDEXED
    ACCESS MODE IS DYNAMIC
    RECORD KEY IS ANIMAL-NUMBER
    ALTERNATE RECORD KEY IS SPECIES WITH DUPLICATES
    ALTERNATE RECORD KEY IS HABITAT WITH DUPLICATES
    FILE STATUS IS ZOO-STAT
```

Creating or Altering a COPY or SOURCE Library

You can create a COPY or SOURCE library with an HP editor or with Data Definition Language (DDL). You can alter a COPY or SOURCE library with an HP editor unless it was created with DDL, in which case you must alter it with DDL.

Topics:

- Using an HP Editor
- Using Data Definition Language (DDL)

Using an HP Editor

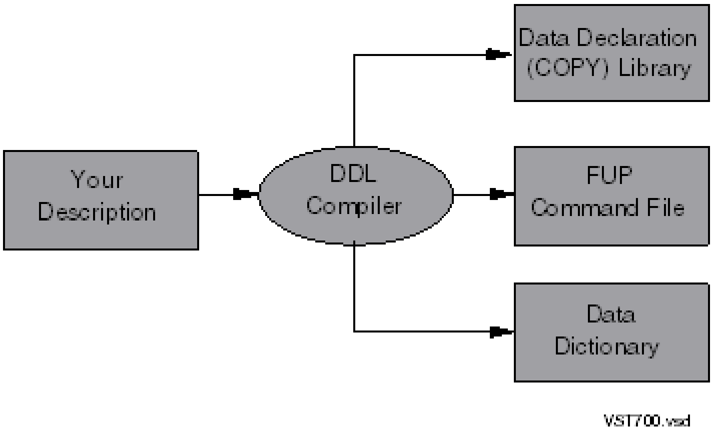
To create or alter a COPY or SOURCE library with either HP editor, EDIT or TEDIT, use the same method as you would use to create or alter an HP COBOL source program (see Creating or Altering an HP COBOL Source Program). Follow the library format described in Library Format (page 514) (see Example 23-2).

Using Data Definition Language (DDL)

Data Definition Language (DDL) is a language with which you describe the data structures of a database. The DDL compiler reads your description and generates a data declaration library (a COPY library) for the database and a File Utility Program (FUP) command file for building the database files. DDL also lets you create and maintain a data dictionary for your database. Because

DDL can create data declarations for other programming languages, it facilitates sharing data structures in mixed-language programs.

Figure 23-3 DDL Input and Output



Topics:

- Describing the Data Structures
- Data Declaration (COPY) Library
- File Utility Program (FUP) Commands
- Data Dictionary

Describing the Data Structures

You describe the data structures of your database with DEFINE statements. Each data item can be an elementary item or a data structure. A data item can be composed of explicit, COBOL-like definitions or it can refer to data items defined earlier in the data dictionary. In the latter case, you define records with RECORD statements. A record definition can include both explicitly defined data items and data items defined earlier in the data dictionary.

In Example 23-3, note the absence of the level number 01.

Example 23-3 Input to the DDL Compiler

```
DEFINE NAME.
 02 LAST-NAME            PIC X(15).
 02 MIDDLE-INITIAL       PIC X(1).
 02 FIRST-NAME           PIC X(15).
END

DEFINE ADDRESS.
 02 STREET-NUMBER        PIC X(6).
 02 STREET-NAME          PIC X(25).
 02 FLAT-NUMBER          PIC X(8).
 02 MUNICIPALITY         PIC X(25).
 02 STATE-OR-PROVINCE    PIC X(25).
 02 NATION               PIC X(25).
 02 POSTAL-CODE          PIC X(11).
END

DEFINE NAME-AND-ADDRESS.
 05 NAME                TYPE *.
 05 ADDRESS             TYPE *.
END

RECORD PERSON-RECORD.       FILE IS "folks" Entry-Sequenced.
 02 NAME-AND-ADDRESS     TYPE *.
 02 OCCUPATION-NUMBER  PIC 9(6).
 02 OCCUPATION-NAME    PIC X(35).
 02 EMPLOYER-NAME      PIC X(35).
 02 EMPLOYER-ADDRESS   TYPE address.
KEY "PN" IS NAME-AND-ADDRESS.NAME.
KEY "EN" IS EMPLOYER-NAME.
END
```

Data Declaration (COPY) Library

Example 23-4 shows the data declaration (COPY) library that the DDL compiler produces from the input shown in Example 23-3. The DDL compiler has assigned (or reassigned) appropriate level numbers—for example:

Data Item	Level Number
NAME	01
STREET-NUMBER in the record ADDRESS	02
STREET-NUMBER in the record NAME-AND-ADDRESS	03
STREET-NUMBER in the record PERSON-RECORD	04

Example 23-4 COPY Library Produced by the DDL Compiler

```
* SCHEMA PRODUCED DATE - TIME : 2/28/93 12:01:42
?SECTION NAME,HP
* Definition NAME created on 02/28/93 at 12:01
  01 NAME.
    02 LAST-NAME                   PIC X(15).
    02 MIDDLE-INITIAL              PIC X(1).
    02 FIRST-NAME                  PIC X(15).
?SECTION ADDRESS,HP
* Definition ADDRESS created on 02/28/93 at 12:01
  01 ADDRESS.
    02 STREET-NUMBER               PIC X(6).
    02 STREET-NAME                 PIC X(25).
    02 FLAT-NUMBER                 PIC X(8).
    02 MUNICIPALITY                PIC X(25).
    02 STATE-OR-PROVINCE           PIC X(25).
    02 NATION                      PIC X(25).
    02 POSTAL-CODE                 PIC X(11).
?SECTION NAME-AND-ADDRESS,HP
* Definition NAME-AND-ADDRESS created on 02/28/93 at 12:01
  01 NAME-AND-ADDRESS.
    02 NAME.
      03 LAST-NAME                 PIC X(15).
      03 MIDDLE-INITIAL            PIC X(1).
      03 FIRST-NAME                PIC X(15).
    02 ADDRESS.
      03 STREET-NUMBER             PIC X(6).
      03 STREET-NAME               PIC X(25).
      03 FLAT-NUMBER               PIC X(8).
      03 MUNICIPALITY              PIC X(25).
      03 STATE-OR-PROVINCE         PIC X(25).
      03 NATION                    PIC X(25).
      03 POSTAL-CODE               PIC X(11).
?SECTION PERSON-RECORD,HP
* Record PERSON-RECORD created on 02/28/93 at 12:01
  01 PERSON-RECORD.
    02 NAME-AND-ADDRESS.
      03 NAME.
        04 LAST-NAME               PIC X(15).
        04 MIDDLE-INITIAL            PIC X(1).
        04 FIRST-NAME                PIC X(15).
      03 ADDRESS.
        04 STREET-NUMBER             PIC X(6).
        04 STREET-NAME               PIC X(25).
        04 FLAT-NUMBER               PIC X(8).
        04 MUNICIPALITY              PIC X(25).
        04 STATE-OR-PROVINCE         PIC X(25).
        04 NATION                    PIC X(25).
        04 POSTAL-CODE               PIC X(11).
    02 OCCUPATION-NUMBER             PIC 9(6).
    02 OCCUPATION-NAME               PIC X(35).
    02 EMPLOYER-NAME                 PIC X(35).
    02 EMPLOYER-ADDRESS.
      03 STREET-NUMBER               PIC X(6).
      03 STREET-NAME                 PIC X(25).
      03 FLAT-NUMBER                 PIC X(8).
      03 MUNICIPALITY                PIC X(25).
      03 STATE-OR-PROVINCE           PIC X(25).
      03 NATION                      PIC X(25).
      03 POSTAL-CODE                 PIC X(11).
```

File Utility Program (FUP) Commands

Example 23-5 shows the FUP command file that the DDL compiler produces from the input shown in Example 23-3. Assuming that the name of the FUP command file is FUPSRC, you create the file FOLKS with this command:

```
110> FUP /IN FUPSRC/
```

The name of the FUP command file is specified in the DDL command FUP; for details, see the *Data Definition Language (DDL) Reference Manual*.

Example 23-5 FUP Command File Produced by the DDL Compiler

```
< SCHEMA PRODUCED DATE - TIME : 2/28/93 12:01:41
< SECTION PERSON-RECORD
< Record PERSON-RECORD created on 02/28/93 at 12:01
RESET
   SET ALTKEY ( "PN", KEYOFF 0, KEYLEN 31, FILE 0 )
   SET ALTKEY ( "EN", KEYOFF 197, KEYLEN 35, FILE 0 )
   SET NO ALTCREATE
   SET ALTFILE ( 0, folks0 )
   SET TYPE E
   SET REC 357
   SET BLOCK 512
CREATE folks
   RESET
   SET TYPE K
   SET KEYLEN 41
   SET REC 41
   SET BLOCK 512
   SET IBLOCK 512
CREATE folks0
```

Data Dictionary

A data dictionary is a set of files that documents the structure and location of each file in your database. You can use the DDL compiler to translate DDL source statements into a data dictionary, and then you can use the DDL compiler to modify the data dictionary and re-create the data declaration (COPY) library. Alternatively, you can keep the DDL source statements in a disk file, and when you want to change a data description, you can recompile the data dictionary. The former procedure is faster (because you are not recompiling the entire data dictionary); the latter procedure is more straightforward.

For complete information about DDL, see the *Data Definition Language (DDL) Reference Manual*.

Including Text From a COPY Library

Text from a COPY library is included in an HP COBOL source program with a COPY statement. Although a COPY statement can appear anywhere in an HP COBOL source program that a character-string or separator can appear, its most common uses are:

- To copy descriptions of records or data structures into the Data Division
- To copy the same small piece of code into several different programs (where “small” means too small to merit being a separately compiled program)
- To copy the occurrence number in an OCCURS clause into more than one data description entry, as if it were a constant

Topics:

- Simple Copying
- Copying With Replacement
- Replacing Substrings
- Copying Into Debugging Lines
- Using Multiple COPY Libraries
- Using COPY Libraries Efficiently

Simple Copying

Simple copying means copying text from a COPY library into a source program. This example uses simple copying to copy the occurrence number in an OCCURS clause into more than one data description entry, as if it were a constant.

Suppose that the present number of sales offices is 24, but you expect it to change. Put this text into a COPY library:

```
?SECTION OFFICES
   24
```

In every table whose length must be the same as the number of sales offices, use a COPY statement; for example:

```
01 RENT RATE          PICTURE 9(6) OCCURS COPY OFFICES. TIMES.
01 EMPLOYEES          OCCURS COPY OFFICES. TIMES.
   03 MANAGEMENT      PICTURE 9(3).
   03 ADMINISTRATIVE  PICTURE 9(3).
   03 SALES           PICTURE 9(3).
   03 SUPPORT         PICTURE 9(3).
   03 OTHER           PICTURE 9(3).
```

Each COPY statement ends with a period. The COPY statement and its period are replaced by the library text:

```
01 RENT RATE          PICTURE 9(6) OCCURS 24 TIMES.
01 EMPLOYEES          OCCURS 24 TIMES.
   03 MANAGEMENT      PICTURE 9(3).
   03 ADMINISTRATIVE  PICTURE 9(3).
   03 SALES           PICTURE 9(3).
   03 SUPPORT         PICTURE 9(3).
   03 OTHER           PICTURE 9(3).
```

If the section OFFICES had been empty, the COPY statements would have been replaced with spaces. If there had been no section named OFFICES, the compiler would have reported an error 72 (Expected `section-name`).

Copying With Replacement

Copying with replacement means copying text from a COPY library, modifying it, and then writing it into a source program. Copying with replacement is achieved with the REPLACING phrase.

Using the preceding example, suppose that the number of sales offices increases from 24 through 27. You have two choices:

- In the COPY library, change 24 to 27:

  ```
  ?SECTION OFFICES
     27
  ```

- In the HP COBOL source program, add REPLACING phrases to the appropriate COPY statements:

  ```
  01 RENT RATE          PICTURE 9(6) OCCURS COPY OFFICES
                          REPLACING 24 BY 27. TIMES.
  01 EMPLOYEES          OCCURS COPY OFFICES
                          REPLACING 24 BY 27. TIMES.
  ...03 MANAGEMENT      PICTURE 9(3).
  ...03 ADMINISTRATIVE PICTURE 9(3).
  ...03 SALES           PICTURE 9(3).
  ...03 SUPPORT         PICTURE 9(3).
  ...03 OTHER           PICTURE 9(3).
  ```

  The resulting source code is:

  ```
  01 RENT RATE          PICTURE 9(6) OCCURS 27 TIMES.
  01 EMPLOYEES          OCCURS 27 TIMES.
  ...03 MANAGEMENT      PICTURE 9(3).
  ...03 ADMINISTRATIVE PICTURE 9(3).
  ...03 SALES           PICTURE 9(3).
  ...03 SUPPORT         PICTURE 9(3).
  ...03 OTHER           PICTURE 9(3).
  ```

  (This COPY library itself does not change.)

Copying with replacement is not the only way to replace text-words in source code; the REPLACE statement is another way. See Replacing Text-Words in an HP COBOL Source Program.

Replacing Substrings

The REPLACING phrase can only replace entire text-words (DAILY-TOTAL but not just TOTAL, for example); however, parenthesized words within pseudo-text are recognized as text-words, allowing you to write statements such as:

```
COPY SUM-IT REPLACING ==(PREFIX)== BY ==DAILY==.
```

Suppose that you want to declare several identifiers that begin with the same prefix (such as DAILY-TOTAL, DAILY-MINIMUM, and DAILY-MAXIMUM), several identifiers that end with the same suffix (such as DAILY-TOTAL, WEEKLY-TOTAL, and MONTHLY-TOTAL), and several identifiers that contain the same strings (such as AVG-DAILY-COST, AVG-MONTHLY-COST, and AVG-ANNUAL-COST).

Example 23-6 Replacing Substrings

Lines in COPY library:

```
?SECTION PREFIX
   77 (PREFIX)-TOTAL       PIC S9(8).99.
   77 (PREFIX)-MINIMUM     PIC S9(8).99.
   77 (PREFIX)-MAXIMUM     PIC S9(8).99.
?SECTION SUFFIX
   77 DAILY-(SUFFIX)       PIC S9(8).99.
   77 WEEKLY-(SUFFIX)      PIC S9(8).99.
   77 MONTHLY-(SUFFIX)     PIC S9(8).99.
```

```
?SECTION PRESUF
   77 (PRE)-DAILY-(SUF)    PIC S9(8).99.
   77 (PRE)-MONTHLY-(SUF)  PIC S9(8).99.
   77 (PRE)-ANNUAL-(SUF)   PIC S9(8).99.
```

Lines in HP COBOL source program:

```
COPY PREFIX REPLACING ==(PREFIX)== BY ==DAILY==.
COPY SUFFIX REPLACING ==(SUFFIX)== BY ==TOTAL==.
COPY PRESUF REPLACING ==(PRE)== BY ==AVG==
                      ==(SUF)== BY ==COST==.
```

HP COBOL source program effectively contains these lines:

```
77 DAILY-TOTAL       PIC S9(8).99.
77 DAILY-MINIMUM     PIC S9(8).99.
77 DAILY-MAXIMUM     PIC S9(8).99.
77 DAILY-TOTAL       PIC S9(8).99.
77 WEEKLY-TOTAL      PIC S9(8).99.
77 MONTHLY-TOTAL     PIC S9(8).99.
77 AVG-DAILY-COST    PIC S9(8).99.
77 AVG-MONTHLY-COST  PIC S9(8).99.
77 AVG-ANNUAL-COST   PIC S9(8).99.
```

Copying Into Debugging Lines

When the source line containing the word COPY is a debug line (has a `D` or `d` in the indicator area), the compiler changes each library line that has a space in its indicator area to a debug line.

Example 23-7 Copying Into Debugging Lines

Source line:

```
D    COPY DEB-DISP.
```

Library lines:

```
?SECTION DEB-DISP
     DISPLAY KINGDOM PHYLUM CLASS ORDER
             FAMILY GENUS SPECIES.
```

Compiled as:

```
D    DISPLAY KINGDOM PHYLUM CLASS ORDER
D            FAMILY GENUS SPECIES.
```

Using Multiple COPY Libraries

There is no limit to the number of COPY libraries the compiler can use in a single compilation. To specify a nondefault COPY library, use an OF or IN phrase in the COPY statement; for example:

```
COPY RECORD1 OF LIBRARY3.
```

(where LIBRARY3 is on the default volume and subvolume) or:

```
COPY RECORD1 IN \NODE4.$VOL2.SUBVOL1.LIBRARY2.
```

(where LIBRARY3 is fully qualified).

Using COPY Libraries Efficiently

You can reduce the time the compiler spends searching for items in COPY libraries by putting the most frequently used items in the front of their COPY libraries. The reason is that the first time the compiler encounters a COPY statement, the compiler opens the default or specified COPY library and reads until it finds the specified item, building a temporary directory of items as it reads them.

Subsequent COPY statements search the directory for items before searching libraries for them; therefore, COPY statements that reference items already entered in the directory compile faster than COPY statements that reference items not yet entered in the directory. If a specified item is not yet in the directory, the compiler resumes reading the library after the last item in the directory, continuing to build the directory.

Including Text From a SOURCE Library

Text from a SOURCE library is included in an HP COBOL source program with a SOURCE directive. A SOURCE directive must be the last directive on its line.

When the compiler reads a SOURCE directive, it suspends reading from its primary source file and begins reading (entire lines) from a specified section of a SOURCE library. The SOURCE directive does not have anything like the COPY statement's REPLACING phrase; it cannot replace library text-words with different text-words.

The SOURCE directive's most common uses are:

- To copy File Descriptions into the Environment Division
- To copy record or group descriptions into the Data Division
- To copy blocks of code lines into the Procedure Division

Example 23-8 Including Text From a Source File

Library file, SRCFILE:

```
?SECTION NAMEREC
 01   NAME-REC.
      03 LAST-NAME       PICTURE X(15).
      03 M-I             PICTURE X(1).
      03 FIRST-NAME      PICTURE X(15).
      03 TITLE           PICTURE X(5).
?SECTION ADDRREC
 01   ADDR-REC.
      03 NUMBER          PICTURE X(5).
      03 STREET          PICTURE X(15).
      03 APT-NUM         PICTURE X(3).
      03 CITY            PICTURE X(15).
      03 STATE           PICTURE X(2).
      03 ZIPCODE         PICTURE X(5).
?SECTION DATEREC
 01   DATE-REC.
      03 MONTH           PICTURE X(9).
      03 DAY             PICTURE X(2).
      03 YEAR            PICTURE X(4).
```

Line in HP COBOL source program:

```
?SOURCE SRCFILE
```

Resulting source code:

```
 01   NAME-REC.
      03 LAST-NAME       PICTURE X(15).
      03 M-I             PICTURE X(1).
      03 FIRST-NAME      PICTURE X(15).
      03 TITLE           PICTURE X(5).
 01   ADDR-REC.
      03 NUMBER          PICTURE X(5).
      03 STREET          PICTURE X(15).
      03 APT-NUM         PICTURE X(3).
      03 CITY            PICTURE X(15).
      03 STATE           PICTURE X(2).
      03 ZIPCODE         PICTURE X(5).
 01   DATE-REC.
      03 MONTH           PICTURE X(9).
      03 DAY             PICTURE X(2).
      03 YEAR            PICTURE X(4).
```

If the HP COBOL source program in Example 23-8 contains the line

```
?SOURCE SRCFILE (NAMEREC,DATEREC)
```

then only the sections NAMEREC and DATEREC are included in the resulting source code, which is:

```
 01   NAME-REC.
      03 LAST-NAME       PICTURE X(15).
      03 M-I             PICTURE X(1).
      03 FIRST-NAME      PICTURE X(15).
      03 TITLE           PICTURE X(5).
 01   DATE-REC.
      03 MONTH           PICTURE X(9).
      03 DAY             PICTURE X(2).
      03 YEAR            PICTURE X(4).
```

Replacing Text-Words in an HP COBOL Source Program

You can replace text-words with other text-words in an HP COBOL source file with a REPLACE statement. A REPLACE statement is active from the time the compiler encounters it until the compiler encounters either another, overriding REPLACE statement or the end of a separately compiled program.

The two Data Divisions in Example 23-9 are equivalent.

Example 23-9 Replacing Text-Words in an HP COBOL Source Program

```
DATA DIVISION.
REPLACE ==OFFICES== BY ==10==
        ==SQ-FT-SIZE== BY ==5==
01 OFFS.
   03 OFFICE-INFO OCCURS OFFICES TIMES.
      05 DISTRICT    PICTURE 99.
      05 SQUARE-FEET PICTURE S9(SQ-FT-SIZE)
PROCEDURE DIVISION.
PERFORM REPORT-OFFICE OFFICE TIMES.
REPLACE OFF.
DATA DIVISION.
01 OFFS.
   03 OFFICE-INFO OCCURS 10 TIMES.
      05 DISTRICT    PICTURE 99.
      05 SQUARE-FEET PICTURE S9(5)
PROCEDURE DIVISION.
PERFORM REPORT-OFFICE 10 TIMES.
```

For restrictions on the use of the REPLACE statement and a detailed explanation of the comparison operation that it uses, see REPLACE Statement (page 516).

Topics:

- Replacing Substrings
- Replacing Text-Words From COPY Libraries
- Replacing Text-Words From SOURCE Libraries

Replacing Substrings

The REPLACE statement can only replace entire text-words (DAILY-TOTAL but not just TOTAL, for example); however, parenthesized words within pseudo-text are recognized as text-words, allowing you to write statements such as:

```
REPLACE ==(PREFIX)== BY ==DAILY==.
```

Suppose that you want to declare several identifiers that begin with the same prefix (such as DAILY-TOTAL, DAILY-MINIMUM, and DAILY-MAXIMUM), several identifiers that end with the same suffix (such as DAILY-TOTAL, WEEKLY-TOTAL, and MONTHLY-TOTAL), and several identifiers that contain the same strings (such as AVG-DAILY-COST, AVG-MONTHLY-COST, and AVG-ANNUAL-COST).

The two sets of source code in Example 23-10 are equivalent.

Example 23-10 Replacing Substrings in an HP COBOL Source Program

```
REPLACE ==(PREFIX)== BY ==DAILY==.
        ==(SUFFIX)== BY ==TOTAL==.
        ==(PRE)== BY ==AVG==
        ==(SUF)== BY ==COST==.
77 (PREFIX)-TOTAL       PIC S9(8).99.
77 (PREFIX)-MINIMUM     PIC S9(8).99.
77 (PREFIX)-MAXIMUM     PIC S9(8).99.
77 DAILY-(SUFFIX)       PIC S9(8).99.
77 WEEKLY-(SUFFIX)      PIC S9(8).99.
77 MONTHLY-(SUFFIX)     PIC S9(8).99.
77 (PRE)-DAILY-(SUF)    PIC S9(8).99.
77 (PRE)-MONTHLY-(SUF)  PIC S9(8).99.
77 (PRE)-ANNUAL-(SUF)   PIC S9(8).99.
77 DAILY-TOTAL       PIC S9(8).99.
77 DAILY-MINIMUM     PIC S9(8).99.
77 DAILY-MAXIMUM     PIC S9(8).99.
77 DAILY-TOTAL       PIC S9(8).99.
77 WEEKLY-TOTAL      PIC S9(8).99.
77 MONTHLY-TOTAL     PIC S9(8).99.
77 AVG-DAILY-COST    PIC S9(8).99.
77 AVG-MONTHLY-COST  PIC S9(8).99.
77 AVG-ANNUAL-COST   PIC S9(8).99.
```

Compare the preceding example to the example in Replacing Substrings.

Replacing Text-Words From COPY Libraries

The compiler processes REPLACE statements after it has processed any COPY statements. REPLACE statements do not change COPY libraries.

In Example 23-11, the compiler substitutes OFFICES for COPY OFFNUM and SQ-FT-SIZE for COPY AREA first and then substitutes 10 for OFFICES and 5 for SQ-FT-SIZE.

Example 23-11 Replacing Text-Words From a COPY Library

Source code:

```
REPLACE ==OFFICES== BY ==10==
        ==SQ-FT-SIZE== BY ==5==
01 OFFS.
   03 OFFICE-INFO OCCURS COPY OFFNUM. TIMES.
      05 DISTRICT    PICTURE 99.
      05 SQUARE-FEET PICTURE S9(COPY AREA.)
REPLACE OFF
```

Lines in default COPY library:

```
?SECTION OFFNUM
   OFFICES
?SECTION AREA
   SQ-FT-SIZE
```

Resulting source code (COPY library is unchanged):

```
01 OFFS.
   03 OFFICE-INFO OCCURS 10 TIMES.
      05 DISTRICT    PICTURE 99.
      05 SQUARE-FEET PICTURE S9(5).
```

If the COPY statement has a REPLACING phrase, the compiler applies the REPLACING phrase before applying the REPLACE statement. For an example of a COPY statement with a REPLACING phrase, see Copying With Replacement.

Replacing Text-Words From SOURCE Libraries

If a REPLACE statement is active when the compiler encounters a SOURCE directive, the REPLACE statement applies to the copy of the library text that the compiler includes in the HP COBOL source program (but not to the original text in the SOURCE library).

Example 23-12 Replacing Text-Words From a SOURCE Library

Source code:

```
REPLACE ==OFFICES== BY ==10==
        ==SQ-FT-SIZE== BY ==5==
?SOURCE SRCFILE
REPLACE OFF
```

Lines in SOURCE library:

```
01 OFFS.
   03 OFFICE-INFO OCCURS OFFICES TIMES.
      05 DISTRICT    PICTURE 99.
      05 SQUARE-FEET PICTURE S9(SQ-FT-SIZE).
```

Resulting source code (SOURCE library is unchanged):

```
01 OFFS.
   03 OFFICE-INFO OCCURS 10 TIMES.
      05 DISTRICT    PICTURE 99.
      05 SQUARE-FEET PICTURE S9(5).
```

Understanding and Controlling the Compiler Listing

By default, the compiler produces a listing that includes:

- Compilation Banner
- Source Program Listing
- Compilation Summary

With compiler directives, you can include one or more of these in the compiler listing:

- Symbol Table Listing
- Symbolic Code Listing

Compiler directives also allow you to suppress parts of the compiler listing (see Table 23-1).

The minimum compiler listing, produced when you compile a program with the SUPPRESS directive, includes only a banner, any diagnostics and the source lines that caused them, and a summary.

When you compile several separately compiled programs at once (each of which could include nested programs), the listing includes one banner and one summary. After the banner, the listing includes a source program listing, symbol table, and symbolic code listing for each program that requested them (by default or directive).

In Table 23-1, the compiler listing parts are in the order in which they appear in the compiler listing.

Table 23-1 Compiler Listing Parts and the Directives That Control Them

Compiler Listing Part	Directive That Lists Part	Directives That Suppress Part
Compilation banner	None—always listed	None
Source program listing	LIST (default)	• NOLIST • SUPPRESS
COPY statements	SHOWCOPY (default)	• NOSHOWCOPY • NOLIST • SUPPRESS
Diagnostic messages*	WARN (default)	NOWARN for warning messages, none for error and failure messages
Symbol table	MAP	• NOMAP • NOLIST • SUPPRESS
Symbolic code listing	INNERLIST	• NOINNERLIST • NOLIST • SUPPRESS
Compilation summary	None—always listed	None
*Included in source program listing		

For further information about any compiler listing part, see the appropriate following topic. The topics are in the order that they appear in Table 23-1.

Compilation Banner

The ECOBOL compilation banner consists of this information:

- Page number of compiler listing, date and time at which compilation began (line 1)
- Version of the compiler (line 4)
- Copyright information (line 5)
- Directives included on compiler command line, if any (line 7)

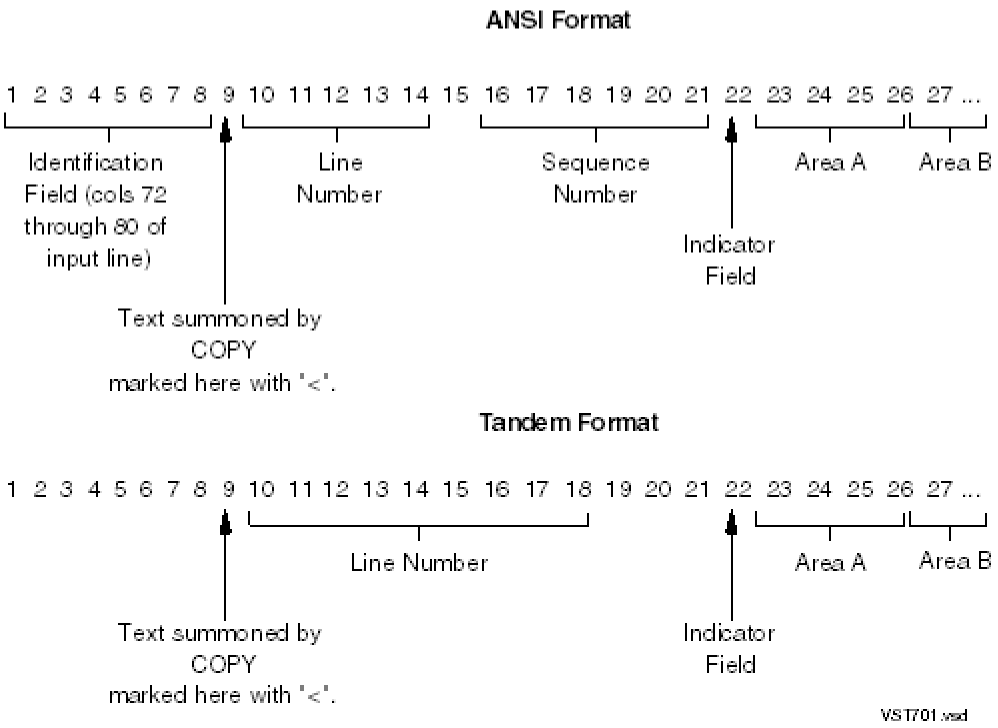
Example 23-13 ECOBOL Compilation Banner

```
Page 1    2004 September 29, 12:14:31
COBOL - T0356H01 - (10SEP2004)
(C)1997 Tandem (C)2003 Hewlett Packard Development Company, L.P.
      Directives: RUNNABLE;OPTIMIZE 1
```

Source Program Listing

The compiler lists your entire source program, unless you specify otherwise. The format of the listing depends in part on the current reference format, ANSI or Tandem. Figure 23-4 compares the two listing formats.

Figure 23-4 Source Program Listing Formats



The two formats are aligned such that, even if your program mixes ANSI and Tandem formats, the COPY and comment indicators and areas A and B are aligned.

Topics:

- Identification Field
- Line Number
- Sequence Number
- Indicator Area and Beyond
- Text Altered by REPLACE Statement
- Text Retrieved by COPY Statement
- Text Retrieved by SOURCE Directive
- SQL/MP and SQL/MX Statements

Identification Field

The identification field is copied from columns 72 through 80 of ANSI format source lines. Some installations use these columns for marking revisions to the program.

Line Number

When the file being listed is an EDIT file, the line number in the listing is the EDIT file line number. If the file is in ANSI format, the fractional part of the number, if any, is omitted.

When the file being listed is not an EDIT file, the line number in the listing is a “synthetic” line number (an integer that the compiler generates). Every time the compiler reads a line of input from the file, it increments a counter by 1.

Sequence Number

When the file is in ANSI format, the compiler copies columns 1 through 6 of the input line to this field. Some COBOL implementations use this field as a sequence number, and if the values are not in ascending order, these implementations report a diagnostic. HP COBOL does no such reporting.

Indicator Area and Beyond

Beginning with the indicator area, the two formats are the same. If, in Tandem format, the input line exceeds 120 characters, it will not fit on the 132-character print line. The compiler wraps such lines onto the next line, beginning at print column 1.

Text Altered by REPLACE Statement

Each REPLACE statement is listed as the compiler encounters it. Each line on which a REPLACE statement has made a change is listed after the change has been made. If the line is too long, the compiler truncates it after a reasonable token and writes as many additional lines (with the same line number) as are necessary to deliver the entire replaced text.

Text Retrieved by COPY Statement

The compiler marks any line containing text that it copied from a COPY library by displaying a less than (>) character in printer column 9.

The compiler directives SHOWCOPY and NOSHOWCOPY determine how the compiler lists lines containing COPY statements. The default, SHOWCOPY, causes the compiler to list both the line containing the COPY statement (as a comment) and the line that results from the COPY statement. The alternative, NOSHOWCOPY, causes the compiler to list only the line that results from the COPY statement. NOSHOWCOPY makes it hard to determine from the listing where the copied lines came from.

The HP COBOL COPY logic attempts to keep your listings readable by combining the original source text line with the copied library text. HP COBOL COPY logic handles these cases:

- Column Position of Library Text Characters

  The compiler preserves the column position of library text characters; it does not shift text to the left or right when it attempts this combining operation; for example:

  ```
  Source line:      MOVE COPY SOURCE1. TO X.
  Library lines:    ?SECTION SOURCE1
                         ABLE
  Printed line:     MOVE ABLE          TO X.
  ```

  If “ABLE” in the library line begins in column 1, you get:

  ```
  Printed lines:    MOVE
                    ABLE               TO X.
  ```

  If “ABLE” in the library line begins in column 22, you get:

  ```
  Printed lines:    MOVE                 ABLE
                                       TO X.
  ```

- Text Before the Word COPY

  If the source line on which the COPY statement begins has text before the word COPY, the compiler tries to combine the original source line with the first line of the library text. This strategy succeeds only if at least one space separates the first nonspace character of the library text line from the last nonspace character of the source line. The combined line retains the line number of the source line that contains the COPY statement.

- Text After the Word COPY

  If the source line on which the COPY statement ends has text following the terminating period, the compiler tries to combine the last line of library text with the text that follows the COPY statement’s period. This strategy succeeds only if at least one space separates the last nonspace character of the library line from the first nonspace character of the remainder of the source line. The combined line retains the line number of the source line that contains the COPY statement.

- ANSI Reference Format

  If the current reference format is ANSI, the listed line begins with the identification field of the appended source line. Example 23-15 shows this behavior.

The compiler does not attempt to combine:

- Lines of different reference formats
- Debug and nondebugging lines
- Library text lines that do not contain a space, `D`, or `d` in the indicator area

When the NOSHOWCOPY directive is absent, the compiler lists the line containing a COPY statement as a comment before listing the line that results from the COPY statement substitution. The error-reporting mechanism can sometimes report a syntax error against the copied text and point to the COPY statement in the comment, as in Example 23-14.

Example 23-14 Diagnostic Reported Against a Copied Line

```
            35       *     MOVE COPY SOURCE1. TO X.
** LINE    35                   ^
** ERROR 240 ** SYNTAX ERROR DETECTED AT TOKEN:  ABLE
            35             MOVE ABLE          TO X.
** LINE    35                                    ^
** WARNING 244 ** PARSING RESUMED AT TOKEN:  X
```

Example 23-15 ANSI Format COPY Expansion

Source Program Lines:

```
PD0023 A.                                                               ANSIFMT
PD0024      COPY MOVE1.                                                 ANSIFMT
PD0024      COPY MOVE2.                                                 ANSIFMT
PD0026      MOVE COPY SOURCE1. TO X.                                    ANSIFMT
PD0027      MOVE COPY SOURCE2. TO X.                                    ANSIFMT
PD0028      MOVE COPY SOURCE3. TO X.                                    ANSIFMT
PD0029      ADD COPY ADD1. COPY ADD2. COPY ADD3. COPY ADD4. TO ANSR.    ANSIFMT
PD0030D     DISPLAY COPY SHOW-LIST OF SOMELIB. .                        ANSIFMT
PD0031      STOP RUN.                                                   ANSIFMT
```

COPY Library Lines:

```
?SECTION MOVE1,ANSI
            MOVE ABLE TO BAKER.                                         MOVE1
?SECTION MOVE2,HP
  MOVE ABLE TO BAKER.
?SECTION SOURCE1,ANSI
                 ABLE                                                   SOURCE1
?SECTION SOURCE2,ANSI
            ABLE                                                        SOURCE2
?SECTION SOURCE3,ANSI
                                 ABLE                                   SOURCE3
?SECTION ADD1,ANSI
000001      ANDY                                                        ADD1
?SECTION ADD2,ANSI
000101      BARBARA                                                     ADD2
?SECTION ADD3,ANSI
?SECTION ADD4
                                       CAL
```

Listing:

```
ANSIFMT     33 PD0023 A.
ANSIFMT     34 PD0024*     COPY MOVE1.
MOVE1   <   34             MOVE ABLE TO BAKER.
ANSIFMT     35 PD0024*     COPY MOVE2.
        <   13         MOVE ABLE TO BAKER.
ANSIFMT     36 PD0026*     MOVE COPY SOURCE1. TO X.
ANSIFMT <   36 PD0026      MOVE ABLE          TO X.
ANSIFMT     37 PD0027*     MOVE COPY SOURCE2. TO X.
ANSIFMT     37 PD0027      MOVE
ANSIFMT <   38             ABLE               TO X.
ANSIFMT     39 PD0028*     MOVE COPY SOURCE3. TO X.
SOURCE3 <   39 PD0028      MOVE                 ABLE
ANSIFMT     40 PD0028                         TO X.
ANSIFMT     41 PD0029*     ADD COPY ADD1. COPY ADD2. COPY ADD3. COPY ADD4. TO ANSR.
ANSIFMT     41 PD0029      ADD
ANSIFMT <   42 000001*     ANDY           COPY ADD2. COPY ADD3. COPY ADD4. TO ANSR.
ANSIFMT <   42 000001      ANDY
ANSIFMT <   43 000101*     BARBARA                   COPY ADD3. COPY ADD4. TO ANSR.
ANSIFMT <   43 000101*     BARBARA                              COPY ADD4. TO ANSR.
ANSIFMT <   43 000101      BARBARA                    CAL                  TO ANSR.
ANSIFMT     44 PD0030*     DISPLAY COPY SHOW-LIST OF SOMELIB. .
SHOWLIST<   44 PD0030D     DISPLAY ABLE BAKER CHARLENE DELTA
ANSIFMT <   45       D             ANDY BARBARA               .
ANSIFMT     46 PD0031      STOP RUN.
```

Text Retrieved by SOURCE Directive

The compiler does not mark lines containing text that it copied from a SOURCE library.

The compiler directives SHOWFILE and NOSHOWFILE determine whether the compiler listing indicates when the compiler stops reading from one source file and starts reading from another. The default, NOSHOWFILE, prevents the compiler from reporting when it switches from one source file to another. The alternative, SHOWFILE, causes the compiler to report each time it starts reading from a new source file by printing a line of the form:

```
Source file:  [n]  filename  yyyy-mm-dd  hh:mm:ss
```

The *`n`* is the ordinal number that the compiler assigned to the source file. Initially, *`n`* is 1, and the compiler increments *`n`* by 1 every time it encounters a SOURCE directive. The *`filename`* is the fully qualified name of the new source file. The remainder of the line is the date and time when that file was last modified.

SQL/MP and SQL/MX Statements

SQL/MP or SQL/MX statements embedded in your HP COBOL program always appear in the compiler listing, but their format depends on whether you used the preprocessor to produce the object file.

For SQL/MP statements, using the preprocessor is optional (see the *SQL/MP Programming Manual for COBOL*).

For embedded SQL/MX statements, you always use the preprocessor to produce the object file (see the *SQL/MX Programming Manual for C and COBOL*).

If you used the preprocessor, the SQL/MP or SQL/MX commands in your HP COBOL source program appear in the compiler listing in their processed form (that is, as SQL/MP or SQL/MX data structures, PERFORM statements, and calls to TAL routines). If the compiler detects an error in a processed SQL/MP or SQL/MX statement, it prints the error message after the offending SQL/MP or SQL/MX data structure, PERFORM statement, or TAL routine call.

If you do not use the preprocessor, the SQL/MP or SQL/MX commands in your HP COBOL source program appear in the compiler listing exactly as they appear in the source program. SQL/MP or SQL/MX data structures, PERFORM statements, or calls to TAL routines that the compiler generated do not appear in the compiler listing (except called or included data structures). If the compiler detects an error in an embedded SQL/MP or SQL/MX statement, it prints the error message after the offending SQL/MP or SQL/MX statement itself.

Diagnostic Messages

Table 23-2 Compiler Diagnostic Messages in the Compiler Listing

Diagnostic Message Category	Subcategory	Numbered	Suppressable	When Printed	Can Be Documented in Error File
Compiler	Warning	Yes	Yes	After detection	Yes
	Error	Yes	No	After detection	Yes
	Failure	Yes	No	After detection	No
Linker (if compiler calls it)	Informational	Yes	Yes	After all compiler diagnostic messages have been printed	No

Topics:

- Compiler Warning Messages
- Compiler Error Messages
- Compiler Failure Messages
- Error Files

For a list of all the compiler diagnostic messages and their probable causes, see Chapter 48: Compiler Diagnostic Messages (page 1133).

Compiler Warning Messages

A warning message reports a questionable condition. A warning does not prevent the generation of code. You can suppress warning messages with the NOWARN directive.

Example 23-16 Warning Message

```
** Warning  25 **  Blank continuation line
```

Table 23-3 Compiler Directives That Produce Optional Warnings

Directive	Produces warnings for ...
DIAGNOSE-74	Statements that might behave differently in HP COBOL and COBOL 74
DIAGNOSE-85	Statements that might behave differently when compiled by the COBOL85 and NM ECOBOL compilers
DIAGNOSEALL	Multiple references to undefined identifiers (instead of the first one only)
FIPS	Statements as required by the Federal Information Processing Standard (FIPS)
SUBSET	HP extensions to ANSI COBOL and obsolete elements

For more information about the directives in Table 23-3, see Compiler Directives (page 542).

Compiler Error Messages

An error message reports a serious violation of HP COBOL syntax or semantics. At the first error message, the compiler stops generating code for the current program unit and deletes any code that it has already generated for that program unit. You cannot suppress error messages.

Example 23-17 Error Message

```
** Error 44 **  Syntax error detected at token COUNTER
```

Like many error messages, this example reports the identifier that the compiler was processing or seeking when it detected the error (in this case, COUNTER).

Although the compiler stops generating code when it encounters the first error, it continues with the compilation, reporting as many diagnostic messages as it can. Because an error can leave the compiler with incorrect or incomplete information, the compiler might report errors in later statements that are actually correct. (The later errors are side effects of the earlier ones.) If you do not understand certain error messages, fix the errors you do understand and the others might disappear or be expressed in an understandable way in the next compilation.

If the compiler finds errors in any program in the IN file, it does not produce any object programs (that is, it produces no object file).

Compiler Failure Messages

A failure message reports a condition so severe that the compiler is unable to continue. Any code that the compiler has already generated during the current compilation session is lost. You cannot suppress failure messages.

Example 23-18 Failure Message

```
*** Failure:
--> message-text [Warning message-number]
```

The brackets are part of the message, not indicators that the bracketed material is optional.

Error Files

The compiler directive ERRORFILE causes the compiler to document its warning and error messages in an error file. For each compiler warning or error message, the compiler writes one record to the error file. Each record contains this information:

- The fully qualified, local file name of the source file in which the warning or error occurred

  If you specified a DEFINE name for the name of the source file, then this entry is the file name that the DEFINE produced.

- The line number that the HP editor (EDIT or TEDIT) assigned to the line in which the warning or error occurred
- The column number in which the warning or error occurred
- The warning or error message text

The error file is not an EDIT file; it is of type 106. You can use it only with the TACL macro FIXERRS. For information about the FIXERRS macro, see FIXERRS Macro (page 708).

In this example, the error file is ERRORS on the default volume and subvolume:

```
90> COBOL85 /IN XYZ, OUT $SPX/ XYZOBJ; ERRORFILE ERRORS
90>     NM ECOBOL /IN XYZ, OUT $SPX/ XYZOBJ; ERRORFILE ERRORS
```

If the file ERRORS does not exist, the compiler creates an entry-sequenced file of type 106 and names it ERRORS. If ERRORS does exist, and is of type 106, the compiler replaces its contents with the new error file. If ERRORS exists but is not of type 106, the compiler terminates with the message:

```
ERRORFILE not created
```

Symbol Table Listing

The compiler lists the symbol table if you explicitly request it with the MAP directive. The symbol table lists, in alphabetic order, each identifier in the program. For each identifier, the listing shows:

- The group(s) of which it is a member
- The type of entity it identifies (its level number)
- The address (including possible offset) with which it is associated
- The size of the entity it identifies, if applicable
- The category of the entity it identifies (such as numeric or alphanumeric)
- The usage of the entity it identifies (such as COMPUTATIONAL or DISPLAY)

Most identifiers in a program identify data items. Example 23-19 shows a few lines of a symbol table listing of the program in Example 15-13: INTEGER-OF-DATE Function (page 675).

Example 23-19 Symbol Table Listing

```
Example of Symbol Table Listing
...
CPU                             05 NM  COMP    OF CPU-PIN
                                H"0        H"2
CPU-PIN                         01 AN  GROUP
                                H"0        H"8
CREATOR-ACCESSOR-ID             01 NM  COMP
                                H"0        H"2
CREATOR-EDITED                  01 AN  GROUP
                                H"0        H"15
CREATOR-GROUP                   05 NME DISPLY  OF CREATOR-EDITED
                                H"6        H"3
CREATOR-MEMBER                  05 NME DISPLY  OF CREATOR-EDITED
                                H"12       H"3
ERROR-RETURN                    01 NM  DISPLY
                                H"0        H"2
EXPLAIN-MYSELF                  PARAGRAPH
                                H"0        H"0
HOME-TERMINAL                   01 AN  DISPLY
                                H"0        H"18
HOME-TERMINAL-LEN               01 NM  NATIVE
                                H"0        H"2
LEFT-BYTE                       05 AN  DISPLY  OF CONSECUTIVE-BYTES
                                H"0        H"1
...
```

Symbolic Code Listing

The compiler lists the instructions of the object program in symbolic (mnemonic) form after each source statement if you explicitly request it with the INNERLIST directive (see Example 23-20).

NOTE: In a symbolic code listing for a very large program, lines at the end of the program have asterisks in place of line numbers.

Compilation Summary

Example 23-20 Symbolic Code Listing

```
COBOL - T0356H01 - (10SEP2004)
(C)1997 Tandem (C)2003 Hewlett Packard Development Company, L.P.
2004 October 1, 17:22:28
     Directives: -Woptimize=0
       1.       ?OPTIMIZE 0,INNERLIST
```

```
2.        IDENTIFICATION DIVISION.
3.        PROGRAM-ID. TESTT.
3.              alloc   r32=ar.pfs,0,17,8,0
3.              add     sp=-928,sp
3.              nop
3.              add     r27=768,sp
3.              stf.spill       [r27]=f2,32
3.              nop
3.              add     r26=784,sp
3.              stf.spill       [r26]=f3,32
3.              nop
3.              stf.spill       [r27]=f4,32
3.              stf.spill       [r26]=f5,32
3.              nop
3.              stf.spill       [r27]=f16,32
3.              stf.spill       [r26]=f17,32
3.              nop
3.              stf.spill       [r27]=f18,32
3.              stf.spill       [r26]=f19,32
3.              nop
3.              stf.spill       [r27]=f20,32
3.              nop
3.              nop
3.              stf.spill       [r26]=f21
3.              stf.spill       [r27]=f22
3.              mov     r34=gp
3.              nop
3.              nop
3.              mov     r33=b0
3.              mov     r35=r0
3.              mov     r36=r0
3.              mov     r37=r0
3.              nop
3.              nop
3.              mov     r38=r0
3.              mov     r39=r0
3.              mov     r40=r0
3.              mov     r41=r0
3.              nop
3.              nop
3.              mov     r42=r0
3.              add     r43=16,sp
3.              st8     [r43]=r0
3.              nop
3.              add     r44=24,sp
3.              st8     [r44]=r0
3.              nop
3.              add     r45=32,sp
3.              st8     [r45]=r0
3.              nop
3.              add     r46=40,sp
3.              st8     [r46]=r0
3.              nop
3.              add     r20=496,sp
3.              st8     [r20]=r34
3.              nop
3.              add     r20=504,sp
3.              st8     [r20]=r35
3.              nop
3.              add     r20=512,sp
3.              st8     [r20]=r36
3.              nop
3.              add     r20=520,sp
3.              st8     [r20]=r37
3.              nop
3.              add     r20=528,sp
3.              st8     [r20]=r38
3.              nop
3.              add     r20=536,sp
3.              nop
3.              nop
3.              st8     [r20]=r39
3.              add     r20=544,sp
3.              nop
3.              st8     [r20]=r40
3.              add     r20=552,sp
3.              nop
3.              st8     [r20]=r41
3.              add     r20=560,sp
3.              nop
3.              st8     [r20]=r42
```

```
3.              add     r20=504,sp
3.              nop
3.              ld8     r34=[r20]
3.              nop
3.              mov     r49=r34
3.              add     r20=512,sp
3.              ld8     r35=[r20]
3.              nop
3.              nop
3.              nop
3.              mov     r50=r35
3.              add     r20=520,sp
3.              ld8     r36=[r20]
3.              nop
3.              nop
3.              nop
3.              mov     r51=r36
3.              add     r20=528,sp
3.              ld8     r37=[r20]
3.              nop
3.              nop
3.              nop
3.              mov     r52=r37
3.              add     r20=536,sp
3.              ld8     r38=[r20]
3.              nop
3.              nop
3.              nop
3.              mov     r53=r38
3.              add     r20=544,sp
3.              ld8     r39=[r20]
3.              nop
3.              nop
3.              nop
3.              mov     r54=r39
3.              add     r20=552,sp
3.              ld8     r40=[r20]
3.              nop
3.              nop
3.              nop
3.              mov     r55=r40
3.              add     r20=560,sp
3.              ld8     r41=[r20]
3.              nop
3.              mov     r56=r41
3.              nop
3.              br.call.sptk    b0=COBLIB_INITIALIZATION_COMPLETE_#
3.              add     r20=496,sp
3.              ld8     r34=[r20]
3.              nop
3.              nop
3.              nop
3.              mov     gp=r34
3.              add     r35=64,sp
3.              st4     [r35]=r0
3.              nop
3.              add     r36=@ltoff(__PUCB__TESTT#),gp
3.              ld8     r37=[r36]
3.              nop
3.              add     r38=30,r37
3.              ld2     r39=[r38]
3.              nop
3.              nop
3.              nop
3.              zxt2    r40=r39
3.              add     r41=32768,r0
3.              and     r42=r40,r41
3.              nop
3.              cmp4.ne p8,p0=r42,r0
3.              add     r19=0,r0
3.      (p8)    add     r19=1,r0
3.              add     r20=568,sp
3.              st8     [r20]=r19
3.              nop
3.              nop
3.              nop
3.      (p8)    br.cond.dpnt.many       .b1_4
3.              add     r34=@ltoff(__PUCB__TESTT#),gp
3.              ld8     r35=[r34]
3.              nop
3.              add     r36=30,r35
```

```
 3.             ld2     r37=[r36]
 3.             nop
 3.             nop
 3.             nop
 3.             zxt2    r38=r37
 3.             add     r39=46080,r0
 3.             or      r40=r38,r39
 3.             nop
 3.             nop
 3.             nop
 3.             zxt2    r41=r40
 3.             add     r42=@ltoff(__PUCB__TESTT#),gp
 3.             ld8     r43=[r42]
 3.             nop
 3.             add     r44=30,r43
 3.             nop
 3.             nop
 3.             st2     [r44]=r41
 4.     ENVIRONMENT DIVISION.
 5.       CONFIGURATION SECTION.
 6.         SOURCE-COMPUTER.  ABD.
 7.         OBJECT-COMPUTER.  ABD.
 8.     DATA DIVISION.
 9.       WORKING-STORAGE SECTION.
10.         01 D PICTURE 9 USAGE COMPUTATIONAL.
11.         01 STD-DATE PICTURE 9(8) USAGE COMPUTATIONAL.
12.     PROCEDURE DIVISION.
13.     STARTT.
14.       PERFORM VARYING STD-DATE FROM 20041004 BY 1 UNTIL STD-DATE > 20041010
14.             movl    r45=0x00131cd2c
14.             add     r46=@gprel(pack_TESTT_STD-DATE_#),gp
14.             st4     [r46]=r45
14.             nop
14.     .b1_5:
14.             add     r34=@gprel(pack_TESTT_STD-DATE_#),gp
14.             nop
14.             nop
14.             ld4     r35=[r34]
14.             movl    r36=0x00131cd32
14.             cmp4.gtu        p8,p0=r35,r36
14.             add     r19=0,r0
14.     (p8)    add     r19=1,r0
14.             add     r20=576,sp
14.             st8     [r20]=r19
14.             nop
14.             nop
14.             nop
14.     (p8)    br.cond.dpnt.many       .b1_8
15.           DISPLAY STD-DATE
15.             nop
15.             nop
15.             mov     r34=r0
15.             add     r35=@gprel(pack_TESTT_STD-DATE_#),gp
15.             ld4     r36=[r35]
15.             nop
15.             nop
15.             nop
15.             zxt4    r37=r36
15.             add     r38=128,r0
15.             add     r39=472,sp
15.             add     r40=r39,r34
15.             add     r41=8,r0
15.             mov     r49=r38
15.             mov     r50=r37
15.             nop
15.             nop
15.             mov     r51=r40
15.             mov     r52=r41
15.             add     r20=584,sp
15.             nop
15.             st8     [r20]=r34
15.             nop
15.             br.call.sptk    b0=_SharedMilli_CQA#
15.             add     r20=496,sp
15.             ld8     r34=[r20]
15.             nop
15.             mov     gp=r34
15.             movl    r35=0xd000000000000000
15.             nop
15.             nop
15.             add     r36=472,sp
```

```
15.                add     r20=584,sp
15.                ld8     r37=[r20]
15.                nop
15.                nop
15.                nop
15.                add     r38=8,r37
15.                mov     r39=r0
15.                mov     r40=r0
15.                mov     r41=r0
15.                nop
15.                nop
15.                mov     r42=r0
15.                mov     r43=r0
15.                add     r44=16,sp
15.                nop
15.                st8     [r44]=r0
15.                add     r45=24,sp
15.                nop
15.                st8     [r45]=r0
15.                add     r46=32,sp
15.                nop
15.                st8     [r46]=r0
15.                add     r47=40,sp
15.                nop
15.                st8     [r47]=r0
15.                mov     r49=r35
15.                mov     r50=r36
15.                mov     r51=r38
15.                nop
15.                nop
15.                nop
15.                nop
15.                mov     r52=r39
15.                mov     r53=r40
15.                mov     r54=r41
15.                mov     r55=r42
15.                mov     r56=r43
15.                nop
15.                br.call.sptk    b0=COBLIB_DISPLAY_#
15.                add     r20=496,sp
15.                ld8     r34=[r20]
15.                nop
15.                mov     gp=r34
16.     MOVE FUNCTION REM (FUNCTION INTEGER-OF-DATE (STD-DATE) 7) TO D
16.                movl    r35=0xc000000000000000
16.                add     r36=@gprel(pack_TESTT_STD-DATE_#),gp
16.                ld4     r37=[r36]
16.                nop
16.                nop
16.                nop
16.                sxt4    r38=r37
16.                add     r39=2,r0
16.                mov     r40=r0
16.                mov     r41=r0
16.                nop
16.                nop
16.                mov     r42=r0
16.                mov     r43=r0
16.                mov     r44=r0
16.                add     r45=16,sp
16.                st8     [r45]=r0
16.                add     r46=24,sp
16.                nop
16.                st8     [r46]=r0
16.                add     r47=32,sp
16.                nop
16.                st8     [r47]=r0
16.                add     r48=40,sp
16.                nop
16.                st8     [r48]=r0
16.                mov     r49=r35
16.                mov     r50=r38
16.                mov     r51=r39
16.                nop
16.                nop
16.                nop
16.                nop
16.                mov     r52=r40
16.                mov     r53=r41
16.                mov     r54=r42
16.                mov     r55=r43
```

```
16.              mov     r56=r44
16.              nop
16.              br.call.sptk    b0=COBLIB_FUNC_DATES_#
16.              add     r20=496,sp
16.              ld8     r34=[r20]
16.              nop
16.              nop
16.              nop
16.              mov     gp=r34
16.              mov     r35=r8
16.              add     r36=128,r0
16.              zxt4    r37=r35
16.              add     r38=7,r0
16.              nop
16.              mov     r49=r36
16.              mov     r50=r37
16.              mov     r51=r38
16.              br.call.sptk    b0=_SharedMilli_DivRem#
16.              add     r20=496,sp
16.              ld8     r34=[r20]
16.              nop
16.              mov     gp=r34
16.              mov     r35=r10
16.              sxt4    r36=r35
16.              nop
16.              nop
16.              sxt4    r37=r36
16.              add     r38=10,r0
16.              cmp.eq  p8,p0=0,r38
16.              nop
16.              nop
16.              nop
16.      (p8)    break   1
16.              setf.sig        f6=r37
16.              add     r39=10,r0
16.              nop
16.              setf.sig        f7=r39
16.              nop
16.              nop
16.              nop
16.              fcvt.xf f2=f6
16.              nop
16.              nop
16.              fcvt.xf f3=f7
16.              nop
16.              nop
16.              frcpa.s1        f4,p9=f2,f3
16.              add     r40=65501,r0
16.              setf.exp        f5=r40
16.              nop
16.              nop
16.              nop
16.      (p9)    fma.s1  f16=f2,f4,f0
16.              nop
16.              nop
16.      (p9)    fnma.s1 f17=f3,f4,f1
16.              nop
16.              nop
16.      (p9)    fma.s1  f18=f17,f16,f16
16.              nop
16.              nop
16.      (p9)    fma.s1  f19=f17,f17,f5
16.              nop
16.              nop
16.      (p9)    fma.s1  f4=f19,f18,f18
16.              nop
16.              nop
16.              mov     f20=f4
16.              nop
16.              nop
16.              fcvt.fx.trunc.s1        f21=f20
16.              nop
16.              nop
16.              xma.l   f22=f7,f21,f0
16.              nop
16.              getf.sig        r41=f22
16.              sub     r42=r37,r41
16.              nop
16.              mov     r43=r42
16.              cmp4.ge p10,p0=r43,r0
16.              add     r19=0,r0
```

```
16.              nop
16.              nop
16.              xma.l   f22=f7,f21,f0
16.              nop
16.              getf.sig        r41=f22
16.              sub     r42=r37,r41
16.              nop
16.              mov     r43=r42
16.              cmp4.ge p10,p0=r43,r0
16.              add     r19=0,r0
16.              nop
16.              nop
16.      (p8)    add     r19=1,r0
16.              add     r20=592,sp
16.              st8     [r20]=r19
16.              add     r19=0,r0
16.              nop
16.              nop
16.      (p9)    add     r19=1,r0
16.              add     r20=600,sp
16.              st8     [r20]=r19
16.              nop
16.              add     r20=608,sp
16.              st8     [r20]=r42
16.              nop
16.              add     r20=616,sp
16.              st8     [r20]=r43
16.              add     r19=0,r0
16.              nop
16.              nop
16.      (p10)   add     r19=1,r0
16.              add     r20=624,sp
16.              st8     [r20]=r19
16.              nop
16.              nop
16.              nop
16.      (p10)   br.cond.dpnt.many       .b1_14
16.              add     r20=608,sp
16.              ld8     r34=[r20]
16.              nop
16.              sub     r35=r0,r34
16.              nop
16.              sxt4    r36=r35
16.              add     r20=632,sp
16.              st8     [r20]=r36
16.              nop
16.              nop
16.              nop
16.              br.cond.sptk.many       .b1_15
16.      .b1_14:
16.              add     r20=616,sp
16.              ld8     r34=[r20]
16.              nop
16.              nop
16.              nop
16.              mov     r35=r34
16.              add     r20=632,sp
16.              st8     [r20]=r35
16.              nop
16.      .b1_15:
16.              add     r20=632,sp
16.              ld8     r34=[r20]
16.              nop
16.              nop
16.              zxt4    r35=r34
16.              zxt2    r36=r35
16.              add     r37=@gprel(pack_TESTT_D_#),gp
16.              st2     [r37]=r36
17.            IF D = 0 THEN DISPLAY "  SUNDAY" ELSE
17.              nop
17.              add     r38=@gprel(pack_TESTT_D_#),gp
17.              ld2     r39=[r38]
17.              nop
17.              nop
17.              zxt2    r40=r39
17.              nop
17.              cmp4.ne p8,p0=r40,r0
17.              add     r19=0,r0
17.      (p8)    add     r19=1,r0
17.              add     r20=640,sp
17.              st8     [r20]=r19
```

```
17.              nop
17.              nop
17.              nop
17.      (p8)    add     r19=1,r0
17.              add     r20=640,sp
17.              st8     [r20]=r19
17.              nop
17.              nop
17.              nop
17.      (p8)    br.cond.dpnt.many       .b1_16
17.              mov     r34=r0
17.              add     r35=472,sp
17.              add     r36=r35,r34
17.              add     r37=@gprel($cob_internal$6#),gp
17.              add     r38=8,r0
17.              mov     r49=r36
17.              mov     r50=r37
17.              mov     r51=r38
17.              add     r20=648,sp
17.              st8     [r20]=r34
17.              nop
17.              br.call.sptk    b0=_SharedMilli_MOVB_FWD_NOOVERLAP#
17.              add     r20=496,sp
17.              ld8     r34=[r20]
17.              nop
17.              mov     gp=r34
17.              movl    r35=0xd000000000000000
17.              nop
17.              nop
17.              add     r36=472,sp
17.              add     r20=648,sp
17.              ld8     r37=[r20]
17.              nop
17.              nop
17.              nop
17.              add     r38=8,r37
17.              mov     r39=r0
17.              mov     r40=r0
17.              mov     r41=r0
17.              nop
17.              nop
17.              mov     r42=r0
17.              mov     r43=r0
17.              add     r44=16,sp
17.              nop
17.              st8     [r44]=r0
17.              add     r45=24,sp
17.              nop
17.              st8     [r45]=r0
17.              add     r46=32,sp
17.              nop
17.              st8     [r46]=r0
17.              add     r47=40,sp
17.              nop
17.              st8     [r47]=r0
17.              mov     r49=r35
17.              mov     r50=r36
17.              mov     r51=r38
17.              nop
17.              nop
17.              nop
17.              nop
17.              mov     r52=r39
17.              mov     r53=r40
17.              mov     r54=r41
17.              mov     r55=r42
17.              mov     r56=r43
17.              nop
17.              br.call.sptk    b0=COBLIB_DISPLAY_#
17.              add     r20=496,sp
17.              ld8     r34=[r20]
17.              nop
17.              mov     gp=r34
17.              nop
17.              br.cond.sptk.many       .b1_42
18.              IF D = 1 THEN DISPLAY "  MONDAY" ELSE
18.      .b1_16:
18.              add     r34=@gprel(pack_TESTT_D_#),gp
18.              ld2     r35=[r34]
18.              nop
18.              nop
```

```
18.              zxt2    r36=r35
18.              nop
18.              cmp4.ne p8,p0=1,r36
18.              add     r19=0,r0
18.      (p8)    add     r19=1,r0
18.              add     r20=656,sp
18.              st8     [r20]=r19
18.              nop
18.              nop
18.              nop
18.      (p8)    br.cond.dpnt.many       .b1_19
18.              mov     r34=r0
18.              add     r35=472,sp
18.              add     r36=r35,r34
18.              add     r37=@gprel($cob_internal$7#),gp
18.              add     r38=8,r0
18.              mov     r49=r36
18.              mov     r50=r37
18.              mov     r51=r38
18.              add     r20=664,sp
18.              st8     [r20]=r34
18.              nop
18.              br.call.sptk    b0=_SharedMilli_MOVB_FWD_NOOVERLAP#
18.              add     r20=496,sp
18.              ld8     r34=[r20]
18.              nop
18.              mov     gp=r34
18.              movl    r35=0xd000000000000000
18.              nop
18.              nop
18.              add     r36=472,sp
18.              add     r20=664,sp
18.              ld8     r37=[r20]
18.              nop
18.              nop
18.              nop
18.              add     r38=8,r37
18.              mov     r39=r0
18.              mov     r40=r0
18.              mov     r41=r0
18.              nop
18.              nop
18.              mov     r42=r0
18.              mov     r43=r0
18.              add     r44=16,sp
18.              nop
18.              st8     [r44]=r0
18.              add     r45=24,sp
18.              nop
18.              st8     [r45]=r0
18.              add     r46=32,sp
18.              nop
18.              st8     [r46]=r0
18.              add     r47=40,sp
18.              nop
18.              st8     [r47]=r0
18.              mov     r49=r35
18.              mov     r50=r36
18.              mov     r51=r38
18.              nop
18.              nop
18.              nop
18.              nop
18.              mov     r52=r39
18.              mov     r53=r40
18.              mov     r54=r41
18.              mov     r55=r42
18.              mov     r56=r43
18.              nop
18.              br.call.sptk    b0=COBLIB_DISPLAY_#
18.              add     r20=496,sp
18.              ld8     r34=[r20]
18.              nop
18.              mov     gp=r34
18.              nop
18.              br.cond.sptk.many       .b1_42
19.                IF D = 2 THEN DISPLAY "  TUESDAY" ELSE
19.      .b1_19:
19.              add     r34=@gprel(pack_TESTT_D_#),gp
19.              ld2     r35=[r34]
19.              nop
```

```
19.              nop
19.              zxt2    r36=r35
19.              nop
19.              cmp4.ne p8,p0=2,r36
19.              add     r19=0,r0
19.      (p8)    add     r19=1,r0
19.              add     r20=672,sp
19.              st8     [r20]=r19
19.              nop
19.              nop
19.              nop
19.      (p8)    br.cond.dpnt.many       .b1_22
19.              mov     r34=r0
19.              add     r35=480,sp
19.              add     r36=r35,r34
19.              add     r37=@ltoff($cob_internal$9#),gp
19.              ld8     r38=[r37]
19.              add     r39=9,r0
19.              nop
19.              mov     r49=r36
19.              mov     r50=r38
19.              mov     r51=r39
19.              add     r20=680,sp
19.              nop
19.              st8     [r20]=r34
19.              nop
19.              br.call.sptk    b0=_SharedMilli_MOVB_FWD_NOOVERLAP#
19.              add     r20=496,sp
19.              ld8     r34=[r20]
19.              nop
19.              mov     gp=r34
19.              movl    r35=0xd000000000000000
19.              nop
19.              nop
19.              add     r36=480,sp
19.              add     r20=680,sp
19.              ld8     r37=[r20]
19.              nop
19.              nop
19.              nop
19.              add     r38=9,r37
19.              mov     r39=r0
19.              mov     r40=r0
19.              mov     r41=r0
19.              nop
19.              nop
19.              mov     r42=r0
19.              mov     r43=r0
19.              add     r44=16,sp
19.              nop
19.              st8     [r44]=r0
19.              add     r45=24,sp
19.              nop
19.              st8     [r45]=r0
19.              add     r46=32,sp
19.              nop
19.              st8     [r46]=r0
19.              add     r47=40,sp
19.              nop
19.              st8     [r47]=r0
19.              mov     r49=r35
19.              mov     r50=r36
19.              mov     r51=r38
19.              nop
19.              nop
19.              nop
19.              nop
19.              mov     r52=r39
19.              mov     r53=r40
19.              mov     r54=r41
19.              mov     r55=r42
19.              mov     r56=r43
19.              nop
19.              br.call.sptk    b0=COBLIB_DISPLAY_#
19.              add     r20=496,sp
19.              ld8     r34=[r20]
19.              nop
19.              mov     gp=r34
19.              nop
19.              br.cond.sptk.many       .b1_42
20.                  IF D = 3 THEN DISPLAY "  WEDNESDAY" ELSE
```

```
20.     .b1_22:
20.             add     r34=@gprel(pack_TESTT_D_#),gp
20.             ld2     r35=[r34]
20.             nop
20.             nop
20.             zxt2    r36=r35
20.             nop
20.             cmp4.ne p8,p0=3,r36
20.             add     r19=0,r0
20.     (p8)    add     r19=1,r0
20.             add     r20=688,sp
20.             st8     [r20]=r19
20.             nop
20.             nop
20.             nop
20.     (p8)    br.cond.dpnt.many       .b1_25
20.             mov     r34=r0
20.             add     r35=480,sp
20.             add     r36=r35,r34
20.             add     r37=@ltoff($cob_internal$10#),gp
20.             ld8     r38=[r37]
20.             add     r39=11,r0
20.             nop
20.             mov     r49=r36
20.             mov     r50=r38
20.             mov     r51=r39
20.             add     r20=696,sp
20.             nop
20.             st8     [r20]=r34
20.             nop
20.             br.call.sptk    b0=_SharedMilli_MOVB_FWD_NOOVERLAP#
20.             add     r20=496,sp
20.             ld8     r34=[r20]
20.             nop
20.             mov     gp=r34
20.             movl    r35=0xd000000000000000
20.             nop
20.             nop
20.             add     r36=480,sp
20.             add     r20=696,sp
20.             ld8     r37=[r20]
20.             nop
20.             nop
20.             nop
20.             add     r38=11,r37
20.             mov     r39=r0
20.             mov     r40=r0
20.             mov     r41=r0
20.             nop
20.             nop
20.             mov     r42=r0
20.             mov     r43=r0
20.             add     r44=16,sp
20.             nop
20.             st8     [r44]=r0
20.             add     r45=24,sp
20.             nop
20.             st8     [r45]=r0
20.             add     r46=32,sp
20.             nop
20.             st8     [r46]=r0
20.             add     r47=40,sp
20.             nop
20.             st8     [r47]=r0
20.             mov     r49=r35
20.             mov     r50=r36
20.             mov     r51=r38
20.             nop
20.             nop
20.             nop
20.             nop
20.             mov     r52=r39
20.             mov     r53=r40
20.             mov     r54=r41
20.             mov     r55=r42
20.             mov     r56=r43
20.             nop
20.             br.call.sptk    b0=COBLIB_DISPLAY_#
20.             add     r20=496,sp
20.             ld8     r34=[r20]
20.             nop
```

```
20.              mov     gp=r34
20.              nop
20.              br.cond.sptk.many       .b1_42
21.                    IF D = 4 THEN DISPLAY "  THURSDAY" ELSE
21.      .b1_25:
21.              add     r34=@gprel(pack_TESTT_D_#),gp
21.              ld2     r35=[r34]
21.              nop
21.              nop
21.              zxt2    r36=r35
21.              nop
21.              cmp4.ne p8,p0=4,r36
21.              add     r19=0,r0
21.      (p8)    add     r19=1,r0
21.              add     r20=704,sp
21.              st8     [r20]=r19
21.              nop
21.              nop
21.              nop
21.      (p8)    br.cond.dpnt.many       .b1_28
21.              mov     r34=r0
21.              add     r35=480,sp
21.              add     r36=r35,r34
21.              add     r37=@ltoff($cob_internal$11#),gp
21.              ld8     r38=[r37]
21.              add     r39=10,r0
21.              nop
21.              mov     r49=r36
21.              mov     r50=r38
21.              mov     r51=r39
21.              add     r20=712,sp
21.              nop
21.              st8     [r20]=r34
21.              nop
21.              br.call.sptk    b0=_SharedMilli_MOVB_FWD_NOOVERLAP#
21.              add     r20=496,sp
21.              ld8     r34=[r20]
21.              nop
21.              mov     gp=r34
21.              movl    r35=0xd000000000000000
21.              nop
21.              nop
21.              add     r36=480,sp
21.              add     r20=712,sp
21.              ld8     r37=[r20]
21.              nop
21.              nop
21.              nop
21.              add     r38=10,r37
21.              mov     r39=r0
21.              mov     r40=r0
21.              mov     r41=r0
21.              nop
21.              nop
21.              mov     r42=r0
21.              mov     r43=r0
21.              add     r44=16,sp
21.              nop
21.              st8     [r44]=r0
21.              add     r45=24,sp
21.              nop
21.              st8     [r45]=r0
21.              add     r46=32,sp
21.              nop
21.              st8     [r46]=r0
21.              add     r47=40,sp
21.              nop
21.              st8     [r47]=r0
21.              mov     r49=r35
21.              mov     r50=r36
21.              mov     r51=r38
21.              nop
21.              nop
21.              nop
21.              nop
21.              mov     r52=r39
21.              mov     r53=r40
21.              mov     r54=r41
21.              mov     r55=r42
21.              mov     r56=r43
21.              nop
```

```
21.              br.call.sptk    b0=COBLIB_DISPLAY_#
21.              add     r20=496,sp
21.              ld8     r34=[r20]
21.              nop
21.              mov     gp=r34
21.              nop
21.              br.cond.sptk.many       .b1_42
22.                      IF D = 5 THEN DISPLAY "  FRIDAY" ELSE
22.      .b1_28:
22.              add     r34=@gprel(pack_TESTT_D_#),gp
22.              ld2     r35=[r34]
22.              nop
22.              nop
22.              zxt2    r36=r35
22.              nop
22.              cmp4.ne p8,p0=5,r36
22.              add     r19=0,r0
22.      (p8)    add     r19=1,r0
22.              add     r20=720,sp
22.              st8     [r20]=r19
22.              nop
22.              nop
22.              nop
22.      (p8)    br.cond.dpnt.many       .b1_31
22.              mov     r34=r0
22.              add     r35=472,sp
22.              add     r36=r35,r34
22.              add     r37=@gprel($cob_internal$12#),gp
22.              add     r38=8,r0
22.              mov     r49=r36
22.              mov     r50=r37
22.              mov     r51=r38
22.              add     r20=728,sp
22.              st8     [r20]=r34
22.              nop
22.              br.call.sptk    b0=_SharedMilli_MOVB_FWD_NOOVERLAP#
22.              add     r20=496,sp
22.              ld8     r34=[r20]
22.              nop
22.              mov     gp=r34
22.              movl    r35=0xd000000000000000
22.              nop
22.              nop
22.              add     r36=472,sp
22.              add     r20=728,sp
22.              ld8     r37=[r20]
22.              nop
22.              nop
22.              nop
22.              add     r38=8,r37
22.              mov     r39=r0
22.              mov     r40=r0
22.              mov     r41=r0
22.              nop
22.              nop
22.              mov     r42=r0
22.              mov     r43=r0
22.              add     r44=16,sp
22.              nop
22.              st8     [r44]=r0
22.              add     r45=24,sp
22.              nop
22.              st8     [r45]=r0
22.              add     r46=32,sp
22.              nop
22.              st8     [r46]=r0
22.              add     r47=40,sp
22.              nop
22.              st8     [r47]=r0
22.              mov     r49=r35
22.              mov     r50=r36
22.              mov     r51=r38
22.              nop
22.              nop
22.              nop
22.              nop
22.              mov     r52=r39
22.              mov     r53=r40
22.              mov     r54=r41
22.              mov     r55=r42
22.              mov     r56=r43
```

```
22.              nop
22.              br.call.sptk    b0=COBLIB_DISPLAY_#
22.              add     r20=496,sp
22.              ld8     r34=[r20]
22.              nop
22.              mov     gp=r34
22.              nop
22.              br.cond.sptk.many       .b1_42
23.                        IF D = 6 THEN DISPLAY "  SATURDAY" END-IF
23.      .b1_31:
23.              add     r34=@gprel(pack_TESTT_D_#),gp
23.              ld2     r35=[r34]
23.              nop
23.              nop
23.              zxt2    r36=r35
23.              nop
23.              cmp4.ne p8,p0=6,r36
23.              add     r19=0,r0
23.      (p8)    add     r19=1,r0
23.              add     r20=736,sp
23.              st8     [r20]=r19
23.              nop
23.              nop
23.              nop
23.      (p8)    br.cond.dpnt.many       .b1_42
23.              mov     r34=r0
23.              add     r35=480,sp
23.              add     r36=r35,r34
23.              add     r37=@ltoff($cob_internal$13#),gp
23.              ld8     r38=[r37]
23.              add     r39=10,r0
23.              nop
23.              mov     r49=r36
23.              mov     r50=r38
23.              mov     r51=r39
23.              add     r20=744,sp
23.              nop
23.              st8     [r20]=r34
23.              nop
23.              br.call.sptk    b0=_SharedMilli_MOVB_FWD_NOOVERLAP#
23.              add     r20=496,sp
23.              ld8     r34=[r20]
23.              nop
23.              mov     gp=r34
23.              movl    r35=0xd000000000000000
23.              nop
23.              nop
23.              add     r36=480,sp
23.              add     r20=744,sp
23.              ld8     r37=[r20]
23.              nop
23.              nop
23.              nop
23.              add     r38=10,r37
23.              mov     r39=r0
23.              mov     r40=r0
23.              mov     r41=r0
23.              nop
23.              nop
23.              mov     r42=r0
23.              mov     r43=r0
23.              add     r44=16,sp
23.              nop
23.              st8     [r44]=r0
23.              add     r45=24,sp
23.              nop
23.              st8     [r45]=r0
23.              add     r46=32,sp
23.              nop
23.              st8     [r46]=r0
23.              add     r47=40,sp
23.              nop
23.              st8     [r47]=r0
23.              mov     r49=r35
23.              mov     r50=r36
23.              mov     r51=r38
23.              nop
23.              nop
23.              nop
23.              nop
23.              mov     r52=r39
```

```
23.              mov     r53=r40
23.              mov     r54=r41
23.              mov     r55=r42
23.              mov     r56=r43
23.              nop
23.              br.call.sptk    b0=COBLIB_DISPLAY_#
23.              add     r20=496,sp
23.              ld8     r34=[r20]
23.              nop
23.              mov     gp=r34
24.                      END-IF
25.                    END-IF
26.                  END-IF
27.                END-IF
28.              END-IF
29.            END-IF
29.              nop
29.              nop
29.      .b1_42:
29.              add     r34=@gprel(pack_TESTT_STD-DATE_#),gp
29.              ld4     r35=[r34]
29.              nop
29.              nop
29.              nop
29.              zxt4    r36=r35
29.              add     r37=1,r0
29.              nop
29.              zxt4    r38=r37
29.              add     r39=r36,r38
29.              movl    r40=0x0fffffff
29.              cmp.leu p8,p0=r39,r40
29.              add     r19=0,r0
29.      (p8)    add     r19=1,r0
29.              add     r20=752,sp
29.              st8     [r20]=r19
29.              nop
29.              nop
29.              nop
29.      (p8)    br.cond.dpnt    .b1_43
29.              add     r34=2,r0
29.              break   2
29.              nop
29.      .b1_43:
29.              add     r34=@gprel(pack_TESTT_STD-DATE_#),gp
29.              ld4     r35=[r34]
29.              nop
29.              nop
29.              nop
29.              add     r36=1,r35
29.              add     r37=@gprel(pack_TESTT_STD-DATE_#),gp
29.              st4     [r37]=r36
29.              nop
29.              nop
29.              nop
29.              br.cond.sptk.many       .b1_5
30.          END-PERFORM.
31.          STOP RUN.
31.      .b1_8:
31.              nop
31.              nop
31.              br.call.sptk    b0=COBLIB_STOP_#
31.              add     r20=496,sp
31.              ld8     r34=[r20]
31.              nop
31.              mov     gp=r34
31.              nop
31.              br.call.sptk    b0=COBLIB_STOP_#
31.              add     r20=496,sp
31.              ld8     r34=[r20]
31.              nop
31.              mov     gp=r34
31.              nop
31.              mov     b0=r33
31.              nop
31.              nop
31.              mov     ar.pfs=r32
31.              add     r27=928,sp
31.              ldf.fill        f22=[r27],-32
31.              nop
31.              add     r26=912,sp
31.              ldf.fill        f21=[r26],-32
```

```
31.              nop
31.              ldf.fill        f20=[r27],-32
31.              ldf.fill        f19=[r26],-32
31.              nop
31.              ldf.fill        f18=[r27],-32
31.              ldf.fill        f17=[r26],-32
31.              nop
31.              ldf.fill        f16=[r27],-32
31.              ldf.fill        f5=[r26],-32
31.              nop
31.              ldf.fill        f4=[r27],-32
31.              nop
31.              nop
31.              ldf.fill        f3=[r26]
31.              ldf.fill        f2=[r27]
31.              add     sp=928,sp
31.              nop
31.              nop
31.              br.ret.sptk.many        b0
31.      .b1_4:
31.              nop
31.              movl    r34=0x8000000000000000
31.              nop
31.              add     r35=127,r0
31.              mov     r36=r0
31.              mov     r37=r0
31.              mov     r38=r0
31.              mov     r39=r0
31.              nop
31.              nop
31.              mov     r40=r0
31.              mov     r41=r0
31.              add     r42=16,sp
31.              nop
31.              st8     [r42]=r0
31.              add     r43=24,sp
31.              nop
31.              st8     [r43]=r0
31.              add     r44=32,sp
31.              nop
31.              st8     [r44]=r0
31.              add     r45=40,sp
31.              nop
31.              st8     [r45]=r0
31.              mov     r49=r34
31.              mov     r50=r35
31.              mov     r51=r36
31.              nop
31.              nop
31.              nop
31.              nop
31.              mov     r52=r37
31.              mov     r53=r38
31.              mov     r54=r39
31.              mov     r55=r40
31.              mov     r56=r41
31.              nop
31.              br.call.sptk    b0=COBLIB_ERROR_#
31.              add     r20=496,sp
31.              ld8     r34=[r20]
31.              nop
31.              nop
31.              nop
31.              mov     gp=r34
31.              mov     r35=r0
31.              mov     r36=r0
31.              mov     r37=r0
31.              nop
31.              nop
31.              mov     r38=r0
31.              mov     r39=r0
31.              mov     r40=r0
31.              mov     r41=r0
31.              nop
31.              nop
31.              mov     r42=r0
31.              add     r43=16,sp
31.              st8     [r43]=r0
31.              nop
31.              add     r44=24,sp
31.              st8     [r44]=r0
```

```
      31.              nop
      31.              add     r45=32,sp
      31.              st8     [r45]=r0
      31.              nop
      31.              add     r46=40,sp
      31.              nop
      31.              nop
      31.              st8     [r46]=r0
      31.              mov     r49=r35
      31.              mov     r50=r36
      31.              mov     r51=r37
      31.              nop
      31.              nop
      31.              nop
      31.              nop
      31.              mov     r52=r38
      31.              mov     r53=r39
      31.              mov     r54=r40
      31.              mov     r55=r41
      31.              mov     r56=r42
      31.              nop
      31.              br.call.sptk    b0=COBLIB_ABEND_#
      31.              add     r20=496,sp
      31.              ld8     r34=[r20]
      31.              nop
      31.              mov     gp=r34
      31.              nop
      31.              mov     b0=r33
      31.              nop
      31.              nop
      31.              mov     ar.pfs=r32
      31.              add     r27=928,sp
      31.              ldf.fill        f22=[r27],-32
      31.              nop
      31.              add     r26=912,sp
      31.              ldf.fill        f21=[r26],-32
      31.              nop
      31.              ldf.fill        f20=[r27],-32
      31.              ldf.fill        f19=[r26],-32
      31.              nop
      31.              ldf.fill        f18=[r27],-32
      31.              ldf.fill        f17=[r26],-32
      31.              nop
      31.              ldf.fill        f16=[r27],-32
      31.              ldf.fill        f5=[r26],-32
      31.              nop
      31.              ldf.fill        f4=[r27],-32
      31.              nop
      31.              nop
      31.              ldf.fill        f3=[r26]
      31.              ldf.fill        f2=[r27]
      31.              add     sp=928,sp
      31.              nop
      31.              nop
      31.              br.ret.sptk.many        b0

Node heap stats: Bytes Malloc'd: 3980, Free'd: 0, Max heap size: 3752 Max Malloc: 78

CPU time spent in the backend: 0:00:00.70
```

The compilation summary is the last page of every compiler listing.

Example 23-21 Compilation Summary

```
COBOL - T0356H01 - (20DEC2004)
No failures detected.
No errors detected.
No warnings reported.
No remarks issued.
Maximum symbol table size = 10884 bytes
eld - TNS/E Native Mode Linker - T0608H01 - 03DEC07
Copyright 2004 Hewlett-Packard Company
eld command line:
\drp12.$system.system.eld -o RUNUNIT RUNUNIT
\DRP12.$SYSTEM.ZDLL031.ZCOBDLL
\DRP12.$SYSTEM.ZDLL031.ZCREDLL
**** INFORMATIONAL MESSAGE **** [1019]:
Using DLL \DRP12.$SYSTEM.ZDLL031.ZCOBDLL.
**** INFORMATIONAL MESSAGE **** [1019]:
Using DLL \DRP12.$SYSTEM.ZDLL031.ZCREDLL.
**** INFORMATIONAL MESSAGE **** [1530]:
Using the zimpimp file $SYSTEM.SYS00.ZIMPIMP.
Output file: RUNUNIT (program file)
Output file timestamp: Jan 6 10:41:44 2005
No errors reported.
No warnings reported.
3 informational messages reported.
Elapsed Time: 00:00:01
Object file: RUNUNIT
Compiler driver: \DRP12.$SYSTEM.SYSTEM.ECOBOL
COBOL DLL: \DRP12.$SYSTEM.ZDLL031.ZCOBDLL
CRE DLL: \DRP12.$SYSTEM.ZDLL031.ZCREDLL
ECOBEXT: \DRP12.$SYSTEM.SYSTEM.ECOBEXT
Compiler statistics
phase CPU seconds elapsed time file name
ECOBFE 0.9 00:00:06 \DRP12.$SYSTEM.SYSTEM.ECOBFE
ELD 0.2 00:00:02 \DRP12.$SYSTEM.SYSTEM.ELD
total 1.1 00:00:07
```

These lines show that that program that produced Example 23-20: Symbolic Code Listing (page 788) had no compilation errors or warnings:

```
No errors detected.
No warnings reported.
```

This line shows that, in Example 23-21, the FIPS directive identified no language elements:

```
No remarks issued.
```

If the ECOBOL compiler had reported any errors, the summary would have contained lines like these:

```
COBOL - T0356H01 - (20DEC2004)
No failures detected.
*** 3 error(s) detected.
*** Last error at page 2 line 51 [source
\DRP12.$DATA4.EXCOB.EX3112CB line 96.].
*** 1 warning(s) reported.
*** Last warning at page 3 line 17 [source
\DRP12.$DATA4.EXCOB.EX3112CB line 116.].
No remarks issued.
Maximum symbol table size = 10906 bytes
No object file produced
```

24 Calling Other Programs and Routines

NOTE: This section applies to the Guardian environment. For information on mixed-language programming in the OSS environment, see Mixed-Language Programs (page 722) and Utility Routines (page 735).

An HP COBOL program that calls one or more non-COBOL routines is called a mixed-language program (as is a non-COBOL program that calls one or more COBOL programs).

HP COBOL programs can call programs compiled by these compilers:

- TNS/E C
- TNS/E C++
- EpTAL

A called routine can have an ordinary, VARIABLE, or EXTENSIBLE parameter list.

Topics:

- Run-Time Environment
- Calling Other COBOL Programs
- Calling Non-COBOL Routines
- Passing Parameters

To call your HP COBOL program from a non-COBOL program, use the non-COBOL language's method of calling an external routine. The method varies from language to language; see the reference manual for the appropriate non-COBOL language.

Run-Time Environment

Native programs always run in the CRE.

In the CRE, each routine in the program appears to be running in its own language-specific run-time environment, regardless of the language of the main routine. For example, if the main routine of a mixed-language program is written in HP COBOL, an HP C routine still has complete access to the HP C run-time library.

The CRE library, a collection of routines that implements the CRE, enables the language-specific run-time libraries to coexist peacefully with each other. User routines and run-time libraries call CRE library routines to access shared resources managed by the CRE, such as the standard files (input, output, and log) and the user heap, regardless of language.

The CRE does not support all possible operations. For example, the CRE supports file sharing only for the three standard files: standard input, standard output, and standard log. The language-specific run-time libraries access all other files by calling run-time procedures directly, whether or not a program uses the CRE.

For more information about writing programs that use the services provided by the CRE, see the *CRE Programmer's Guide*.

Calling Other COBOL Programs

An HP COBOL program calls another HP COBOL program with a CALL statement. The called program can be either a separately compiled HP COBOL program or a nested HP COBOL program. A nested program is usually accessible to more calling programs if it is a common program.

If a called program is an initial program, its program state is initialized whenever it is called.

The CALL statement can call another program statically (with the program specified at compilation time) or dynamically (with the program specified at run time). The former is more efficient; the latter is more flexible.

The CALL statement can pass parameters from the calling program to the called program either by reference (the default) or by content.

Topics:

- Separately Compiled HP COBOL Programs
- Nested HP COBOL Programs
- Common Programs
- Initial Programs
- Static Calls
- Dynamic Calls

Separately Compiled HP COBOL Programs

A separately compiled program is an HP COBOL source program that is not nested within any other program. A compilation unit—a collection of source statements presented to a compiler in one compilation—contains one or more separately compiled programs. Separately compiled programs in the same compilation unit can call each other.

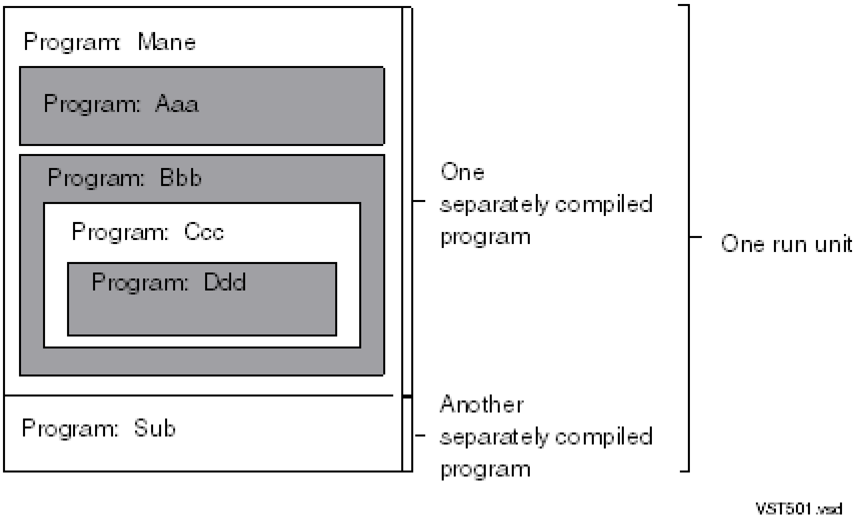
Any separately compiled program with no Linkage Section can be a main program (that is, the program with which execution begins). If more than one of the programs in a compilation unit has no Linkage Section, one program must include a MAIN directive to identify it as the main program.

Nested HP COBOL Programs

Nested HP COBOL programs are contained in other HP COBOL programs. Nested programs can be directly or indirectly contained. In Figure 24-1, the compilation unit contains two separately compiled programs (Mane and Sub), which contain nested programs, and the programs have these relationships:

The program ...	Directly contains ...	And indirectly contains ...
Mane	Aaa, Bbb	Ccc, Ddd
Aaa	Nothing	Nothing
Bbb	Ccc	Ddd
Ccc	Ddd	Nothing
Ddd	Nothing	Nothing
Sub	Nothing	Nothing

Figure 24-1 Directly Contained Programs and Indirectly Contained Programs



An HP COBOL program can call any program directly contained within itself and any other separately compiled program. It cannot call a program indirectly contained within itself. The programs in Figure 24-1 can call each other:

	Can call the program ...					
The program ...	**Mane**	**Aaa**	**Bbb**	**Ccc**	**Ddd**	**Sub**
Mane		X	X			X
Aaa						X
Bbb				X		X
Ccc					X	X
Ddd						X
Sub	X					

Common Programs

If a program has the COMMON clause in its Identification Division, it is called a common program, and it can be called by any program directly or indirectly contained in the program that directly contains it. In Figure 24-1, if Aaa, Bbb, and Ccc are common, the programs in Figure 24-1 can call each other:

	Can call the program ...					
The program ...	**Mane**	**Aaa**	**Bbb**	**Ccc**	**Ddd**	**Sub**
Mane		X	X			X
Aaa			X			X
Bbb		X		X		X
Ccc		X	X		X	X
Ddd		X	X	X		X
Sub	X					

You gain no performance benefits by making Ddd common.

Initial Programs

If a program has the INITIAL clause in its Identification Division, it is an initial program. Its program state is initialized whenever it is called. The data in its Working-Storage Section is allocated when it is called rather than being statically allocated. For more information about initial programs, see Initial Programs (page 61).

Static Calls

A static call is achieved with the statement CALL *program-name*, where *program-name* is an alphanumeric literal whose value is known at compilation time. *program-name* is the PROGRAM-ID of the called program, and it can be qualified with the phrase OF *file-mnemonic* or IN *file-mnemonic*.

Quotation marks around *program-name* are not required, but are recommended. If, in some eventual program maintenance, someone adds a variable named APROG to a program that already calls a program named APROG, and the name APROG is not in quotation marks, the compiler assumes that APROG refers to the variable.

At run time, if the variable APROG does not contain the name of a valid program that is included in the object file, the program terminates with the error message “Called program not found.” If APROG is enclosed in quotes, the compiler recognizes it as a program name rather than a variable.

A static call is more efficient than a dynamic call, but a dynamic call is more flexible.

Dynamic Calls

To perform a dynamic call, use the statement CALL *identifier*, where *identifier* is an alphanumeric data item whose value is not known until run time. The value of *identifier* is the PROGRAM-ID of the called program, but it cannot be qualified with the phrase OF *file-mnemonic* or IN *file-mnemonic*.

Example 24-1 shows a CALL identifier statement in context. The identifier is ROUTINE, and at run time its value will be either POSRTN or NEG0RTN, depending on the value of TALLY.

Example 24-1 CALL Identifier Statement

```
DATA DIVISION.
01 ROUTINE  PICTURE X(7).
...
PROCEDURE DIVISION.
...
IF TALLY IS GREATER THAN 0 MOVE "POSRTN" TO ROUTINE
ELSE                       MOVE "NEG0RTN" TO ROUTINE
CALL ROUTINE
...
```

A program that is to be called with a CALL *identifier* statement cannot use standard (16-bit) addressing.

The compilation is unable to validate the parameters of dynamic calls (that is, to determine whether the actual parameters of the calling program correspond to the formal parameters of the called program).

Every program that a dynamic call could possibly call (in the preceding example, the programs POSRTN and NEG0RTN) must be linked into the run unit or user library before you execute the program (see Linking HP COBOL Programs (page 830)).

If you want an HP COBOL program to be entirely free to call any other COBOL program that is prepared to be called, make the programs separate processes. Separate processes communicate through the file process $RECEIVE—the sender writes to a named process, and the receiver reads a file named $RECEIVE. For more information about $RECEIVE, see $RECEIVE (page 934).

Calling Non-COBOL Routines

The way an HP COBOL program can call a non-COBOL routine depends on whether the calling program was compiled with the PORT directive. An HP COBOL program that was not compiled with the PORT directive calls a non-COBOL routine with the ENTER statement. An HP COBOL program that was compiled with the PORT directive calls a non-COBOL routine with either the ENTER statement or the CALL statement, which behaves like an X/Open CALL statement.

NOTE: If the types of the actual and formal parameters do not match, the ENTER statement attempts to convert the actual parameters to the types of the formal parameters; the X/Open CALL statement reports an error instead.

Applying a CANCEL statement to a non-COBOL routine that was called with an X/Open CALL statement terminates the run unit.

Topics:

- ENTER Statement
- X/Open CALL Statement
- Functions (Routines That Return Values)
- Operating System Routines
- HP COBOL Run-Time Routines
- COBOL Utility Routines
- ENFORM Programs
- Restrictions on Calling Non-COBOL Routines

ENTER Statement

If an HP COBOL program is not compiled with the PORT directive, it can call routines written in native HP C, native HP C++, and pTAL by using the ENTER statement.

In the ENTER statement, you do not need to specify the language of the called routine, because the compiler can determine the language in which the program is written. If you do specify the language, it must be C or TAL. If you specify TAL, the compiler expects a pTAL program (it does not accept TAL programs). If you specify C, the compiler expects a routine written in either HP C or HP C++. Although an HP COBOL program can use the ENTER statement to call HP C++ and pTAL programs, you cannot specify the language C++ or pTAL.

Any non-COBOL routine called by an HP COBOL program must have already been compiled, and its object program forms must be available to the compiler through a DLL, import library, or object file, specified through the SEARCH or CONSULT directive.

X/Open CALL Statement

If an HP COBOL program is compiled with the PORT directive, it can call non-COBOL routines with the CALL statement, which behaves like an X/Open CALL statement.

If the called program is a function, the X/Open CALL statement returns its value in the special register RETURN-CODE. For more information, see Special Register RETURN-CODE.

CAUTION: If a program compiled with the PORT directive calls a program not compiled with the PORT directive, differences in the way BINARY/COMPUTATIONAL/COMP data items are aligned in the two programs can cause data to be misread.

Functions (Routines That Return Values)

A function is a routine that returns a value to the calling program. COBOL has no mechanism for returning a function's value to the calling program, but HP COBOL has two:

- GIVING Phrase
- Special Register RETURN-CODE

GIVING Phrase

The GIVING phrase allows you to specify a data item, *`return-value`*, to hold the value that the function returns. The storage operation is performed after any necessary scaling and follows the rules for an elementary MOVE TO statement (for these rules, see MOVE TO (page 378)).

Special Register RETURN-CODE

The special register RETURN-CODE is accessible to an HP COBOL program that was compiled with the PORT directive and uses the X/Open CALL statement to call a function.

The value of the function is returned in the special register RETURN-CODE, whose description is:

```
01  RETURN-CODE  EXTERNAL  PICTURE S9(5)  COMPUTATIONAL.
```

If the value of the function is greater than 99,999, arithmetic overflow occurs.

If the calling program cannot access RETURN-CODE, the value of RETURN-CODE is unchanged.

Operating System Routines

Operating system routines are written in pTAL, HP C, and HP C++. The language in which an operating system routine is written does not matter to the HP COBOL program that calls it. Operating system routines execute in system code space, using both system data space (for system tables) and user data space (for temporary storage). When an HP COBOL program calls an operating system routine, it uses the ENTER statement.

Some operating system routines never need to be called by HP COBOL programs—file system routines, for example. File system routines are called by the HP COBOL run-time routines to do their input and output operations. Although it is possible to call file system routines explicitly from an HP COBOL program, you are advised not to apply both HP COBOL I-O statements and explicit calls to file system routines to the same file, as the HP COBOL run-time routines do significant preprocessing and postprocessing.

NOTE: If you use operating system routines to do nowait I-O (you call a routine to do an input or output operation and then call AWAITIO to wait for the operation to finish), be aware that the HP COBOL run-time routines perform nowait I-O in some cases. The AWAITIO routine can wait for any I-O operation to complete, or wait for the I-O operation on a specific file to complete. It is best to apply AWAITIO to a specific file. Using the command PARAM WAITED-IO ON allows you some latitude, however.

Examples of operating system routines that are useful to an HP COBOL program are ABEND, which signals an abnormal termination, and PURGEDATA, which erases all data in a file but does not purge the file itself.

Topics:

- Parameters
- Resolution
- Extensible and Variable Parameter Lists
- Generations of Operating System Routines

Parameters

Guardian operating system routines are documented in the *Guardian Procedure Calls Reference Manual*. When calling system routines from an HP COBOL program, use Table 24-1 to determine the appropriate form for the COBOL parameters.

You must verify that the actual parameters in the ENTER statement (including `return-value` in the GIVING phrase) are of the HP COBOL types that correspond to the TAL types of the formal parameters in the operating system routine. The compiler performs the necessary conversions of sending parameters. Parameters passed by reference must match exactly. The HP COBOL data type INDEX does not correspond to a pTAL data type.

Table 24-1 HP COBOL and pTAL Parameter Correspondence

		pTAL Data Type	
COBOL Data Category		**Passed by Reference**	**Passed by Value**
Alphabetic		STRING	Not applicable
Numeric	COMP 9 (1) - 9 (4)	INT	INT INT(32) FIXED REAL REAL(64)
	COMP 9 (5) - 9 (9)	INT(32)	INT INT(32) FIXED REAL REAL(64)
	COMP 9 (10) - 9 (18)	FIXED	INT INT(32) FIXED REAL REAL(64)
	NATIVE-2	INT	INT INT(32) FIXED REAL REAL(64)
	NATIVE-4	INT(32)	INT INT(32) FIXED REAL REAL(64)
	NATIVE-8	FIXED	INT INT(32) FIXED REAL REAL(64)

Table 24-1 HP COBOL and pTAL Parameter Correspondence *(continued)*

		pTAL Data Type	
COBOL Data Category		**Passed by Reference**	**Passed by Value**
	DISPLAY	STRING	INT INT(32) FIXED REAL REAL(64)
	INDEX	Not applicable	Not applicable
Numeric Edited		STRING	Not applicable
Alphanumeric Edited		STRING	Not applicable
Alphanumeric		STRING	Not applicable

Resolution

The compiler leaves operating system routines unresolved, and the loader resolves them during fixup (see Fixup (page 830)Fixup). The compiler is able to leave references to operating system routine calls unresolved because a file named ECOBEXT, which is part of every HP COBOL RVU, contains dummy versions of all the operating system routines. Another file, EXTDECS, contains pTAL external declarations for the environment routines and is part of every operating environment RVU. A third file, ZSYSCOB, contains HP COBOL source declarations of data items and structures for Guardian procedures and operating environment messages.

There are two reasons to have your system manager change the files ECOBEXT and EXTDECS:

- You have additional routines that you want to be left unresolved until fixup (ask your system manager to add dummy versions of them to the ECOBEXT or EXTDECS file).
- The formal parameter list of one or more system routines changes from extensible to variable, in which case the EXTDECS file and the ECOBEXT file must also change.

For more information on the ZSYSCOB file, see the *Guardian Application Conversion Guide*.

Extensible and Variable Parameter Lists

An operating system routine can have an extensible or a variable parameter list. Both extensible and variable parameter lists allow you to omit unnecessary actual parameters in the ENTER statement. The difference between extensible and variable parameter lists is in what happens when new formal parameters are added to the routine, as sometimes happens to operating system routines with a new RVU of the operating environment.

When new formal parameters are added to an operating system routine that has an extensible parameter list, you do not have to recompile programs that call the routine.

If an operating system routine’s formal parameter list changes from variable to extensible, the code that was compiled to expect a variable parameter list can usually call the new version of the routine that has an extensible parameter list (at a slight performance penalty). For more information about variable and extensible parameter lists, see the *pTAL Reference Manual*.

Generations of Operating System Routines

Like the file ECOBEXT, the files ECOBEX1 and ECOBEX0 are part of every HP COBOL RVU. Like the file EXTDECS, the files EXTDECS1 and EXTDECS0 are part of every operating environment RVU. This table shows the relationship between these files.

File of Operating System Routine Declarations	Generation of Operating System Routines	File of Dummy Versions of operating system routines
EXTDECS0	Latest	ECOBEX0
EXTDECS1	Next-to-latest	ECOBEX1
EXTDECS	Second-next-to-latest	ECOBEXT

Different generations of the same operating system routine might have different formal parameter lists (see Extensible and Variable Parameter Lists). Because of this, programs compiled with ECOBEX0 might execute only on the very latest RVU of the operating environment, programs compiled with ECOBEX1 might execute only on the two most recent RVUs of the operating environment, and programs compiles with ECOBEXT might execute only on the three most recent RVUs of the operating environment.

Your system manager must verify that the appropriate file has the name ECOBEXT when you compile your program. Any of the files ECOBEX0, ECOBEX1, or ECOBEXT can be named ECOBEXT.

If you need to use a version of the ECOBEXT file that is available on your system under a name other than ECOBEXT, there are two ways to do it:

- Create a subvolume containing the ECOBOL compiler and all the files it needs. Install the appropriate version of ECOBEXT in that subvolume, under the name ECOBEXT. Use this subvolume for your compilations; that is, instead of using a compilation command like

  ```
  ECOBOL /IN XYZ/
  ```

  use a compilation command like

  ```
  RUN MYCOBVOL.ECOBOL /IN XYZ/
  ```

  where MYCOBVOL is the subvolume you created.

- Use the system volume for your compilations, but use a CONSULT directive to tell the compiler to resolve external references from a file other than ECOBEXT.

HP COBOL Run-Time Routines

Your HP COBOL program calls HP COBOL run-time routines implicitly, not with ENTER statements. HP COBOL run-time routines are in the dynamic link library ZCOBDLL. Their names begin with “COBLIB_.” They execute in DLL code space.

COBOL Utility Routines

HP provides many COBOL utility routines that your COBOL program can call with the ENTER statement. These routines are described in Chapter 14: Libraries and Utility Routines (page 607).

ENFORM Programs

ENFORM, a component of the Encompass distributed database management system, is a powerful nonprocedural language for querying or developing reports on a relational database.

If your database is described in the Data Definition Language (DDL), you can write an ENFORM query to specify the records you want from the database and the order in which you want ENFORM to deliver them. You then compile that ENFORM query into a disk file and include code in a COBOL program to start ENFORM as a separate process.

The operating environment procedure ENFORMSTART initiates the query processor. Each time the ENFORMRECEIVE procedure executes, it accepts either a record that ENFORM built and transmitted to the program through the interprocess message system or an error value. The ENFORMFINISH procedure terminates the processing with the query processor. This mechanism is called the host-language interface and is documented in the *ENFORM User's Guide*. You cannot use this mechanism from a Pathway server if the ENFORM routines use $RECEIVE.

Restrictions on Calling Non-COBOL Routines

An HP COBOL program can call routines written in native HP C, native HP C++, and pTAL.

These topics explain the restrictions on calling non-COBOL routines from HP COBOL. For information about passing HP COBOL parameters to such routines, see Passing Parameters to Non-COBOL Routines.

Topics:

- HP C Routines
- HP C++ Function Name Consideration

HP C Routines

An HP COBOL program must not call an HP C function that directly or indirectly accesses the HP C function `getenv`.

An HP COBOL program indirectly calls HP C functions that allocate and deallocate memory and perform HP C input-output operations. Do not directly call the HP C function `getenv` from an HP COBOL program.

If a program has a COBOL MAIN program, and calls HP C or HP C++ functions that perform HP C or HP C++ I/O operations on the standard files `stdin`, `stdout`, or `stderr`, the HP C library function `fopen_std_file` must be called before any such operations take place. You need `fopen_std_file (0,x )` for input, `fopen_std_file (1,x )` for output, and `fopen_std_file (2,x )` for the `stderr`. This function must be called from an HP C or HP C++ routine. Also, the HP C program must be compiled "with extensions" if you are going to call this function. See the *Guardian Native C Library Calls Reference Manual* for more information on the `fopen_std_file` function.

An HP COBOL program cannot call an HP C function that has these characteristics:

- Has lowercase letters in its name, if the HP C function is called by ENTER (the compiler upshifts them). This restriction does not apply to HP C functions called by CALL.
- Has a variable parameter list.
- Returns a structured value.
- Specifies a formal parameter whose type has no corresponding HP COBOL type (to see which HP C types have corresponding HP COBOL types, see Appendix B: Data Type Correspondence (page 1239).

A TNS/E HP COBOL program and a TNS/E HP C program can share data under these conditions:

- Level-01 HP COBOL data items are described with the EXTERNAL clause.
- HP C data names that the HP COBOL program references have no lowercase letters or underscores.
- If the HP COBOL and HP C programs are sharing strings, the strings end in the zero byte that HP C expects.

In Example 24-2, the HP COBOL and HP C programs share the data item MYVAR.

Example 24-2 HP COBOL and HP C Programs Sharing Data

HP COBOL Code:

```
* The SEARCH directive references the HP C object file.
?SYMBOLS; SEARCH SHOWEXTO

IDENTIFICATION DIVISION.
   PROGRAM-ID.    TEST-EXTERNAL.
ENVIRONMENT DIVISION.
   CONFIGURATION  SECTION.
     SPECIAL-NAMES.  SYMBOLIC NULLCHAR IS 1.
DATA DIVISION.
   EXTENDED-STORAGE SECTION.
     01 MYVAR EXTERNAL.
        05 DATA-01 PIC X(20).
        05 NULL-TERM PIC X.
PROCEDURE DIVISION.
   STARTIT.
   MOVE "abcdefghijlmnopqrstu" TO DATA-01.
   MOVE NULLCHAR TO NULL-TERM.
   ENTER C "SHOWEXT".
   DISPLAY "Back in COBOL: " DATA-01.
```

HP C Code:

```
#include  nolist
extern char MYVAR[21];

void SHOWEXT(void){
int i;

/* In the Guardian environment, the call to fopen_std_file
is needed because the main program is in HP COBOL.
In the OSS environment, comment out or delete the call to fopen_std_file */
fopen_std_file(1, 1); /* stdout, die_on_error = TRUE */

printf(" Input value: %s\n", MYVAR);
for ( i=0; i < 20; i++) MYVAR[i] -= 32; /* shift case */
printf(" Output value: %s\n", MYVAR);
}
```

HP C++ Function Name Consideration

When a COBOL program calls a function compiled by HP C++, the function definition must include the extern "C" specification for the function name to be used in its original form. HP C++, by default, adjusts function names to accommodate generic functions and class methods. For example, a definition such as:

```
void EXT(char *p) {}
```

results in the creation of an entry point called `EXT_FPc`; however, a definition of the form:

```
extern "C" void EXT(char *p) {}
```

produces an entry point called `EXT`.

This is true in both the Guardian environment and the OSS environment.

Passing Parameters

Topics:

- Addressing Modes
- What HP COBOL Can Pass by Content
- What HP COBOL Can Pass by Reference
- What HP COBOL Can Pass by Value.

- Passing Parameters to COBOL Programs
- Passing Parameters to Non-COBOL Routines

Addressing Modes

Native programs use 32-bit addressing for all data items. Many HP COBOL items are byte-addressed. Some data items in other languages are 2-byte addressed. When you pass a byte-addressed item to a routine that expects a 2-byte-addressed parameter, the data item must be aligned on a 2-byte boundary; otherwise, problems can arise, because the called routine is expecting an aligned parameter.

Special registers are not addressable data items. If passed as parameters, they must be passed by value.

What HP COBOL Can Pass by Content

An HP COBOL program can pass these types of items to non-COBOL routines by content:

- Elementary Data Items
- Tables
- Numeric Literals
- Values of Arithmetic Expressions
- HP COBOL File Names

Elementary Data Items

The most practical categories and sizes for elementary data item parameters to be passed by content are:

Parameter Category and Size	USAGE
Numeric 2-byte	NATIVE-2
Numeric 4-byte	NATIVE-4
Numeric 8-byte	NATIVE-8
Alphanumeric	PICTURE X(*n*)

The names of elementary data item parameters can be qualified, subscripted, or both, but cannot include reference modifiers.

Passing a COMPUTATIONAL numeric data item as a parameter is not recommended. Although a data item described as

```
PICTURE S9(4) USAGE COMPUTATIONAL
```

is allocated 2 bytes, the value that can be stored in them must be in the COMPUTATIONAL range -9999 through 9999. Anything outside that range is truncated on the left by the MOVE operation.

The compiler generates any code necessary to convert numeric actual parameters to the form required by the external routine, including scaling and conversion to the required storage size and data type. Similarly, the compiler generates any code that is necessary to convert the numeric value that an external routine returns to the form required by the data item in the GIVING phrase of the HP COBOL program's ENTER statement.

The other CRE languages handle nonnumeric data items as strings.

Tables

To pass a table (an array) by content, you must create a data structure that consists solely of the array elements and pass the name of that data structure. The corresponding formal parameter must be defined to use a content parameter. For example, to pass the table of Cs in Example 24-3, you must pass the name C-ARRAY.

Example 24-3 Record Containing a Table

```
01 A-RECORD.
   03 A   PIC S999V99.
   03 B   PIC S999V99.
   03 C-ARRAY.
      05 C OCCURS 7 TIMES USAGE NATIVE-2.
   03 D   PIC X(15).
```

HP COBOL handles any data structure as a stream of characters, so the compiler does not perform numeric conversions on table elements. If you have a table of elements whose descriptions do not match those of their corresponding formal parameters, you must create a new table whose element descriptions match exactly, copy the values form the original table to the new table, and pass the name of the new table as the actual parameter.

Numeric Literals

HP COBOL can pass numeric literals, but not nonnumeric literals, as content parameters. The compiler generates any code that is necessary to convert them to the form required by the external routine.

Values of Arithmetic Expressions

HP COBOL can pass the value of an arithmetic expression to native HP C, native HP C++, and pTAL routines by content. The compiler generates any code that is necessary to convert the value of an arithmetic expression to the form required by the external routine.

HP COBOL File Names

HP COBOL can pass COBOL file names to ZCOBDLL routines that accept them. The COBOL file name is the file name recognized by the HP COBOL program's input-output statements, not the file name recognized by the operating environment. The data structures that enable the ZCOBDLL routines to accept COBOL file names are proprietary to HP and are not available to users.

What HP COBOL Can Pass by Reference

An HP COBOL program can pass these to non-COBOL routines by reference:

- Elementary Data Items
- Records

Elementary Data Items

The only elementary data items that HP COBOL programs can always pass by reference are NATIVE-2, NATIVE-4, NATIVE-8, and PICTURE X (*n*) data items. If an external routine stores an integer value in a COMPUTATIONAL numeric data item, and that value exceeds HP COBOL's limits (4 decimal digits for 16 bits of storage, 9 decimal digits for 32 bits of storage, and 18 decimal digits for 64 bits of storage), results from using the contents of that data item are unpredictable—the program might work as expected, not work as expected, or terminate abnormally.

HP COBOL programs can also pass by reference a numeric data item with a PICTURE containing *V* or *P*, but only if the corresponding formal parameter is the equivalent of an 8-byte integer and the HP COBOL program and external routine agree on an interpretation.

Some values on NonStop systems that fit into an HP COBOL data item described as PICTURE S9(*n* -*s*)V9(*s*) COMPUTATIONAL exceed HP COBOL's limits. More importantly, HP COBOL generates computational code based on the descriptions of data items, so if an external routine installs a value in a PICTURE S9(*n* -*s*)V9(*s*) COMPUTATIONAL field that HP COBOL could not have stored, the computational results could be invalid.

Records

HP COBOL programs must pass record parameters by reference.

What HP COBOL Can Pass by Value

An HP COBOL program can pass these types of items to non-COBOL routines by value:

- Numeric literals
- Numeric data items
- Special registers
- Arithmetic expressions

If a parameter is passed by value, it is evaluated, scaled, and converted to the storage size and type of the formal parameter. The resulting value is passed to the called routine. This conversion might cause an arithmetic overflow.

Passing Parameters to COBOL Programs

A COBOL program can pass parameters to another COBOL program with the CALL statement.

A non-COBOL routine can pass parameters to a COBOL program with the non-COBOL language's method of calling an external routine. The method varies from language to language; see the reference manual for appropriate non-COBOL language.

Topics:

- From HP COBOL Programs
- From Non-COBOL Routines

From HP COBOL Programs

By default, the CALL statement passes parameters by reference. This means that the called program can change the value of the parameter that is stored in the calling program.

You can specify that the CALL statement is to pass a parameter by content instead of by reference. This means that the called program cannot change the value of the parameter that is stored in the calling program. To specify that a parameter is to be passed by content, specify CONTENT in a USING phrase in the CALL statement.

A COBOL program can pass a program name (a *program-name* in a PROGRAM-ID paragraph) as a parameter to another COBOL program (see Dynamic Calls).

From Non-COBOL Routines

When calling a COBOL program from a non-COBOL program, remember that:

- A non-COBOL program must pass any parameters to COBOL program by reference.
- A non-COBOL program cannot pass a program name (a *program-name* for a PROGRAM-ID paragraph) to a COBOL program as a parameter.
- COBOL has no single-byte numeric data type that corresponds to:
  - HP C and C++ data types `char` and `unsigned char`
  - pTAL data types UNSIGNED(8) and STRING

  Enabling HP COBOL to obtain a numeric value from a single byte requires a work-around in the HP COBOL program. In your HP COBOL program, define a corresponding parameter and a temporary storage area, as in Example 24-4.

Example 24-4 TAL/pTAL Structures for Passing Parameters to COBOL Program

TAL or pTAL declaration:

```
STRUCT M;
  BEGIN
  INT X;
  STRING Y;
  UNSIGNED(8) Z;
  END;
```

Corresponding HP COBOL parameter:

```
01 STRUCT-M.
   03 X USAGE NATIVE-2.
   03 YZ.
      05 Y PIC X.
      05 Z PIC X.
```

HP COBOL temporary storage area:

```
01 TEMP-COMP.
   03 ONE-WORD-NUMERIC USAGE NATIVE-2.
   03 CHEATER REDEFINES ONE-WORD-NUMERIC.
      05 FILLER   PIC X.
      05 LOW-HALF PIC X.
```

In Example 24-4, when a non-COBOL routine (or a READ statement) has delivered a value to STRUCT-M, the HP COBOL program must move a numeric zero to ONE-WORD-NUMERIC and move the single-byte numeric value from either Y or Z into LOW-HALF. Then the HP COBOL program can use ONE-WORD-NUMERIC as a numeric value. For example, this code tests whether the value of Y is greater than 9:

```
MOVE ZEROS TO ONE-WORD-NUMERIC.
MOVE Y TO LOW-HALF
IF ONE-WORD-NUMERIC > 9
...
```

Passing Parameters to Non-COBOL Routines

Whether the ENTER statement passes parameters by reference or by value depends on what the called routine requires. The compiler determines whether the called routine requires a reference parameter or a value parameter. If the called routine requires a reference parameter, the compiler generates code that passes an address; if the called routine requires a value parameter, the compiler generates code that passes a value. (A called program can change the value of a reference parameter but not a value parameter.)

If the compiler cannot locate the object form of the called program or routine, or at least a dummy version that describes the expected parameters, then it cannot determine whether the parameters are to be passed by reference; therefore, it cannot generate the appropriate code, so it reports an error and does not produce an object program.

If a parameter is passed by reference so that the called routine can return a value to the calling program, the parameter declaration must match the expectation of the called routine, because HP COBOL code does not convert the values upon return from the routine.

A COBOL program cannot pass a program name (a *program-name* in a PROGRAM-ID paragraph) to a non-COBOL routine as a parameter.

Topics:

- HP C or HP C++
- pTAL

HP C or HP C++

- An HP COBOL program can pass numeric arguments to an HP C or HP C++ routine either by reference or by value.
- An HP COBOL program must pass nonnumeric arguments to an HP C or HP C++ routine by reference.
- If the data type of a value parameter differs from the data type of the corresponding formal parameter, the compiler converts the value of the value parameter to the data type of the formal parameter. If the value is outside the range of values allowed for the HP C or HP C++ data type, an arithmetic trap occurs.
- The compiler cannot convert reference parameters to different data types.
- An HP COBOL program can pass an integer (NATIVE-2, NATIVE-4, or NATIVE-8) data item to an HP C or HP C++ routine either as a reference parameter or by an external declaration. In both cases, the HP C or HP C++ routine receives the integer as a pointer type.
- The GIVING phrase cannot be specified in an ENTER statement that accesses an HP C or HP C++ function whose type is a pointer.

For more information about HP C or HP C++ routines and modules, see the *C/C++ Programmer's Guide* and the *Guardian Native C Library Calls Reference Manual*.

In Example 24-5, an HP COBOL program uses reference parameters to pass integer data items to a C routine. On return from the ENTER statement in the HP COBOL code, SQ contains the product of PAR1, PAR2, and PAR3, and the value of PAR3 is 25.

Example 24-5 Using Reference Parameters to Pass Integers to C Routine

HP COBOL code:

```
01 PARAMS.
   03 PAR1  USAGE IS NATIVE-2.
   03 PAR2  USAGE IS NATIVE-4.
   03 PAR3  USAGE IS NATIVE-8.
01 SQ USAGE IS NATIVE-8.
...
   ENTER C "blog" USING PAR1 PAR2 PAR3 GIVING SQ
...
```

C code:

```
long long blog (int *a, long *b, long long *c)
{
   long long result;
...
   result = *a * *b * *c;
   c = 25;
   return result;
}
```

In Example 24-6, an HP COBOL program uses external declarations to pass integer data items to a C routine.

Example 24-6 Using External Declarations to Pass Integers to C Routine

External declarations in HP COBOL:

```
01 FLIP USAGE IS NATIVE-2 EXTERNAL.
01 FLAP USAGE IS NATIVE-4 EXTERNAL.
01 FLOP USAGE IS NATIVE-8 EXTERNAL.
```

C code:

```
extern short FLIP;

extern long FLAP;
extern long long FLOP;

FLOP = (long long)((long)FLIP * FLAP);
```

Example 24-7 Passing a String to a C Routine

HP COBOL code:

```
DATA DIVISION.
WORKING-STORAGE SECTION.
01 A-STRING  PICTURE X(25).
...
PROCEDURE DIVISION.
...
ENTER C "FUNCT1" USING A-STRING
```

C routine:

```
FUNCT1 (char *d)
...
return void;
```

pTAL

- If a pTAL parameter is to be passed by value, an HP COBOL program can pass a numeric literal, a numeric data item, or the value of an arithmetic expression. The compiler generates code that evaluates, scales, and converts the value to the form that the called pTAL procedure expects.
- If a pTAL parameter is to be passed by reference, an HP COBOL program can pass an FD file name (to pass the address of the file control block) or a data item (to pass the address of the data item).
- An HP COBOL program that calls a pTAL procedure can omit any optional parameters, substituting the keyword OMITTED for each omitted optional parameter.
- In pTAL, if you declare a routine to be EXTENSIBLE or VARIABLE, its formal parameters are optional (meaning that the calling program need not supply the actual parameters). If an HP COBOL program calls an extensible or variable pTAL routine, the HP COBOL program must use the keyword OMITTED in place of each omitted optional parameter, except in the case of trailing omitted optional parameters. For example, if the pTAL procedure declaration is:

  ```
  PROC my_proc (a, b, c) VARIABLE;
     INT .EXT a;
     INT .EXT b;
     INT .EXT c;
  BEGIN
  ...
  END;
  ```

  Then an HP COBOL program can call the pTAL routine with any of these statements:

  ```
  ENTER TAL "MY_PROC" USING A
  ENTER TAL "MY_PROC" USING A B
  ENTER TAL "MY_PROC" USING A B C
  ENTER TAL "MY_PROC" USING A OMITTED C
  ```

- pTAL has a parameter type called `string:length` that is defined:

  ```
  a^proc (a^string:length)
     string .ext a^string
     int         length;
  ```

  An HP COBOL program that calls a pTAL routine that has a `string :length` parameter only passes one parameter, the string part. The compiler computes the length of the string and passes it to the pTAL routine.

For example, suppose that your HP COBOL program calls the pTAL routine FILE_GETINFOBYNAME_, which has the parameter `file-name :maxlen`. To have your HP COBOL program pass the entire item, you can use this code:

```
01 A-FILENAME PIC X(20)
...
ENTER "FILE_GETINFOBYNAME_" USING A-FILENAME ...
```

To have your HP COBOL program pass part of the item, you can use either reference modification or an OCCURS clause.

The code for reference modification is:

```
01 A-FILENAME PIC X(20)
...
MOVE 0 TO A-COUNT
INSPECT FUNCTION REVERSE (A-FILENAME) TALLYING A-COUNT
   FOR LEADING SPACES
ENTER "FILE_GETINFOBYNAME_"
       USING A-FILENAME
       (1: FUNCTION LENGTH (A-FILENAME) - A-COUNT) ...
```

The code for using an OCCURS clause is:

```
01  A-FILENAME.
    02 PIC X OCCURS 1 TO 20 TIMES DEPENDING ON
       A-FILENAME-LENGTH.
01  A-FILENAME-LENGTH PIC 99 COMP.
```

Set A-FILENAME-LENGTH to the right length and call FILE_GETINFOBYNAME_ without using reference modification.

For more information about pTAL procedures, see the *pTAL Reference Manual*.

Example 24-8 Passing Parameters to a pTAL Routine

HP COBOL code:

```
DATA DIVISION.
WORKING-STORAGE SECTION.
01 ONE-WORD USAGE NATIVE-2.
01 TWO-WORD USAGE NATIVE-4.
01 FOUR-WORD USAGE NATIVE-8.
01 A-STRING PICTURE X(25).
...
PROCEDURE DIVISION.
...
ENTER TAL "proc1" USING ONE-WORD
                        TWO-WORD
                        FOUR-WORD
                        A-STRING
```

pTAL routine:

```
PROC proc1 (a, b, c, d);
INT a;
INT(32) b;
FIXED(0) c;
STRING .d; -- pointer to array
BEGIN
...
END;
```

25 Issues Related to Linking

NOTE: Most of this section applies to the Guardian environment. If you are binding HP COBOL programs in the OSS environment, see Chapter 20: Using HP COBOL in the OSS Environment (page 721) and Mixed-Language Programs (page 722).

Concepts and Terms

These the concepts and terms are relevant to linking:

- Memory Areas (page 608)
- Code and Data Blocks
- Linking
- Loadfiles, Linkfiles, and Processes
- Comment for Top of Page

Code and Data Blocks

The compiler accepts HP COBOL source code and produces code and data blocks. The code blocks can call each other.

Topics:

- How the Compiler Produces Code and Data Blocks
- How One Code Block Calls Another

How the Compiler Produces Code and Data Blocks

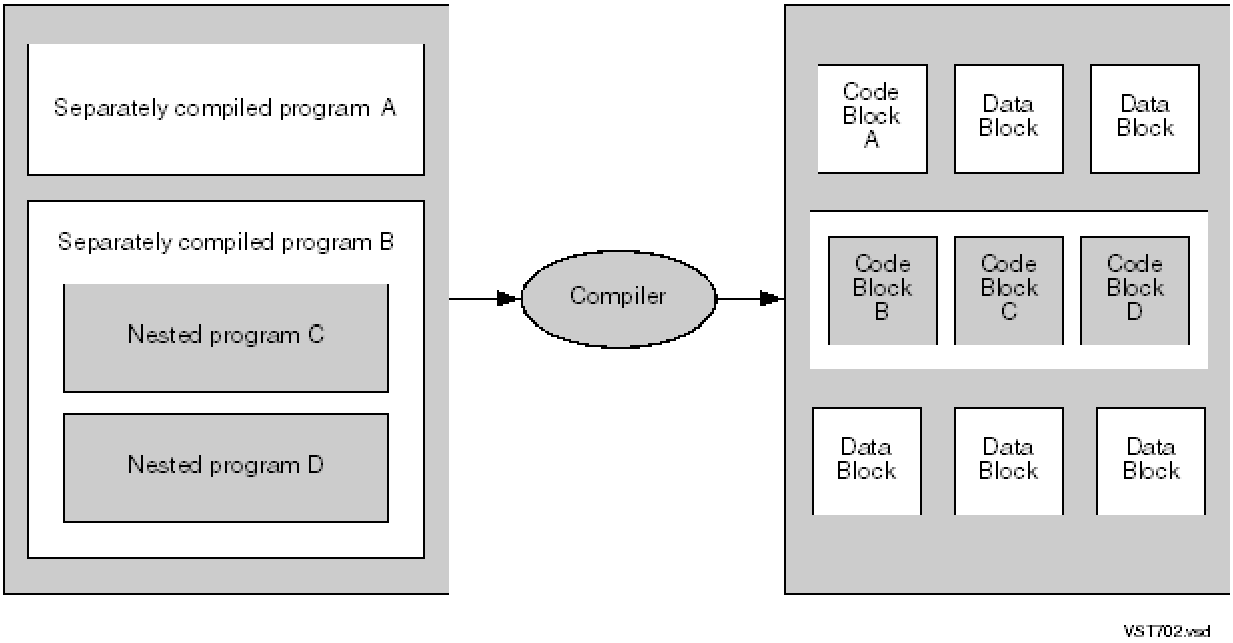
To understand how the compiler produces code and data blocks from HP COBOL source code, you must know about separately compiled programs and compilation units.

A separately compiled program is an HP COBOL source program that is not nested within any other program. It can have other HP COBOL source programs nested within it.

A compilation unit is a collection of source statements presented to a compiler in one compilation. It contains one or more separately compiled programs.

For each compilation unit, the compiler produces a single object file. For each separately compiled program in the compilation unit, the object file contains one relocatable code block for the outermost source program, one relocatable code block for each program nested inside the outermost source program, and one or more data blocks. Each nest of programs produces an indivisible nest of code blocks. In the object file, the code and data blocks exist solely as named code blocks and data blocks; they do not have a collective identity.

Figure 25-1 Code and Data Blocks



These compilers also produce one object file per compilation unit:

- TNS HP C
- TNS HP C++
- FORTRAN
- Pascal
- TAL
- Native (TNS/R and TNS/E) HP C
- Native (TNS/R and TNS/E) HP C++
- pTAL

The object file contains one relocatable code block for each procedure and zero or more data blocks.

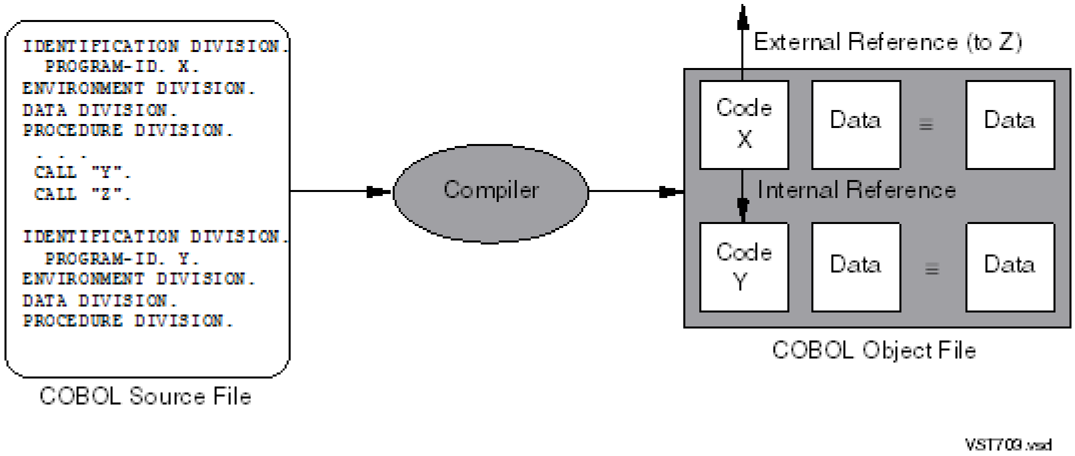
How One Code Block Calls Another

At the source program level, one program calls another with a CALL or ENTER statement; at the object program level, the calling program's code block calls the called program's code block.

A CALL statement calls another COBOL program. Each CALL statement in a compilation unit makes either an internal reference (calling another program in the compilation unit) or an external reference (calling a program that is not part of the compilation unit). In Figure 25-2, the first CALL statement makes an internal reference, and the second CALL statement makes an external reference.

An ENTER statement calls a non-COBOL program. Each ENTER statement makes an external reference.

Figure 25-2 Internal and External References



```
IDENTIFICATION DIVISION.
  PROGRAM-ID. X.
ENVIRONMENT DIVISION.
DATA DIVISION.
PROCEDURE DIVISION.
 . . .
 CALL "Y".
 CALL "Z".

IDENTIFICATION DIVISION.
  PROGRAM-ID. Y.
ENVIRONMENT DIVISION.
DATA DIVISION.
PROCEDURE DIVISION.
```

When external references in two code blocks are resolved to entry points in each other, the two code blocks are called “crossbound.” The system code block and the system library blocks are all crossbound—a routine in any of these blocks can call a routine in any other of these blocks. The blocks of a multiblock user code space are all crossbound, as are the blocks of user library space.

Linking

Linking is the operation of examining, collecting, and modifying code and data blocks from one or more object files to produce a single object file. Two important aspects of linking a program are:

- Validating references to other programs or routines—determining whether the actual parameters of the calling program or routine correspond to the formal parameters of the called program or routine.
- Resolving references to other programs or routines—generating the code that will transfer control from the calling program or routine to the called program or routine at execution time.

A typical compilation resolves some references and leaves others unresolved. (In Figure 25-2, the reference to program Y is resolved, and the reference to program Z is unresolved.)

Topics:

- How the Compiler Validates and Resolves References
- Unresolved References

How the Compiler Validates and Resolves References

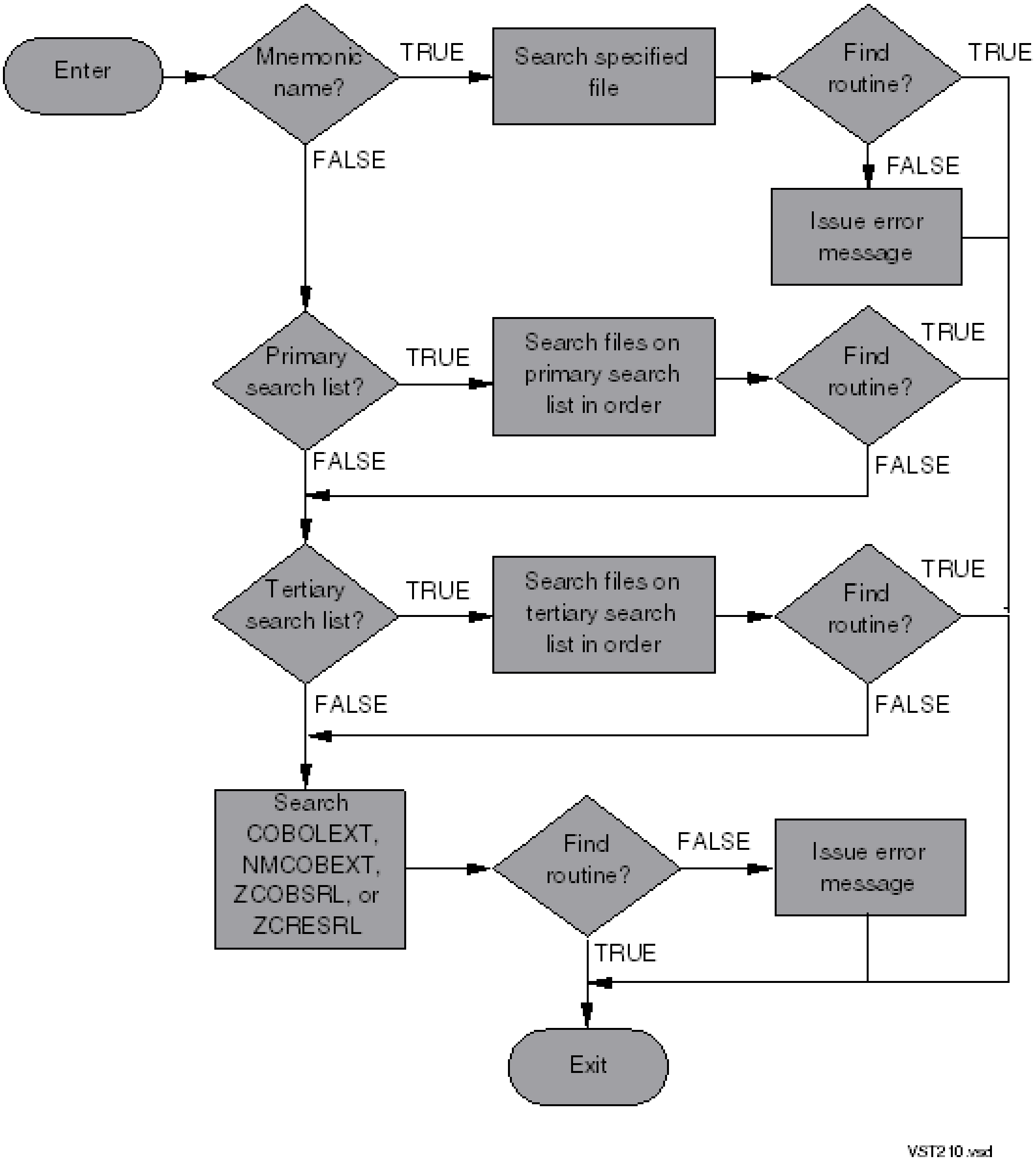
The compiler translates each CALL statement and each ENTER statement in an HP COBOL source program into an external reference. When the compiler generates an implicit call to a routine that is to execute in another memory area, the parameters match those expected by the routine. When you explicitly call a routine that is to execute in another memory area (with an ENTER or X/Open CALL statement), the compiler must verify that the parameters in the ENTER or X/Open CALL statement match those expected by the external routine. To verify this, the compiler searches for the routine (and as Figure 25-3: How External References Are Resolved (page 829) illustrates):

- If the ENTER or X/Open CALL statement identifies the file that contains the routine with a mnemonic name, the compiler searches that file only. If the compiler does not find the routine in that file, it reports an error.
- If the compiler does not identify the file that contains the routine, it searches for the routine in the files on the primary search list if one exists. (One or more SEARCH directives build the primary search list; see Putting an Object File on the Primary Search List.)
- If the compiler does not find the routine on the primary search list, it searches for the routine in the files on the tertiary search list if one exists. (One or more CONSULT directives build the tertiary search list; see Putting an Object File on the Tertiary Search List (page 832).)
- If the compiler does not find the routine in a file on the tertiary search list, it searches for the routine in the file ECOBEXT, ZCOBDLL, and ZCREDLL.
- If the compiler does not find the routines in ECOBEXT, ZCOBDLL or ZCREDLL, it reports an error.

If the compiler cannot validate the parameters of an ENTER statement in a program unit, it does not produce an object file for that program unit.

When the compiler generates an implicit reference to one of its own run-time routines or to an operating system routine, the compiler leaves these references unresolved so that the code of the corresponding routines will execute in system code space.

Figure 25-3 How External References Are Resolved



Unresolved References

If an external reference in an object file refers to an entry point that is not among the code blocks in that object file, the external reference is called unresolved. Unresolved external references cause warning messages, but are allowed. They are expected to be resolved by the fixup operation of the system loader. If the compiler cannot resolve an external reference to a non-COBOL routine, it must at least be able to validate it.

When the compiler processes a CALL statement that calls another COBOL program, but cannot find the other COBOL program, it delivers a warning message and produces an object file.

When the compiler processes an ENTER statement that calls a routine, but cannot find the routine, it delivers an error message and does not produce object code for the calling program. The reason that the compiler cannot produce object code for the calling program is that the compiler is unable to verify that the parameters of the call match those expected by the routine (and, where necessary, to generate code that converts parameters in the HP COBOL program into the form required by the routine).

For more information, see How the Compiler Resolves Unqualified References (page 526).

Loadfiles, Linkfiles, and Processes

The single object file that results from linking is a loadfile if exactly one program in it is a main program; otherwise, it is a linkfile.

A single execution of a loadfile is called a process. The operating environment creates a process by loading a loadfile into memory, creating entries for the process in various operating environment tables, and transferring control to the entry point of the main routine of the process.

In memory, a process occupies code spaces, data spaces, and a process file segment. The code spaces and data spaces (memory areas) are described in Table 14-2: Memory Area Characteristics (page 608). The process file segment is an extended data segment available only to the operating environment, which uses it for things such as managing the status of communication with files.

Fixup

The first time you instruct the operating environment to execute a loadfile as a process, the system loader performs a fixup operation on the file and attempts to resolve all unresolved external references.

External references call routines in:

- Other code blocks in the same user code space
- Code blocks in DLLs
- The system code space

If the system loader cannot resolve an external reference, it reports that there are still unresolved externals (?EXT: *name*). The process executes until it attempts to call an unresolved external procedure, and then it calls the selected debugger.

The fixup operation is the reason that the first execution of a newly compiled program is less efficient than subsequent executions. After the system loader has performed the fixup, subsequent executions of the program do not need this operation until the contents of the loadfile are changed or the operating environment is changed.

This run-time resolution helps to reduce object file size and compilation times. It also reduces the likelihood that you will have to recompile an HP COBOL program when a run-time routine that it calls is enhanced or corrected.

Linking HP COBOL Programs

An HP COBOL program can be linked automatically, as part of compilation: If you specify the RUNNABLE or SEARCH directive, the ECOBOL compiler also calls the eld utility. If you do not link the program automatically, you must use the eld utility to link it after compilation.

Topics:

- Linking Automatically (page 830)
- Linking Programs to Be Called Dynamically (page 832)

Linking Automatically

If you specify the RUNNABLE directive, the compiler automatically links the program, interacting with the `eld` utility.

For simplicity, this explanation attributes actions of internal processes and the `eld` utility to the compiler.

Unless all the external routines that your program calls are in the file ECOBEXT, ZCOBDLL, or ZCREDLL, you must tell the compiler which object files contain the external routines that your program calls. There are several ways to tell the compiler about a specific object file:

- Establish a mnemonic name for the object file and use it in CALL or ENTER statements.
- Put the object file's name on the primary search list.

  If the compiler finds an external routine in an object file on the primary search list, it validates the routine's parameters and includes the entire object file in your program's object file.
- Put the object file's name on the tertiary search list.

  If the compiler finds an external routine in an object file on the tertiary search list, it validates the routine's actual parameters but does not include the routine in your program's object file. The system loader resolves the routine at fixup.

You can use a predefined DEFINE, =_OBJECT_SEARCH, to specify one or more subvolumes for the compiler to search for unqualified object files (see Specifying Subvolumes to Be Searched for Unqualified Files (page 767)).

Topics:

- Establishing a Mnemonic Name for an Object File (page 831)
- Putting an Object File on the Primary Search List (page 831)
- Putting an Object File on the Tertiary Search List (page 832)

Establishing a Mnemonic Name for an Object File

If a CALL or ENTER statement contains a mnemonic name that specifies the object file that contains the called program, the compiler searches that file (and only that file) for the specified routine. If the compiler does not find the routine, it reports an error.

You establish a mnemonic name for an object file with a File-Mnemonic clause in the SPECIAL-NAMES paragraph of the Environment Division. Suppose that the object file has the system file name MYLIB and you want to give it the mnemonic name HERLIB. The File-Mnemonic clause is:

```
FILE "MYLIB" IS HERLIB.
```

Suppose that MYLIB contains a COBOL program named APROG. Your HP COBOL program can call APROG with the statement:

```
CALL "APROG" OF HERLIB.
```

Quotation marks around the program name (APROG) are not required, but are recommended. If, in some eventual program maintenance, someone adds a variable named APROG to a program that already calls a program named APROG, and the name APROG is not in quotation marks, the compiler assumes that APROG refers to the variable. At run time, if the variable APROG does not contain the name of a valid program that is included in the object file, the program terminates with the error message "Called program not found." If APROG is enclosed in quotes, the compiler recognizes it as a program name rather than a variable.

Putting an Object File on the Primary Search List

If a CALL or ENTER statement does not contain a mnemonic name that specifies the object file that contains the called program, the compiler searches for the called routine in the object files on the primary search list. If the compiler finds the routine, it validates the routine's actual parameters and includes the entire object file (from the primary search list) in the object file that it is creating.

You can put one or more object files on the primary search list with one or more SEARCH directives. The first SEARCH directive establishes the primary search list and adds files to it in the specified order. Subsequent SEARCH directives append files to the primary search list in the specified order. The compiler searches the object files of the primary search list in order. If the compiler finds the routine, it stops searching for it.

If you put a SEARCH directive on the compilation command line when you compile your program, the compiler sees that SEARCH directive first, so the primary search list begins with that SEARCH directive's object files.

Putting an Object File on the Tertiary Search List

If the compiler does not find an external routine in the user library, it searches for the routine in the object libraries on the tertiary search list. If the compiler finds the routine in a library on the tertiary search list, it validates the routine's actual parameters but does not include the routine in your object file. The system loader resolves the routine at fixup.

You can put one or more object libraries on the tertiary search list with one or more CONSULT directives. The first CONSULT directive establishes the tertiary search list, adding files to it in the specified order. Subsequent CONSULT directives append libraries to the tertiary search list in the specified order. The compiler searches the libraries of the tertiary search list in order. If the compiler finds the routine, it stops searching for it. Object libraries on the tertiary search list are not bound into the loadfile.

Suppose that the files RTNS1985 and RTNS1988 contain object code for external routines that your HP COBOL program calls, and that most of the external routines that your program calls are in RTNS1988. For greatest efficiency, put RTNS1988 first on the tertiary search list. You can use the directive:

```
?CONSULT RTNS1988,RTNS1985
```

You could also use multiple CONSULT directives, which can be anywhere in your HP COBOL program. This code produces the same tertiary search list as the preceding directive:

```
?CONSULT RTNS1988
?CONSULT RTNS1985
```

If you also put a CONSULT directive on the compilation command line when you compile your program, the compiler sees it first, so the tertiary search list begins with the object libraries in the CONSULT directive on the compilation command line.

If the compiler does not find the routine in any object file on the tertiary search list, it searches for the routine in the file ECOBEXT, ZCOBDLL, or ZCREDLL. If the ECOBOL compiler does not find the routine in ECOBEXT, ZCOBDLL, or ZCREDLL, the compiler reports an error.

Linking Programs to Be Called Dynamically

A dynamic call is one in which the name of the called program is not known until run time (see Dynamic Calls (page 810)). Every program that a dynamic call could possibly call must be bound into the run unit or user library before you execute the program. There are several ways to do this:

- Compile all the programs in the run unit from a single source file.
- Use the linker to build the required program units into the run unit.
- Refer to each possible called program explicitly in the source program—in a paragraph that is not executed—and then call the ones you want with CALL `identifier` statements.

At run time, if `identifier` specifies the name of a program that is unavailable, the run-time routines raise an exception.

COBOL Segmentation

NOTE: The 1985 COBOL Standard classifies the segmentation module as **obsolete**, so you are advised not to use it.

HP COBOL ignores segment numbers, but issues warning messages for transfers of control forbidden by the segmentation rules of the ANSI COBOL Standard. HP COBOL also issues warning messages for altered GO TO statements in independent segments, because such alterations will not be restored when control returns to the independent segments.

26 Executing and Debugging HP COBOL Programs

NOTE: This section applies to the Guardian environment. If you are compiling HP COBOL programs in the OSS environment, see Running HP COBOL Programs (page 722).

After you have compiled and bound your program, you can execute and, if necessary, debug it.

Preparing to Execute an HP COBOL Program

If your program has certain characteristics, or if you want certain things from a particular execution of your program, you must prepare appropriately before you execute your program.

Table 26-1 Conditions Requiring Pre-Execution Preparation

Condition	Preparation Required Before Program Execution
Program assigns HP COBOL file names to DEFINE names.	Add DEFINEs that have those DEFINE names (see Adding DEFINEs).
Program assigns HP COBOL file names to system file names and you want to override those assignments.	Override those assignments (see Overriding File Assignments Made at Compilation Time).
Program creates some of the files it uses and you want to specify their characteristics.	Specify their characteristics (see Specifying Characteristics of Files That a Program Creates).
Program was compiled with a NONSTOP directive that you want to override.	Override the NONSTOP directive (see Overriding the NONSTOP Directive).
You do not want your home terminal to be the default input-output device.	Specify the default input-output device that you want (see Specifying a Default Input-Output Device).
You want the operating environment to return control to your program if a certain printer is unavailable or out of paper.	Specify this preference (see Providing for an Unavailable Printer).
Program declares external switches in the SPECIAL-NAMES paragraph and you want some of them to be ON when your program begins to execute.	Turn on switches (see Turning On External Switches).
You want the program to call a debugger, rather than ABEND, after printing any fatal error messages.	Specify this preference (see Requesting a Debugger Instead of ABEND).
You want the program to do all of its input-output in wait mode.	Specify this preference (see Specifying Waited Input-Output).
You want to establish one or more sets of preparation commands for your program and put a set into effect with a single command.	Put each set of preparation commands into an OBEY command file and execute the appropriate OBEY command file (see Using OBEY Files to Prepare for Execution).
You want to know what preparation commands are already active.	Find out what these commands are (see Finding Out What Preparation Commands Are Already Active).
You want to change or clear active preparation commands.	Clear commands (see Changing or Clearing Unwanted Preparation Commands).

Preparation commands stay active until you override them, clear them, or end your TACL session.

Adding DEFINEs

If your program assigns COBOL file names to DEFINE names, you must add DEFINEs that have those DEFINE names before executing your program. You add a DEFINE with the TACL command ADD DEFINE. Before you can add DEFINEs, you must set your TACL process DEFMODE attribute to ON (the default is OFF) with the TACL command:

```
SET DEFMODE ON
```

DEFINE Names

A DEFINE name begins with an equal sign (=) followed by a letter and up to 22 additional characters, which can be letters, digits, hyphens (-), underscores (_), and circumflexes (^).

Example 26-1 DEFINE Names

```
=NEWFILE
=FILE25
=TAPE-FILE
=FTEST_RESULTS
=JEFFS^FILE
```

DEFINE Attributes

Every DEFINE has a CLASS attribute, which determines its other attributes. For details, see the section of this manual that explains input-output for the system file that you want to associate with the DEFINE name:

System file is a ...	Section Number and Name
Tape file	Chapter 28: Tape Input and Output (page 855)
Disk file	Chapter 29: Disk Input and Output (page 873)
Terminal	Chapter 30: Terminal Input and Output (page 907)
Printer or spooler	Chapter 31: Printer and Spooler Output (page 917)

Overriding File Assignments Made at Compilation Time

To override a file assignment made at compile time—an assignment of a COBOL file name to a system file name—you use an ASSIGN command.

Suppose that your program, PROG1, assigns the COBOL file name MAJORACCT to the system file name \AKRON.$SLB.MAJ.ACC with this SELECT clause:

```
SELECT MAJORACCT ASSIGN TO "\AKRON.$SLB.MAJ.ACC"
```

When you execute PROG1 and it opens MAJORACCT, the file it actually opens is \AKRON.$SLB.MAJ.ACC. Suppose that you want MAJORACCT to be another file, \NICE.$FRNC.SIGNIF.CUST, instead. You can override the original file assignment before executing PROG1 with the command:

```
ASSIGN PROG1.MAJORACCT,\NICE.$FRNC.SIGNIF.CUST
```

When you execute PROG1 and it opens MAJORACCT, the file it actually opens is \NICE.$FRNC.SIGNIF.CUST rather than \AKRON.$SLB.MAJ.ACC.

The number of ASSIGN commands the command interpreter can store is limited by the amount of memory allocated to the command interpreter. The default amount of memory allocated to the command interpreter is 8 pages, which accommodates up to 19 ASSIGN commands. To allocate more memory to the command interpreter, use the MEM option in the RUN command that you use to run the command interpreter. Each additional page of memory allocated to the command interpreter accommodates an additional 19 ASSIGN commands.

Specifying Characteristics of Files That a Program Creates

If your program is to create a file, you can specify these characteristics of that file with an ASSIGN command:

- The actual file name with which the COBOL file name is to be associated
- The size of the file (extent size)

- A file code
- An exclusion mode (EXCLUSIVE, SHARED, or PROTECTED) for reading and writing (but it does not override any exclusion mode that the program explicitly specifies)

The ASSIGN command Example 26-2 specifies that the file that the program calls NEWFILE has the system file name \AKRON.$SLB.MAJ.NEWFILE, is allocated five file pages, is an EDIT file (type 101), and EXCLUSIVE exclusion mode (unless the OPEN statement that opens the file specifies a different exclusion mode).

Example 26-2 ASSIGN Command

```
ASSIGN PROG1.NEWFILE,
       \AKRON.$SLB.MAJ.NEWFILE,
       EXT 5,
       CODE 101,
       EXCLUSIVE
```

You can also specify file characteristics with a DEFINE. For details on specifying file characteristics with either the ASSIGN command or DEFINEs, see ASSIGN Command (page 590) and DEFINEs (page 601).

Overriding the NONSTOP Directive

If your program was compiled with a NONSTOP directive (which causes it to execute as a process pair), and you want to override the NONSTOP directive (until you have debugged the program, for example), you must execute the command PARAM NONSTOP OFF before executing your program. The PARAM NONSTOP OFF command is:

```
PARAM NONSTOP OFF
```

You can use a single PARAM command to do several things; for example, this PARAM command overrides the NONSTOP directive and enables USE DEBUGGING SECTIONS:

```
PARAM NONSTOP OFF, DEBUG ON
```

For another example of a PARAM command that does several things, see Specifying Waited Input-Output (page 838).

Specifying a Default Input-Output Device

First, verify that you want to specify a default input-output device. The alternative is to specify separate default input and output devices.

The default input device (or the default input-output device) provides input to unqualified ACCEPT statements (those without FROM clauses).

The default output device (or the default input-output device) displays run-time diagnostic messages and output from unqualified DISPLAY statements (those without UPON clauses).

If a program has unqualified ACCEPT statements, the default input device (or the default input-output device) must be a disk file, terminal, or process. If a program has unqualified DISPLAY statements, the default output device (or the default input-output device) must be a disk file, terminal, process, or printer. If a program has both unqualified ACCEPT statements and unqualified DISPLAY statements, the default input-output device must be a terminal or process.

If you want separate default input and output devices, do not specify a default input-output device. See Specifying Default Input and Output Devices.

If you want a default input-output device other than your home terminal, specify the default input-output device as explained in the *CRE Programmer's Guide*.

If default input-output device is a system file name, the file must exist before you execute the program. If the default input-output device is a DEFINE name, you must add it before you execute the program (see Adding DEFINEs). If default input-output device is an asterisk, all run-time messages are discarded.

This command specifies the default input-output device \AKRON.$SLB.MAJ.ERRMSG:

```
PARAM EXECUTION-LOG \AKRON.$SLB.MAJ.ERRMSG
```

This command specifies the DEFINE name =ERRORS as the default input-output device. You must add the DEFINE name =ERRORS before you execute the program.

```
PARAM EXECUTION-LOG =ERRORS
ADD DEFINE =ERRORS, FILE \AKRON.$SLB.MAJ.ERRMSG
```

This command causes all run-time messages to be discarded:

```
PARAM EXECUTION-LOG *
```

Providing for an Unavailable Printer

If you want the operating environment to return control to your program if a certain printer is unavailable or out of paper, you must execute a PARAM PRINTER-CONTROL command before executing your program.

This command specifies that control is to return to the program if the printer assigned to THE-PRINTER is unavailable or out of paper:

```
PARAM PRINTER-CONTROL THE-PRINTER
```

Turning On External Switches

If your program declares external switches in the SPECIAL-NAMES paragraph and you want any of them to be in the ON position when your program begins to execute, you must turn them on with PARAM SWITCH-*nn* commands before executing your program. You can turn on several switches with a single PARAM SWITCH-*nn* command or use multiple PARAM SWITCH-*nn* commands.

This command turns on SWITCH-1, SWITCH-4, and SWITCH-15:

```
PARAM SWITCH-1 ON, SWITCH-4 ON, SWITCH-15 ON
```

This series of commands is equivalent to the previous command:

```
PARAM SWITCH-1 ON
PARAM SWITCH-4 ON
PARAM SWITCH-15 ON
```

Requesting a Debugger Instead of ABEND

If you want your program to call a debugger, rather than ABEND, after printing any fatal error messages, you must execute a PARAM INSPECT ON command before executing your program. The PARAM INSPECT ON command is:

```
PARAM INSPECT ON
```

Specifying Waited Input-Output

To cause your HP COBOL program to do all its input-output in the wait mode, execute the command PARAM WAITED-IO ON before you execute the program. For more information, see PARAM Command (page 594).

Using OBEY Files to Prepare for Execution

If you expect to have to enter the same set of preparation commands repeatedly, you can save time by putting those commands in an OBEY command file. You can create an OBEY command file with an HP editor (see the example in Running the Compiler (page 765)).

Suppose that you almost always need to execute these preparation commands before executing your program, PROG1, so you put them into an OBEY command file named PREP1:

```
ADD DEFINE =BIGCUST, FILE \AKRON.$SLB.MAJ.ACC
ASSIGN PROG1.NEWFILE,&
       \AKRON.$SLB.MAJ.NEWFILE,&
       EXT 5, CODE 101, EXCLUSIVE
PARAM NONSTOP OFF
PARAM INSPECT ON
PARAM EXECUTION-LOG \AKRON.$SLB.MAJ.ERRMSG
PARAM PRINTER-CONTROL THE-PRINTER
PARAM SWITCH-1 ON, SWITCH-2 ON
```

Then you can execute all of the preceding commands with the single command:

```
OBEY PREP1
```

Suppose that you occasionally want to execute PROG1 with the file \NICE.$FRNC.SIGNIF.CUST in place of the file \AKRON.$SLB.MAJ.ACC, have PROG1 create the file \NICE.$FRNC.SIGNIF.NEWFILE instead of the file \AKRON.$SLB.MAJ.NEWFILE, and send run-time library routine messages to the file \NICE.$FRNC.SIGNIF.ERRMSG instead of the file \AKRON.$SLB.MAJ.ERRMSG. You can put these commands in an OBEY command file named PREP1A:

```
DELETE DEFINE =BIGCUST
ADD DEFINE =BIGCUST, FILE \NICE.$FRNC.SIGNIF.CUST
ASSIGN PROG1.NEWFILE,&
       \NICE.$FRNC.SIGNIF.NEWFILE,&
       EXT 5, CODE 101, EXCLUSIVE
PARAM EXECUTION-LOG \NICE.$FRNC.SIGNIF.ERRMSG
```

Then you can execute the commands in PREP1 (some of which you still want) and override them with the commands in PREP1A with this command series:

```
OBEY PREP1
OBEY PREP1A
```

You can achieve the same result with one command, OBEY PREP2, by putting these commands in an OBEY command file named PREP2:

```
ADD DEFINE =BIGCUST, FILE \NICE.$FRNC.SIGNIF.CUST
ASSIGN PROG1.NEWFILE,&
       \NICE.$FRNC.SIGNIF.NEWFILE,&
       EXT 5, CODE 101, EXCLUSIVE
PARAM NONSTOP OFF
PARAM INSPECT ON
PARAM EXECUTION-LOG \NICE.$FRNC.SIGNIF.ERRMSG
PARAM PRINTER-CONTROL THE-PRINTER
PARAM SWITCH-1 ON, SWITCH-2 ON
```

An equivalent command sequence with the PARAM commands combined is:

```
ADD DEFINE =BIGCUST, FILE \NICE.$FRNC.SIGNIF.CUST
ASSIGN PROG1.NEWFILE,&
       \NICE.$FRNC.SIGNIF.NEWFILE,&
       EXT 5, CODE 101, EXCLUSIVE
PARAM NONSTOP OFF, INSPECT ON, SWITCH-1 ON, SWITCH-2 ON,&
      EXECUTION-LOG \NICE.$FRNC.SIGNIF.ERRMSG,&
      PRINTER-CONTROL THE-PRINTER
```

NOTE: The maximum number of DEFINES, ASSIGN commands, and PARAM commands that can be active at one time depends on the amount of memory allocated to the TACL command interpreter. For more information, see the *TACL Reference Manual*.

Finding Out What Preparation Commands Are Already Active

To find out if any DEFINEs are active, use this TACL command:

```
INFO DEFINE =*
```

To find out the characteristics of a DEFINE that you know is active, such as DEFINE FILE $BEA.LEACH.HYER, use either of these commands:

```
SHOW DEFINE FILE
SHOW DEFINE *
```

To find out what ASSIGN commands are active, give the ASSIGN command with no parameters:

```
ASSIGN
```

To find out what PARAM commands are active, give the PARAM command with no parameters:

```
PARAM
```

Changing or Clearing Unwanted Preparation Commands

To change or clear a DEFINE, you must use these TACL commands:

TACL Command	Function
ALTER DEFINE	Changes the attribute values of an active DEFINE
DELETE DEFINE	Clears (deletes) an active DEFINE

For details on the preceding TACL commands, see the *TACL Reference Manual*.

To change or clear an ASSIGN or PARAM command, you use the TACL command CLEAR. Some examples of the CLEAR command follow.

Active Command	TACL Command That Clears It
ASSIGN PROG1.MAJORACCT,& \NICE.$FRNC.SIGNIF.CUST	CLEAR ASSIGN PROG1.MAJORACCT
PARAM NONSTOP OFF	CLEAR PARAM NONSTOP
One or more ASSIGN commands	CLEAR ALL ASSIGN
One or more PARAM commands	CLEAR ALL PARAM
One or more ASSIGN or PARAM commands	CLEAR ALL

Executing an HP COBOL Program

You can start your program executing either from a TACL prompt (that is, from your TACL process) or from any other process. In either case, the operating environment creates a new process—your executing program—and that process is affected by any ASSIGN commands, PARAM commands, and DEFINEs that are active when the process is created (see Preparing to Execute an HP COBOL Program).

Like every new process, your executing HP COBOL program receives a startup message from the operating environment. The startup message contains names for the IN file, the OUT file, the home terminal, and the default volume and subvolume. As an HP COBOL application programmer, you need not get involved with the startup message itself. If you need to manipulate the startup message, you can use Saved Message Utility (SMU) routines.

Starting an HP COBOL Program From a TACL Prompt

To start an HP COBOL program from the TACL prompt, use the TACL command RUN. In the simplest case, the program retrieves input for unqualified ACCEPT statements from your home terminal and displays run-time diagnostic messages and the output of unqualified DISPLAY statements on your home terminal. If the file ID of your loadfile is $SYSTEM3.ECOBOL.PROG1, you can execute your program with this command:

```
RUN $SYSTEM3.ECOBOL.PROG1
```

You can omit the keyword RUN and the subvolume that your program is on if a default subvolume is specified on #PMSEARCHLIST. To specify $VOLUME.SUBVOL as a default on #PMSEARCHLIST, execute this TACL command before starting your program:

```
#SET #PMSEARCHLIST #DEFAULTS $VOLUME.SUBVOL $SYSTEM.SYSTEM
```

Then you can start the program whose file ID is $VOLUME.SUBVOL.PROG1 from the TACL prompt with this command:

```
PROG1
```

The RUN command has many options, some of which allow you to specify these program attributes:

- Default input and output devices
- Home terminal
- Debugger
- Process name (required if program is to run as a process pair)
- Ability to run in the background

For more information about the RUN command and its options, see the *TACL Reference Manual*.

To start an HP COBOL program from another HP COBOL program, see Initiating a Process From an HP COBOL Program (page 934).

Specifying Default Input and Output Devices

If you want your program to retrieve input for unqualified ACCEPT statements from a device other than your home terminal, you must specify that device.

If you want your program to display run-time diagnostic messages and output from unqualified DISPLAY statements to a device other than your home terminal, you must specify that device.

If you want the default input device to be the same as the default output device, you can specify a default input-output device with a PARAM EXECUTION-LOG command before you execute your program (see Specifying a Default Input-Output Device) and ignore the remainder of this topic.

IN and If you want separate default input and output devices, you must specify them with the OUT options of the RUN command and must not use a PARAM EXECUTION-LOG command (because the PARAM EXECUTION-LOG command takes precedence over the IN and OUT options of the RUN command).

The default input and output devices can be system files (including disk files) or DEFINE names that you have already associated with system files. The default input file must be a terminal, disk file, or process; the default output file must be a terminal, disk file, line printer, or process.

This command specifies the default input device INFILE and the default output device OUTFILE for the program PROG1. (Assume that PROG1, INFILE, and OUTFILE are on the default subvolume.)

```
PROG1 /IN INFILE, OUT OUTFILE/
```

This command specifies the default input device INFILE for the program PROG1 but no default output device. The default output device will be the home terminal.

```
PROG1 /IN INFILE/
```

This command specifies the default output device OUTFILE for the program PROG1 but no default input device. The default input device will be the home terminal.

```
PROG1 /OUT OUTFILE/
```

Specifying the Home Terminal

By default, the home terminal for your executing HP COBOL program is the same as the home terminal for your TACL process. This is a problem if all of these conditions are true:

- The default input-output device, default input device, or default output device is the home terminal.
- The TACL process has control of the home terminal, causing HP COBOL run-time routines to wait for the terminal to become available.
- One of the waiting routines is a Pathway server.

If the suspended Pathway server's requester is also suspended, the performance of the Pathway application is degraded.

If your program has the previously described problem, specify a different home terminal for your ECOBOL process with the TERM option of the RUN command. Verify that the new home terminal is available.

This command specifies the home terminal $TE1.#E02 for the program PROG1. Because the default input and output devices are not specified, they will be the home terminal, $TE1.#E02, unless an earlier PARAM EXECUTION-LOG command specified a default input-output device.

```
PROG1 /TERM $TE1.#E02/
```

This command specifies the default input device INFILE for the program PROG1, no default output device, and the home terminal $TE1.#E02. The default output device will be the home terminal, $TE1.#E02.

```
PROG1 /IN INFILE, TERM $TE1.#E02/
```

To find the name of a terminal, use the TACL command WHO. Example:

```
223> WHO
Home terminal: $TE1.#E02
...
```

Naming the New Process

If your HP COBOL program is to run as a process pair, it must have a process name. If your program is not to run as a process pair, a process name is optional but useful: it allows you to stop the process by name, which is easier than determining its processor and Process Identification Number (PIN) and then stopping it with those.

You verify that your program has a process name in either of these two ways:

- Include the NAME Option in the RUN command.
- Compile the program with the RUNNAMED Directive.

NAME Option

The NAME option of the RUN command verifies that the program is run as a named process. You can specify the process name or not. If you do, it must begin with a dollar sign ($). (For the syntax of a process name, see the *Guardian Programmer's Guide*.) If you do not specify the process name, the operating environment assigns a timestamp process name to the new process.

This command assigns the process name “$PROG1” to the process created when you run $VOL1.SUB2.PROG1:

```
RUN $VOL1.SUB2.PROG1 /NAME $PROG1/
```

This command causes the operating environment to assign a timestamp process name to the process created when you run $VOL1.SUB2.PROG1:

```
RUN $VOL1.SUB2.PROG1 /NAME/
```

This command runs the program $VOL1.SUB2.PROG1 as an unnamed process (unless you specified RUNNAMED in the object file).

```
RUN $VOL1.SUB2.PROG1
```

RUNNAMED Directive

If you compile your program with the RUNNAMED directive, the operating environment assigns it a timestamp process name when you execute it. You do not need to include the NAME option in the RUN command (but you can, if you want to give it a specific name).

The RUNNAMED directive works only if the RUNNABLE directive is active.

Running the New Process in the Background

By default, your TACL process suspends itself while your HP COBOL program executes. You can prevent this by running your program in the background with the NOWAIT option of the RUN command.

Beware of using the NOWAIT option if your program has ACCEPT and DISPLAY statements that use the home terminal. With NOWAIT, your HP COBOL program cannot accept input from, or display output to, the home terminal until you pause the terminal. Without NOWAIT, you can access your TACL process by pressing Break when you are certain that the COBOL-created process is done with the home terminal.

This command runs the program PROG1 in the background:

```
PROG1 /NOWAIT/
```

Run-Time Errors

Run-time errors are errors that arise during the execution of a process. Examples of their causes are:

- The process tries to open a nonexistent file.
- The process tries to open a file that is inaccessible because another process has it open exclusively.
- The process tries to call a non-COBOL program that is not in the process's own loadfile, the system library, or the TNS or user library.

Diagnostic Messages

When a run-time error occurs, the run-time routine that detects it sends a run-time diagnostic message to the default output device. The default output device is the home terminal unless you changed it with the PARAM EXECUTION-LOG command (see Specifying a Default Input-Output Device) or the OUT option of the RUN command (see Specifying Default Input and Output Devices).

A run-time diagnostic message identifies the error, the process in which the error occurred, and the loadfile from which the process was loaded. The error number and format of a run-time diagnostic message depend on the run-time environment. For details, see Section 48, Run-Time Diagnostic Messages.

One category of run-time diagnostic messages is not reported to the default output device: input-output exceptions with I-O status codes less than “30” that your program handles with its own error-handling procedures.

Error-Handling Procedures

In your HP COBOL program, you can define error-handling procedures with USE AFTER EXCEPTION statements. When a file system error occurs, control passes to a USE AFTER EXCEPTION statement, which can access two values to determine what action to take. The first value is the I-O status code, which identifies the error; the second value is that of the special register GUARDIAN-ERR, which identifies the cause of the error.

One I-O error can have several possible causes; therefore, one I-O code status value can be associated with several GUARDIAN-ERR values. For example, a WRITE statement error (I-O code status value “24”) can be caused by a relative key that is out of bounds (GUARDIAN-ERR value 23), a full file (GUARDIAN-ERR value 45), or an invalid key (GUARDIAN-ERR value 46). The USE AFTER EXCEPTION statement can interpret the value of GUARDIAN-ERR and act accordingly.

After a USE AFTER EXCEPTION statement executes, it returns control to the routine that called it, except when the I-O status code is “4*x* ” or “90,” in which case the process terminates abnormally.

For more information about the USE AFTER EXCEPTION statement and diagnosing input-output errors, see USE AFTER EXCEPTION (page 491).

Getting a Program to Enter Debugging Mode

You can get a program to enter debugging mode (that is, to enter the debugger for which you compiled the program) at any of these times:

- Before Execution
- During execution:
  - Programmatically
  - Forcefully

For information on specific debuggers, see Chapter 16: Debugging Tools (page 707).

Before Execution

To have an ordinary HP COBOL program (as opposed to one that is a Pathway server) enter debugging mode before executing any instructions, start it with the RUND command instead of the RUN command. Use the same options that you would use with the RUN command (see Executing an HP COBOL Program).

To have a Pathway server enter debugging mode before executing any instructions, use the PATHCOM command SET SERVER DEBUG ON. For considerations regarding errors that can occur when you are debugging a Pathway server, see the *TS/MP Pathsend and Server Programming Manual*.

Programmatically

To cause a program to put itself into debugging mode, put this statement at the point in the program where you want to start debugging:

```
ENTER "DEBUG"
```

The process prompts you for debugging commands at its home terminal.

Forcefully

To force an HP COBOL program to enter debugging mode while it is executing, use the TACL command DEBUG. If you do not have a TACL prompt on your terminal (because you are not running the HP COBOL program in the background), press Break to get a TACL prompt. If you include the TERM option in the RUN command with which you start your program, you can debug the program from a terminal other than the one the program is running on.

If the HP COBOL program is a Pathway server process, you can use the TACL DEBUG command only from a terminal that is connected to a command interpreter, not from a terminal that is configured to run a SCREEN COBOL terminal program. The reason is that you must be able to tell the debugger which process to debug. For considerations regarding errors that can occur when you are debugging a Pathway server, see the *TS/MP Pathsend and Server Programming Manual*.

NOTE: If you press Break too soon after you start a process, you get the messages:

```
WARNING - STARTUP MESSAGE NOT READ
ABENDED: nn,nnn
```

followed by a TACL prompt. The process that you started was not able to complete its initialization and was terminated.

27 Input and Output Concepts

Input and output are involved in virtually every HP COBOL program. The typical batch HP COBOL program obtains data (input), manipulates it, and produces a report (output). The typical interactive HP COBOL program is a server that obtains a request, collects data (input), and sends a reply (output).

Input and output always involve files. Input for an HP COBOL program comes from a disk file or nondisk file (such as a terminal); output from an HP COBOL program goes to a disk file or nondisk file (such as a terminal or printer).

Process communication involves files—one process calls another by handling it as a file—and operating system messages. An HP COBOL process ignores operating environment messages unless you arrange to have it intercept them.

Files

Files of alphanumeric data are ASCII files; that is, they are encoded in American National Standard Code for Information Interchange (ASCII).

Files are controlled by a file system, a combination of input-output hardware and operating system routines that mediate between the hardware and application programs. The NonStop operating system supports two file systems: the Guardian file system and the OSS file system.

Both file systems recognize a file by its system file name; an HP COBOL program recognizes a file by its COBOL file name, a name that the program defines. For an HP COBOL program to access or manipulate an actual file, the file's COBOL file name must be associated with a system file name.

Under certain conditions, the HP COBOL run-time routines preread file records (starting the read for record *n* +1 when returning record *n* to the program), saving execution time by overlapping reading and processing.

For information about creating files, see the *Guardian Programmer's Guide*.

System File Names

A system file name is the name by which the Guardian or OSS file system recognizes a system file. That name is unique, not only on the system where the file is physically located, but also within that system's network.

These rules and the remainder of this topic apply to all system file names in the Guardian file system. For information about system file names in the OSS file system, see Files in the OSS Environment (page 723).

- System file names are composed of alphanumeric characters and these special characters, which are used as delimiters:
  - Backward slash (\)
  - Dollar sign ($)
  - Number sign (#)
  - Colon (:)
  - Period (.)
- System file names are not case-sensitive; \NODE1.$VOLUME1.SUBVOL1.FILE1 and \node1.$volume1.subvol1.file1 refer to the same file.

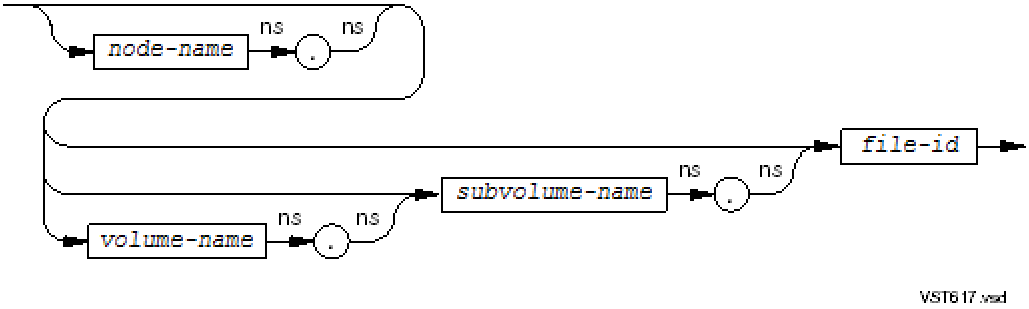
Other rules depend on the category of the system file name. The categories are:

- Permanent Disk File Names
- Temporary Disk File Names
- Nondisk File (Device) Names

For information about process file names, see Processes Handled as Files.

Permanent Disk File Names

Permanent disk files are named when they are created. After being created, a permanent disk file remains on the disk until explicitly purged.



node-name

is a backward slash (\) followed by an alphabetic character and up to six alphanumeric characters. Example: \mynode. Default: default node.

volume-name

is a dollar sign ($) followed by an alphabetic character and up to six alphanumeric characters. Example: $volume3. Default: default volume.

subvolume-name

is an alphabetic character followed by up to seven alphanumeric characters. Example: subvol10. Default: default subvolume.

file-id

is an alphabetic character followed by up to seven alphanumeric characters. Example: file1234.

An example of a permanent disk file name is \mynode.$volume3.subvol10.file1234. If \mynode.$volume3.subvol10 is the default subvolume, then these are equivalent:

```
\mynode.$volume3.subvol10.file1234
$volume3.subvol10.file1234
subvol10.file1234
file1234
```

The default node, volume, and subvolume are established with the =_DEFAULTS DEFINE and can be changed with TACL commands. For details, see the *TACL Reference Manual*.

Temporary Disk File Names

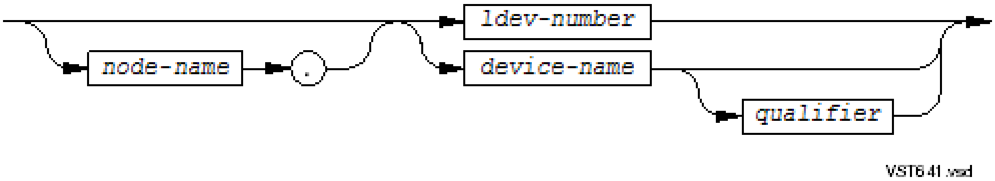
The two ways to create a temporary disk file are:

- In the file-control entry, with the clause SELECT … ASSIGN #TEMP
- With the TACL command ASSIGN … #TEMP

Temporary disk files are named by the file system when a program creates them. The program cannot specify a system file name for a temporary file. When the program closes a temporary disk file, the file system purges it. For more information about temporary disk files, see the *Guardian Programmer's Guide*.

Nondisk File (Device) Names

Nondisk files are devices such as terminals, printers, magnetic tape drives, and data communications lines. A nondisk file can be accessed by its name (the recommended method) or its logical device number, both of which are assigned by system management.



node-name

is a backward slash (\) followed by an alphabetic character and up to six alphanumeric characters. Example: \mynode.

ldev-number

is a dollar sign ($) followed by an integer whose value is less than or equal to 34492. Example: $12345.

device-name

is a dollar sign ($) followed by an alphabetic character and up to six alphanumeric characters. Example: $printr2.

qualifier

is a number sign (#) followed by one to seven alphanumeric characters. Example: #PRINTR2.

An example of a nondisk file name is \mynode.$printer2.#S. If \mynode is the default node, then these are equivalent:

```
\mynode.$printer2.#S
$printer2.#S
```

COBOL File Names

A COBOL file name is the name by which a particular HP COBOL program recognizes a system file. It is unique within the HP COBOL program. A COBOL file name is the *file-name* in the file-control entry (in the Environment Division) and the file description entry (in the Data Division). A *file-name* must meet these criteria:

- Consist of one or more (a maximum of 30) alphanumeric characters (letters, digits, and hyphens)
- Contain at least one nonnumeric character (letter or hyphen)
- Not begin or end with a hyphen

Letters can be uppercase or lowercase.

NOTE: HP COBOL requires a *file-name* to contain at least one nonnumeric character; COBOL requires a *file-name* to contain at least one letter.

Example 27-1 COBOL File Names

```
MAJRACCT
MAJOR-ACCOUNT
MajorAccount-3
```

Associating COBOL File Names With System File Names

The file-control entry for a file associates its COBOL file name with one of:

- System File Name
- Special Name
- Nonnumeric and National Literals

If you want to associate the COBOL file name with a system file name at run time, see #DYNAMIC.

System File Name

The most straightforward way to associate a COBOL file name with a system file name is to use the file-control entry to assign a COBOL file name (the `file-name` in the SELECT clause) to a system file name (the `system-file-name` in the ASSIGN clause).

These SELECT and ASSIGN clauses associate the COBOL file name MAJRACCT with the system file name \AKRON.$SLB.MAJ.ACC:

```
SELECT MAJRACCT ASSIGN TO "\AKRON.$SLB.MAJ.ACC"
```

When the HP COBOL program opens MAJRACCT, the file it actually opens is \AKRON.$SLB.MAJ.ACC.

In the Guardian environment, you can override the file assignment that you made at compilation time before executing the program (see Overriding File Assignments Made at Compilation Time (page 836)). You cannot do this in the OSS environment.

For the syntax of a system file name in the Guardian environment, see the *Guardian Programmer's Guide*. For the syntax of a system file name in the OSS environment, see Files in the OSS Environment (page 723).

Special Name

A special name is a place holder for a specific file whose system file name is determined at run time.

Table 27-1 Special Names for System Files

Special System File Name	Place holder for ...
#IN	In the Guardian file system, the file named in the IN parameter of startup message of current process In the OSS file system, the default input device (FD 0)—do not use it in SELECT clauses or the SPECIAL-NAMES paragraph as you can in the Guardian environment
#OUT	In the Guardian file system, the file named in the OUT parameter of startup message of current process In the OSS file system, the default output device (FD 1)—do not use it in SELECT clauses or the SPECIAL-NAMES paragraph as you can in the Guardian environment
#TERM	Home terminal of current process
#TEMP	Temporary disk file on default volume
#DYNAMIC	File name specified with the run-time library routine COBOL_ASSIGN_ during the execution of the current process

In the Guardian environment, you can override the file assignment that you made at compilation time before executing the program (see Overriding File Assignments Made at Compilation Time (page 836)). The new `system-file-name` in the overriding ASSIGN command can also be a special name—any special name except #DYNAMIC.

#IN

In the Guardian environment, #IN represents the file specified as the IN parameter in the startup message of the current process. If you started the current process from a TACL prompt, #IN refers to the file specified by the IN option of the RUN command (or its default). (See Specifying Default Input and Output Devices (page 841).)

In the program PROG1, these SELECT and ASSIGN clauses associate the COBOL file name MAJRACCT with the special system file name #IN:

```
SELECT MAJRACCT ASSIGN TO #IN
```

Equivalently, these ASSIGN command associates the COBOL file name MAJRACCT with the special system file name #IN:

```
ASSIGN PROG1.MAJRACCT, #IN
```

If you run the program PROG1 with this command, the special name #IN represents the system file INFILE, so INFILE is associated with the COBOL file name MAJRACCT.

```
PROG1 /IN INFILE/
```

In the OSS environment, #IN is the default input device (FD 0). You cannot use it in SELECT clauses or the SPECIAL-NAMES paragraph as you can in the Guardian environment.

#OUT

In the Guardian environment, #OUT represents the file specified as the OUT parameter in the startup message of the current process. If you started the current process from a TACL prompt, #OUT refers to the file specified by the OUT option of the RUN command (or its default). (See Specifying Default Input and Output Devices (page 841).)

In the program PROG1, these SELECT and ASSIGN clauses associate the COBOL file name REPORT with the special name #OUT:

```
SELECT REPORT ASSIGN TO #OUT
```

Equivalently, these ASSIGN command associates the COBOL file name REPORT with the special name #OUT:

```
ASSIGN PROG1.REPORT, #OUT
```

If you run the program PROG1 with this command, the special name #OUT represents the system file OUTFILE, so OUTFILE is associated with the COBOL file name REPORT.

```
PROG1 /OUT OUTFILE/
```

In the OSS environment, #OUT is the default output device (FD 1). You cannot use it in SELECT clauses or the SPECIAL-NAMES paragraph as you can in the Guardian environment.

#TERM

The special name #TERM represents the home terminal of the current process. If you started the current process from a TACL prompt, #TERM refers to the file specified in the TERM option of the RUN command (or its default). (See Specifying the Home Terminal (page 842).)

In the program PROG1, these SELECT and ASSIGN clauses associate the COBOL file name HOME-TERMINAL with the special name #TERM:

```
SELECT HOME-TERMINAL ASSIGN TO #TERM
```

Equivalently, these ASSIGN command associates the COBOL file name HOME-TERMINAL with the special name #TERM:

```
ASSIGN PROG1.HOME-TERMINAL, #TERM
```

If you run the program PROG1 with this command, the system file $TE1.#E02 is associated with the special name #TERM and, therefore, with the COBOL file name HOME-TERMINAL.

```
PROG1 /TERM $TE1.#E02/
```

#TEMP

The special name #TEMP represents a temporary disk file on the default volume (which is specified by the program's startup message). When the program opens a file that is assigned to #TEMP, the HP COBOL run-time routines create a disk file on the default volume. When the program closes a temporary disk file, the file system purges it. For more information about temporary disk files, see the *Guardian Programmer's Guide*.

In the program PROG1, these SELECT and ASSIGN clauses associate the COBOL file name TEMPFILE with the special name #TEMP, which stands for the system file name that the file system gave the temporary file.

```
SELECT TEMPFILE ASSIGN TO #TEMP
```

Equivalently, this ASSIGN command associates the COBOL file name TEMPFILE with the special name #TEMP:

```
ASSIGN PROG1.TEMPFILE, #TEMP
```

#DYNAMIC

The special name #DYNAMIC represents the file specified with the run-time library routine COBOL_ASSIGN_ during the execution of the current process.

If you want your HP COBOL program to accept or construct a file name and then open the file with that name, you must:

1. Assign the COBOL file name to #DYNAMIC (in the file-control entry).
2. Associate #DYNAMIC with a system file name by calling the routine COBOL_ASSIGN_.
3. Open the file (using its COBOL file name).

Suppose that you want your program to use a file that the user specifies at run time. Following Step 1, you assign the COBOL file name (ADJUSTABLE in this example) to the special name #DYNAMIC in the file-control entry:

```
SELECT ADJUSTABLE ASSIGN TO #DYNAMIC.
```

To follow Step 2, you must declare Working-Storage data items to hold the name and the error value returned by the COBOL_ASSIGN_ routine, prompt the user for a file name, and then call the COBOL_ASSIGN_ routine:

```
01 ADJUSTABLE-STUFF.
   03 ADJUSTABLE-NAME     PICTURE X(34).
   03 ADJUSTABLE-ERROR    PICTURE S9(4).
. .
DISPLAY "What file shall I read?".
ACCEPT ADJUSTABLE-NAME.
ENTER "COBOL_ASSIGN_" USING  ADJUSTABLE
                           ADJUSTABLE-NAME
                    GIVING ADJUSTABLE-ERROR.
IF ADJUSTABLE-ERROR = ZERO
   OPEN INPUT ADJUSTABLE
ELSE
   PERFORM ADJUSTABLE-ERROR-ROUTINE
   ...
```

To lock a file that has been opened with dynamic assignment, use the LOCKFILE statement. You cannot perform a CLOSE WITH LOCK operation on such a file.

NOTE: Do not use #DYNAMIC in the TACL command ASSIGN. You will get an error when you try to assign a system file name to the associated COBOL file name with the COBOL_ASSIGN_ routine.

For further information about the COBOL_ASSIGN_ routine, see COBOL_ASSIGN_ (page 654).

DEFINE Name

NOTE: This topic applies only to the Guardian file system. DEFINEs do not accept OSS pathnames.

If the file-control entry assigns the COBOL file name (the *file-name* in the SELECT clause) with a DEFINE name (the *define-name-literal* in the ASSIGN clause), then you must associate the DEFINE name with a system file name before executing the program (see Adding DEFINEs (page 835)). Even when the program executes in the OSS environment, the DEFINE must specify a Guardian file name, tape device, or spooler.

These SELECT and ASSIGN clauses associate the COBOL file name MAJRACCT with the DEFINE name =BIGCUST:

```
SELECT MAJRACCT
       ASSIGN TO "=BIGCUST"
```

This command associates =BIGCUST with the system file \AKRON.$SLB.MAJ.ACC:

```
ADD DEFINE =BIGCUST, FILE \AKRON.$SLB.MAJ.ACC
```

Prereading File Records

To save execution time by overlapping reading and processing, the HP COBOL run-time routines perform record prereading (starting the read for record *n* +1 when returning record *n* to the program) when all of these conditions are true:

- File access mode is SEQUENTIAL.
- File open mode is INPUT.
- File exclusion mode is PROTECTED (for a disk file) or EXCLUSIVE (for a disk or nondisk file).
- The process is not running as a process pair.
- The file is not open for timed I-O.

Verify that the prereading requirements are not met when the HP COBOL process does any of:

- Executes the read operation as part of a TMF transaction (enters BEGINTRANSACTION, executes a READ statement, then enters ENDTRANSACTION)
- Executes a CLOSE NO REWIND statement on a tape file and subsequently executes an OPEN NO REWIND statement on the same file to continue from the previous position

Processes Handled as Files

One process calls another by handling it as a file. If the calling process is an HP COBOL program, it refers to the called process by a COBOL file name. Like COBOL file names of files that are not processes, the COBOL file name of a process must be associated with a file name that the operating environment recognizes (the process name or process descriptor). For more information about processes, see Chapter 32: Process Initiation, Communication, and Management (page 933).

Intercepting Operating System Messages

The operating system sends messages to processes, reporting such things as:

- Break was pressed on a terminal.
- A descendant process has terminated (normally or abnormally).
- Another process with which you are communicating has opened you or closed you.
- ASSIGN, PARAM, and startup messages were sent to you from your parent process (Guardian environment only).

An HP COBOL process ignores operating system messages unless you arrange to have it intercept them. To have an HP COBOL process intercept operating system messages, you must include REPORT and MESSAGE SOURCE clauses in its RECEIVE-CONTROL paragraph.

The MESSAGE SOURCE clause provides a mechanism through which a COBOL program can discover the sender of each message received. During the successful execution of a read operation on $RECEIVE, the $RECEIVE mechanism places a set of values into the storage space designated in the MESSAGE SOURCE clause.

Example 27-2 MESSAGE SOURCE Format

```
01  SOURCE-MESSAGE.
    05  SYSTEM-FLAG        PICTURE  S9
                           USAGE IS COMPUTATIONAL.
    05  ENTRY-NUMBER       PICTURE  999
                           USAGE IS COMPUTATIONAL.
    05  FILLER             PICTURE  X(4).
    05  PHANDLE            PICTURE  X(20).
    05  FILLER             PICTURE  X(4).
```

If the value of SYSTEM-FLAG is -1, the message came from the operating system.

For more information about the MESSAGE SOURCE phrase, see MESSAGE SOURCE Phrase (page 161). For more information about the Receive-Control paragraph, see Chapter 32: Process Initiation, Communication, and Management (page 933).

28 Tape Input and Output

HP COBOL batch programs must often read or write files that are on magnetic tape (tape files).

If you need further information on reading and writing tape files, see the *Guardian Programmer's Guide*.

If you want to archive disk files on magnetic tape, use the operating environment's BACKUP utility. To restore archived disk files from a tape back to a disk (which you must do before an HP COBOL program can read them), use the operating environment's RESTORE utility. For information on the BACKUP and RESTORE utilities, see the *Guardian User's Guide*.

Reading and Writing Tape Files

An HP COBOL program can read and write unlabeled tape files, labeled tape files, and tape files of types other than HP.

By default, the shortest record an HP COBOL program can write to a tape file is 24 bytes. If you need shorter records, use the COBOL_SETMODE_ routine to set function 52 (see the *Guardian User's Guide*).

Preventing Prereading

HP COBOL prereads tapes; that is, it initiates a new read operation after processing the last (or only) record in a block. Prereading can overlap I-O with program processing. If you do not want this to happen, verify that the records of a tape file will not be preread by opening the file for timed I-O (include a TIME LIMITS phrase in the OPEN statement or use the command PARAM WAITED-IO). For an explanation of prereading file records, see Prereading File Records (page 853). You cannot give a tape file the access mode I-O or the exclusion mode PROTECTED or SHARED.

Saving Tape and Time

You can save both tape and time by blocking tape file records efficiently—that is, by specifying that each physical record is to contain more than one logical record (the default is one logical record per physical record).

Because consecutive physical records are separated by an interrecord gap, longer physical records mean fewer interrecord gaps (less wasted tape). Because the tape drive must start and stop each time you read or write a physical record, having fewer interrecord gaps save time.

Interrecord gap size depends on tape drive model, but the typical interrecord gap is about 0.6 inch (1.5 cm). If you are writing 1,600 bytes per inch—19,200 bytes, or characters, per foot (30 cm) of tape—then each interrecord gap occupies 960 bytes (1,600 bytes per inch*0.6 inch = 960 bytes).

Suppose that each logical record is 80 bytes. If each physical record contains only one logical record (the default), you can write 18.5 physical records (18.5 logical records) per foot of tape (19,200 bytes/(80 bytes per physical record + 960 bytes per physical record) = 18.5 records). If, instead, you specify 10 logical records per physical record, giving each physical record 800 bytes, you can write 10.9 physical records (109 logical records) per foot of tape. If you specify 50 logical records per physical record, giving each physical record 4 KB, you can write 3.87 physical records (193.5 logical records) per foot of tape.

You specify the number of logical records or characters per physical record (block) with a BLOCK CONTAINS clause in the file description entry of the file associated with the tape drive. The BLOCK CONTAINS clause works if these conditions are met:

- The file's organization is sequential.
- The file has fixed-length records.

- When the block size is specified in characters, it is a multiple of the number of characters in the logical record size. Also, when a RECORD CONTAINS clause extends the record size, the block size expressed is a multiple of the number of characters in that specified record size.
- The file description entry does not have a LINAGE or ALTERNATE RECORD KEY clause.

On NonStop systems, physical record (block) sizes for tapes range from 24 through 32,767 bytes. The recommended maximum tape block sizes for application programs are:

Density (bytes per inch)	Block Size (bytes)
800	4,096
1,600	8,192
6,250	32,767

For further information on the BLOCK CONTAINS clause, see BLOCK CONTAINS Clause (page 174).

Unlabeled Tape Files

An unlabeled tape file is a tape file that does not have standard ANSI or IBM labels.

If an HP COBOL program creates a tape file, the tape file is unlabeled, and any other HP COBOL program can easily read or write it. If a system that is not an HP system creates a tape file, an HP COBOL program can still read it, but might encounter features that are not HP features.

The first five steps for reading or writing an unlabeled tape file are the same:

1. In the file-control entry:
   a. Assign the COBOL file name of the tape to either:
      - The device name of the tape drive that holds the tape that you want to read or write (established when the system was configured)
      - A DEFINE name

   b. In the ORGANIZATION clause, declare SEQUENTIAL organization.
   c. In the ACCESS MODE clause, declare SEQUENTIAL access.
2. If you used dynamic file assignment in Step 1a, then in the Procedure Division, use the COBOL_ASSIGN_ routine to establish the system file name of the tape at run time (see #DYNAMIC (page 852)) and go to Step 4.
3. If you used a DEFINE name in Step 1a, add the appropriate DEFINE before executing your program (see Adding DEFINEs for Tape Files) and go to Step 4.
4. Open the file with the appropriate `file-specification` in the OPEN statement. To prevent prereading of file records, include a TIME LIMITS phrase in the OPEN statement or use the command PARAM WAITED-IO.

file-specification	Effect
INPUT	Opens the tape for input so you can read it
OUTPUT	Opens the tape for output so you can write it, starting with its first record
EXTEND	Opens the tape for extension so you can write it, starting immediately after its last record

   (You cannot open a tape with the keyword I-O, which allows both input and output.)
5. If you opened the file for input, read it with the READ statement; if you opened it for output or extension, write it with the WRITE statement.

After Step 5, the instructions depend on whether you are reading or writing one file on one tape, several files on one tape, or one file that spans several tapes.

One File on One Tape

To read or write one file on one tape, follow Step 1 through Step 5 under Unlabeled Tape Files and then execute this step:

- Close the file with the CLOSE statement. Do not use a NO REWIND phrase. Use a LOCK phrase if the file was not dynamically assigned and you want to prevent the tape from being reopened (the run-time tape routine ignores LOCK for a dynamically assigned file).

Example 28-1 assigns the COBOL file name TAPE1 to the system file name $DRIVE2 (a name that the system administrator assigned to a tape drive) and declares its organization and access modes to be sequential; then it opens TAPE1 for exclusive input, reads, closes, and locks it.

Example 28-1 One File on One Tape

```
...
ENVIRONMENT DIVISION.
...
INPUT-OUTPUT SECTION.
...
FILE-CONTROL
  SELECT TAPE1 ASSIGN $DRIVE2
  ORGANIZATION SEQUENTIAL
  ACCESS MODE SEQUENTIAL.
...
PROCEDURE DIVISION.
...
OPEN INPUT TAPE1 EXCLUSIVE
READ TAPE1 . .
CLOSE TAPE1 LOCK
...
```

Several Files on One Tape

To read or write more than one file on one tape, follow Step 1 through Step 5 under Unlabeled Tape Files and then execute these steps:

1. Close the file with the CLOSE statement, specifying NO REWIND. The NO REWIND phrase prevents the run-time routines from rewinding the tape, so that you can read or write the file that immediately follows the file you just closed.
2. Reopen the file specifying NO REWIND (see Step 4 under Unlabeled Tape Files).
3. Read or write the file again (see Step 5 under Unlabeled Tape Files).
4. Close the file with the CLOSE statement. Do not use a NO REWIND phrase. Use a LOCK phrase if the file was not dynamically assigned and you want to prevent the tape from being reopened (the run-time tape routine ignores LOCK for a dynamically assigned file).

Example 28-2 writes two files on one tape and then closes and locks the tape.

Example 28-2 Several Files on One Tape

```
 ...
 ENVIRONMENT DIVISION.
 ...
 INPUT-OUTPUT SECTION.
 ...
 FILE-CONTROL
   SELECT TAPE1 ASSIGN $DRIVE2
   ORGANIZATION SEQUENTIAL
   ACCESS MODE SEQUENTIAL.
 ...
 PROCEDURE DIVISION.
 ...
*WRITE FIRST FILE
 OPEN OUTPUT TAPE1 EXCLUSIVE
 WRITE TAPE1 ...
 CLOSE TAPE1 NO REWIND
*WRITE SECOND FILE
 OPEN OUTPUT TAPE1 EXCLUSIVE NO REWIND
 WRITE TAPE1 ...
 CLOSE TAPE1 LOCK
 ...
```

One File on Several Tapes

To read or write one file that spans several tapes, follow Step 1 through Step 5 under Unlabeled Tape Files and then execute these steps:

1. Close the file with the CLOSE statement, specifying REEL FOR REMOVAL (or UNIT FOR REMOVAL). The REMOVAL phrase rewinds the tape for removal so that you can remove it from the tape drive and replace it with the next tape.
2. Read or write the file again (see Step 5 under Unlabeled Tape Files).
3. Repeat Step 4 and Step 5 under Unlabeled Tape Files and Step 1 above for each tape that contains part of the file.
4. Close the file with the CLOSE statement. Do not use a NO REWIND phrase. Use a LOCK phrase if the file was not dynamically assigned and you want to prevent the tape from being reopened (the run-time tape routine ignores LOCK for a dynamically assigned file).

Mount Messages

When a run-time routine needs to have a reel of tape mounted, it sends a message to the process' home terminal.

Example 28-3 Mount Messages

For an input file:

```
Mount the next reel of fd-name = device-name in sequence
```

For an output file:

```
Mount the next reel of fd-name = device-name, with a write ring
```

Message sent to the home terminal:

```
Type return or $volume (or 'NO' to end input)
```

Respond to the message sent to the home terminal after the next tape reel is mounted. If the next reel is ready on the device named in the mount message, press Return. If the next reel is ready on a different tape device, type that device name (beginning with $) before you press Return.

If you do not want to continue with the next reel of tape, type "NO" in response to the message. If the program is reading the tape, the next READ statement will discover an end of file.

For example, suppose that a file is stored on three tapes. Its COBOL file name is HUGEFILE. If HP COBOL end-of-tape sequences were used when the tapes were created, HP COBOL automatically asks for each reel; otherwise, you must read and then close each reel.

Example 28-4 assigns the COBOL file name HUGEFILE to the system file name $DRIVE2 (a name that the system administrator assigned to a tape drive) and declares its organization and access modes to be sequential. The first tape is on the tape drive $DRIVE2. The program opens HUGEFILE for exclusive input, reads it, and closes it, rewinding it for removal. You replace the first tape with the second tape, and the program reads it and closes it, rewinding it for removal. You replace the second tape with the third tape, and the program reads, closes and locks it, having now read the entire file.

Example 28-4 Multitape File

```
...
 ENVIRONMENT DIVISION.
...
 INPUT-OUTPUT SECTION.
...
 FILE-CONTROL
   SELECT HUGEFILE ASSIGN $DRIVE2
   ORGANIZATION SEQUENTIAL
   ACCESS MODE SEQUENTIAL.
...
 PROCEDURE DIVISION.
...
 OPEN INPUT HUGEFILE EXCLUSIVE.

*READ FIRST TAPE
 READ HUGEFILE...
 CLOSE HUGEFILE REEL FOR REMOVAL.

*READ SECOND TAPE
 READ HUGEFILE...
 CLOSE HUGEFILE REEL FOR REMOVAL.

*READ THIRD TAPE
 READ HUGEFILE...
 CLOSE HUGEFILE LOCK.
...
```

Avoid having an HP COBOL program read a multitape file that was created by a File Utility Program (FUP). HP COBOL and FUP mark the ends of nonfinal reels of a multiple-reel tape file differently: an HP COBOL program writes an end-of-file (EOF) mark, a trash record, and a pair of consecutive EOF marks; a FUP writes only a pair of consecutive EOF marks.

If your HP COBOL program must read a FUP-created multitape file, try this:

1. In the SELECT clause of the file-control entry, declare the file to be OPTIONAL.
2. Have the program close and reopen the file when it detects EOF.
3. Have the operator mount the next tape. When the program detects EOF immediately after an OPEN statement, the last reel has been read.
4. When all tapes have been read, have the operator respond NO to the tape mount message to signal the actual end of the file.

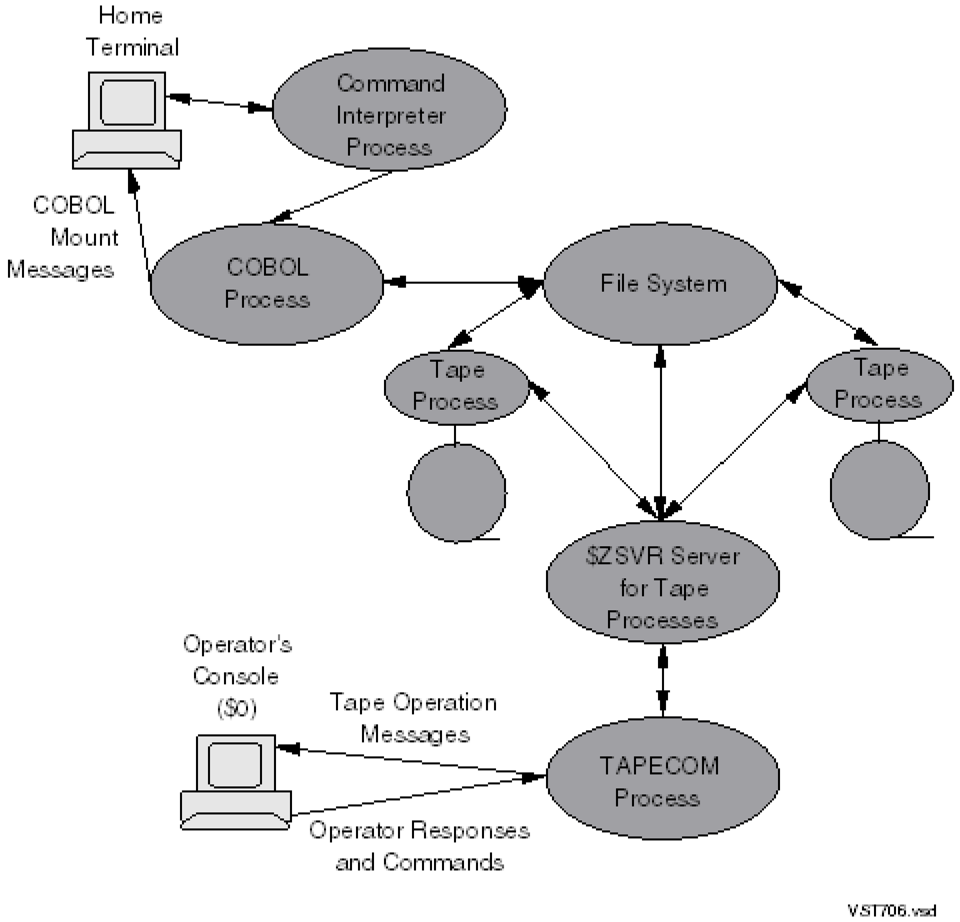
Labeled Tape Files

A labeled tape file is a tape file that has standard ANSI or IBM labels. If a tape file has labels of any other type, an HP COBOL program must bypass them, handling the tape file as unlabeled.

Labeled-tape processing on NonStop systems involves these software elements:

- ANSI or IBM standard tape labels (which catalog tape files)
- HP COBOL statements and clauses that describe labeled tape files and operate on them
- The TACL command DEFINE (which associates the COBOL file name of a labeled tape file with its system file name and defines its attributes)
- Tape processes (which control tape drives)
- $ZSVR, the operating environment server for tape processes
- MEDIACOM utility
- File-system procedures and functions that support labeled-tape processing

Figure 28-1 Software Involved in Labeled-Tape Processing



To use a labeled tape file with an HP COBOL program, you must:

1. Describe the labeled tape file in a file-control entry in the HP COBOL program, assigning the file's COBOL file name to a DEFINE name.
2. Enable the creation of DEFINEs with the TACL command SET DEFMODE ON.
3. Add a DEFINE that has the name assigned to the file in Step 1 and the appropriate attributes (see Adding DEFINEs for Tape Files).
4. Execute the program (see Executing HP COBOL Programs That Use Tape Files).

Describing Labeled Tape Files in an HP COBOL Program

In the file-control entry for a labeled tape file, assign the COBOL file name of the labeled tape file to a DEFINE name (for the syntax of a DEFINE name, see DEFINE Names (page 836)).

In the file description entry for a labeled tape file, include a RECORD CONTAINS clause to specify whether the file's records are of fixed or variable length and to define their size.

Example 28-5 assigns the COBOL file name SEQ-FILE to the DEFINE name =PROG1D1 (giving it sequential organization and access mode as required for a tape file) and then describes SEQ-FILE as having fixed-length records of 30 characters each.

Example 28-5 Labeled Tape File

```
ENVIRONMENT DIVISION.
...
INPUT-OUTPUT SECTION.
FILE-CONTROL.
SELECT SEQ-FILE ASSIGN TO "=PROG1D1"
       ORGANIZATION SEQUENTIAL
       ACCESS MODE SEQUENTIAL.
...
DATA DIVISION.
FILE SECTION.
FD  SEQ-FILE
    RECORD CONTAINS 30 CHARACTERS
01  SEQ-RECORD      PIC X(30).
...
```

Consistency Checking of Labeled Tape Information

HP COBOL run-time routines check that the HP COBOL program specifications are consistent with these DEFINE attributes:

- LABELS
- RECFORM
- BLOCKLEN
- RECLEN
- FILESEQ
- USE

If a program specification is inconsistent with an attribute, an HP COBOL run-time routine takes one of these actions, depending on the attribute:

- Allows the DEFINE attribute to override the program specifications
- Issues an open error because of inconsistency

The system label-processing software checks that the DEFINE attributes are consistent with the tape label specifications; if they are not, the operating environment issues an open error.

If the tape label and the data on the tape are inconsistent (for example, if the tape label specifies fixed-length records but the tape contains variable-length records), results are unpredictable.

Converting HP COBOL Programs to Use Labeled Tape Files

To convert an HP COBOL program to use a labeled tape file, set up an OBEY command file that:

1. Assigns the *file-name* in the file description entry to a DEFINE name
2. Enables the creation of DEFINEs
3. Deletes any existing DEFINE that has the name selected in Step 1
4. Creates the DEFINE
5. Runs the program
6. Clears the assignment (from Step 1) and the DEFINE (from Step 4) to prevent their affecting other processes created by the same TACL process

In Example 28-6, the commands in the OBEY command file commands convert the HP COBOL program to use a labeled tape file. The *file-name* in the file description entry is MONTHLY-SALES, the DEFINE name is =SALES^JAN86, and the program name is QUOTAS.

The DEFINE in Example 28-6 is for a tape file with ANSI standard labels and fixed-length records 80 characters long. Because the file description entry contains the clause LABEL RECORDS ARE OMITTED, the HP COBOL run-time library sets the file status code, STATUS-KEY, to “97.” If the program does not contain code that takes specific action when STATUS-KEY is “97.” the program can use a labeled tape file without changes to the source code or recompilation.

Example 28-6 Converting a COBOL95 Program to Use Labeled Tape Files

HP COBOL file-control entry:

```
SELECT MONTHLY-SALES ASSIGN TO $TAPE2
        ORGANIZATION SEQUENTIAL
        ACCESS MODE SEQUENTIAL
        FILE STATUS IS STATUS-KEY.
...
```

HP COBOL file description entry for a tape file:

```
FD MONTHLY-SALES
    RECORD CONTAINS 80 CHARACTERS
    LABEL RECORDS ARE OMITTED.
```

OBEY command file commands:

```
ASSIGN MONTHLY-SALES, =SALES^JAN86
SET DEFMODE ON
DELETE DEFINE =SALES^JAN86
ADD DEFINE =SALES^JAN86, CLASS TAPE, LABELS ANSI,&
           VOLUME SLS1Q86, RECFORM F, RECLEN 80,&
           BLOCKLEN 4000
RUN QUOTAS
CLEAR ASSIGN MONTHLY-SALES
DELETE DEFINE =SALES^JAN86
```

Tape Files of Types Other Than HP

If a system that is not an HP system creates a tape file, an HP COBOL program can still read it, but might encounter features that are not HP features, such as:

- Labels
- Extended Binary Coded Decimal Interchange Code (EBCDIC)
- Packed-decimal or floating-point binary data
- Variable-length records

Labels are explained in Labeled Tape Files.

EBCDIC Files

To read an EBCDIC file, an HP COBOL program must include a CODE-SET clause in the file description entry. The CODE-SET clause causes translation between the native USASCII character code convention and the EBCDIC code convention for all input-output operations. For more information about the CODE-SET clause—including restrictions—see CODE-SET Clause (page 184).

If you cannot include a CODE-SET clause in the file description entry for an EBCDIC file, you must copy the EBCDIC file to an ASCII file with the File Utility Program (FUP) COPY command. The ASCII file can be a tape file or an existing disk file. To create a disk file, use the FUP CREATE command.

The FUP command in Example 28-7 copies the EBCDIC tape file on tape drive $DRIVE1 to a tape file on tape drive $DRIVE2, converting it to ASCII in the process.

Example 28-7 Copying an EBCDIC File to a New ASCII File

```
FUP COPY $DRIVE1,$DRIVE2,EBCDICIN
```

The FUP command in Example 28-8 copies the EBCDIC tape file on tape drive $DRIVE1 to the existing disk file $TEN.APP3.PQ5, converting it to ASCII in the process.

Example 28-8 Copying an EBCDIC File to an Existing ASCII File

```
FUP COPY $DRIVE1,$TEN.APP3.PQ5,EBCDICIN
```

The FUP command in Example 28-9 creates the disk file $TEN.APP3.PQ5 from an EBCDIC tape file composed of 10K records of 80 characters each. Your HP COBOL program can access $TEN.APP3.PQ5 sequentially.

Example 28-9 Copying an EBCDIC File to an Existing ASCII File

```
FUP CREATE $TEN.APP3.PQ5, TYPE E, &
EXT (1000 RECS,1000 RECS), REC 80
```

The preceding discussion does not completely explain FUP. There are many considerations involved in choosing parameters for the COPY and CREATE commands. For details, see the *Guardian Programmer's Guide*.

△ **CAUTION:** The EBCDIC-to-ASCII conversion mechanism described in this topic handles data of the type DISPLAY, but not data of the type COMPUTATIONAL. If an EBCDIC file does not have a character format, do not use FUP or CODE-SET to change EBCDIC to ASCII, because this disturbs the stored numeric values. Instead, convert directly from the foreign EBCDIC representation to an HP COBOL representation. (See Converting Other Data Types to HP COBOL Data Types.)

Converting Other Data Types to HP COBOL Data Types

If your HP COBOL program must use packed-decimal or floating-point binary data from a tape file that was not created on an HP system, be aware that the data from the tape file might be represented differently from HP COBOL packed-decimal and floating-point binary data. For details on HP COBOL packed-decimal and floating-point binary data representation, see USAGE Clause (page 214).

HP COBOL data types:

- BINARY or COMPUTATIONAL

  BINARY and COMPUTATIONAL are synonyms. HP COBOL stores a COMPUTATIONAL data item as a 2-byte, 4-byte, or 8-byte binary value, depending on the number of digits in its picture:

Picture	Storage
S9(01) - S9(04)	1 byte
S9(05) - S9(09)	2 bytes
S9(10) - S9(18)	4 bytes

- DISPLAY

  HP COBOL stores a DISPLAY data item as a sequence of ASCII characters, one for each digit. If the data description entry specifies that the sign is separate from the value, a separate byte is allocated for the sign, which is stored as an explicit plus (+) or a minus (-) character.

  If the data description entry does not specify that the sign is separate from the value, the sign is stored by setting the high-order bit of one byte of the data item—the high-order byte if the sign is specified as LEADING; the low-order byte otherwise.

  The value zero has neither a positive nor a negative sign.

  Some examples of data descriptions and their corresponding stored values are:

Data Description	Binary Value Stored
PIC S99 VALUE 5	00110000 00110101
PIC S99 VALUE -5	00110000 10110101
PIC S99 VALUE 0	00000000 00000000
PIC S99 SIGN LEADING VALUE 5	00110000 00110101

Data Description	Binary Value Stored
PIC S99 SIGN LEADING VALUE -5	10110000 00110101
PIC S99 SIGN LEADING VALUE -0	00000000 00000000
PIC S99 SIGN TRAILING VALUE 5	00110000 00110101
PIC S99 SIGN TRAILING VALUE -5	00110000 10110101
PIC S99 SIGN TRAILING VALUE 0	00000000 00000000

- NATIVE-*n*

  HP COBOL stores a NATIVE-*n* data item as a signed, numeric, twos-complement, binary integer. A NATIVE-*n* data type is stored in *n* bytes:

Data Type	Storage
NATIVE-2	2 bytes
NATIVE-4	4 bytes
NATIVE-8	8 bytes

Handling Variable-Length Records

Software for systems other than NonStop systems sometimes provides a mechanism for writing variable-length blocked records on tape—usually a block-length field at the beginning of each block and a record-length field at the beginning of each record. HP software does not have a similar mechanism. If your HP COBOL program must read a file tape that has variable-length blocked records and was not created on an HP system, you must write HP COBOL or pTAL code to deblock the records. It might be easier to produce another tape with fixed-length records (blocked or not) on the system on which the file was created.

Adding DEFINEs for Tape Files

If your HP COBOL program assigns a tape file to a DEFINE name (which is optional for an unlabeled tape file, but required for a labeled tape file), then you must add that DEFINE name before executing your program, or the program will not be able to open the file.

The values of the DEFINE attributes must match corresponding values in the tape label. The tape process compares corresponding values when the program tries to open the tape file, and if any corresponding values do not match, the tape process issues an error message, and the open operation fails (see Consistency Checking of Labeled Tape Information).

These topics explain the attributes of DEFINEs for:

- Unlabeled Tape Files
- Labeled Tape Files

In the Guardian environment, you must enable the creation of DEFINEs with the TACL command SET DEFMODE ON before you can add them. In the OSS environment, you can add DEFINEs with the command `add_define`.

An HP COBOL program running in the OSS environment can assign a file to “GUARDIAN tape-device” or “GUARDIAN define-name.” In the latter case, you must use the `add_define` command to create a TAPE DEFINE before running the program.

Unlabeled Tape Files

The attributes of unlabeled tape files are:

- CLASS Attribute (Required)
- DEVICE Attribute (Required)
- BLOCKLEN Attribute (Conditional)
- LABELS Attribute (Optional)
- RECFORM Attribute (Optional)
- RETENTION or EXPIRATION Attribute (Optional)

CLASS Attribute (Required)

The value of the CLASS attribute must be TAPE. The CLASS attribute must be the first attribute in the ADD DEFINE command, because it determines what other attributes the DEFINE can have and clears any existing attribute settings, including any that precede it in the ADD DEFINE command.

DEVICE Attribute (Required)

The value of the DEVICE attribute must be the system file name of a tape drive.

BLOCKLEN Attribute (Conditional)

The BLOCKLEN attribute is required if the tape file has fixed-length records; that is, if the RECORD CONTAINS clause in the file description entry is of the form

```
RECORD CONTAINS length-fixed CHARACTERS
```

the value of the BLOCKLEN attribute must be the same as:

- The block size that the tape label specifies
- The value of *blk-2* in the BLOCK CONTAINS clause (if the tape file description entry has one)

If the tape file has variable-length records, omit the BLOCKLEN attribute from the DEFINE. HP COBOL programs cannot process blocked variable-length records.

LABELS Attribute (Optional)

The LABELS attribute must have the value OMITTED. With LABELS OMITTED, the system checks the mounted tape for standard ANSI or IBM labels; if the tape has them, the system rewinds and unloads the tape.

The LABELS attribute of the DEFINE overrides any LABEL RECORDS clause in your HP COBOL program.

NOTE: The 1985 COBOL Standard classifies the LABEL RECORDS clause as obsolete, so you are advised not to use it.

Table 28-1 Effect of LABELS Attribute and LABEL RECORDS Clause on Unlabeled Tape File

	LABEL RECORDS Clause	
LABELS Attribute Value	**LABEL RECORDS STANDARD**	**LABEL RECORDS OMITTED**
OMITTED	HP COBOL sets file status code to 97* and execution proceeds	Execution proceeds
None (no LABELS attribute)	Open error	Execution proceeds

* File status code 97 means that the statement executed successfully, but the circumstances were not entirely as expected.

RECFORM Attribute (Optional)

The value of the RECFORM attribute must correspond to the RECORD CONTAINS clause in the file description entry in the program.

Table 28-2 Correspondence Between RECORD CONTAINS Clause and RECFORM Attribute Value

Record Length	RECORD CONTAINS Clause	RECFORM Attribute Value
Variable (undefined)	RECORD CONTAINS *min-length* TO *max-length* CHARACTERS	U
Fixed	RECORD CONTAINS *length-fixed* CHARACTERS	F

RETENTION or EXPIRATION Attribute (Optional)

Use the RETENTION or EXPIRATION attribute when you want to save a file for any period of time, even part of a day; otherwise, the file expires as soon as you write it, and the system might overwrite the file.

The value of the RETENTION or EXPIRATION attribute is an integer, the number of days before the file expires.

△ **CAUTION:** On a multiple-file volume, the system might overwrite an unexpired file that follows an expired one; a volume on which the first file has expired is a scratch volume.

Example 28-10 adds a DEFINE for an unlabeled tape file with variable-length records. The file is on the tape on the tape drive named $DRIVE.

Example 28-10 DEFINE for Unlabeled Tape File With Variable-Length Records

```
ADD DEFINE =ABC, CLASS TAPE, DEVICE $DRIVE
```

Example 28-11 adds a DEFINE for an unlabeled tape file with fixed-length records. The file is on the tape on the tape drive named $DRIVE, and it will not expire for five days.

Example 28-11 DEFINE for Unlabeled Tape File With Fixed-Length Records

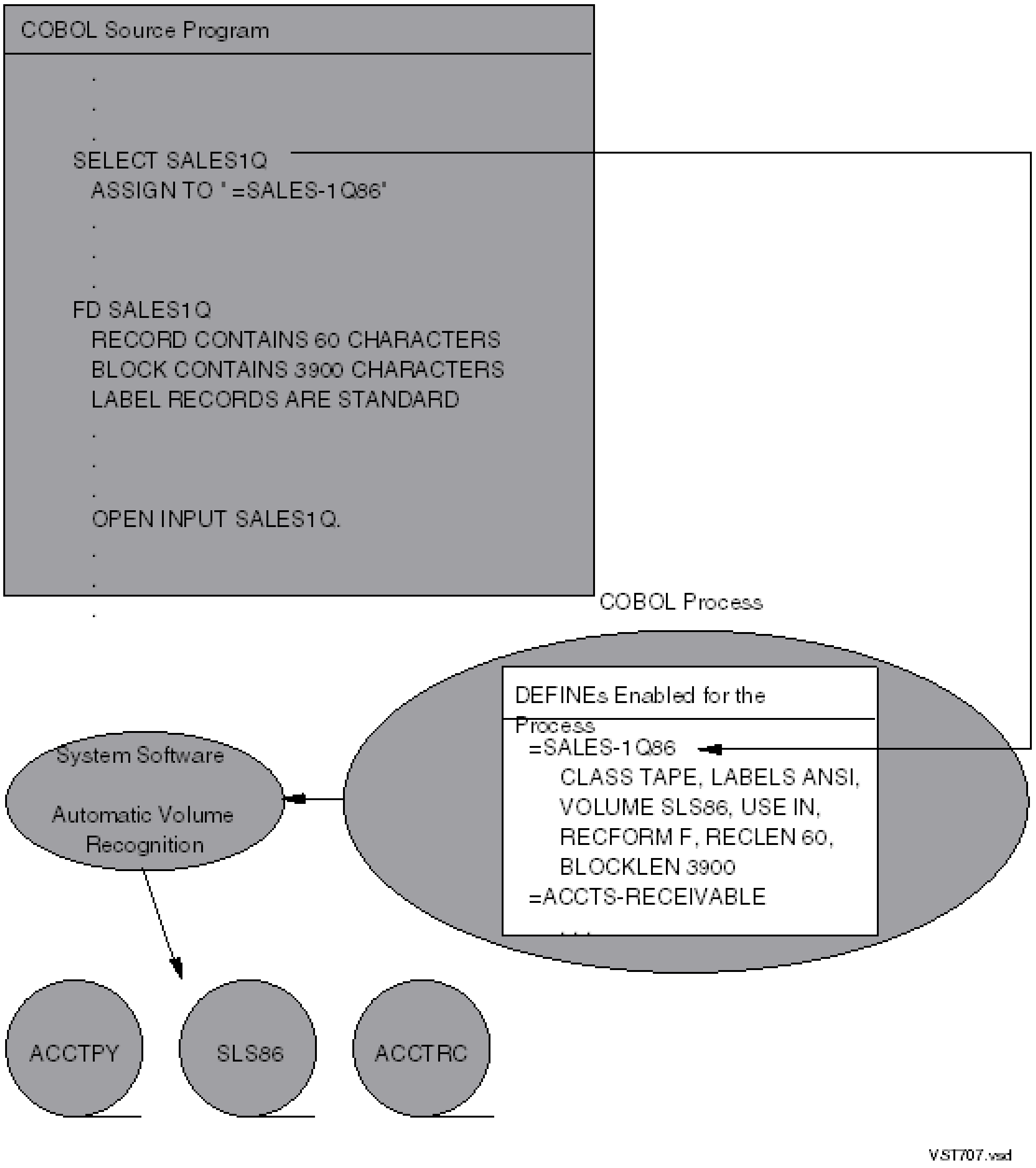
```
ADD DEFINE =DEF, CLASS TAPE, DEVICE $DRIVE,&
BLOCKLEN 132, RETENTION 5
```

Labeled Tape Files

The attributes of labeled tape files are:

- CLASS Attribute (Required)
- LABELS Attribute (Required)
- RECFORM Attribute (Conditional)
- BLOCKLEN Attribute (Conditional)
- RECLEN Attribute (Conditional)
- DEVICE Attribute (Conditional)
- FILESEQ Attribute (Conditional)
- USE Attribute (Optional)
- VOLUME Attribute (Optional)
- RETENTION or EXPIRATION Attribute (Optional)

Figure 28-2 DEFINE Names and COBOL File Names



```
    .
    .
    .
  SELECT SALES1Q
    ASSIGN TO "=SALES-1Q86"
    .
    .
    .
  FD SALES1Q
    RECORD CONTAINS 60 CHARACTERS
    BLOCK CONTAINS 3900 CHARACTERS
    LABEL RECORDS ARE STANDARD
    .
    .
    .
    OPEN INPUT SALES1Q.
    .
    .
    .
```

```
=SALES-1Q86
   CLASS TAPE, LABELS ANSI,
   VOLUME SLS86, USE IN,
   RECFORM F, RECLEN 60,
   BLOCKLEN 3900
=ACCTS-RECEIVABLE
   . . .
```

CLASS Attribute (Required)

The value of the CLASS attribute must be TAPE. The CLASS attribute must be the first attribute in the ADD DEFINE command, because it determines what other attributes the DEFINE can have and clears any existing attribute settings, including any that precede it in the ADD DEFINE command.

LABELS Attribute (Required)

The value of the LABELS attribute for a labeled tape file must reflect the type of labels the tape has:

Labels	LABELS Attribute Value
ANSI standard	ANSI
IBM standard	IBM
Any other	BYPASS

BYPASS tells the system to handle a labeled tape file like an unlabeled tape file. The labels are interpreted as data. Use BYPASS if either of these is true:

- The tape has labels that are not ANSI or IBM standard.
- The HP COBOL program does its own label processing.

The LABELS attribute of the DEFINE overrides any LABEL RECORDS clause in your HP COBOL program.

Table 28-3 Effect of LABELS Attribute and LABEL RECORDS Clause on Labeled Tape File

	LABEL RECORDS Clause	
LABELS Attribute Value	**LABEL RECORDS STANDARD**	**LABEL RECORDS OMITTED**
ANSI or IBM	Execution proceeds	HP COBOL sets file status code to 97* and execution proceeds
BYPASS	HP COBOL sets file status code to 97*, execution proceeds, and HP COBOL handles any labels as data	Execution proceeds and HP COBOL handles any labels as data

* File status code 97 means that the statement executed successfully, but the circumstances were not entirely as expected.

RECFORM Attribute (Conditional)

The RECFORM attribute is required unless you are bypassing labels (the value of the LABELS attribute is BYPASS). The value of the RECFORM attribute must reflect the type of records the tape has and must correspond to the RECORD CONTAINS clause in the file description entry in the program.

Table 28-4 Correspondence Between RECORD CONTAINS Clause and RECFORM Attribute Value

Record Length	RECORD CONTAINS Clause	RECFORM Attribute Value
Variable (undefined)	RECORD CONTAINS `min-length` TO `max-length` CHARACTERS	U
Fixed	RECORD CONTAINS `length-fixed` CHARACTERS	F

BLOCKLEN Attribute (Conditional)

The BLOCKLEN attribute is required if the tape file has variable-length records (the value of the RECFORM attribute is U) or if the tape file has fixed-length records (the value of the RECFORM attribute is F), and the file description entry does not have a BLOCK CONTAINS clause that specifies that the number of records per block is an integer greater than one.

If the tape file has variable-length records, the value of the BLOCKLEN attribute depends on the file's maximum record size and open mode. If the open mode is INPUT, the value of the BLOCKLEN attribute must be less than or equal to the maximum record size; if the open mode is OUTPUT or EXTEND, the value of the BLOCKLEN attribute must be greater than or equal to the maximum record size.

If the file has fixed-length records, the value of the BLOCKLEN attribute must be the maximum record size, which is obtained from the BLOCK CONTAINS clause or from the record description entry for the largest record.

RECLEN Attribute (Conditional)

The RECLEN attribute is required if the file has fixed-length records. Its value must be the same as the value of the BLOCKLEN attribute (that is, the maximum record size).

If the record has variable-length records, the RECLEN attribute is unnecessary; if you use it, its value is ignored.

DEVICE Attribute (Conditional)

The DEVICE attribute is required if you are bypassing labels (the value of the LABELS attribute is BYPASS); otherwise, it is optional.

When the DEVICE attribute is required, its value must be the system file name of a tape drive.

When the DEVICE attribute is optional, its value overrides the system's automatic volume recognition. You cannot specify a different device in response to the system mount message.

FILESEQ Attribute (Conditional)

The FILESEQ attribute is required if the file you want to use is not first on the tape or if your HP COBOL program has a MULTIPLE FILE clause. If you omit FILESEQ and the COBOL program does have a MULTIPLE FILE clause, the open operation fails.

NOTE: The 1985 COBOL standard classifies the MULTIPLE FILE clause as **obsolete**, so you are advised not to use it.

The value of the FILESEQ attribute specifies the position of a labeled tape file on a multiple-file tape. It must be an integer in the range 1 through 9999 (relative file positions are consecutive integers from 1 through 9999) and must match the value in the corresponding field of the tape label.

USE Attribute (Optional)

The USE attribute prevents inadvertent overwriting of a tape file. The value of the USE attribute must correspond to the tape file's open mode.

Table 28-5 Corresponding Open Modes and USE Attribute Values

Open Mode	USE Attribute Value
INPUT	IN or OPENFLAG
OUTPUT	OUT or OPENFLAG
EXTEND	EXTEND or OPENFLAG

The values IN, OUT, and EXTEND have the same meanings as the corresponding open modes; OPENFLAG means that the file can be opened in any mode that the OPEN statement specifies.

VOLUME Attribute (Optional)

The VOLUME attribute identifies the tape. The value of the VOLUME attribute must be the name of the tape volume as it appears in the tape label. This automatic volume recognition enables you or the operator to mount the labeled tape on any available tape drive before or after you start your HP COBOL program.

After locating the tape volume on a device, the operating environment passes the DEFINE to the tape process that controls the device. The tape process checks the values in the tape labels against the values of the DEFINE attributes. If the values do not match, the operating environment rejects the open request and issues an error message. If the values match, the operating environment approves the open request, and the HP COBOL process can use the tape file.

RETENTION or EXPIRATION Attribute (Optional)

Use the RETENTION or EXPIRATION attribute when you want to save a file for any period of time, even part of a day; otherwise, the file expires as soon as you write it, and the operating environment might overwrite the file.

The value of the RETENTION or EXPIRATION attribute is an integer, the number of days before the file expires.

CAUTION: On a multiple-file volume, the operating environment might overwrite an unexpired file that follows an expired one; a volume on which the first file has expired is a scratch volume.

Example 28-12 adds a DEFINE for a labeled tape file with variable-length records whose labels are to be bypassed. The file is on the tape on the tape drive named $DRIVE.

Example 28-12 DEFINE for Labeled Tape File With Variable-Length Records and Labels to Be Bypassed

```
ADD DEFINE =ABC, CLASS TAPE, LABELS BYPASS,&
BLOCKLEN 132, DEVICE $DRIVE
```

Example 28-13 adds a DEFINE for a labeled tape file with standard IBM labels and variable-length records. The file is on the tape whose volume ID is 45329, and it will not expire for 100 days.

Example 28-13 DEFINE for Labeled Tape File With Variable-Length Records and Standard IBM Labels

```
ADD DEFINE =DEF, CLASS TAPE, LABELS IBM, RECFORM U,&
BLOCKLEN 132, VOLUME 45329, RETENTION 100
```

Example 28-14 adds a DEFINE for a labeled tape file with standard ANSI labels and fixed-length records. The file is the second file on the tape, and it will not expire for 10 days.

Example 28-14 DEFINE for Labeled Tape File With Fixed-Length Records and Standard ASCII Labels

```
ADD DEFINE =DEF, CLASS TAPE, LABELS ANSI, RECFORM F,&
BLOCKLEN 132, RECLEN 132, FILESEQ 2, EXPIRATION 10
```

Executing HP COBOL Programs That Use Tape Files

During execution of an HP COBOL program that uses tape files, the operating environment software and the HP COBOL run-time library monitor:

- Tape mounting
- Adherence to rules for using DEFINEs
- For labeled tapes, consistency of labeled-tape information in the HP COBOL program, DEFINE, and tape label

You can mount tapes yourself or have an operator mount them.

Mount Messages

Mount messages for unlabeled tape files (and labeled tapes files being handled as such) are issued by an HP COBOL run-time routine; mount messages for labeled tape files are issued by the labeled-tape processing software.

Table 28-6 Effect of LABELS Attribute on Mount Messages

LABELS Attribute Value	Software That Issues Mount Messages
None (no LABELS attribute)	HP COBOL run-time routine
OMITTED	HP COBOL run-time routine
BYPASS	HP COBOL run-time routine
ANSI	Labeled-tape processing software
IBM	Labeled-tape processing software

Unlabeled Tapes

Without labels, the operating environment cannot use automatic volume recognition to locate the tape, so the mount message from the HP COBOL run-time routine specifies the tape drive on which to mount the tape—the tape drive specified by the DEVICE attribute of the file's DEFINE.

If the mount message appears on a terminal display, you can specify a different tape drive on which to mount the tape. Respond to the mount message by typing the device name of the new tape drive after the colon:

```
TYPE RETURN OR $DEVICE (OR 'NO' TO END INPUT): $TAPE
```

HP COBOL opens the tape drive you specify.

If a DEFINE exists for the tape file and you specify a different device in response to an HP COBOL mount message, the HP COBOL run-time library alters the DEVICE attribute of the file's DEFINE.

If no DEFINE exists for the tape file, the HP COBOL run-time library creates a DEFINE with the required attributes (including the DEVICE attribute).

The new DEVICE attribute value or the new DEFINE remains active until one of these occurs:

- The HP COBOL process terminates.
- The HP COBOL process uses a DEFINE procedure to change the value of the DEVICE attribute or to delete the DEFINE.
- The HP COBOL run-time library changes the value of the DEVICE attribute when you specify a different device for the file in response to another HP COBOL mount message.

Labeled Tapes

A mount message from the labeled-tape processing software usually does not specify a device on which to mount the tape. The labeled-tape processing software uses automatic volume recognition to find a tape volume on one of the system's tape drives, so you or the operator can mount the tape on any available tape drive. For a continuation reel of a multitape file, however, a mount message from the labeled-tape processing software specifies the same device as the first reel.

Run-Time Errors

Table 28-7 Run-Time Errors Specific to Tape File Use

Message Number (CRE)	Text of Message (in Non-CRE Environment)
170	MULTIPLE FILE TAPE file not on tape
171	OPEN positioning for MULTIPLE FILE TAPE failed
214	OPEN I-O for file not on input-output device
215	Wrong or missing LABELS attribute
216	Wrong or missing USE attribute
217	Wrong or missing RECFORM attribute
218	Wrong or missing RECLEN attribute
219	Wrong or missing BLOCKLEN attribute
220	Wrong or missing FILESEQ attribute
221	Wrong or missing DEVICE attribute
222	A DEFINE procedure failed with error *nnn*
223	DEFINE required for LABEL RECORDS STANDARD

Input-output errors that are not specific to tape files can also occur. For a complete list of HP COBOL run-time errors and suggestions for recovering from them, see Chapter 49: Run-Time Diagnostic Messages (page 1193).

29 Disk Input and Output

Both batch programs and transaction-supporting servers use disk files.

Common Disk File Topics

Allocation

The file system allocates physical storage for a disk file in the form of file extents. A file extent is a contiguous block of storage, starting on a disk sector boundary and containing a multiple of 2,048 bytes (up to 134,215,680 bytes). The file system permits a disk file to have up to 978 extents—one primary extent and up to 977 secondary extents. The primary extent can be a size different from that of the secondary extents. The file system allocates extents only when needed. The file extents that constitute a file are not necessarily contiguous. When all extents are allocated, writing to the file causes I-O file status code "34."

HP COBOL has no mechanism for specifying extents. If an HP COBOL program creates a disk file, it allocates a primary extent of 4 x 2,048 bytes and a secondary extent of 20 x 2,048 bytes. An HP COBOL program can call the routine COBOL_CONTROL_ to allocate and deallocate extents.

If you create a disk file outside of HP COBOL and do not specify extents, the file system allocates a primary extent of 2,048 bytes. If you do not explicitly specify a secondary extent, the file system allocates secondary extents, as needed, that are the same size as the primary extent.

Partitioned Files

An Enscribe disk file can be composed of up to 16 partitions. Every partition consists of a primary extent plus up to 977 secondary extents. Each partition must reside on a different volume. If your system is connected to a network, either through Expand or through the fiber optic extension (FOX) of the interprocessor bus, the partitions can even reside on different systems (at a performance penalty). For information about Expand, see the *Expand Network Management and Troubleshooting Guide*.

An HP COBOL program can read or write a partitioned file but cannot create one. You must use FUP to create a partitioned file; for details, see the *Guardian Programmer's Guide*.

Partitioned files have these significant advantages:

- They can be up to 16 times as large as nonpartitioned files.
- They can be accessed faster because separate read heads are used.
- The loss of access to a disk need not mean the loss of access to the entire file.

Purging Files or Their Data

You can purge a file from the disk from your TACL session with the TACL command PURGE. An HP COBOL program can purge a file from the disk by calling the operating system routine PURGE. Purging a file from the disk merely removes the disk file from the directory and frees the disk space for use by other files; it does not erase the data on the disk.

You can erase the contents of a file without purging the file with the FUP command PURGEDATA. An HP COBOL program erases the contents of a file without purging the file simply by opening the file for output. An HP COBOL program can erase the contents of a file by calling routine COBOL_CONTROL_. The data remains on the disk—only the current-record, next-record, and EOF pointers are reset, and the EOF pointer in the file label on the disk are changed to indicate that the file is logically empty.

To erase the data on the disk, you must:

1. Use either the FUP SECURE command with its CLEARONPURGE option or call the routine COBOL_SETMODE_ to set function 52 (see the *Guardian User's Guide*).

2. After setting the CLEARONPURGE flag, call the PURGE routine (directly or through FUP).

The CLEARONPURGE flag has no effect on the outcome of a PURGEDATA operation performed through FUP or through the CONTROL routine.

Locking

An HP COBOL program can lock and unlock an entire file with the LOCKFILE and UNLOCKFILE statements. Batch programs lock entire files more often than transaction processing programs do.

An HP COBOL program can lock individual records in a file with the LOCK phrase of the READ statement and unlock them with either the UNLOCKRECORD statement or the REWRITE statement with UNLOCK.

You must lock files when one process needs exclusive access to a record, a set of records, or an entire file for some logical operation to complete successfully (as in the case of TMF). For more information, see Avoiding Deadlock).

Ownership and Security

Each disk file on the HP system has an owner and a file security. Initially, the owner is the user who creates the file and the security is the default file security. The owner can transfer ownership to another user with the FUP GIVE command. The owner of a file can change the file's security with the FUP SECURE command.

The security system recognizes a file owner by the user ID, which consists of a group number and a user number. The operating environment obtains the group number and user number from the group name and user name when the user logs on; for example, if you log on as PROJECT3.SANDY, your group name is PROJECT3 and your user name is SANDY, and the operating environment translates these names to their corresponding group number and user number.

Security Attributes

Security attributes of a disk file allow the owner to control access to the file. If you own a file, you can change its security at any time with the FUP SECURE command:

```
FUP SECURE file-name,"RWEP"
```

Security Attribute	Meaning
R	Reading
W	Writing
E	Executing
P	Purging

The value of each security attribute determines the class of user who has permission to perform the corresponding operation on the file (see Table 29-1).

Table 29-1 Security Attributes for Disk Files[1]

Who Can Perform the Corresponding Operation on the File	On Local System Only	From Anywhere on Expand Network[2]
Owner only	O	U
Any member of owner's user group	G	C
Any user	A	N
Local super ID only	-	Not applicable

[1] Uppercase letters are shown, but the corresponding lowercase letters are equivalent.

[2] If your HP system is part of an Expand network.

To display a file's current security attributes, use the FUP INFO command.

To alter a file's security from an HP COBOL program, call the routine COBOL_SETMODE_ (page 642).

Suppose that your HP system is part of an Expand network and you own the disk file FILE3. The command in Table 29-1 prevents everyone but you from reading, writing, executing, or purging FILE3, but you can only read, write, execute, or purge the file on your local system.

Example 29-1 File Accessible Only to Owner and Only Locally

```
FUP SECURE FILE3,"OOOO"
```

The command in Table 29-2 is the same as the command in Table 29-1, except that you can read, write, execute, or purge FILE3 from anywhere on the Expand network.

Example 29-2 File Accessible Only to Owner

```
FUP SECURE FILE3,"UUUU"
```

The command in Table 29-3 allows any member of your user group to read or execute FILE3 from anywhere on the Expand network, but allows only you to write or purge it, and only from your local system.

Example 29-3 File Readable and Executable to Owner's User Group

```
FUP SECURE FILE3,"COCO"
```

The command in Table 29-4 allows any user to read FILE3, only you to write or purge it, and only a local super ID user to execute it—and these rules apply from anywhere on the Expand network.

Example 29-4 File Readable to Any User

```
FUP SECURE FILE3,"NU-U"
```

Default File Security

A newly created file has the default file security. You can set the default file security for the duration of your TACL session with the TACL command VOLUME; you can set it until further notice with the TACL command DEFAULT (further notice being another DEFAULT command).

The command in Example 29-5 sets the default file security to "UUUU" until you override it with a VOLUME command or another DEFAULT command.

Example 29-5 Setting Default File Security to "UUUU"

```
DEFAULT, "UUUU"
```

The command in Example 29-6 sets the default file security to "CUCU" for the duration of your TACL session. If you log off and log back on, the default file security reverts to "UUUU."

Example 29-6 Setting Default File Security to "CUCU"

```
VOLUME, "CUCU"
```

If a Pathway server written in HP COBOL creates a file, the security of the file is that of the DEFAULT command active when the monitor process (PATHMON process) governing the server class starts to execute.

Accessing Files on Other Nodes

For you, as a user on one node, to have any access at all to a file on another node, even when the file's security attributes permit it, you must have two remote passwords declared on each node. The TACL command REMOTEPASSWORD enables you to set the remote passwords on your own node. You must arrange with the system manager of the other node to have the same two passwords established on that node. For further information about remote passwords, see the *Guardian User's Guide*.

Fixed-Length and Variable-Length Records

A file has variable-length records if its file description entry has either:

- A RECORD CONTAINS *m* TO *n* CHARACTERS clause
- A RECORD IS VARYING FROM *m* TO *n* clause
- No RECORD clause and different-sized record descriptions

If your source program does not explicitly describe a file as having variable-length records, the HP COBOL run-time write routine writes fixed-length physical records. The length of each physical record in the file is the larger of:

- The length of the longest record description entry for the file
- The maximum record length explicitly stated in the RECORD clause of the file description entry

If a file contains records of different lengths, your program must declare it as having variable-length records to be able to read it. If the program declares the file (explicitly or implicitly) as having fixed-length records, and the program then tries to read a record shorter than the declared length, the read operation succeeds with a file status code "04."

Unstructured files must have fixed-length records.

Exclusion Modes

The exclusion mode is a feature of a file that determines whether other processes can read or write to the file being opened. If two separate processes are both writing to the same file, it is possible that both will attempt to write the same record and thereby corrupt the database.

Table 29-2 Exclusion Modes and Their Meanings

Exclusion Mode	While the process that opened the file has it open ...
EXCLUSIVE	No other process can read or write the file.
PROTECTED	Any other process can read the file, but no other process can write the file.
SHARED	Any other process can read or write the file.

The exclusion mode can be specified in the OPEN statement or in an ASSIGN command. The exclusion mode in an OPEN statement takes precedence over the exclusion mode in an ASSIGN command. If neither the OPEN statement or ASSIGN command specifies an exclusion mode, the exclusion mode is PROTECTED if the file is being opened for input; EXCLUSIVE otherwise.

Most server files have the exclusion mode SHARED so that several servers of the same class can all access the file. The servers use record locking to prevent conflicting write operations to the file.

If you have a batch program that needs to open a file that one or more servers have opened for shared access, give careful consideration to whether the batch program should open the file for shared, protected, or exclusive access. If the batch program is doing something that affects the servers, it should probably open the file for exclusive access and thus fail if any server has it open. If the batch program cannot affect the servers, and the activity of the servers does not affect the activity of the batch program, then the batch program can open the file for shared access.

Time Limits

If you include a TIME LIMITS phrase in an OPEN statement, you can then include a TIME LIMIT phrase in a LOCKFILE, READ, or START statement to cause the statement to be abandoned if it does not finish executing in a certain amount of time. The TIME LIMIT phrase in a LOCKFILE, READ, or START statement sets the file's status code so that the requester can tell that the data is not currently accessible. This action helps avoid deadlock.

Reading Files From Called Programs

Two separately compiled programs cannot share files; that is, if one program opens the file and then calls another separately compiled program, the caller cannot pass the file (or access to the file) to the called program. A pair of separately compiled calling and called programs can both have the same disk file open, but each has its own record area, its own current record pointer, its own file status data item, and so on.

When an HP COBOL run unit opens a file, the file remains open until one of these happens:

- The run unit explicitly closes the file.
- The calling program cancels the called program that has the file open (with the CANCEL statement).
- The run unit terminates execution.

A well-structured program unit has all the file activity for a certain file within one called program. The other programs in the run unit call that program to open, read, write, position, or close the file.

In the called program, include a data item whose initial value indicates that the file is not open. When the program opens the file, set the data item to a value that indicates that the file is open. When the program closes the file, set the data item to a value that indicates that the file is closed. The value of the data item is retained between calls to that program unless a CANCEL statement intervenes.

Sharing Files Among HP COBOL Programs

Two programs in a run unit can refer to common file connectors in these circumstances:

- Any program that has described an external file connector can refer to that file connector.
- If a program G is contained within another program H, both programs can refer to a common file connector. They do so by referring to an associated global file name (or associated global record-name, in the case of the WRITE statement) described in either:
  - The containing program H
  - Any program that directly or indirectly contains H

If several programs define a file connector as external (causing its storage location to be a single location outside all programs) and they also define the file connector as having a global name, then all such programs and all programs nested within each of them have access to the file connector.

Sharing Files Among Different-Language Modules

If your program consists of modules written in different languages, the modules can share the standard files—the predefined files called “standard input,” “standard output,” and “standard log.”

For more information about mixed-language programs sharing standard files, see the *CRE Programmer’s Guide*.

Modification

With the ENABLE utility, you can build a file-maintenance application that allows you to modify a disk file. ENABLE uses a Data Definition Language (DDL) file description to create a Pathway application that you can use to read and change the data in the disk file. For details, see the *ENABLE User’s Guide*.

Maximum Number of Files

The maximum number of files on a volume is determined during system configuration. The maximum number of files that a single process can have open concurrently depends on the amount of space available for file control blocks and buffers in the upper 64 KB of the process’s user data space.

Types of HP Disk Files

NonStop systems provide two types of disk files:

- Unstructured (Sequential) Files
- Structured Files

Unstructured (Sequential) Files

An unstructured file consists of a stream of bytes, and Enscribe allows each byte to be addressed directly. The organization of an unstructured file is the responsibility of the program that uses it. An HP COBOL program can only use an unstructured file as a sequential file.

Two examples of unstructured files are object code files (file code 100) and EDIT files (file code 101).

An EDIT file can have variable-length records, but all other unstructured files must have fixed-length records. No unstructured file can have alternate record keys.

A program written entirely in HP COBOL can open an EDIT file for both input and output (or extension). The HP COBOL program cannot access the line numbers of the EDIT file, only the text of the lines themselves.

A program written entirely in HP COBOL can open unstructured files that are not in EDIT format for input or output, but cannot create an unstructured file.

To use an existing unstructured file, your HP COBOL program must declare it:

- With ORGANIZATION SEQUENTIAL
- With ACCESS MODE SEQUENTIAL
- With fixed-length records (if the file is not an EDIT file)
- Without alternate keys

When using an unstructured file, an HP COBOL program still operates on the basis of records, but they are strictly logical records. If one HP COBOL program writes an unstructured file of 80-character records, another program can read it with any record length. If the reading program expects 37-character records, the run-time routines deliver 37-byte pieces of the continuous byte stream. You can rewrite a record with another record of the same length, but you cannot delete a record.

The HP COBOL run-time routines that access unstructured disk files manage those files directly. An unstructured file has no records, as far as Enscribe is concerned; the declarations in your HP COBOL program determine how the HP COBOL run-time routines manipulate the unstructured files.

Structured Files

Structured disk files are either entry-sequenced, relative, or key-sequenced.

Table 29-3 Corresponding HP and HP COBOL Disk File Terms

HP Term	HP COBOL Term
Relative	Relative
Entry-sequenced	Sequential
Key-sequenced	Indexed

The HP COBOL run-time routines that access structured disk files (entry-sequenced, relative, and key-sequenced) do so with the help of the operating system routines that are collectively called the Enscribe database record manager.

Table 29-4 Comparison of Structured File Characteristics

	Type of Structured File		
Characteristic	**Entry-Sequenced (Sequential)**	**Relative**	**Key-Sequenced (Indexed)**
Records ordered by ...	Order in which they were entered	Record number	Value of prime record key
Access is by ...	Record address or alternate record key Maximum alternate record key length = 253 bytes (249 if not unique)	Record number or alternate record key Maximum record number = 1,048,575 Maximum alternate-key length = 253 bytes (249 if not unique)	Prime or alternate record key Maximum prime record key length = 522 bytes Maximum alternate record key length = 253 bytes (253 minus prime length if not unique)
Space occupied by a record ...	Depends on length specified when record is written Maximum logical record length = 4,072 bytes	Is specified when file is created Maximum logical record length = 4,072 bytes	Depends on length specified when record is written Maximum logical record length = 2,035 bytes
Record can be shortened or lengthened	No	Yes	Yes, and space freed by shortening a record can be reused within its block
Record can be deleted	No, but its space can be used for another record of the same size	Yes, and its space can be reused	Yes, and its space can be reused within its block

In HP COBOL, the DEPENDING phrase of the RECORD IS VARYING clause enables you to designate a data item to receive the record length upon the successful completion of a READ statement and to control the record length for a WRITE statement.

HP COBOL allows writing and reading zero-length records. To write a zero-length record, specify

```
RECORD IS VARYING FROM 0 TO ... DEPENDING ON data-name
```

and move the value zero into `data-name` before writing. After you read a file with the same record description, `data-name` has the value zero. TMF writes zero-length records when it backs out of transactions.

If an HP COBOL program describes fixed-length records, any zero-length records are discarded when read. If an HP COBOL program describes variable-length records, upon return from the READ statement, the entire record area is undefined. For this reason, you are advised to specify the DEPENDING phrase when reading variable-length records, and to process zero-length records appropriately (by ignoring them, for example).

Under Enscribe, entry-sequenced files can have alternate keys. You must create these files by calling Enscribe directly in one of these ways:

- From your TACL session, use FUP.
- From within an HP COBOL program, call the FILE_CREATE_ procedure.

Entry-Sequenced (Sequential) Files

An entry-sequenced file created by an HP COBOL program is either a series of fixed-length records or a series of variable-length records.

Your program ordinarily reads entry-sequenced files sequentially. You can read the records in the order in which they appear in the file by opening the file for input and reading record after record. You can append records to the file by opening the file with an extension and writing records to the file.

Enscribe allows each physical record of an entry-sequenced file to have any length from zero bytes (empty) to the maximum record length that was declared when the file was created. HP COBOL allows each logical record to have any length from zero bytes to the maximum record length declared explicitly in the file description entry or implicitly in the record definitions following the file description entry.

If your source program describes the file as having variable-length records, each physical record uses only as many bytes of disk space as needed; therefore, the number of records for each block can vary according to the length of the records in each block (although some space at the end of a block can be wasted).

If an entry-sequenced file has alternate keys, you can use a START statement to specify the next record to read.

Relative Files

Relative files contain records that exist independent of each other. Each record is associated with a unique relative record number—an ordinal number. The first record is number 1, the second is number 2, and so forth. The number exists independent of the contents of the record. You can read any record by number, but the record number is not part of the record.

The record numbers of a relative file need not be consecutive; Enscribe does not require that there be a record for each ordinal number between that of the first record and the highest number associated with any record in the file. If the HP COBOL run-time routines ask Enscribe for the record associated with a certain number and there is no such record in the file, Enscribe reports that no such record exists, and the HP COBOL run-time routine raises the invalid-key condition (and, if you defined a file status code data item, the run-time routine stores an appropriate value in it).

You can also read the file in record-number order. In this case, Enscribe does not report absent records—it returns the records that exist. Enscribe can report the record number of each record as it is read. When you read a relative file this way, the HP COBOL run-time routines set the relative key data item to reflect the record number.

Both Enscribe and HP COBOL (as an HP extension) provide two special record numbers: -1 and -2. If you specify record number -1, a record is written at the end of the file; if you specify record number -2, a record is written in the next available position.

Enscribe allocates the same fixed amount of disk storage to each record. Each record can vary in length from empty to that fixed limit. In HP COBOL terms, you can write fixed-length records of any length up to the limit, or variable-length records of any length up to the limit. If HP COBOL creates the file for you, that limit is the larger of:

- The length of the longest record description entry for the file
- The maximum record length explicitly stated in the RECORD clause of the file description entry for the file

If a relative file has alternate keys, you can use a START statement not only to specify the next record to read, but also to change the key of reference.

You can write anywhere in the file, replacing existing records or installing a record at an ordinal position that had no record associated with it. You can replace an existing record with a record of a different length. You can delete a record. A deleted record is not the same as a record of length zero. A read operation with an invalid key leaves the contents of the record area undefined (that is, unpredictable).

Key-Sequenced (Indexed or Queue) Files

Key-sequenced files contain records that are stored in independent positions on the disk, but accessed in ascending sequence ordered by the unique value of a field within each record. Enscribe calls this field the prime record key; HP COBOL calls it the prime record key. You can read any record by specifying its key value.

A key-sequenced file can be incomplete; Enscribe does not require that there be a record for each possible key value. If HP COBOL run-time routines ask Enscribe for the record associated with a certain key value, and there is no such record in the file, Enscribe reports that no such record exists and the HP COBOL run-time routine raises the invalid-key condition (and, if you defined a file status code data item, stores an appropriate value in it).

You can also read the file in key-value order. In this case, Enscribe does not report absent records—it returns the records that exist. Because the key is part of the record, you have access to the key value when you have read the record.

Enscribe allocates only as much space on disk as necessary for each key-sequenced file record. Each record can vary in length from empty to the maximum record length defined for the file. In HP COBOL terms, you can write fixed-length or variable-length records (whichever the file was declared to have) of any length up to the limit. If HP COBOL creates the file for you, that limit is the larger of:

- The length of the longest record description entry for the file
- The length declared in the RECORD CONTAINS clause

You can write records anywhere in the file, introducing new records or replacing existing records. You can replace an existing record with a record of a different length. You can delete a record. A deleted record is not the same as a record of length zero.

In HP COBOL, the DEPENDING phrase of the RECORD CONTAINS clause enables you to designate a data item to receive the record length upon the successful completion of a READ statement and to control the record length for a WRITE statement. A read operation with an invalid key leaves the contents of the record area undefined (that is, unpredictable).

Creating and Using HP COBOL Sequential Files

In a sequential file—a file with sequential organization—records are arranged in a fixed predecessor-successor relationship that is established as the records are entered in the file.

HP COBOL sequential files fall into these categories:

- Entry-Sequenced Files
- Unstructured Files
- Unstructured EDIT Files

HP COBOL programs can create, read, write, and purge all of the preceding forms of sequential file. To create an entry-sequenced file that has alternate keys, however, the COBOL program must use the FILE_CREATE_ procedure (see Unstructured Files).

Entry-Sequenced Files

An HP COBOL program can create and write an entry-sequenced file that any other HP product can later read. An HP COBOL program can read an entry-sequenced file created by any HP product.

An HP COBOL program can use the FILE_CREATE_ procedure to create an entry-sequenced file that has alternate keys. An HP COBOL program cannot create an entry-sequenced file that has alternate keys directly, because the operating environment maintains the alternate keys in one or more separate files, whose names are recorded with the operating environment information about the entry-sequenced file (see Alternate Record Keys).

Outside an HP COBOL program, you can use the TACL command FUP CREATE to create an entry-sequenced file that has alternate keys.

Unstructured Files

An HP COBOL program cannot create an unstructured file directly because the HP COBOL language cannot describe a file as unstructured. To create an unstructured file indirectly, an HP COBOL program calls the FILE_CREATE_ procedure.

An easier way to create an unstructured disk file is to use FUP outside of any HP COBOL program. You can use the routine COBOL_ASSIGN_ to set *file-code* to 101 (for an EDIT file) or to set *file-type* to 0 (for a non-EDIT file).

Suppose that your HP COBOL program needs an unstructured disk file named MYTEMP on volume $MYVOL that accommodates 5K eighty-byte records in one extent. Suppose that your HP COBOL program includes this data description:

```
01 WS-TEMP-NAME.
   03 WS-VOL-NAME    PIC X(8).
   03 WS-SUBVOL-NAME PIC X(8).
   03 WS-FILE-NAME   PIC X(8).
```

This HP COBOL code creates MYTEMP:

```
ENTER "COBOL_ASSIGN_" USING MYTEMP, OMITTED, 0
```

The value zero in the third parameter causes the file to be unstructured. The omitted parameter is for the file code (which defaults to zero).

The FUP command that is equivalent to the preceding HP COBOL code that created MYTEMP is:

```
FUP CREATE $MYVOL.MYTEMP, TYPE U
```

Additional FUP parameters enable you to specify these attributes:

- File code
- Primary extent size
- Secondary extent size
- Whether the file is audited by TMF
- Whether the file's label is to be copied to the disk every time the file's EOF value changes

For more information about FUP, see the *Guardian Programmer's Guide*.

EDIT Files

An EDIT file is a special form of unstructured file whose file code is 101. HP COBOL programs can read, write, and create EDIT files. The file code 101 must be assigned to the file in one of these ways before the HP COBOL program opens the file:

- Create the file outside the program (with an HP editor or FUP).
- Have the HP COBOL program create the file by calling the FILE_CREATE_ procedure.
- Before executing the HP COBOL program, use the TACL command ASSIGN to associate the file's name with the file code 101.
- Have the program call the routine COBOL_ASSIGN_ to associate the file's name with the file code 101.

Creating and Using HP COBOL Relative Files

In a relative file—a file with relative organization—records are accessed by their record numbers. A record number is the position, relative to the beginning of the file, at which the record is stored. It is not related to the order in which the records were entered in the file. Record numbers need not be consecutive; for example, you can create a relative file with only three records whose numbers are 12, 19435, and 237.

An HP COBOL program can create and write a relative file that any other HP product can later read. An HP COBOL program can read a relative file created by any HP product.

An HP COBOL program can create, read, write, and purge relative files. An HP COBOL program must use the FILE_CREATE_ procedure to create a relative file that has alternate keys.

An HP COBOL program cannot create a relative file that has alternate keys directly, because the operating environment maintains the alternate keys in one or more separate files whose names are recorded with the operating environment information about the relative file (see Alternate Record Keys).

An HP COBOL program can create a relative file that does not have alternate keys simply by describing the file as having relative organization (but without alternate keys) and opening the file for OUTPUT, EXTEND, or I-O. The HP COBOL run-time routines create the relative file.

There is one important difference between the way HP COBOL uses relative files and the way many other HP utilities use them: HP COBOL calls the first record in a relative file record 1, and the operating environment and most other HP software call the first record in a relative file record 0.

The HP COBOL run-time routines that manipulate relative files convert HP COBOL record numbers to Enscribe record numbers by subtracting 1 from the value of each HP COBOL record number (except in the case of the two special record numbers -1 and -2 explained in Types of HP Disk Files).

If a file contains numeric values that are to be used as record numbers, all programs that use the file must use the same record numbering scheme.

Example 29-7 Relative File Without Alternate Keys

```
ENVIRONMENT DIVISION.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
  SELECT RELFILE
    ASSIGN TO "$FIDDL.DEE.D"
    ORGANIZATION IS RELATIVE
    ACCESS MODE IS RANDOM
    RELATIVE KEY IS FIDLKEY.
...
DATA DIVISION.
FILE SECTION.
FD  RELFILE
```

```
    RECORD CONTAINS 5 TO 200 CHARACTERS
    LABEL RECORDS ARE OMITTED.
01  REL-RECORD.
...
WORKING-STORAGE SECTION.
...
01  KEYS-GROUP.
    03 FIDLKEY    PIC 9(8) COMPUTATIONAL.
```

These FUP commands declare the same relative file as the HP COBOL code in Example 29-7:

```
SET TYPE R
SET EXT (1000 RECS, 2000 RECS)
SET REC 200
SET BLOCK 2048
CREATE $FIDDL.DEE.D
```

Creating and Using HP COBOL Indexed Files

In an indexed file—a file with indexed organization—records are accessed by the values of a key field within each record.

An HP COBOL program can create and write an indexed file that any other HP product capable of handling indexed files can later read. An HP COBOL program can read an indexed file created by any HP product.

An HP COBOL program can create, read, write, and purge indexed files. An HP COBOL program must use the FILE_CREATE_ procedure to create an indexed file that has alternate keys.

An HP COBOL program cannot directly create an indexed file that has alternate keys, because the operating environment maintains the alternate keys in one or more separate files, whose names are recorded with the operating environment information about the indexed file (see Alternate Record Keys).

An HP COBOL program can create an indexed file that does not have alternate keys simply by describing the file as having indexed organization (but without alternate keys) and opening the file for OUTPUT, EXTEND, or I-O. The HP COBOL run-time routines create the indexed file.

Example 29-8 Indexed File Without Alternate Keys

```
ENVIRONMENT DIVISION.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
  SELECT INDFILE
    ASSIGN TO "$COBOL.PROGS.JAKE"
    ORGANIZATION IS INDEXED
    ACCESS MODE IS DYNAMIC
    RECORD KEY IS JAKE-KEY.
...
DATA DIVISION.
FILE SECTION.
FD  INDFILE
    RECORD CONTAINS 5 TO 200 CHARACTERS
    LABEL RECORDS ARE OMITTED.
01  IND-RECORD.
...
WORKING-STORAGE SECTION.
...
01  KEYS-GROUP.
    03 JAKE-KEY   PIC X(18).
```

These FUP commands declare the same indexed file as the preceding HP COBOL code:

```
SET TYPE K
SET EXT (1000 RECS, 2000 RECS)
SET REC 200
SET BLOCK 2048
CREATE $COBOL.PROGS.JAKE
```

Creating and Using Queue Files

A queue file is an indexed file (and therefore, a key-sequenced file) that can function as a queue. Processes can queue and dequeue records in a queue file.

Queue files contain variable-length records that are accessed by values in designated key fields. Unlike other key-sequenced files, queue files have prime keys but cannot have alternate keys. The prime key for a queue file includes an 8-byte timestamp; you can add a user key if desired. The disk process inserts the timestamp when each record is inserted into the file, and maintains the timestamp during subsequent file operations.

For more information about queue files, see the *Enscribe Programmer's Guide*.

Creating a Queue File

Before you can use a queue file, you must create it (with the FUP CREATE command, for example). Its prime key must have at least 8 bytes (for the timestamp). If you want to add a user key, describe it before describing the timestamp, as in Example 29-9.

Example 29-9 Creating a Queue File With a Timestamp and a User Key

```
SELECT queue-file ASSIGN to queuef
  ORGANIZATION IS INDEXED
  ACCESS MODE IS SEQUENTIAL
  RECORD KEY IS queue-key
FD queue-file
  RECORD IS VARYING FROM 10 TO 100 CHARACTERS
    DEPENDING ON rec-len.
01 queue-rec.
   02  queue-key.
       03  user-key  PIC XX.
       03  timestamp  PIC X(8).
   02  the-rest  PIC X(90).
```

Opening a Queue File

Open a queue file for shared access and SYNCDEPTH 0; for example:

```
OPEN queue-file INPUT SHARED SYNCDEPTH 0
```

Reading a Queue File

When you read a record of a queue file, you have a choice of:

- Removing the Record From the Queue
- Leaving the Record in the Queue

Removing the Record From the Queue

To read a record and remove it from the queue, use the READ ... NEXT WITH LOCK statement. The compiler translates the READ ... NEXT WITH LOCK statement to a call to the Guardian routine READUPDATELOCK[X].

Leaving the Record in the Queue

To read a record but leave it in the queue, use the READ ... NEXT statement (without LOCK). The compiler translates the READ ... NEXT statement to a call to the Guardian routine READ[X].

Writing to a Queue File

To write to a queue file, be sure that it exists and then open it for input and output; for example:

```
OPEN queue-file I-O SHARED SYNCDEPTH 0
```

In the file-control entry, specify ACCESS MODE IS RANDOM.

After the file is open, use any form of the WRITE statement to write to it. The value that you write to the timestamp field does not matter, because the system replaces that value with its own timestamp.

Establishing Starting Points in Files

When you read a file using a key, you sometimes want to start at the record associated with a given key value and continue to read successive records according to the value of that key until you reach the record associated with a certain other value of that key. The key can be the prime key of a file of relative or indexed organization or an alternate key of any structured file.

Such a key is termed the key of reference. The mechanisms in HP COBOL that implement this form of reading are the DYNAMIC access mode, the START statement, and the sequential form of the READ statement (a READ statement with a NEXT phrase).

Key of Reference

COBOL defines the "key of reference" as "the prime or alternate key currently being used to access records within an indexed file." HP COBOL extends that definition to "the prime or alternate key currently being used to access records within an indexed or relative file, or the alternate key currently being used to access records in an entry-sequenced file."

Enscribe uses the term "access path" for the sequence of records accessed by the key of reference.

You establish the key of reference by executing a START statement or a READ statement containing a KEY phrase. In a START statement, the key can specify the identifier of an entire key field or the identifier of a leftmost subordinate of the key field. For example, in the record in Example 29-10, if LAST-NAME, DEPARTMENT-NUMBER, and JOB-TITLE are alternate keys, you can use LN5, LN10, LN15, DN3, and J-T-PORTION as keys.

Example 29-10 Key of Reference

```
01 PERSONNEL-REC.
   03 NAME-FIELDS.
      05 LAST-NAME.
         07 LN1ST.      PIC X.
         07 LN2ND.      PIC X.
         ...
         07 LN25TH.     PIC X.
      05 FIRST-NAME     PIC X(25).
      05 M-I-NAME       PIC X.
   03 DEPARTMENT-NUMBER PIC 9999.
   03 DN REDEFINES DEPARTMENT-NUMBER.
      05 DN3            PIC 999.
      05 FILLER         PIC X.
   03 JOB-TITLE.
      05 J-T-PORTION.
         07 J-T         PIC X OCCURS 1 TO 25 TIMES
                              DEPENDING ON J-T-LENGTH.
   66 LN5  RENAMES LN1ST THRU LN5TH.
   66 LN10 RENAMES LN1ST THRU LN10TH.
   66 LN15 RENAMES LN1ST THRU LN15TH.
```

After you have established the key of reference, you can execute sequential READ statements (READ NEXT statements) to read successive records according to that key: records in a relative file in ascending record-number order, records in an indexed file in ascending prime-key order, or records in any structured (sequential) file in ascending alternate-key order. You can also execute sequential READ REVERSED statements to read records in reverse order.

The key-of-reference concept enables you to select one of several possible keys and then use a single sequential READ statement to read a subset of records in a file.

Suppose that you have the personnel record in Example 29-10, and that the employee number (which is not even part of the record) is the record number and therefore the prime key. By selecting an alternate key as the key of reference, you can start reading at the first "Adams" in the file, the first member of department 3141, or the first person having the job "Clerk." Then you can use a single READ statement to read successive records for employees named "Adams," successive records for employees whose names follow "Adams" alphabetically, records of employees in the department number 3141 or higher, or records of employees whose job title is "Clerk" or follows "Clerk" in collating sequence order.

Uppercase and lowercase letters are not equivalent in keys. When a data item with a mixed-case value is to serve as a key, you have these choices:

- Leave the value alone, and accommodate any case differences.
- Shift the value into uppercase in the existing data item.
- Create an uppercase copy of the value in another data item in the record and use the uppercase copy for key operations.

Alternate Record Keys

An alternate record key (or alternate key) is a data item, other than the prime record key, whose value identifies a record in a structured file. COBOL restricts alternate keys to indexed files. HP COBOL also allows alternate keys in entry-sequenced (sequential) and relative files.

In HP COBOL, an alternate key can occur anywhere in a record. The DUPLICATES phrase of the ALTERNATE RECORD KEY clause of the file-control entry determines whether alternate key values must be unique. If unique, an alternate key can have up to 253 characters; otherwise, it can have up to 253 characters minus the length of the prime key. Alternate keys can overlap both each other and the prime key, but no two alternate keys can start at the same character position (offset) in the record.

Enscribe implements alternate keys using one or more alternate-key files. For each file with one or more alternate keys (primary file), Enscribe maintains at least one alternate-key file. For each unique alternate key with a different key length, Enscribe maintains a separate alternate-key file. You can refer to multiple unique alternate keys of the same key length through a single alternate-key file. The directory entry for the primary file with alternate keys includes the names and other attributes of the associated alternate-key files.

Each record of an alternate-key file has three fields: the alternate key specifier, the value of the alternate key, and the value of the prime key. An alternate key specifier is a two-character code that identifies an alternate key defined for a primary file.

An alternate-key file is a key-sequenced file. For every record in the primary file, there are at most as many records in the alternate-key file as there are alternate keys declared for that alternate-key file.

Using FUP CREATE, you can specify a null-value character for each alternate key. Any record with an alternate-key field composed entirely of the null-value character is not represented in the alternate-key file. This strategy saves space when you have alternate keys for which many records have the same value, such as zeros or spaces, and you are not interested in locating these records by their alternate key.

If you have a structured file without alternate keys and you want it to have alternate keys, you can use FUP to create the alternate-key file and to load the alternate-key file.

If you describe your database with DDL, the DDL compiler can produce the necessary FUP commands to create prime-key and alternate-key files.

For more information about alternate keys, see the *Guardian Programmer's Guide*.

CAUTION: Because updating the alternate-key file can require multiple write operations, certain types of failures can cause the operating environment to fail to record alternate keys. The record and some of its alternate keys might be updated while other alternate keys might not be updated. To prevent this, use TMF to verify that an update to a file with alternate keys is either completed or aborted. For information about TMF, see Chapter 33: Fault-Tolerant Processes (page 963).

Positioning

Enscribe provides three positioning modes: approximate, exact, and generic. Only approximate positioning is available in COBOL; both approximate and generic positioning are available in HP COBOL.

Neither COBOL nor HP COBOL supports exact positioning; it involves a variable-length key. In using Enscribe directly, you specify a maximum key length for a key-sequenced file's prime key or for any alternate key. When you call the KEYPOSITION routine (analogous to HP COBOL's START statement), you can specify a length and a value: a compare length that is shorter than or equal to the maximum key length, and a value of that length for the key. Exact positioning means that the only records delivered are those whose key is the same length as the compare length and whose value is the same as the key value. If the key is unique, at most one such record exists in the file.

Table 29-5 Enscribe File Positioning Modes

Positioning Mode	Starting Position Defined by	End of File Is Signaled	Available in HP COBOL?
Exact Positioning	Whole key value	When key no longer identical to start key	No
Approximate Positioning	Whole or partial key value	At physical end of file	Yes
Generic Positioning	Whole or partial key value	When key or partial key no longer identical to start key	Yes

Approximate Positioning

With approximate positioning, you execute the START statement with a starting key, and the next READ statement retrieves the next record in the access path that contains a value in the corresponding data item that bears the stated relationship (equal to, greater than, or not less than) to the starting key. Subsequent READ statements receive the records that follow in the access path. An EOF condition arises when the actual end of the file is reached.

Approximate positioning is the default, but you can include an APPROXIMATE phrase in the START statement for documentation purposes if necessary.

Generic Positioning

Generic positioning is like approximate positioning except that you specify a starting key that (usually) has a shorter length than the key data item in the record. Also, the only relation permitted with the key is the equal relation. Subsequent READ statements retrieve the records in the access path whose key values match the starting key for as many characters as the start key has.

Suppose you want to read only the Adamses in the personnel file described in Key of Reference. With the GENERIC phrase of the START statement (an HP extension to COBOL), you specify that when a READ NEXT statement obtains a record with a LAST-NAME value that does not begin with the five characters "Adams," the statement must raise the EOF condition and not deliver that record to the program.

You usually use generic positioning with partial keys. Using key values that include trailing spaces, you can achieve an effect similar (but not identical) to exact positioning. Consider the case of “Adams.” If you want to find only the Adamses (but not any Adamsons), you can use a complete key of “Adams” plus enough spaces to fill out the key.

Suppose that an HP COBOL program includes the code in Example 29-11.

Example 29-11 Generic Positioning

```
INPUT-OUTPUT SECTION.
  FILE-CONTROL.
    SELECT NAME-FILE ASSIGN TO "$JUICE.APPLE.NAME"
    ORGANIZATION IS INDEXED
    ACCESS MODE IS DYNAMIC
    RECORD KEY IS KEY-NUMBER
    ALTERNATE RECORD KEY IS KEY-NAME WITH DUPLICATES
...
DATA DIVISION.
  FILE SECTION.
    FD NAME-FILE
       RECORD CONTAINS 5 TO 100 CHARACTERS
       LABEL RECORDS ARE OMITTED.
    01 NAME-FILE-REC.
       02 KEY-NUMBER PIC X(5).
       03 FILLER     PIC X(3).
       02 KEY-NAME   PIC X(35).
       02 REDEF-KEY REDEFINES KEY-NAME.
          04 FIRST-5 PIC X(5).
          04 FILLER  PIC X(30).
...
 PROCEDURE DIVISION.
   THE-TOP.
       OPEN I-O NAME-FILE.
       PERFORM SHOW-NAMES.
       STOP RUN.
    SHOW-NAMES.
       DISPLAY "SHALL I START THE FILE GENERIC?".
       ACCEPT WHAT-FILE.
       IF WHAT-FILE = "Y"
          DISPLAY "WHAT FIRST 5 LETTERS SHALL I USE?"
          ACCEPT FIRST-5
          START NAME-FILE KEY = FIRST-5 GENERIC
       ELSE
          DISPLAY "WHAT APPROXIMATE NAME SHALL I START AT?"
          ACCEPT KEY-NAME
          START NAME-FILE KEY = KEY-NAME APPROXIMATE.
       PERFORM GET-AND-DISPLAY UNTIL NOT NAME-FILE-OK.
    GET-AND-DISPLAY.
       READ NAME-FILE NEXT;
          AT END MOVE "99999-----EOF" TO NAME-FILE-REC.
       DISPLAY KEY-NUMBER, "---", KEY-NAME.
```

Suppose that NAME-FILE contains these records:

```
00005   ADAMS
00010   ADAMSKI
00121   JOHNSON
01010   ADAMS
12532   REITWIESNER
43132   SMITH
52353   ROTH
54116   ADAMS-JONES
54396   ADAMSSOHN
```

Without the GENERIC phrase, if the first key is “ADAMS” followed by 30 spaces, the program displays:

```
00005---ADAMS
01010---ADAMS
54116---ADAMS-JONES
00010---ADAMSKI
54396---ADAMSSOHN
00121---JOHNSON
12532---REITWIESNER
52353---ROTH
43132---SMITH
99999-----EOF
```

With the GENERIC phrase, if the first key specifies that its first five characters must be "ADAMS," the program displays:

```
00005---ADAMS
01010---ADAMS
54116---ADAMS-JONES
00010---ADAMSKI
54396---ADAMSSOHN
99999-----EOF
```

Repositioning to New Record With Same Alternate Key

COBOL provides no mechanism for repositioning to one of several records having the same alternate key. HP COBOL provides a mechanism: the POSITION phrase of the START statement.

The POSITION phrase enables you to write context-free servers that must return large sets of records to their requesters. Suppose you want your server to return one record for each product that includes part number 1345-55433. To keep message length and buffer space to a reasonable size, the server can return only 10 records. A server should not retain a context from a requester. Each server should be able to serve any requester.

The POSITION phrase can specify the prime key, a leftmost subordinate of the prime key, an alternate key, or a leftmost subordinate of an alternate key in the KEY phrase. The prime-key value is in a separate data item; it need not be inserted in the record area of the file.

The POSITION phrase enables a server to pass the value of the prime key of the most recently processed record back to a requester. The requester can then send the prime-key value to a server with its next request, and the server can use that value in a POSITION phrase of the START statement (and also as the alternate key in the KEY phrase) to uniquely specify a record at which to resume processing.

The POSITION phrase includes the optional keyword, AFTER. Without AFTER, the START statement establishes the current file position at the same record specified by the prime-key value. With AFTER, the START statement establishes the current file position at the next record (if any) after the record specified by the prime-key value in the POSITION phrase.

The role of the POSITION phrase is to reestablish the environment after the last read using a partial key. The execution of a START statement with a POSITION phrase does not report an invalid-key condition—the next READ statement executed could report an EOF condition.

To resume processing a file after an interruption, take the same START statement that you used to initiate processing and add a POSITION phrase, specifying the key relation EQUAL.

It might be possible to collapse the initial START statement and the repositioning START statement into a single statement when it is not important for the initial positioning operation to detect an invalid-key condition if no record matches the key relation in a normal initial start operation. If this is acceptable, write the program for just the continuation repositioning case and use LOW-VALUES for the prime record key (or 1 for the relative key) in the initial case. This strategy is especially useful for servers. The server need only be implemented with a continuation request and can depend on the requester to supply LOW-VALUES (or 1) for the prime key in the initial request.

Example 29-12 is a skeleton program for a server. It accepts several types of request, one of which calls for a list of part records (beginning at a specified part name) to be returned to the requester.

If more than 10 such records must be returned, the requester must accept 10 at a time and request the next portion by passing back the last part number it received and the part name. This program is incomplete and is provided only to illustrate the POSITION phrase of the START statement.

Example 29-12 Use of START With the POSITION Phrase

```
IDENTIFICATION DIVISION.
  PROGRAM-ID. PART-LIST-SERVER.
  AUTHOR.  KELLY COBOL.
ENVIRONMENT DIVISION.
  CONFIGURATION SECTION.
    SOURCE-COMPUTER. TXP.
    OBJECT-COMPUTER. TXP.
  INPUT-OUTPUT SECTION.
    FILE-CONTROL.
      SELECT MESSAGE-IN
         ASSIGN TO "$RECEIVE"
         FILE STATUS IS RECEIVE-FILE-STATUS.
      SELECT MESSAGE-OUT
         ASSIGN TO "$RECEIVE"
         FILE STATUS IS RECEIVE-FILE-STATUS.
      SELECT PART-FILE
         ASSIGN TO "PART"
         ORGANIZATION IS INDEXED
         ACCESS MODE IS DYNAMIC
         RECORD KEY IS PART-NO OF PART-DATA-RECORD
         ALTERNATE RECORD KEY IS
           PART-NAME OF PART-DATA-RECORD WITH DUPLICATES
         FILE STATUS IS FILE-STAT.
    RECEIVE-CONTROL.
      TABLE OCCURS 5 TIMES
      SYNCDEPTH LIMIT IS 1
      REPLY CONTAINS 1100 CHARACTERS.
DATA DIVISION.
   FILE SECTION.
   FD  MESSAGE-IN
       LABEL RECORDS ARE OMITTED
       RECORD CONTAINS 1 TO 200 CHARACTERS.
    01  PART-DEL-GET-LIST-MSG.
       02 PRT-HEADER.
          03 REPLY-CODE                  PIC S9(4)      COMP.
          03 APPLICATION-CODE            PIC X(2).
          03 FUNCTION-CODE               PIC X(02).
          03 TRANS-CODE                  PIC 9(2).
          03 TERM-NO                     PIC X(15).
          03 LOG-REQUEST                 PIC X(01).
       02 PART-NO                        PIC X(10).
       02 PART-NAME                      PIC X(50).
   FD  MESSAGE-OUT
       LABEL RECORDS ARE OMITTED
       RECORD CONTAINS 1 TO 1100 CHARACTERS.
  01  SERVER-REPLY.
       02 PRT-HEADER.
          03 REPLY-CODE                  PIC S9(4)      COMP.
           ... {same as PRT-HEADER of PART-DEL-GET-LIST-MSG }
       02 ERROR-CODE                     PIC S9(4)      COMP.
       02 GUARDIAN-ERR                   PIC S9(4)      COMP.
       02 ERROR-MESSAGE.
          03 ERROR-FILE-NO               PIC X(05).
          03 ERROR-TEXT                  PIC X(65).
    01  PART-GET-REPLY.
       02 PRT-HEADER.
          03 REPLY-CODE                  PIC S9(4)      COMP.
          ... {same as PRT-HEADER of PART-DEL-GET-LIST-MSG }
       02 PART-DATA.
          03 PART.
            04 PART-NO                          PIC X(10).
            04 PART-NAME                        PIC X(50).
       ...
1  PART-LIST-REPLY.
      02 PRT-HEADER.
         03 REPLY-CODE                         PIC S9(4)      COMP.
         ... {same as PRT-HEADER of PART-DEL-GET-LIST-MSG }
      02 LINE-COUNT                            PIC 9(04).
      02 ALL-PART-LINES.
         03 PART-LINE                     OCCURS 10 TIMES.
            04 PART-NO                         PIC X(10).
            04 PART-NAME                       PIC X(50).
            04 CUST-NUM                        PIC 9(06).
            ...
0  FD  PART-FILE
      LABEL RECORDS ARE OMITTED.
  01  PART-DATA-RECORD.
      02 PART.
         03 PART-NO                            PIC X(10).
         03 PART-NAME                          PIC X(50).
         ...
  WORKING-STORAGE SECTION.
   ...
PROCEDURE DIVISION.
MAIN SECTION.
BEGIN-COBOL-SERVER.
   PERFORM A-INIT.
   PERFORM B-TRANS UNTIL CLOSE-FROM-REQUESTER.
   PERFORM C-EOJ.
   STOP RUN.
A-INIT.
   OPEN INPUT  MESSAGE-IN
        OUTPUT MESSAGE-OUT        SYNCDEPTH 1
        INPUT  PART-FILE SHARED.
C-EOJ.
   CLOSE MESSAGE-IN
         MESSAGE-OUT
         PART-FILE.
B-TRANS.
   MOVE SPACES TO PART-DEL-GET-LIST-MSG.
   PERFORM 920-GET-SERVER-MSG.
   IF NOT CLOSE-FROM-REQUESTER
      PERFORM B-TRANS-CASE.
B-TRANS-CASE.
   IF TRANS-CODE OF PART-DEL-GET-LIST-MSG = LIST-BY-NO-TRANS
                                         OR LIST-BY-NAME-TRANS
      PERFORM 200-GET-PART-LIST-RCD
   ELSE
      PERFORM 960-INVALID-T-CODE
   .
100-GET-PART-RCD.
   PERFORM 110-INIT-REPLY.
   PERFORM 120-READ-PART-INFO.
110-INIT-REPLY.
  ...
120-READ-PART-INFO.
   MOVE PART-NO OF PART-DEL-GET-LIST-MSG TO PART-NO OF PART-DATA-RECORD.
   PERFORM 945-READ-PART-KEY.
   IF NO-ERROR
         PERFORM 914-GET-REPLY
   ELSE
         PERFORM 990-BUILD-ERROR-REPLY
         PERFORM 910-SERVER-REPLY.
```

```
200-GET-PART-LIST-RCD.
   PERFORM 210-INIT-REPLY.
   PERFORM 220-BUILD-REPLY.
210-INIT-REPLY.
   MOVE SPACES TO SERVER-REPLY.
   MOVE PRT-HEADER OF PART-DEL-GET-LIST-MSG TO PRT-HEADER OF SERVER-REPLY.
   MOVE ZERO TO REPLY-CODE OF PART-LIST-REPLY
                LINE-COUNT OF PART-LIST-REPLY
   MOVE SPACES TO ALL-PART-LINES OF PART-LIST-REPLY.
220-BUILD-REPLY.
   MOVE PART-NO   OF PART-DEL-GET-LIST-MSG TO PART-NO   OF PART-DATA-RECORD.
   MOVE PART-NAME OF PART-DEL-GET-LIST-MSG TO PART-NAME OF PART-DATA-RECORD.
   PERFORM 935-START-PART-REPOSITIONED.
   IF NO-ERROR
      PERFORM 230-GET-PART VARYING I FROM 1 BY 1
              UNTIL I > MAXLIST OR FILE-ERROR
      IF NO-ERROR OR END-OF-FILE
         PERFORM 916-LIST-REPLY
      ELSE
         PERFORM 990-BUILD-ERROR-REPLY
         PERFORM 910-SERVER-REPLY
   ELSE
      PERFORM 990-BUILD-ERROR-REPLY
      PERFORM 910-SERVER-REPLY.
230-GET-PART.
   MOVE I TO LINE-COUNT OF PART-LIST-REPLY.
   PERFORM 940-READ-PART-NEXT.
   IF NO-ERROR
      MOVE PART-NO   OF PART-DATA-RECORD TO PART-NO   OF PART-LIST-REPLY (I)
      MOVE PART-NAME OF PART-DATA-RECORD TO PART-NAME OF PART-LIST-REPLY (I)
      ...
   ELSE
      IF END-OF-FILE
         MOVE PART-FILE-EOF TO REPLY-CODE OF PART-LIST-REPLY
         SUBTRACT 1 FROM LINE-COUNT OF PART-LIST-REPLY.
910-SERVER-REPLY.
    WRITE SERVER-REPLY.
914-GET-REPLY.
    WRITE PART-GET-REPLY.
916-LIST-REPLY.
    WRITE PART-LIST-REPLY.
935-START-PART-REPOSITIONED.
    START PART-FILE KEY IS = PART-NAME OF PART-DATA-RECORD
          POSITION PART-NO OF PART-DATA-RECORD APPROXIMATE.
960-INVALID-T-CODE.
    MOVE INVALID-TRANSACTION TO REPLY-CODE OF SERVER-REPLY.
    PERFORM 910-SERVER-REPLY.
```

Optimizing Disk File Processing

The features of Enscribe that protect the database from corruption also slow down the Enscribe input and output routines. If you can afford less protection, you can improve the performance of the HP COBOL disk input and output routines. You can afford less protection when a process has exclusive access to a file, or when it is acceptable that other processes that read the file might not receive the most recent copy of a changing record.

These topics explain how to use these types of files to optimize disk file processing:

- Unstructured Files
- Structured Files
- Files With Alternate Keys
- Partitioned Files

Unstructured Files

A process can read and write unstructured files faster than it can read structured files, whether the logical records of the unstructured file are blocked or not. A process can read and write blocked unstructured files considerably faster than it can read and write unblocked unstructured files.

HP COBOL performs record blocking (for write operations) and deblocking (for read operations) of unstructured files when all of these conditions are met:

- The program describes the file's organization as sequential.
- The program does not describe the file as being composed of variable-length records (except in the case of EDIT files, which can be described as having variable-length records).
- The program includes a BLOCK CONTAINS clause that specifies an even multiple of the established record size.
- The program does not describe the file with a LINAGE clause or an ALTERNATE RECORD KEY clause.

Structured Files

For structured files, the file system provides two independent record blocking performance enhancements—cache buffering and sequential block buffering. In addition, HP COBOL has Fast I-O, a faster enhancement built on top of sequential block buffering.

The type of record blocking used for a file is determined by *number* in the RESERVE clause of the file-control entry.

The value of *number* must be in the range 1 through 32 and is interpreted:

Value of number	Effect on Record Blocking
1	No buffering or HP COBOL Fast I-O
2	Sequential block buffering on input and buffered cache on output if the assigned file qualifies
3 or greater	HP COBOL Fast I-O if the assigned file qualifies; if not, sequential block buffering for input and buffered cache for output if the assigned file qualifies; otherwise, normal I-O*number* is the number of blocks to buffer

The BLOCK CONTAINS clause has no effect when a RESERVE clause is present—the block size of the existing file is used.

Cache Buffering

Cache buffering is a disk-process feature that speeds up the writing of structured disk files. A cache, or buffer pool, is an area of system memory reserved for buffering blocks of data for transfer to or from a disk.

If cache buffering is not used, each logical write operation causes the cache block containing the record to be written immediately to the disk. This is called write-through cache.

If cache buffering is used, the records are held in the cache (in system memory) and not immediately written through to the disk. The blocks are written to disk only when certain situations require it. This is called buffered cache.

By default, files that are audited by TMF use cache buffering, and files that are not audited do not.

CAUTION: Do not use cache buffering with a program that requires that each record actually be written on the disk before the next statement in the program is executed. If your program is writing to a file that is opened with the exclusion mode SHARED or PROTECTED, give careful consideration to the use of cache buffering, because other processes can read the file and could read a record that your writing process has updated but which the file system has not yet delivered to the disk.

When a file is opened, the call to the FILE_OPEN_ procedure includes a parameter that determines whether or not cache buffering is used—that is, whether the file has the attribute BUFFERED. You can give a file the BUFFERED attribute by using FUP routines or the corresponding Enscribe routines through environment procedure calls; then all output to the file is done with cache buffering.

Each disk has its own cache. Each cache is configured to have space for a certain quantity of disk data. Each disk device has an associated disk process that performs the physical operations of reading data from the disk into cache memory and writing data from cache memory to the disk.

When a process requests a record from a given disk, the Enscribe record manager checks that particular cache for the block that contains the record. If that block is not present in the cache, Enscribe must perform a physical read from the disk and then return the record to the application process. If the block is present, Enscribe does not need to do the read operation; it can simply return the record to the application process.

Sequential Block Buffering

Sequential block buffering (SBB) is an Enscribe feature that speeds the sequential reading of a structured file by reading a block of records together into a memory buffer. Enscribe allocates the buffer in the process file segment outside the data space of your process.

NOTE: With the advent of the DP2 disk processing system, “normal” I/O could be faster than sequential block buffering, depending on the number of records per block. For example, a file containing eight 4K blocks and 1600 records requires one physical I/O operation and 1600 interprocess messages in normal mode, but eight physical I/O operations and only eight interprocess messages in SBB mode; on the other hand, a file with eight 4K blocks and only eight records requires one physical I/O and eight interprocess messages in normal mode, but eight physical I/O operations and eight interprocess messages in SBB mode

If the file has no alternate keys, the size of the buffer is the block size of the file. If the file has alternate keys, the alternate-key file (not the primary file) is buffered, because the alternate-key file is being read sequentially, while the primary file is probably being read randomly.

Without sequential block buffering, every time an application program performs a read operation, the HP COBOL run-time routines must call the READ routine, which causes the file system to send a message to the disk process and causes the disk process to return a message containing the record. With sequential block buffering, only entire blocks are sent, and any records in the block after the first one are delivered from the process file segment without any message traffic.

HP COBOL provides sequential block buffering for a file when all of these conditions are met:

- The file is structured.
- The file's open mode is INPUT or I-O.
- The program declares the file to have sequential access mode (although organization can be sequential, relative, or indexed).

CAUTION: Do not use sequential block buffering for a file opened for shared access. If you do, a process could read data that is not up to date while another process alters the file.

HP COBOL Fast I-O

Fast I-O is an HP COBOL enhancement in input-output performance beyond that of sequential block buffering or buffered cache. With HP COBOL Fast I-O, the HP COBOL run-time routines use an auxiliary block buffer and perform the blocking and deblocking in local storage. This process operates 2 or 3 times faster than when the operating environment performs the blocking and deblocking operations.

HP COBOL Fast I-O is available if the HP COBOL program and the structured file upon which it is to operate meet these criteria:

- The file description includes a RESERVE clause with a *`number`* specifying the number of blocks to buffer. The *`number`* must be greater than 2.
- The file description does not include a LINAGE clause or a CODE-SET clause.
- The file was not opened with time limits (as with the TIME LIMITS phrase in the OPEN statement).
- The program is not compiled with the NONSTOP directive.
- The specifications in the OPEN statement, or the attributes derived during the open operation by some other means (such as from a TACL ASSIGN command), conform to:
  - The open mode is INPUT or OUTPUT.
  - If the open mode is INPUT, the exclusion mode is PROTECTED or EXCLUSIVE.
  - If the open mode is OUTPUT, the exclusion mode is EXCLUSIVE.

NOTE: HP COBOL Fast I-O is not recommended for an indexed file that has had many records inserted or deleted since it was created. If you want to use HP COBOL Fast I-O on such a file, first use FUP RELOAD to reorganize it.

If all of the preceding qualifications are satisfied at run time, HP COBOL Fast I-O is used only if memory resources are available. For a file opened for output, the loader creates the file. The loader is also used by FUP, SQL/MP, and other products. The loader manages its own extended data segment to write the file. The loader requires local storage, which is allocated by the run-time system. The amount of local storage is approximately 800 bytes (the exact amount depends on the type of the file and other factors). If the necessary storage is not available, the OPEN statement returns file status "07" and buffered cache is used to write the file.

For a file opened for input, HP COBOL Fast I-O allocates space for double buffering from a buffer pool that is in its own extended data segment and is of a fixed size. Buffer size is determined by multiplying the number in the RESERVE clause by the data block size of the file (to see this size, use the command `FUP INFO file-name, DETAIL`). Then, based on various factors such as the maximum transfer allowed, a read buffer size that is not larger than 30,720 bytes is selected. The total space allocated from the buffer pool is twice the read buffer size. If there is not enough space (as when several files are opened for input using HP COBOL Fast I-O at the same time), the OPEN statement returns file status "07" and sequential block buffering is used for the file.

The buffer pool space returns to the pool when the file is closed. The SAME AREA clause has no effect on allocation or deallocation from this buffer pool.

If the file is being created and the maximum file size is exceeded, the corrupt bit is set in the file header and the program terminates abnormally. Attempting to open the file causes a file system error. If the file is sequential, you can use FUP to clear the corrupt bit and read what was written to the file (all of the records you released with WRITE will probably not be there). If the file is relative or indexed, the file is unusable.

Files With Alternate Keys

You can improve the performance of alternate-key file access by:

- Declaring Null Values for Alternate Keys
- Making the Main File Entry-Sequenced or Relative
- Not Updating the Alternate-Key File Automatically

Declaring Null Values for Alternate Keys

When you create an alternate-key file with FUP CREATE, you can declare a null value for any alternate key. When you insert a record, if each byte in the alternate-key field contains that null value, the alternate-key reference is not added to the alternate-key file. This reduces both the size of the alternate-key file and the access time for the records of a file being accessed according to that alternate key. It also makes the records that have null values invisible when you read the file by an alternate key for which a null value has been defined.

If you update a record in the file in such a way that the alternate-key field receives a null value, the alternate-key reference for that record is deleted from the alternate-key file.

If you read a file according to an alternate key for which a null value has been defined, records containing the null value of the alternate key are not read.

The most common null values are the ASCII space and the binary zero.

Making the Main File Entry-Sequenced or Relative

If your main file has several keys defined but no one key is used substantially more than any other keys, consider making that main file a relative or entry-sequenced file rather than a key-sequenced file.

Every time you read a key-sequenced file by prime key, you cause two possible physical read operations: one for the key and one for the record. Every time you read a key-sequenced file by alternate key, you cause four possible physical read operations: one for the alternate key, one for the alternate-key record, one for the prime key, and one for the record. If the main file has two keys and the probability of reading the file by any given key is about even, then half the time you could cause two physical read operations and half the time you could cause four. For each logical read operation, you average three physical read operations.

If the main file has two keys and the probability of reading the file by any given key is about even, you also average three physical read operations for each logical read operation if you choose entry-sequenced or relative organization for the main file; however, if the main file has more than two keys, the average number of physical read operations per logical read operation improves.

The average number of physical read operations per logical read operation improves even further when several records in the alternate-key file or main file are in the same physical block and no physical read is required, such as when you are reading sequentially on an alternate key.

Not Updating the Alternate-Key File Automatically

Another way to improve the performance of alternate-key file access is to specify that the alternate-key file not be automatically updated by the operating environment when the value of an alternate-key field changes (use the FUP parameter NOUPDATE). This is appropriate when your only means of updating the file is a batch update program that runs when no interactive processes are reading the file; you can run the batch update, then re-create the alternate-key file and reload it. Do not use the NOUPDATE parameter if the file is updated interactively, because the changed records do not cause any changes in the alternate-key file until the next batch update.

Partitioned Files

Partitioned files can be accessed faster because separate read heads are used for each partition. Loss of access to one disk need not mean loss of access to the entire file.

You can divide an indexed file according to the value of the prime key and have, for example, `A` through `C` on the first disk volume and `U` through `Z` on the last disk volume. If one of the disks is taken offline for maintenance, or if one system of the network is inaccessible due to communication problems, the remainder of the partitioned file is still accessible.

To open a partitioned file when some partitions cannot be accessed, an HP COBOL program must use the routine COBOL_SPECIAL_OPEN_. If all partitions can be accessed, an HP COBOL program can open a partitioned file with the OPEN statement.

For more information about partitioned files, see Partitioned Files.

Optimizing Disk File Storage

You can specify at file creation time that an indexed or data file is to be compressed for more space-efficient storage at a slight expense of speed. Enscribe compresses records by eliminating duplicate leading characters from one record to the next and replacing them with a one-byte count of the duplicate characters. If a file has a significant amount of duplicated data, the storage saving can be substantial.

The prime key of a compressed file must be at the beginning of the record.

Compression is recommended for an indexed file in which the first records of successive blocks have similar prime-key values. Compression is recommended for an alternate-key file in which several alternate keys have the same value.

For more information about compressing indexed and data files, see the *Guardian Programmer's Guide*.

Avoiding Deadlock

Deadlock is a situation in which two processes are in contention for a single-user resource. If both processes require exclusive use of both resources, they are deadlocked. Neither can proceed until the other surrenders control of the resource it has under its control. If two servers get into a deadlock, their respective requesters are also deadlocked because neither requester receives a reply from its server.

One example of deadlock is when one process has opened a file (call it ABLE) and needs to open another file (call it BAKER), while another process has opened BAKER and now needs to open ABLE.

Another example of deadlock is when one process has locked a record in a file (call it record `n`) and wants to read another record in the file (call it record `m`), while some other process has locked record `m` and wants to read record `n`.

The ways to avoid deadlock are:

- Locking and Unlocking Files and Records
- Setting Time Limits on Input-Output Operations

Locking and Unlocking Files and Records

The most complete way for a process to lock a file is to open the file for exclusive access (open it with the exclusion mode EXCLUSIVE). This strategy prevents all other processes from reading or writing the file.

If it is not feasible for one process to have exclusive control of a file for when it has the file open, there are alternatives.

One alternative is protected access. When a process opens a file for protected access (opens it with the exclusion mode PROTECTED), other processes can read and write the file while the opening process has it open, but cannot write the file until the opening process closes the file.

Another alternative is shared access. When a process opens a file for shared access (opens it with the exclusion mode SHARED), other processes can read and write the file while the opening process has it open.

When a file is open for shared access and one process needs exclusive access to the file temporarily, that process can lock the file with a LOCKFILE statement, operate on it, and then unlock it with an UNLOCKFILE statement. If a process only needs temporary exclusive access to a record of the file, the process can temporarily lock the record with a READ statement with a LOCK phrase and then unlock it with an UNLOCKRECORD statement.

To avoid deadlock when processes share and lock files, have each process lock and unlock records and files in the same sequence.

NOTE: If a process is protected by TMF, all locks are retained (despite unlocking statements) until the current transaction is completed or backed out.

Setting Time Limits on Input-Output Operations

Even when each program that shares files carefully arranges to lock files and records in the same order, a newly introduced program or a change to an existing program can fail to conform and cause deadlock. In HP COBOL, you can prevent this deadlock by using the TIME LIMITS phrase with the statements OPEN, START, READ, and LOCKFILE.

Without the TIME LIMITS phrase, if your process attempts to open a file, establish a starting position in a file, read a file, or lock a file and that file or the necessary record in that file is already locked, the process suspends activity until the lock is removed.

With the TIME LIMITS phrase, you can specify a time limit on the suspension. If the time expires before the inhibiting lock is removed, the file operation terminates with file status code “30” and GUARDIAN-ERR special register value 40. The program can diagnose the failure of the file operation and either retry the operation or abandon the current group of operations to allow another process to complete its operations.

Suppose you have a file declared by these entries:

```
SELECT OAK-FILE ASSIGN TO "$OAK.ACORN.TREE"
       ORGANIZATION IS SEQUENTIAL
       ACCESS MODE IS SEQUENTIAL
       FILE STATUS IS OAK-STATUS.
FD   OAK-FILE
     LABEL RECORDS ARE OMITTED.
01   OAK-RECORD.
     02 ...
```

Suppose that you open the file with this statement:

```
OPEN INPUT OAK-FILE WITH TIME LIMITS SHARED.
```

Suppose that your program includes a declaratives section such as:

```
DECLARATIVES.
  OAK-FILE-USE-SECTION.
    USE AFTER STANDARD ERROR PROCEDURE ON OAK-FILE.
  OAK-DECL.
  IF OAK-STATUS = "30"
     AND GUARDIAN-ERR = 40
     MOVE "Operation timed out on OAK file" TO REPORT
     MOVE TIME-OUT-CODE TO OAK-ERROR
     DISPLAY REPORT
```

```
   END-IF.
END DECLARATIVES.
```

Suppose that you attempt to read the file OAK with a statement such as:

```
READ OAK-FILE RECORD TIME LIMIT 5 AT END PERFORM END-OAK.
IF OAK-ERROR = TIME-OUT-CODE
...
```

When the read operation cannot be completed within 5 seconds, control passes to the USE procedure for the OAK file, which can adjust the program's behavior appropriately. The USE procedure displays a diagnostic to the OUT file. The program could retry a certain number of times, or (if it is a server) report back to its requester that the read operation could not be completed, which is another way to handle a file that might be locked at the file or record level.

Using Enscribe and Operating System Routines

Enscribe routines are a subset of operating system routines that function collectively as the operating environment's database record manager. All processes use Enscribe routines to interact with records in disk files in the database. HP COBOL run-time routines that perform input-output operations do so by calling Enscribe routines, but they also handle HP COBOL-related processing that Enscribe does not, such as declaratives and file status code data items.

An HP COBOL program can call an Enscribe or other operating system routine with the ENTER statement. Some Enscribe and operating system routines are not safe to use from an HP COBOL program; others are.

△ **CAUTION:** Generally, it is not safe to mix HP COBOL input-output operations and direct calls to Enscribe input-output routines for the same open file. Although you can use HP COBOL statements to open, close, and checkpoint a file and also call Enscribe routines to perform operations that HP COBOL does not provide, calling Enscribe routines disables useful HP COBOL tools such as declaratives and file-status data items, causing corruption or loss of data, abnormal termination, and other failures. Although such calls might work in one RVU of the software, they are not guaranteed to work in subsequent RVUs.

Physical File Names

A physical file name is the file name by which the Enscribe disk file handling routines recognize a file (as opposed to the COBOL file name, which is also called the internal file name).

In the non-CRE environment, Enscribe disk file handling routines demand file names in one of two rigid formats—one for local files and one for files on other nodes of the Expand network. They correspond to these HP COBOL declarations:

```
01 Enscribe-LOCAL-FILE-NAME.
   03 VOLUME-SPECIFIER.
      05 FILLER                 PICTURE X VALUE IS "$".
      05 VOLUME-NAME            PICTURE X(7).
   03 SUBVOLUME-NAME            PICTURE X(8).
   03 FILE-NAME                 PICTURE X(8).
01 Enscribe-NETWORK-FILE-NAME.
   03 SYSTEM-SPECIFIER.
      05 FILLER                 PICTURE X VALUE IS "\".
      05 ONE-BYTE-SYSTEM-NUMBER PICTURE X.
   03 VOLUME-SPECIFIER.
      05 VOLUME-NAME            PICTURE X(6).
   03 SUBVOLUME-NAME            PICTURE X(8).
   03 FILE-NAME                 PICTURE X(8).
```

The data item ONE-BYTE-SYSTEM-NUMBER is a 1-byte integer whose value is in the range 0 through 254. Each node in an Expand network has a unique system name and a unique system number. The number and name of a node are established during system generation.

Obtaining File Numbers and Other File Attributes

When a process opens a file, Enscribe assigns the file a number that is unique within that process. File number zero is always set aside for $RECEIVE, regardless of whether a process opens it or not. The first other file that a process opens is assigned file number one. Enscribe always assigns the lowest available file number. After a process closes a file, the associated file number becomes available.

Many Enscribe routines require file numbers instead of file names. Each of these routines returns a file number:

- HP COBOL routine COBOLFILEINFO or COBOL_FILE_INFO_
- Enscribe routine FILE_GETINFO_

Each of the preceding routines can return other file attributes in addition to the file number; use the routine best suited to your needs.

The HP COBOL routine COBOLFILEINFO or COBOL_FILE_INFO_ accepts a COBOL file name and returns:

- The file number of the file
- The internal form of the file name associated with the file description
- The file-system error code of the last input-output operation attempted for that file

For more information about the COBOLFILEINFO routine, see COBOLFILEINFO. For more information about the COBOL_FILE_INFO_ routine, see COBOL_FILE_INFO_.

The FILE_GETINFO_ procedure can return more attributes than the HP COBOL routine COBOLFILEINFO can. For details, see the *Guardian Procedure Calls Reference Manual*.

Determining Whether Two COBOL File Names Specify the Same Physical File

Through programmer error or a TACL ASSIGN command, two COBOL file names in the same run unit can refer to the same physical file. This situation is not forbidden; in fact, it can be useful: you can develop a server by having the input come from a disk file and the output go to a printer, and then putting the server into production with both the input file and the output file assigned to $RECEIVE.

Having two COBOL file names refer to the same physical file can also cause problems. If, for example, a program has two output files that are intended as two separate reports, and both are assigned to the same printer (not a spooler collector but the same mechanical device), then the program terminates abnormally. If both files have the exclusion mode SHARED, the lines of the two reports are interspersed on the paper.

To enable your HP COBOL program to detect that two of its HP COBOL file names refer to the same external file, call operating system routine FILENAME_COMPARE_. FILENAME_COMPARE_ accepts two file names in internal form and returns a numeric value:

Value Returned	Meaning
-1	The two internal file names do not refer to the same physical file.
0	The two internal file names refer to the same physical file.
1	The two internal file names refer to the same volume name, device name, or process name, but the remainder of the two file names do not match (such as two instances of the same spooler collector that specify different locations).

A negative value other than -1 represents a file-system error number having the corresponding positive value; for example, if at least one of the parameters specifies an illegal file name, the value returned is -13.

For more information about the FILENAME_COMPARE_ routine, see the *Guardian Procedure Calls Reference Manual*.

Purging a File From an HP COBOL Program

The HP COBOL language provides no mechanism for deleting a file from the system directory; an HP COBOL program can purge a file only through an explicit call to the FILE_PURGE_ procedure. The file being purged must be closed.

The FILE_PURGE_ procedure accepts the internal form of a file name as a parameter. To determine whether the purge operation was successful, call the HP COBOL routine COBOLFILEINFO with the file name as its parameter. COBOLFILEINFO returns an error value. If the purge operation was successful, the error value is zero.

One common reason for a purge operation to fail is that the process's accessor ID was not authorized to purge the file according to the security attributes of the file.

Example 29-13 Purging a File From an HP COBOL Program

```
WORKING-STORAGE SECTION.
01  FS-ERROR NATIVE-2
01  EXTERNAL-NAME  PIC X(30).
01  INTERNAL-NAME.
    03 VOL-NAME    PIC X(8).
    03 SUBVOL-NAME PIC X(8).
    03 FILE-NAME   PIC X(8).
01  PURGE-RESULT     PIC S9(4) COMP.
    88 PURGED-OK     VALUE ZERO.
...
PROCEDURE DIVISION.
...
MOVE "$FIFTY.NIFTY.SHIFTY" TO EXTERNAL-NAME.
MOVE "$FIFTY" TO VOL-NAME.
MOVE "NIFTY" TO SUBVOL-NAME.
MOVE "SHIFTY" TO FILE-NAME.
  ENTER "FILE_PURGE_" USING EXTERNAL-NAME (1:19)
    GIVING FS-ERROR.
  ENTER TAL "COBOLFILEINFO" USING INTERNAL-NAME PURGE-RESULT
  IF PURGED-OK
...
```

If the reason you want to purge a file is that you want to have a temporary file for the duration of a single process and the name of the file is not of any real interest, use the special name #TEMP in the ASSIGN clause of the file-control entry at compilation time (or in a TACL ASSIGN command at execution time). When you specify #TEMP, the run-time routines create a temporary file on the process's default volume.

You can also create a temporary file on a specific volume by just assigning the HP COBOL file to the volume name, specifying spaces. It is often useful to locate a temporary file on a different volume to improve performance.

The TACL ASSIGN command enables you to allocate specific primary and secondary extents for temporary and permanent disk files that a process is to create. If you do not specify extents, the run-time routines use a primary extent size of 4 disk pages (each page holds 2048 bytes) and a secondary extent size of 20 disk pages.

Unless the CLEARONPURGE flag is set for a file, purging the file does not obliterate the data on the disk. If security is a major consideration, use one of these to set the CLEARONPURGE flag:

- The FUP SECURE command with the CLEARONPURGE option
- The SETMODE routine

For more information about the FILE_PURGE_ and SETMODE routines, see the *Guardian Procedure Calls Reference Manual*.

Purging the Contents of a File From an HP COBOL Program

It is sometimes convenient to purge the contents of a file but keep the file in the directory for later use. The HP COBOL run-time routines do this when you open a file for output (if you have read access to the file). You might choose (for security) to purge the contents of a temporary work file when you have finished working with it, because merely purging the file does not erase the data from the disk.

The HP COBOL library routine COBOL_CONTROL_ is a multipurpose routine that calls the operating system routine CONTROL, and whose function is determined by an operation code. The operation code 20 enables a process to purge only the contents of a file if the process has write access to the file. The process can even purge the data from a temporary file whose name was assigned by Enscribe because you specified only a volume name or assigned it to #TEMP.

The COBOL_CONTROL_ routine call for purging data requires two parameters. The first parameter is the *`file-name`* described in the SELECT clause of the file-control entry for the file; the second is the number 20 (which causes the file system to purge the data in the file). Example:

```
ENTER "COBOL_CONTROL_" USING MYFILE 20
```

For more information about the CONTROL routine, see the *Guardian Procedure Calls Reference Manual*.

Renaming a File From an HP COBOL Program

The FILE_RENAME_ procedure enables a process to rename a closed disk file, provided the process has purge access to the file. If the file is a temporary file, the process can call FILE_RENAME_ to make the file a permanent file. For more information about the RENAME or FILE_RENAME_ routine, see the *Guardian Procedure Calls Reference Manual*.

Creating a File Having Alternate Keys From an HP COBOL Program

At times, it is necessary within an HP COBOL process to create and operate on a file that has alternate keys. Suppose the file must be named and built during execution. The only way to do this is through a series of steps involving a few Enscribe routines:

1. Declare the records of the file in a file description entry.
2. Use the FILE_CREATELIST_ procedure to create the primary file in conformance with the specification in the file-control entry and the file description.
3. Use the FILE_CREATELIST_ procedure to create the alternate-key file in conformance with the specification of the keys in the file-control entry and the record description for the file.
4. Open the loadfile and use it.

Suppose you have a server process that must create a private, indexed file having one alternate key.

NOTE: This explanation does not elaborate all the data definitions and all the code necessary to manage the creation of an indexed file with an alternate key. Only the significant declarations and code are shown. In some cases, the code mentions data items that are presumed to have been defined and given values by statements not shown.

The file-control entry and file description entry are:

```
SELECT T-file-1 ASSIGN TO "ISFILE"
        ORGANIZATION IS INDEXED
        RECORD KEY IS K1
        ALTERNATE RECORD KEY IS AK1
        ACCESS MODE IS DYNAMIC
        FILE STATUS IS Fs
...
FD  T-file-1 LABEL RECORDS OMITTED.
```

```
 01  T-rec-1.
     02  K1.
         03  K1-fp  PIC XX.
         03  FILLER PIC X.
     02  Ak1.
         03  Fp  PIC X(5).
         03  Lp  PIC X(5).
     02  FILLER  PIC X(20).
     02  rw-flag PIC X(4).
     02  T-rec-no  PIC 999.
...
```

Declare parameters to pass to the FILE_CREATELIST_ procedure to create the indexed file (see the *Guardian Procedure Calls Reference Manual* for details on these parameters):

- The file type (key-sequenced)
- The record length
- The prime record key parameters (key length = 16, key offset = 0, and index block length = default):

```
01  Create-error   NATIVE-2.
01  Error-item     NATIVE-2.
01  File-name      PIC X(128) VALUE "ISFILE".
01  File-name-len  PIC 999 COMP VALUE 6.
01  Fs             PIC XX.
01  Fp-chg         PIC S9 COMP  VALUE -1.

01 Item-list.
   02 File-type NATIVE-2 VALUE 41.
   02 Lrl NATIVE-2 VALUE 43.
   02 Key-offset NATIVE-2 VALUE 45.
   02 Key-length NATIVE-2 VALUE 46.
   02 Number-keys NATIVE-2 VALUE 100.
   02 Key-descriptor NATIVE-2 VALUE 101.
   02 Num-ak-files NATIVE-2 VALUE 102.
   02 Ak-file-name-len NATIVE-2 VALUE 103.
   02 Ak-file-name NATIVE-2 VALUE 104.

01 Number-of-items NATIVE-2 VALUE 9.

01 Values-array.
   02 File-type-v NATIVE-2 VALUE 3.
   02 Lrl-v NATIVE-2.
   02 Key-offset-v NATIVE-2 VALUE 0.
   02 Key-length-v NATIVE-2.
   02 Number-keys-v NATIVE-2 VALUE 1.
   02 Key-descriptor-v.
      03 A-key-spec PIC XX VALUE "AA".
      03 A-key-len NATIVE-2.
      03 A-key-off NATIVE-2.
      03 A-key-filenum NATIVE-2 VALUE 0.
      03 A-null-value NATIVE-2 VALUE 0.
      03 A-attributes NATIVE-2 VALUE H"4000".
   02 Num-ak-files-v NATIVE-2 VALUE 1.
   02 Ak-file-name-len-v NATIVE-2 VALUE 7.
   02 Ak-file-name-v PIC X(8) VALUE "ISFILEA ".
01 Values-length NATIVE-2.
01 Akf-rec-len NATIVE-2.
01 Akf-key-len NATIVE-2.
01 Create-error NATIVE-2.
01 Error-item NATIVE-2.
01 File-name PIC X(128) VALUE "ISFILE".
01 File-name-len PIC 999 COMP VALUE 6.
01 Fs PIC XX.
01 Fp-chg PIC S9 COMP VALUE -1.
```

Example 29-14 Dynamic File Assignment

```
...
     MOVE FUNCTION LENGTH (T-rec-1) TO Lrl-v
     MOVE FUNCTION LENGTH (K1) TO Key-length-v, A-key-off
     MOVE FUNCTION LENGTH (AK1) TO A-key-len
     MOVE FUNCTION LENGTH (Values-array) TO Values-length
* Create the primary file
     ENTER "FILE_CREATELIST_" USING File-name,
                                    File-name-len,
                                    Item-list,
                                    Number-of-items,
                                    Values-array,
                                    Values-length,
                                    Error-item
         GIVING Create-error
     IF Create-error NOT = 0
         DISPLAY "Creation failed with error ", Create-error,
           " in parameter " Error-item
         STOP RUN
     END-IF
* Create the alternate key file
*    The key length has to be 2 greater than the real one
     COMPUTE Akf-key-len = 2 + A-Key-len
*    The record length has to be 5 greater than the real key length
     COMPUTE Akf-rec-len = 5 + A-Key-len
     ENTER "FILE_CREATE_" USING Ak-file-name-v,
                                Ak-file-name-len-v,
                                OMITTED,
                                OMITTED,
                                OMITTED,
                                OMITTED,
                                3,
                                OMITTED,
                                Akf-rec-len,
                                OMITTED,
                                Akf-key-len,
                                0
         GIVING Create-error
     IF Create-error NOT = 0
         DISPLAY "AK creation failed with error ", Create-error,
           " in parameter " Error-item
         STOP RUN
     END-IF
```

30 Terminal Input and Output

Most interaction between users and an HP system is conducted through video display terminals. The HP COBOL language has no mechanisms for full-screen block input-output. On an HP system, a typical application uses Pathway/TS to write requester programs in SCREEN COBOL to handle terminal input and output and uses HP COBOL or pTAL server programs to handle database input and output. Occasionally, an HP COBOL process must communicate with a terminal directly.

For very small amounts of data transfer and for free-format collection and reporting of numeric values, such processes can use the ACCEPT and DISPLAY statements. Each ACCEPT or DISPLAY statement either specifies a mnemonic name of a device or defaults to the process's IN or OUT file, respectively (only if the IN and OUT files are suitable for ACCEPT and DISPLAY statement use).

If a process needs uninterrupted access to a terminal or wants to determine at run time which of several terminals to communicate with, the process must declare the terminal process as a sequential file and operate on it with the usual OPEN, READ, WRITE, and CLOSE statements.

The system generation process includes operations to define each terminal attached to an HP system. Each terminal has a device name and a logical device number.

The file system supports transfers from terminals in both page mode and conversational mode. Conversational mode is easy to use from an HP COBOL process, but page mode is awkward. The normal way to use page-mode input and output in conjunction with an HP COBOL process is to write a SCREEN COBOL requester program and have it communicate with an HP COBOL server process.

Using ACCEPT and DISPLAY With a Terminal

ACCEPT and DISPLAY statements transfer small amounts of data between a device and a process. The device is not a file; it is neither mentioned in the HP COBOL program's FILE-CONTROL paragraph nor described in the program's File Section.

Whenever a process executes an ACCEPT or DISPLAY statement, the HP COBOL run-time routines automatically open the device, perform the operation, and close the device.

Any ACCEPT and DISPLAY statements that do not include a mnemonic name to specify a device interact with the standard input and output devices of their process, respectively.

Guardian Environment

In the Guardian environment, each process has an IN file, an OUT file, and a home terminal file. You can specify these files in the RUN command, using the IN, OUT, and TERM run options or with the PARAM EXECUTION-LOG command (see Chapter 26: Executing and Debugging HP COBOL Programs (page 835)). If you do not specify one or all of the files in the RUN command, the process inherits the corresponding file of the command interpreter that accepted the RUN command.

If you want a process's ACCEPT and DISPLAY statements to interact with a terminal other than that on which you issued the RUN command to start the process, you have two alternatives:

- Use the SPECIAL-NAMES paragraph to associate a mnemonic name with the other terminal and qualify the ACCEPT and DISPLAY statements with that name.
- Mention the other terminal in the RUN command IN and OUT options when you initiate the process.

If a process needs to choose among several terminals for input or output, the use of the mnemonic name requires that all device assignments be established at compile time; therefore, several ACCEPT or DISPLAY statements, each associated with a fixed device, must be present in the program, and the process must choose the ACCEPT or DISPLAY to use.

Example 30-1 shows the use of ACCEPT and DISPLAY statements with the IN file and the OUT file and with a terminal that is known to the file system as $TRM053.

Example 30-1 ACCEPT and DISPLAY Statements With a Terminal

```
 IDENTIFICATION DIVISION.
   PROGRAM-ID.    TERMINAL-ACCEPT-DISPLAY.
   AUTHOR.        MO COBOL.
   DATE-WRITTEN.  29 FEBRUARY 1984.
   DATE-COMPILED.
******************************************************
*  This program illustrates ACCEPT and DISPLAY with  *
*  and without mnemonic names.                       *
******************************************************
 ENVIRONMENT DIVISION.
   CONFIGURATION SECTION.
     SOURCE-COMPUTER.  HP TXP.
     OBJECT-COMPUTER.  HP TXP.
     SPECIAL-NAMES.
        FILE "$TRM053" IS OUT-THERE.
 DATA DIVISION.
   WORKING-STORAGE SECTION.
    01 A-TEXT-LINE PICTURE X(30).
 PROCEDURE DIVISION.
 A.
*    Deliver a value to the default terminal (the OUT file)
   DISPLAY "Who's there?".
*    Get a line from the default terminal (the IN file)
   ACCEPT A-TEXT-LINE.
*    Deliver the entered-line from the default terminal to
*    OUT-THERE
   DISPLAY A-TEXT-LINE " is at the default terminal."
      UPON OUT-THERE.
*    Deliver a value to a specific terminal
   DISPLAY "Who's there?" UPON OUT-THERE.
*    Get a line from a specific terminal
   ACCEPT A-TEXT-LINE FROM OUT-THERE.
*    Deliver the entered-line from OUT-THERE
   DISPLAY A-TEXT-LINE " is at terminal $TRM053".
   STOP RUN.
```

OSS Environment

These are true in the OSS environment but not in the Guardian environment:

- No prompt is given for an ACCEPT statement.
- If a DISPLAY statement includes *mnemonic-name*, it must be either the OSS pathname of a Guardian file or the name of an OSS text file.
- If an ACCEPT statement includes *mnemonic-name*, it must be the OSS pathname of a Guardian process or terminal. If *mnemonic-name* is an OSS device, a diagnostic is issued and the default input device (#IN) is used instead.

Using a Terminal as a File

All files that an HP COBOL process uses must be specified in the FILE-CONTROL paragraph and described in the File Section of the Data Division of the source program.

As with any other file, the process must explicitly open the terminal file, perform its read and write operations, and close the file.

It is always advisable to declare a terminal file to have variable-length records by including a RECORD CONTAINS or RECORD IS VARYING clause in the file description of the file. If the file is not declared to have variable-length records, the HP COBOL run-time routines expect each

READ statement to receive a record of exactly the length declared (explicitly or implicitly) in the file description, and they raise a run-time error if too short a record is typed at the terminal. If a record of more than the declared number of characters is typed at the terminal, the run-time routine does not deliver the excess characters to the process.

When an HP COBOL process opens a file assigned to a terminal, it establishes communication with that terminal. The default exclusion mode for terminals is SHARED. If a process is to have private use of a terminal, with no command interpreter assigned to the terminal, the OPEN statement should specify the exclusion mode EXCLUSIVE.

Use the ASSIGN clause of the file-control entry for a given file to associate a COBOL file name with a device name or logical device number of a terminal. You can override this assignment at the start of a process's execution by using the TACL command ASSIGN. Establish the assignment dynamically during process execution by specifying #DYNAMIC in the ASSIGN clause of the file-control entry and calling the COBOL_ASSIGN_ routines to associate the terminal device name or logical device number with a COBOL file name.

Example 30-2 shows the use of terminal $TRM053 as a file.

Example 30-2 Using a Terminal as a File

```
 IDENTIFICATION DIVISION.
   PROGRAM-ID.    TERMINAL-READ-WRITE.
   AUTHOR.        BO COBOL.
   DATE-WRITTEN.  29 FEBRUARY 1984.
   DATE-COMPILED.
*************************************************************
*  This program illustrates the use of a terminal as a      *
*  file                                                     *
*************************************************************
 ENVIRONMENT DIVISION.
   CONFIGURATION SECTION.
     SOURCE-COMPUTER.  HP TXP.
     OBJECT-COMPUTER.  HP TXP.
   INPUT-OUTPUT SECTION.
     FILE-CONTROL.
       SELECT A-TERM
              ASSIGN TO "$TRM053"
              ORGANIZATION IS SEQUENTIAL
              ACCESS MODE IS SEQUENTIAL.
 DATA DIVISION.
   FILE SECTION.
     FD A-TERM
        RECORD CONTAINS 1 TO 79 CHARACTERS
        LABEL RECORDS ARE OMITTED.
*           Using 80 characters causes a blank line after
*           the WRITE
     01 A-TEXT-RECORD PICTURE X(79).
 PROCEDURE DIVISION.
 A.
*    Open the terminal as a file, excluding other users.
   OPEN I-O A-TERM EXCLUSIVE.
 B.
*    Read one 79-character record.  If a CTRL/Y was entered,
*    quit.
   READ A-TERM
        AT END GO TO HIT-EOF.
*    Insert response.
   MOVE "So what?" TO A-TEXT-RECORD.
*    Deliver response.
   WRITE A-TEXT-RECORD.
*    Loop.
   GO TO B.
 HIT-EOF.
   STOP RUN.
```

Prompting the Terminal Operator for Input

This topic assumes that you are using the terminal as a file (see Using a Terminal as a File).

HP COBOL has a useful extension that can be used with terminal input and output. The HP COBOL READ statement is extended to include a PROMPT phrase. If you want to enter a dialogue with a user at a terminal, just using alternate WRITE statements for prompts and READ statements to collect the reply causes the prompt to appear on one line of the screen and the reply to be collected from the next line. You can achieve a more natural effect of having the prompt and its reply appear on the same line by using the PROMPT phrase.

When an HP COBOL process executes a READ statement that includes a PROMPT phrase, the HP COBOL run-time routine that performs terminal input and output first delivers the characters in the data item whose name appears in the PROMPT phrase to the terminal, leaving the terminal cursor at the next available character position. Next, the routine clears the file's record area to spaces. When the terminal operator enters a sequence of zero or more characters, terminated by pressing Return, the routine delivers the sequence of characters into the record area and returns control to the process.

Example 30-3 READ Statement With PROMPT Phrase

```
 IDENTIFICATION DIVISION.
   PROGRAM-ID.    TERMINAL-READ-WRITE.
   AUTHOR.        FLO COBOL.
   DATE-WRITTEN.  29 FEBRUARY 1984.
   DATE-COMPILED.
*************************************************************
*  This program illustrates the use of READ PROMPT          *
*  for a terminal file                                      *
*************************************************************
 ENVIRONMENT DIVISION.
   CONFIGURATION SECTION.
     SOURCE-COMPUTER.  HP TXP.
     OBJECT-COMPUTER.  HP TXP.
   INPUT-OUTPUT SECTION.
     FILE-CONTROL.
       SELECT A-TERM
              ASSIGN TO "$TRM053"
              ORGANIZATION IS SEQUENTIAL
              ACCESS MODE IS SEQUENTIAL.
 DATA DIVISION.
   FILE SECTION.
     FD A-TERM
        RECORD CONTAINS 1 TO 79 CHARACTERS
        LABEL RECORDS ARE OMITTED.
*           Using 80 characters causes a blank line after the
*           WRITE.
     01 A-TEXT-RECORD PICTURE X(79).
   WORKING-STORAGE SECTION.
     01 TAUNT PIC X(17) VALUE "What do you want?".
 PROCEDURE DIVISION.
 A.
*    Open the terminal as a file, excluding other users.
   OPEN I-O A-TERM EXCLUSIVE.
 B.
*    Read one 79-character record.  If a CTRL/Y was entered,
*    quit.
   READ A-TERM WITH PROMPT TAUNT
        AT END GO TO HIT-EOF.
*    Insert response.
```

```
   MOVE "So what?" TO A-TEXT-RECORD.
*    Deliver response.
   WRITE A-TEXT-RECORD.
*    Loop.
   GO TO B.
 HIT-EOF.
   STOP RUN.
```

Sharing a Terminal

For development, you normally use a terminal that is associated during system generation with a command interpreter (such as TACL). That command interpreter uses your terminal as its IN file, its OUT file, and its home terminal.

Most terminals on a production system do not have command interpreters associated with them. Each production terminal is directly associated with a production process, either as a terminal running under the control of a Pathway/TS terminal control process (TCP) or as a terminal that can be accessed by a process.

Terminal Associated With a Command Interpreter

When a command interpreter has a terminal open for its IN and OUT file, an HP COBOL process can open that terminal as a file only if it opens it for shared access (the default). It cannot open such a terminal for exclusive access, even if the command interpreter does not accept messages from or send messages to that terminal for the duration of the HP COBOL process's execution. If you want an HP COBOL process to have exclusive use of a terminal, no command interpreter can have the terminal open.

To free a terminal that is running TACL for use as a file by an HP COBOL program, enter the TACL command PAUSE.

Terminal Not Associated With a Command Interpreter

When a terminal is not connected to a command interpreter, an HP COBOL process can open it, read from it, write to it, or close it like any other file. See Using a Terminal as a File.

Non-COBOL Modules

If your program consists of modules written in different languages, the modules can share the standard files—the predefined files called “standard input,” “standard output,” and “standard log. The standard input, output, and log file can be a terminal.

For more information about mixed-language programs sharing standard files, see the *CRE Programmer's Guide*.

Getting Break Ownership

When you press Break on a terminal, the process that owns Break receives a system message on $RECEIVE. A process discovers that Break has been pressed by reading $RECEIVE. An HP COBOL process, lacking the ability to perform NOWAIT input and output directly, must open $RECEIVE with an OPEN statement with a TIME LIMITS phrase and then try to read $RECEIVE from time to time. If a READ statement fails because it timed out, Break was not pressed.

When a command interpreter starts a process, whether or not the process uses the command interpreter's terminal for any input or output, the command interpreter retains ownership of

Break unless another process explicitly obtains ownership of Break from the operating environment. All a process needs to do to obtain that ownership is:

- Open the terminal for I-O (to get a stable file number, because DISPLAY and ACCEPT open and close the terminal each time they are executed)
- Call the COBOL_SETMODE_ routine with:
  - — The *`file-name`*
  - — The function code 11
  - — The CPU, PIN (processor number, process number) of the process to receive ownership as the first parameter
  - — Zero as the second parameter

If you start a process from the command interpreter, and that process takes ownership of Break but does not return ownership before terminating, the command interpreter does not regain title to Break. You might have to get a system operator to stop and restart your command interpreter in order for it to regain Break ownership.

Transferring Break Ownership

For one process to pass ownership of Break to another process, the first process must determine the CPU and PIN of the second process. Then the first process can transfer Break ownership to the other process by calling the operating system routine SETMODE with the appropriate values for parameters 1 and 2 (see Getting Break Ownership).

The most common reason to transfer Break ownership to another process is to return it to its previous owner after you have accessed it.

If the process will return Break ownership to its previous owner after a time, another parameter must be included in the call to receive the 4-byte value of the restoration parameters. This enables the process to call SETMODE with the 4 bytes as parameters 1 and 2 when it surrenders the ownership of the Break, as Example 30-4 shows.

Because the message that Break has been pressed is a system message, a nonprivileged process cannot directly transfer the message to another process. Some other protocol must be established between the two processes.

Example 30-4 Transferring Break Ownership

```
WORKING-STORAGE SECTION.
01  FILE-NUMBER         PIC S9(4).
01  PROCESS-HANDLE      PIC X(20).
01  ERROR-NUMBER        PIC S9(5) COMP.
01  PREV-BREAK-OWNER.
    03 PREV-OWNER       NATIVE-2.
    03 PREV-ACCESS-MODE NATIVE-2....
PROCEDURE DIVISION....
   OPEN I-O A-TERM.
   ENTER TAL "COBOLFILEINFO" USING A-TERM
                                   OMITTED
                                   OMITTED
                                   FILE-NUMBER.
*  Get ownership of Break, saving previous info.
   ENTER TAL "SETMODE" USING FILE-NUMBER,
                             11,
                             1,
                             1,
                             PREV-BREAK-OWNER....
...  { HP COBOL program now owns Break }...
*  Restore Break ownership according to saved info.
   ENTER "COBOL_SETMODE_ " USING FILE-NUMBER,
                             11,
                             PREV-OWNER,
                             PREV-ACCESS-MODE.
```

Communicating With an Operator or Process

A running HP COBOL program can use the DISPLAY statement to write to the operator console (a write-only device). Likewise, a running HP COBOL program can use the DISPLAY statement to write characters to any terminal or printer and can use the ACCEPT statement to read characters from any terminal or process.

When a process includes code from an HP COBOL program and that HP COBOL code reports a run-time error, it reports the error to the process’ home terminal. The home terminal for a process must be chosen with care. Run-time diagnostic messages can easily get lost if an unattended terminal is used for them.

Writing to a Console

The system operator console terminal device on any HP system is named $0 (“dollar zero”). It can be a hard-copy device or the Operations and Service Processor (OSP). In either case, the system operator (any super group member) can redirect the messages to another device or disable the delivery of messages entirely.

Minimize messages to the operator in both development programs and production programs.

If your HP COBOL application needs to announce something to the system operator, it must either associate a mnemonic name with $0 and use DISPLAY statements to communicate with the operator (as Example 30-5 does) or assign a file to $0 in the Environment Division and use WRITE statements.

Example 30-5 Using DISPLAY Statements to Write to a Console

```
IDENTIFICATION DIVISION.
   PROGRAM-ID.    TELL-THE-LOG.
   AUTHOR.        KILROY COBOL.
   INSTALLATION.  TRANSACTIONS ANONYMOUS.
   DATE-WRITTEN.  29 FEBRUARY 1984.
   DATE-COMPILED.
*********************************************************
*  This program delivers a line to the system console.  *
*********************************************************
 ENVIRONMENT DIVISION.
   CONFIGURATION SECTION.
     SOURCE-COMPUTER.  HP TXP.
     OBJECT-COMPUTER.  HP TXP.
     SPECIAL-NAMES.
         FILE "$0" IS RECORD-FOR-POSTERITY.
 DATA DIVISION.
   WORKING-STORAGE SECTION.
     01  ANNOUNCEMENT  PICTURE X(16) VALUE "Kilroy ws here".
 PROCEDURE DIVISION.
 A.
     DISPLAY "Program TELL-THE-LOG: "
             ANNOUNCEMENT
             UPON RECORD-FOR-POSTERITY.
     STOP RUN.
```

Each console terminal device on your node has a unique name. The names of consoles other than the system operator console are determined by the system generation. A program can contain any number of mnemonic names, each associated with a terminal device. The syntax of the DISPLAY statement limits the name in the UPON phrase to be a mnemonic name. The mnemonic name cannot be a data item; therefore, if you have a message that must be displayed on several devices, you need one DISPLAY statement for each device.

Writing to a Terminal or Printer

When a process includes code from an HP COBOL program having a DISPLAY statement without a mnemonic name attached, the process delivers characters to its OUT file. If no OUT file was specified in the initiation of the process, the process inherits the OUT file of its parent (the command interpreter, if the process was initiated with a RUN command). If the name of the specified OUT file is all spaces, the characters are discarded.

To display characters to device $DDD or printer $PPP that is not the OUT file of the process, use the qualified form of the DISPLAY statement; for example:

```
DISPLAY "Hello" UPON MY-NAME-FOR-DOLLAR-DDD.
```

The compiler accepts this form of the DISPLAY statement only if it has already encountered an entry in the SPECIAL-NAMES paragraph associating MY-NAME-FOR-DOLLAR-DDD with the device $DDD; for example:

```
SPECIAL-NAMES.
   FILE "$DDD" IS MY-NAME-FOR-DOLLAR-DDD.
```

Each time a DISPLAY statement in a process executes, the process opens the terminal or printer, transmits the characters, and closes the terminal or printer. Some other process could open the terminal or printer for exclusive use and hold it open, causing the next DISPLAY or ACCEPT statement that tries to reach it to get a “device in use” error.

Reading From a Terminal or Process

When a process includes code from an HP COBOL program having an ACCEPT statement without a mnemonic name attached, the process reads characters from its IN file; if no IN file was specified, the process reads characters from its home terminal.

To read characters from terminal $TTT or process $QQQ that is not the IN file of the process, use the qualified form of the ACCEPT statement; for example:

```
ACCEPT ADVICE FROM MY-NAME-FOR-DOLLAR-QQQ.
```

The compiler accepts this form of the ACCEPT statement only if it has already encountered an entry in the SPECIAL-NAMES paragraph associating MY-NAME-FOR-DOLLAR-QQQ with some device or process; for example:

```
SPECIAL-NAMES.
   FILE "$QQQ" IS MY-NAME-FOR-DOLLAR-QQQ.
```

Each time an ACCEPT statement in a process executes, the process opens the terminal or process, collects the characters, and closes the terminal or process. Some other process could open the terminal or process for exclusive use and hold it open, causing the next ACCEPT or DISPLAY statement that tries to reach it to get a “device in use” error.

Reading and Writing Numeric Data

The ANSI standard is not very specific on the details of the behavior of ACCEPT. The ACCEPT statement, as implemented in HP COBOL, allows you some flexibility in entering numeric data that some other implementations do not.

HP COBOL allows you to enter a sequence of characters that correspond to a numeric literal, such as would be legal in an HP COBOL source program. The ACCEPT run-time routine converts this numeric representation into an appropriate form for storage in the designated data item, only as if you had a MOVE statement that specified the numeric literal as its source and the data item as its destination.

The DISPLAY statement is equally versatile, converting a stored data value to a convenient external format. In Example 30-6, both ACCEPT and DISPLAY handle signed and unsigned numeric, numeric edited, and computational items.

Example 30-6 Reading and Writing Numeric Data

```
 IDENTIFICATION DIVISION.
   PROGRAM-ID.    FRIENDLY-ACCEPT.
   AUTHOR.        KIT COBOL.
   INSTALLATION.  TRANSACTIONS ANONYMOUS.
   DATE-WRITTEN.  29 FEBRUARY 1984.
   DATE-COMPILED.
******************************************************
*  This program shows the flexibility of the ACCEPT  *
*  statement.                                        *
******************************************************
 ENVIRONMENT DIVISION.
   CONFIGURATION SECTION.
     SOURCE-COMPUTER.  HP TXP.
     OBJECT-COMPUTER.  HP TXP.
     SPECIAL-NAMES.
DATA DIVISION.
   WORKING-STORAGE SECTION.
     01  A      PICTURE 99999.
     01  B      PICTURE 999V999.
     01  C      PICTURE S999V999.
     01  D      PICTURE 999.999.
     01  E      PICTURE 9999V9999 USAGE IS COMPUTATIONAL.
     01  F      PICTURE
 USAGE IS COMPUTATIONAL.
  PROCEDURE DIVISION.
 Z.
     ACCEPT A.  DISPLAY A.
     ACCEPT B.  DISPLAY B.
     ACCEPT C.  DISPLAY C.
     ACCEPT D.  DISPLAY D.
     ACCEPT E.  DISPLAY E.
     ACCEPT F.  DISPLAY F.
     STOP RUN.
```

A record of the output of one execution of the program in Example 30-6 follows. Each ACCEPT statement interprets the characters entered at each prompt (?) the same way as it would interpret those characters as a numeric literal in a source program. Each DISPLAY statement converts the value of a data item to a character-string that represents a numeric value for display.

```
?-1234.5678
01234
?-1234.5678
234.567
?-1234.5678
-234.567
?-1234.5678
234.567
?-1234.5678
1234.5678
?-1234.5678
-0001234.5678000
```

Logging Program Activity Information to a Terminal

Whenever a process sends a message to the system operator console terminal device ($0), the operating environment can automatically duplicate the message to the standard log file, which can be a terminal.

If you use the DISPLAY statement to send messages to the console, and therefore to the standard log file terminal, you must associate a mnemonic name with $0 in the SPECIAL-NAMES paragraph. This association is established at compilation time. Each DISPLAY statement transmits characters to a single terminal or device.

If you use the WRITE statement to send messages to the console, and therefore to the standard log file terminal, you can determine the assignment between the file and the device in any one of these ways:

- During program compilation, through the SELECT clause of the FILE-CONTROL paragraph
- Before program execution, through the ASSIGN or ADD DEFINE command
- During program execution, with the COBOL_ASSIGN_ routine.

If your program consists of modules written in different languages, the modules can share the standard log file terminal.

For more information about mixed-language programs sharing standard files, see the *CRE Programmer's Guide*.

31 Printer and Spooler Output

There are two ways that a process can write a report to paper: it can write directly to a printer or it can write to a spooler collector.

When a process writes directly to a printer, it has exclusive use of the printer from the time it opens the file until the time it closes the file.

When a process writes to a spooler collector, the spooler collector accumulates the lines of the report on a disk and does not engage a printer until the process closes the file.

Some installations insist that processes write only to a spooler, because this can even out the printer usage or speed up jobs that would be output bound if they had to wait for a slow printer.

An installation can configure its spooler locations such that a printer is either accessible only to the spooler and never by other processes or accessible to other processes whenever the spooler is not using it.

HP has no file structure that is strictly for printer files. To establish a COBOL file for use with a printer, declare it to have sequential organization and to be accessed sequentially.

Using a Printer Directly

Each installation chooses names for its printers. To write to a printer, an HP COBOL program declares a file with sequential organization and sequential access and associates that file with the printer device name.

The HP COBOL process tries to open the printer file with the open mode OUTPUT or EXTEND. If the printer is not available, the OPEN statement fails with file status code "30" and a run-time error. The HP COBOL run-time routines deliver a diagnostic message to the process' home terminal. If no declarative procedure is available for the printer file, the process terminates. If a declarative procedure is available for the printer file, the declarative procedure can examine the special register, GUARDIAN-ERR, and take appropriate action.

GUARDIAN-ERR contains the file-system error number that specifies the reason the printer was not available. Typical values for GUARDIAN-ERR are 14 (device does not exist) and 12 (file in use). If the file is in use, the process can call the operating system routine DELAY to let some time pass and then try the open operation again.

If the printer is available, the open operation succeeds, and the process can go ahead and write to the printer. By default, an HP COBOL program opens a printer file with exclusion mode EXCLUSIVE. All other processes that attempt to write to the printer are excluded until the process that has the printer open closes it.

Understanding Spoolers

The HP spooler is a set of processes that acts as an interface between the print devices of a system and the users and their application programs. A spooler receives output from a process and stores it on disk, queued for delivery to a print process. This unit of storage is called a job.

For more information on spoolers, see the *Spooler Programmer's Guide*.

Spooler Components

The spooler has these components:

- Supervisor Process
- Collector Process
- Print Process
- SPOOLCOM
- PERUSE

The two spooler components that you are most likely to use are SPOOLCOM and PERUSE.

Unlike the supervisor, collectors, and print processes that run continuously and are named processes, SPOOLCOM and PERUSE are processes you start when you need one and they usually are not named. Both interact with the supervisor.

Supervisor Process

The supervisor process communicates with and monitors the other processes and decides when and where to print output. The supervisor is an HP product and is started and managed by the system manager. A system can have more than one spooler supervisor; this can be useful for systems with vast numbers of print jobs or systems with special reporting needs. If a system has only one supervisor, its name is usually $SPLS.

Collector Process

The collector process receives the output. Your spooler system has one or more collectors. Each installation chooses its own names, but typical process names for spooler collectors are $S and $C.

Print Process

Each print device accessible to a spooler has an associated print process that executes in coordination with the spooler supervisor. The print process provided by HP reads the records stored on disk and delivers them to its associated print device. An installation can provide its own print processes that read the stored records and perform other operations on them, such as performing a statistical analysis of data in the records or formatting the data for a special-purpose output device.

SPOOLCOM

SPOOLCOM is an HP product. It is a loadfile that can be started by an interactive user or a system operator. System operators, who are members of the super group, can do more with SPOOLCOM than application programmers can.

As an application programmer running SPOOLCOM interactively, you can examine the job queue and the status of print devices and change attributes of jobs you own.

A system operator running SPOOLCOM can examine the queue and the status of devices but can also create and initialize the other components of the spooler system and make changes to the attributes of any job in the spooler.

For more information about SPOOLCOM, see the *Spooler Programmer's Guide*.

PERUSE

HP provides a perusal program named PERUSE, with which you can control and monitor your jobs in the spooler. PERUSE is oriented more to the interactive user than to the system operator. When a member of the super group calls PERUSE, that user must handle individual jobs rather than groups of jobs, and PERUSE does not give that user the control over devices that SPOOLCOM does.

Your installation can write its own perusal processes. For more information about PERUSE, see the *Spooler Programmer's Guide*.

Jobs

A job is analogous to a file. When you write to the spooler, you are creating a job. The spooler assigns each job a number. It starts at 1 and advances by 1 up to a maximum (specified when the spooler was started), not to exceed 4095. When the spooler reaches the maximum job number, it starts over with 1. If the job number the spooler tries to use is currently being used, the spooler advances until it finds a job number that is not being used.

Each job has an owner. When an application program opens the spooler collector for output, the created job is marked as owned by the user number of the application program. Unlike a file, a spooler job does not have four-fold security; only the individual owner (or a system operator in the super-group) can print, delete, or otherwise manipulate a job.

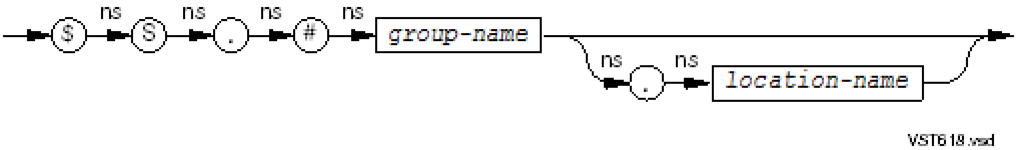
Using SPOOLCOM, you can transfer ownership of a job to another user. If one spooler reroutes a job to another spooler on the same system or on a different system in an Expand network, the owner of the job on the second spooler is the same as the owner was on the first spooler (typically, the system operator; therefore, if your spooler reroutes your job to another spooler, you are no longer the owner of the job.

Jobs have six primary attributes:

- Location
- State
- Number of Copies
- Priority
- Report Name
- How an HP COBOL Program Calls a Non-COBOL Program

Location

A job's location is the job's logical destination. Its physical destination is governed by a print process. A job's location name consists of a group name and a destination name. The format of a location name is:



Suppose your installation has 12 printers, of which 4 are stocked with narrow paper. Three of them are at the other end of the building (in room 145), and one is near your work place (in room 301). Suppose your spooler designates this group of printers as NARROW, and within that group designates the printers as RM145A, RM145B, RM145C, and RM301. If you do not care which narrow-paper printer your report is printed on, you can assign the output file to $S.#NARROW. The spooler directs the job to whichever of the four printers could finish it first. If you want the job printed on the nearby printer, assign the output file to $S.#NARROW.RM301.

There are two ways that your system manager can configure the spooler to print jobs that have only a group name but no destination name specified:

- On the printer that can finish the job first
- On all printers in the group

The latter configuration is called broadcast mode.

If you specify a location that is unknown to the spooler, your job sits in the spooler until such a location is made known to the spooler or you or a system operator changes the job's location with PERUSE or SPOOLCOM.

You can send the job to a fictitious location and use PERUSE to examine the contents of a report before it is printed. If there is any problem with the report, you can delete it from the spooler, fix the program that produced the report, and re-create the report.

If you do not specify a location, the spooler uses the location #DEFAULT. Each installation determines for itself the identity of the device or devices that compose the default group.

State

A job is always in one of four states:

State	Meaning
OPEN	A process is writing to the collector.
READY	The originating process has closed its file and the job is ready to be printed, but the device on which it is to print is not currently available or its location is not associated with a device.
PRINT	The job is being printed on the device associated with the job's location.
HOLD	The job is awaiting disposition.

If the job is in PRINT state, the spooler stops printing the job as soon as you request that the job be put into the HOLD state.

You can put a job into the HOLD state with PERUSE or SPOOLCOM. PERUSE has a HOLD command; SPOOLCOM has a JOB command with a HOLD subcommand.

By default, the spooler deletes a job as soon as it has printed it. If you want the spooler to put the job into the HOLD state after printing it instead, use the PERUSE command HOLDAFTER or the SPOOLCOM command JOB with the subcommand HOLDAFTER. You can issue the HOLDAFTER command for a job in any state.

A job in the HOLD state remains in the spooler until you or a system operator changes the job's state or deletes the job. Remember that the number of jobs the spooler can hold is limited, as is the amount of disk space available for the text of those jobs.

Number of Copies

The default number of copies is 1. If you want several copies of your report, specify a number of copies (a number in the range from 1 through 32,767). When the spooler finds an available device for the job, it prints that many copies before freeing the device for another job.

Your installation might have rules limiting how many copies you may run or how many pages you may print with a single job; check with your operations staff.

Priority

A job's priority determines when the job is printed in relation to other jobs queued for the same device. The default priority is 4. The system operator can configure each device known to the spooler to print jobs of the same priority in a first-come, first-served order or in shorter-jobs-before-longer-jobs order.

Report Name

The spooler passes the report name to the print process. If your installation uses the print process supplied by HP, the system operator can configure it to either leave a blank page between jobs or print a header page. The header page displays the report name in banner-head letters on the first page of the job, along with the location name. In ordinary type at the bottom of the first page, the print process reports the date and time, the job number, and the form name of the job.

A report name can have up to 16 characters. No character can be a hyphen. The default report name is your group name and user name. You can use PERUSE or SPOOLCOM to change the report name.

If you are using a different print process, that print process can do anything with the report name.

Form Name

If a job must be printed on a specific form, give the job a form name. Each device known to the spooler can have a form name.

If your installation prints invoices, you might specify INVOICE as the form name for a report. When the time comes to print the invoices, the operator loads the proper paper forms on the chosen device and sets the device's form name to INVOICE. When both the location and form name of a spooler job match the location and form name of the device, the spooler prints the job. You probably cannot expect the operator to notice that a job is waiting for a form name to be associated with a device. Your facility might have a forms schedule for regularly scheduled jobs, but you would have to alert the operator for other spooler jobs that need special forms.

A form name can have up to 16 characters.

A form name must not:

- Start with a number or a space character
- Contain any special characters or embedded blanks

The default form name is blank (that is, no form name).

When a device has a nonblank form name associated with it, the spooler routes only jobs with the same form name to that device. When a device has no form name associated with it, the spooler can route any job with no form name to it.

Using a Spooler

There are two contexts in which you, as an HP COBOL application programmer, use a spooler: compilation and execution. When you compile an HP COBOL program and produce a listing, it can go to the spooler. When you execute an HP COBOL program, it can send output to the spooler. There are three levels of spooling. You interact with the spooler through the PERUSE and SPOOLCOM processes.

Spooling Compiler Listings

To send your compiler listing to a spooler, specify a spooler collector and location for your OUT file when you compile your HP COBOL program; for example:

```
COBOL85 /IN WHIZBANG, OUT $S.#WIZZ/;SYNTAX
```

The compiler uses level 3 spooling when the OUT file is a spooler collector (device subtype 31).

Spooling Program Output

For its output, an HP COBOL program can use any of:

- Level-1 Spooling
- Level-2 Spooling
- Level-3 Spooling

Level-1 Spooling

Level-1 spooling uses the spooler collector as a simple output file: you assign the COBOL file name of the output file to a system name that is a spooler collector (with or without a location name) and operate on the file with the HP COBOL statements OPEN, WRITE, REWRITE, and CLOSE. The LINAGE clause and the ADVANCING phrase of the WRITE statement work as expected. All attributes of the spooler job except the location name assume their default values: single copy, priority 4, report name set to group name followed by user name, and no form name.

Example 31-1 Level-1 Spooling

```
SELECT SPOOLER-FILE
  ASSIGN TO "$S.#MYSTUF"
  ORGANIZATION IS SEQUENTIAL
  ACCESS MODE IS SEQUENTIAL....
FD  SPOOLER-FILE
    RECORD CONTAINS 80 TO 132 CHARACTERS
    LABEL RECORDS ARE OMITTED.
01  SPOOLER-SHORT-LINE    PIC X(80).
01  SPOOLER-LONG-LINE     PIC X(132)....
PROCEDURE DIVISION....
   OPEN OUTPUT SPOOLER-FILE...
   WRITE SPOOLER-SHORT-LINE...
```

Level-2 Spooling

You use level-2 spooling the way you use level-1 spooling, except that instead of opening the output file with the OPEN statement, you open it with the routine COBOL_SPECIAL_OPEN_ , giving the first parameter, open-type, the value 1, and either omitting the parameter level-3 or giving it the value 0.

The other parameters of the COBOL_SPECIAL_OPEN_ routine enable you can specify:

- Exclusion mode
- Sync depth
- Whether a page-eject precedes the first page printed
- Location name
- Form name
- Report name
- Number of copies
- Page size
- Flags that set the job priority and the attributes HOLD and HOLDAFTER
- Owner
- Maximum number of lines
- Maximum number of pages
- Whether to create a regular or code-129 spooler file
- Whether a form feed is to be used to position the file at the top of a new page rather than using spacing when LINAGE is specified for the file
- An error return code

For details, see COBOL_SPECIAL_OPEN_ (page 644).

The preferred way to apply a declarative to a file that is used for level-2 spooling is to explicitly name the file in a USE AFTER EXCEPTION statement. All ordinary HP COBOL file-manipulating statements then transfer control to the declarative if an exception arises. The statement ENTER COBOL_SPECIAL_OPEN_ (in the CRE environment) does not transfer control if it encounters an exception, so include a GIVING phrase in the ENTER statement and then test the file status code data item after the ENTER statement executes.

Example 31-2 uses level-2 spooling in the non-CRE environment. In the CRE, use COBOL_SPECIAL_OPEN_ instead of COBOL85^SPECIAL^OPEN.

Example 31-2 Level-2 Spooling

```
SELECT SPOOLER-FILE
  ASSIGN TO "$S"
  ORGANIZATION IS SEQUENTIAL
  ACCESS MODE IS SEQUENTIAL
  FILE STATUS IS SPOOLER-FILE-STAT....
```

```
FD  SPOOLER-FILE
     RECORD IS VARYING FROM 80 TO 132 CHARACTERS.
 01  SPOOLER-SHORT-LINE     PIC X(80).
 01  SPOOLER-LONG-LINE      PIC X(132).
WORKING-STORAGE SECTION.
 01  SPOOLER-FILE-STAT      PIC XX.
 01  LOCATION-NAME          PIC X(16) VALUE "#MYSTUF".
 01  REPORT-NAME            PIC X(16) VALUE "PRIVATE PROPERTY".
 *     Flag for HOLDAFTER (32), PRIORITY 2 (+2 = 34)
 01  FLAGS                  PIC S9(4) VALUE 34.
 01  ERROR-CODE             PIC S9(4)....

PROCEDURE DIVISION.
...
ENTER "COBOL_SPECIAL_OPEN_"

                  USING SPOOLER-FILE
                     1
                     OMITTED
                     OMITTED
                     OMITTED
                     0
                     LOCATION-NAME
                     OMITTED
                     REPORT-NAME
                     OMITTED
                     OMITTED
                     FLAGS
              GIVING ERROR-CODE...
    WRITE SPOOLER-SHORT-LINE...
```

Level-3 Spooling

Level-3 spooling causes the compiler to block output and minimizes the number of interactions with the spooler. There are two ways to get level-3 spooling:

- The same way you get level-2 spooling, except that when you open the output file with the COBOL_SPECIAL_OPEN_ routine, you give the parameter *level-3* a nonzero value.
- Instead of using HP COBOL output statements on the file, use ENTER statements to call the Guardian environment routines listed in Table 31-1.

Table 31-1 Guardian Environment Routines for Level 3 Spooling

Routine	Action
CLOSE	Closes the spooler collector
OPEN	Opens the spooler collector
SPOOLCONTROL	Handles VFU channel skipping
SPOOLCONTROLBUF	Handles the loading of programmable VFU for the model 5220 printer
SPOOLEND	Signals the end of a spooler job and can change attributes before a new spooler job starts without closing and reopening the spooler collector
SPOOLSETMODE	Sets automatic perforation skipping, system spacing control, baud rate, form length, vertical tabs, automatic-answer or control-answer mode, horizontal pitch, prespacing or post-spacing, get device status, or reset to configured values
SPOOLSTART	Establishes the level-3 buffer, location name, form name, report name, number of copies, page size, and attributes such as HOLD and PRIORITY
SPOOLWRITE	Writes text to the spooler

For information about these routines, see the *Spooler Programmer's Guide*.

Using PERUSE

If you direct a process's output to a spooler location that is known to the spooler, the job prints as soon as the device associated with that location becomes available. If the location you chose is unknown to the spooler, the job sits in READY state until one of these occurs:

- An operator (using SPOOLCOM) defines that location to the spooler.
- You or the operator changes the location attribute of the job to a known location.
- You or the operator deletes the job from the spooler.

The HP perusal process PERUSE enables you to examine the contents of the job, change its attributes, or delete it from the spooler. To call PERUSE, run it like any other process:

```
103> PERUSE
```

PERUSE introduces itself with a header. If any jobs in the spooler belong to you, PERUSE lists them.

In Example 31-3, job 1171 is ready to be printed. It consists of three pages, one copy is to be printed, its priority is 4, its HOLDAFTER attribute is not set, its location is #EXCEP R01, and it belongs to DEVELOP.JAN. Job 2423 does have its HOLDAFTER attribute set. The underscore (_) is the PERUSE prompt.

Example 31-3 PERUSE Output

```
PERUSE - T9101D20 - (8JUN92)     SYSTEM  \LEO

  JOB    STATE   PAGES   COPIES   PRI   HOLD   LOCATION             REPORT
  1171   READY   3       1        4            #EXCEP  R01          DEVELOP JAN
  2423   READY   4       1        4      A     #MURPHY              DEVELOP JAN
_
```

For more information about PERUSE, see the *Spooler Programmer's Guide*.

Current Job

The current job is the job to which the spooler is currently devoting its attention. Almost all commands that you give to PERUSE apply only to the current job. You cannot tell the spooler to list job number 2423; you must establish job 2423 as the current job and then tell the spooler to list the current job.

The current job is undefined when you first start the spooler and after a DELETE command executes. To define the current job, use the JOB command.

Example	Effect
`JOB 1171`	Establishes job 1171 as the current job
`JOB *`	Establishes the job most recently added to the spooler (and owned by the current user) as the current job
`JOB #MURPHY`	Establishes the job most recently sent to location #MURPHY (and owned by the current user) as the current job

If the first command you give to PERUSE is not a JOB command, any command except DELETE establishes the job most recently added to the spooler (and owned by the current user) as the current job.

Current Position

At any given time during your PERUSE session, the spooler has a current position within its current job, just as a file in an HP COBOL program has a current record position. If you press

Return at the PERUSE prompt, PERUSE lists the next line of the job and advances its current position.

Common PERUSE Operations

After you have established a current job, you can perform the operations such as those listed in Table 31-2 (all operations apply to the current job unless otherwise specified). Table 31-2 is not a complete list of PERUSE commands; for more information, see the *Guardian User's Guide*.

Table 31-2 PERUSE Operations and Commands (Partial List)

Operation	Command
Set number of copies	COPIES
Set form name	FORM
Set location	LOC
Set report name	REPORT
Set priority	PRI
Put the job into the HOLD state	HOLD or HOLD ON
Remove the job from HOLD state	HOLD OFF
Set the HOLDAFTER attribute	HOLDAFTER or HOLDAFTER ON
Clear the HOLDAFTER attribute	HOLDAFTER OFF
List some or all of the current job at your screen or on a device or to another job	LIST
List last page of the job (this is often a summary, showing any errors)	LIST L
Set the current position as the first line of a chosen page in the job	PAGE
Display the page number and line number of the current position	PAGE
Find the next instance of a certain character-string in the job	FIND
Delete the job	DELETE
Give the job to a new owner	OWNER
Get information about PERUSE commands	HELP

Here are some examples of spooler commands that involve the current job and the current position. Suppose you have just started the spooler. The spooler always begins by listing all jobs in its queues that belong to you. Suppose that the list is:

```
PERUSE - T9101D20 - (8JUN92)    SYSTEM  \LEO

  JOB   STATE  PAGES  COPIES  PRI  HOLD  LOCATION          REPORT
  1171  READY  3      1       4          #EXCEP  R01       DEVELOP JAN
  2423  READY  4      1       4     A    #MURPHY           DEVELOP JAN

_
```

To establish job 1171 as the current job, you enter:

```
_JOB 1171
```

Suppose that the last page of the listing contains a summary, and you want to check that first. You enter:

```
_LIST L
```

The last page of the listing appears on your terminal screen. If it looks acceptable, you can relocate the listing to a defined device by entering a command such as:

```
_LOC #LP5
```

If the printer associated with #LP5 is not busy, your job starts printing there. As soon as the job is printed, the spooler deletes the job unless its HOLDAFTER attribute is set.

If you want to browse through your job, use the function keys to list groups of lines. Function key *n* lists the next 2 *n* lines, starting at the current position; for example, function key 1 lists the next 2 lines, function key 2 lists the next 4 lines, function key 3 lists the next 8 lines, and function key 6 lists the next 64 lines (approximately a page in a typical report).

To find certain texts in your job, use the FIND command. Suppose you compile an HP COBOL program, sending its listing to a spooler location that is not associated with a device. The job remains in the spooler in a READY state. If you list the last page of the job (using the LIST L command), you can view the summary and see whether the compiler found any errors. If it did find errors, you can use the FIND command to track them down easily.

If you have a job in the PRINT state and you suddenly remember that you wanted to make additional copies, you can issue a HOLDAFTER command to keep the spooler from automatically deleting the job at the end of the printing operation. If the spooler collects your HOLDAFTER command before it finishes listing the job, then after the job finishes printing, it goes into the HOLD state. You can then request that three more copies be sent to the same location:

```
_COPIES 3
_HOLD OFF
```

If you want a different banner on the additional copies, you can change the job's report name before setting the HOLD state to OFF. If you want to list the three extras copies at a different location, you can use the LOC command to change the location before setting the HOLD state to OFF.

If your current job is 300 pages, and you want to print 5 copies of pages 150 through 159 at the printer associated with location #LP2.A, you can issue the commands:

```
_LIST /OUT $S.#ABBREV/ 150/159 C
_JOB #ABBREV
_COPIES 5
_LOC #LP2.A
```

This creates a new spooler job with the location ABBREV, containing only the chosen pages. The letter *C* in the LIST command transmits all CONTROL and SETMODE commands embedded in the job to the new job. Without *C*, the pagination of the new job does not match that of the old. The JOB command makes the job most recently sent to #ABBREV the current job.

If you have a job in the HOLD or READY state, you can delete it by making it the current job and issuing the DELETE command.

Using SPOOLCOM

Although most of your interaction with the spooler is likely to be through PERUSE or another perusal process, there are a few things you might want to do for which SPOOLCOM can be more helpful than PERUSE. The main value of SPOOLCOM is that it shows you the date each job was created, and it also enables you to perform operations on groups of jobs.

SPOOLCOM is intended primarily for system operators (members of the super group). These users have privileged access to the system. If you do not have privileged access, SPOOLCOM restricts the variety and complexity of operations available to you.

SPOOLCOM does have the ability to accept commands on the (implicit) RUN command line and to accept commands from an IN file specified as a RUN option, but the way you are most likely to use SPOOLCOM is run it like any other process:

```
104> SPOOLCOM
```

SPOOLCOM introduces itself with this header:

```
SPOOLCOM - T9101C00 - (15JUL87)  SYSTEM \LEO
)
```

The right parenthesis is the SPOOLCOM prompt. Table 31-3 lists some of the commands available to you, but it is not a complete list of SPOOLCOM commands. For more information, see the *Guardian User's Guide*.

Table 31-3 SPOOLCOM Operations and Commands (Partial List)

Operation	Command
Report the status of a particular collector or all collectors	COLLECT
Report the state of a device in the spooler system	DEV
Skip pages on a report that a particular device is printing	DEV
Restart a device that has been stopped by a device error	DEV
Escape from SPOOLCOM	EXIT or Control-*Y*
Show proper syntax of all SPOOLCOM commands	HELP
Operate on one or more of your jobs in the spooler, such as reporting their status, changing their attributes or ownership, deleting them, starting them (shifting them from HOLD state to READY state), and so on	JOB
Report the status of one or more locations (including their destinations)	LOC
Report a cross-reference list of locations, devices, and print processes	LOC
Specify a spooler supervisor with which SPOOLCOM is to communicate	OPEN
Report the status of the spooler's print processes	PRINT
Report the status of the spooler	SPOOLER

The subcommands most useful to the application programmer are:

- JOB
- DEV

JOB

The SPOOLCOM command JOB is very powerful and flexible. Suppose that you are user 5,40, and you give the command:

```
) JOB (OWNER 5,40)
```

You get a report like this:

```
JOB  STATE  FLAGS  OWNER   TIME  COPY  PAGE  REPORT          LOCATION
1203 HOLD   4 A    5,40    01/28 1     4     DEVELOP QUINN   #ANON
1595 READY  4 A    5,40    01/20 1     59    DEVELOP QUINN   #ANON
1671 HOLD   4 A    5,40    12/15 1     1     DEVELOP QUINN   #LP3
1672 PRINT  4 A    5,40    12/15 1     1     DEVELOP QUINN   #LP3
1737 READY  4      5,40    01/28 1     8     DEVELOP QUINN   #ANON
1739 HOLD   4      5,40    01/11 1     1     DEVELOP QUINN   #HT
1779 HOLD   4      5,40    01/28 1     1     DEVELOP QUINN   #HT
2423 READY  4 A    5,40    01/20 1     4     DEVELOP QUINN   #MURPHY
```

This is much like the report that PERUSE gives you at the beginning of its execution, except that you have to ask for jobs belonging to user 5,40, or SPOOLCOM lists all its jobs, not only yours.

Suppose that you want to delete the two jobs belonging to you that have been around since December 15 (jobs 1671 and 1672). In PERUSE, you must establish 1671 as the current job, delete it, establish 1672 as the current job, and delete it. With SPOOLCOM you can give the command:

```
JOB (OWNER 5,40, DATE FROM DEC 15 1988 THRU DEC 15 1996)
```

SPOOLCOM reports only your jobs created on that date. When you are certain that the jobs of 12/15 are the right ones, you can delete the chosen jobs with the command:

```
JOB (OWNER, DATE FROM DEC 15 1988 THRU DEC 15 1996), DELETE!
```

If you do not use the exclamation mark (!) SPOOLCOM asks you about each job. If you use OWNER without a user ID, SPOOLCOM uses the creator accessor ID (your user ID).

For two jobs, SPOOLCOM requires more keystrokes than PERUSE, but suppose you wanted to delete all your jobs created on or before January 11. The SPOOLCOM command for that is:

```
JOB (OWNER, DATE THRU JAN 11 1997), DELETE!
```

Similarly, you can delete all jobs whose current location is #ANON by using the command:

```
JOB (OWNER, LOC #ANON), DELETE!
```

You can delete all jobs whose current state is HOLD by using the command:

```
JOB (OWNER, STATE HOLD), DELETE!
```

If you want the spooler to print one copy each job in a certain set but direct them to another user at another spooler location, and delete those jobs after they are printed, you can give the command:

```
JOB (OWNER, LOC #ANON), HOLD, REPORT DEVELOP LEE, HOLDAFTER OFF, LOC #PRT.A, START
```

DEV

The SPOOLCOM command DEV is useful for discovering what devices are configured for your spooler, and for skipping pages on a spooler job. If you give SPOOLCOM a simple DEV command with no parameters, you get a report like this:

```
DEVICE                           STATE          FLAGS  PROC    FORM
$LP1                             JOB 159        H      $SPLP
$LP2                             WAITING          ET   $SPLP   RED
$SPECL.#TYPE1                    WAITING        H !T   $SPLP
$SPECL.#TYPE2                    WAITING        H !    $SPLP
\ARIES.$S                        WAITING        H      $SPLX
\VIRGO.$S                        WAITING        H      $SPLX
```

This report shows:

- Two print devices, $LP1 (which is printing job 159) and $LP2 (which is free, but has a form name established, so the spooler routes only jobs with the form name RED to $LP2)
- Two print processes, $SPLP and $SPLX
- A device named $SPECL that apparently performs at least two distinct services (selected by the specification #TYPE1 or #TYPE2, which the $SPECL process obtains from its open message), which is free
- The spooler is configured to enable you to send jobs to spooler collectors on two other systems— \ARIES and \VIRGO, which are free.

Suppose you have a 100-page report in the spooler and you want to print its last 3 pages on $LP1. First, use either PERUSE or the SPOOLCOM JOB command to get the job printing on $LP1. Then use this form of the DEV command:

```
DEV $LP1, SKIPTO 98
```

This command tells the spooler to skip forward to page 98 of the job. The skipping capability is related to the device, not to the job.

Skipping capability is useful is when the printer has jammed or the ribbon failed and ruined part of your listing. If the spooler delivered 56 pages of the job, but the pages from 50 onward are not usable, the job's owner (or the system operator) can take the printer offline, fix the mechanical problem, put the printer back online, and issue the SPOOLCOM command:

```
DEV $LP1, SKIPTO 49
```

For the full syntax of the JOB and DEV commands, see the *Guardian User's Guide*. You can often get the information you need from SPOOLCOM command HELP; for example, this command gives the syntax and capabilities of the JOB command:

```
HELP JOB
```

Controlling Vertical Spacing in a Printed Report

When you write to a printer or to a spooler, you can transmit not only data but also control information. Control information skips numbers of lines or skips to VFU channels. Some printers supplied by HP use a punched paper tape to control vertical formatting; others use an electronic device to perform that function.

The control information is not embedded in each print line but is transmitted separately from your program to the device or to the spooler collector. The spooler stores the information in a manner that enables it to generate the same image on paper as would have been generated if your program had been writing directly to a printer.

You control vertical spacing in a report with the ADVANCING clause of the sequential form of the WRITE statement.

If you do not use an ADVANCING clause, the WRITE statement behaves as if it contained the clause AFTER ADVANCING 1 LINE; that is, the printer prints the record in the WRITE statement after advancing one record.

Skipping Lines

To leave a blank line in your report before printing the record DETAIL-X, use this form of the WRITE statement:

```
WRITE DETAIL-X AFTER ADVANCING 2 LINES
```

To print the record DETAIL-Y and then leave a blank line, use this form of the WRITE statement:

```
WRITE DETAIL-Y BEFORE ADVANCING 1 LINE
```

Do not follow the preceding statement with another WRITE AFTER statement, or you will not get a blank line.

Overprinting Lines

To overprint the record named HEADER-X for emphasis, print it three times on the same line:

```
WRITE HEADER-X
WRITE HEADER-X AFTER ADVANCING 0 LINES
WRITE HEADER-X AFTER ADVANCING 0 LINES
```

This sequence also causes overprinting:

```
WRITE HEADER-X AFTER ADVANCING 0 LINES
WRITE HEADER-X BEFORE ADVANCING 0 LINES
```

To underline a line, print a string of underscore characters over it.

Skipping to a Printer Control Tape Channel Punch

If you are writing to a printer that has a VFU control feature, handled either by punched paper tape or an electronic equivalent, you can specify that the printer is to advance to a certain vertical position before (or after) the record delivered by the WRITE statement is written on the paper. To do this, you must:

- Know what channels are defined for the printer
- Use the SPECIAL-NAMES paragraph of the Environment Division to associate your own mnemonic name with the channel you plan to use
- Use a BEFORE or AFTER ADVANCING phrase having that mnemonic name

So, if you have a printer on which a skip to channel 6 puts you at line 50 of a page, your program must specify something like:

```
SPECIAL-NAMES.
   CHANNEL-6 IS CH-6.
```

Then the WRITE statement to print the contents of record SUBTOTAL-Z on line 50 of the current page is:

```
WRITE SUBTOTAL-Z AFTER ADVANCING CH-6
```

Skipping to a New Page

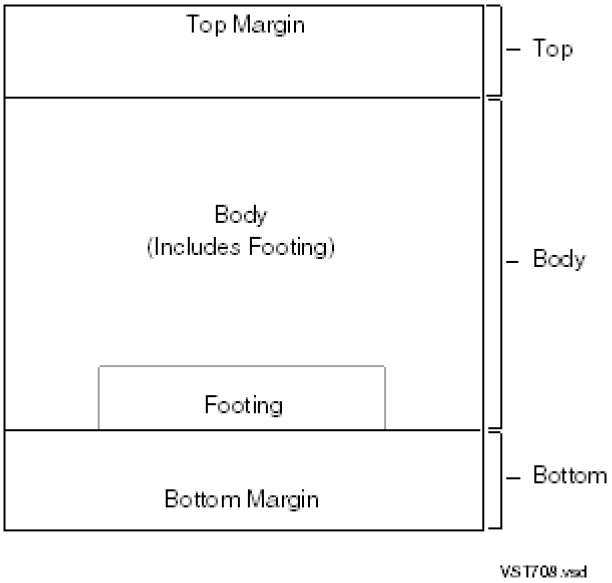
To have your program deliver a print line to the top of a new page, include an AFTER ADVANCING PAGE phrase in the WRITE statement. If you want the program to deliver a print line, then skip to the top of a page, include a BEFORE ADVANCING PAGE phrase in the WRITE statement.

Formatting Pages of a Printed Report

By using the LINAGE clause in the file description of a report file and by including the AT END-OF-PAGE phrase in each WRITE statement that adds lines to the report, you can produce attractive reports with little effort. This technique requires you to use a bit of structured programming.

The idea behind the LINAGE clause is that each page of the report has a top-margin area and a bottom-margin area, where the program writes no characters, and a body area, where the program does write characters. The body area ends with a footing area, made up of zero or more lines, where the program can write when it detects that the remainder of the body area is full. A typical use of this is for subtotals and totals for columns of data in the remainder of the body.

Figure 31-1 LINAGE Clause Layout



A standard 11-inch page printed at 6 lines per inch can accommodate 66 lines. You could establish 3 blank lines at the top, 3 blank lines at the bottom, and 60 lines of body between them. If you establish a footing line at 55, then the end-of-page condition arises after line 54 is written. If you want a one-line footing, use

```
FOOTING AT 66
```

This file description illustrates the LINAGE clause for this type of page:

```
FD SHOW-LINAGE
   LABEL RECORDS ARE OMITTED
   LINAGE IS 60 LINES
          WITH FOOTING AT 55
          LINES AT TOP 3
          LINES AT BOTTOM 3.
```

One tenet of structured programming is that there be one WRITE procedure for each output file; the program builds the line for delivery and then performs the write procedure. This situation is a perfect basis for using the LINAGE clause.

If your code writes each text line before advancing one line, you can use AT END-OF-PAGE phrase to detect that the program has already written the last line in the body of the report page. If the page needs subtotals or column footings, you can specify a number of footing lines in the LINAGE clause. This action causes the end-of-page condition to arise when the process attempts to write into the footing area. You can then use an AT END-OF-PAGE phrase in the main WRITE statement and specify that when the end-of-page condition arises, a footing-area procedure is to be performed. Example 31-4 illustrates this technique.

Example 31-4 END-OF-PAGE Phrase

```
WRITE-PROCEDURE SECTION.
...
   WRITE SHOW-LINAGE-REC BEFORE ADVANCING 1
         AT END-OF-PAGE PERFORM DO-FOOTING.
...
DO-FOOTING.
   MOVE SUMMATION-LINES TO SHOW-LINAGE-REC.
   WRITE SHOW-LINAGE-REC BEFORE ADVANCING 2.
   PERFORM COLLECT-SUBTOTALS-TO-PRINT.
   WRITE SHOW-LINAGE-REC BEFORE ADVANCING PAGE.
```

If you have trouble tracing the interactions of BEFORE phrases, AFTER phrases, numbers of lines to advance, page advances, and channel skips, you can always discover exactly what printer control commands are being used by following this procedure:

1. Direct your output to a spooler collector with a fictitious location.
2. Use the PERUSE command LIST to specify that the lines for at least two pages be listed to a printer or to the spooler collector in octal, showing all control codes:

   ```
   LIST /OUT $S.#LPF/ 1/2 O C
   ```

   This step lists each print line, each SETMODE operation, and each CONTROL operation.
3. See the *Spooler Programmer's Guide* to identify the codes in the SETMODE and CONTROL operations.

Logging Program Activity Information to a Printer

To record a log of program activity to a printer, you can use either of these statements:

- DISPLAY
- WRITE (usually preferable)

DISPLAY

If you use the DISPLAY statement to send messages to a printer for shared access, you must associate a mnemonic name with the printer device name (such as $LP) in the SPECIAL-NAMES paragraph. This association is established at compilation time. Each DISPLAY statement transmits characters to a single device—there is no mechanism for changing the destination of a DISPLAY statement.

Every time your process executes a DISPLAY statement, it must open the printer, write one line of characters, and close the printer. The printer is then available to other processes until the next time your process executes a DISPLAY statement. You cannot use the DISPLAY statement to deliver characters to a spooler.

In the OSS environment, if a DISPLAY statement includes `mnemonic-name`, it must be either the OSS pathname of a Guardian file or the name of an OSS text file.

WRITE

If you use the WRITE statement to send messages to a printer, you can determine the assignment between the file and the device:

- At compilation through the SELECT clause of the FILE-CONTROL paragraph
- At the beginning of execution with the TACL command ASSIGN or ADD DEFINE (in the Guardian environment only)
- During execution through the invocation of the routine COBOLASSIGN

The WRITE statement has these advantages over the DISPLAY statement:

- If there are several printers and the one you expected to use is busy, you can redirect the file to another printer.
- You can open the file for exclusive access (the default exclusion mode for printers) and be certain that no other process can send characters to the device while you are using it.
- You can recover from input-output errors by using declaratives.
- You can use the PARAM PRINTER-CONTROL command to handle the condition of the printer’s being out of paper (in the Guardian environment only).

32 Process Initiation, Communication, and Management

On a NonStop system, a process is a running program. More specifically, it is the unique executing entity created when someone runs object code from a loadfile in one of these ways:

- By entering an explicit RUN command (for example, RUN COBOL85)
- By entering an implicit RUN command (for example, COBOL85)
- By calling the PROCESS_CREATE_ procedure from another process

If a user runs two separate programs, the operating environment creates two processes. The operating environment also creates two processes if the user runs two instances of the same program concurrently, or if two users run the same program concurrently.

Physically, each process consists of at least:

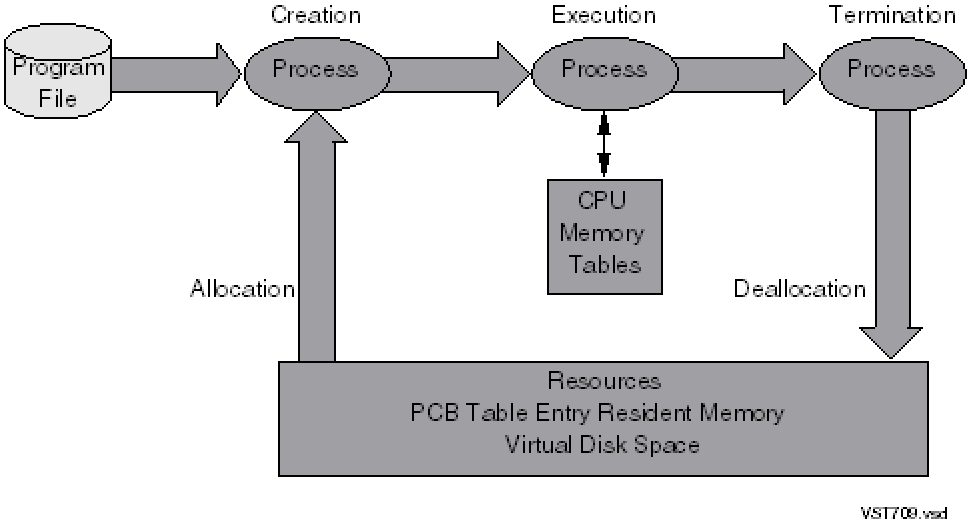
- A shareable, unmodifiable code area containing instructions and constants
- An exclusive, modifiable data area called a stack
- An exclusive entry in the process control block (PCB), a system table that uniquely defines the process within the system

This arrangement permits two users of the same loadfile to use only one code area. Although two processes can share a code area, each process has its own data area and PCB.

Although several processes in a particular processor module can share resources and attempt to run, only one process actually executes in a given processor module at a given instant. Actual process execution requires the use of the processor's hardware registers, which are allocated to the running process by the operating environment. Before the running process yields the processor module to another process or to an interrupt handler, the operating environment saves information about the current executing environment in the process's PCB entry. This strategy permits the operating environment to restore that environment when the process resumes execution.

A process comes into being when the operating environment takes the code and data produced by compilation and either binding or linking and combines them with the memory and other resources of the computer system. The process exists until it requests termination (or is terminated) and surrenders its resources.

Figure 32-1 Process Creation, Execution, and Termination



Memory and Virtual Memory

Memory is one resource consumed by a process. Each processor in an HP system has its own physical memory. Each processor in a system can have one to four increments of memory, and different processors in the same system can have different amounts of memory. The size of the increments depends on the type of processor—see the system description manual for your processor.

The operating environment allows several processes to occupy different areas of physical memory concurrently. It manages these processes with a virtual memory mechanism. The active portions of a process reside in physical memory. The inactive portions remain in physical memory only as long as the number of active processes remains small. As the number of active processes grows, the inactive portions of processes tend to remain on disk in virtual memory. The unit of memory allocation is the page, whose size depends on the processor. See the system description manual for your processor.

The virtual memory for process code consists of the loadfile.

The virtual memory for process data consists of temporary disk storage, allocated at the time the process is created.

A process can modify its data but not its code; therefore, the operating environment can fetch code pages needed from disk but does not need to restore them to disk; the operating environment must rewrite data pages to disk when their physical memory is surrendered.

The transfer of code and data pages between virtual memory on disk and physical memory is directed by the memory manager process of the operating environment with the help of the disk process.

The performance of any process is affected by the amount of time that is spent managing memory instead of executing process instructions.

For more information about memory and its management, see the system description manual for your processor.

Initiating a Process From an HP COBOL Program

Suppose you want to make your running HP COBOL program start a new process. This is different from having your HP COBOL program call another program with a CALL or ENTER statement—you want to cause an entirely independent process to execute asynchronously, perhaps on a different processor of the system or on a different HP system. Use the CLU_PROCESS_CREATE_ routine.

For information on the CLU_PROCESS_CREATE_ routine, see the *CRE Programmer's Guide*.

Communicating With a Process

On an HP system, all processes communicate directly with one another through the message system. The basic mechanism for access to the message system is $RECEIVE.

$RECEIVE

$RECEIVE is like one end of a communications conduit. It is similar to a courier who brings you a message and can be dismissed or sent back with a reply.

From the perspective of a process, $RECEIVE appears to be a sequential file. The syntax for communicating with $RECEIVE in an HP COBOL program is, therefore, based on the syntax for communicating with an ordinary HP COBOL sequential file.

To receive a message from another process (including the operating environment), a process must read $RECEIVE.

To reply to a message, a process must write to $RECEIVE. The operating environment delivers the reply to the originator of the message. The process does not need to determine where the message came from to reply to it.

As with any other read operation, a process waiting to receive a message on $RECEIVE waits until a message is delivered, the process is stopped, or a timeout occurs on the read operation.

Before a process can receive a message, some process must have sent a message to it.

To send a message, a process must have the name of the process that is to receive the message (see Process Names). The sending process opens a file having the same name as the receiving process (using an OPEN statement). Then the sending process can write to its file and cause the operating environment to deliver the message to the receiving process. The receiving process then can read the message on its $RECEIVE.

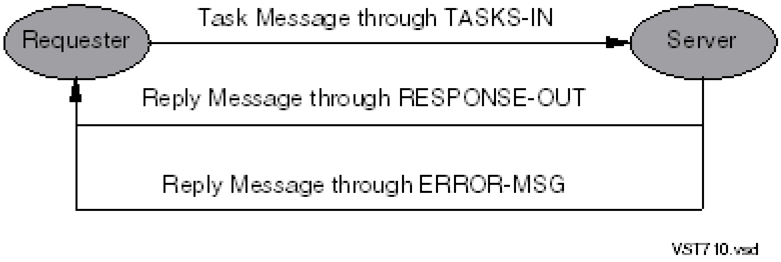
If a process tries to open a nonexistent process, both the operating environment and the HP COBOL run-time routines issue error messages. If no declarative intercepts the error, the process terminates.

The sender of a message specifies whether a reply is expected or not. An HP COBOL process uses a WRITE statement to send a message to which no reply is expected and a READ WITH PROMPT statement to send a message that requests a reply. The READ WITH PROMPT statement reads the reply to the message that is transmitted as the prompt. The READ WITH PROMPT statement lets you write a requester in HP COBOL, because it enables you to send a request to a server and await a reply from that server. For more information on READ WITH PROMPT, see PROMPT phrase.

$RECEIVE as Separate Input and Output Files

The most common way to use $RECEIVE is to open it in INPUT mode for one file, and in OUTPUT or EXTEND mode for one or more other files.

Figure 32-2 $RECEIVE as Separate Input and Output Files


Requester
Task Message through TASKS-IN
Server
Reply Message through RESPONSE-OUT
Reply Message through ERROR-MSG
VST710.vsd

All files are assigned to $RECEIVE; however, when testing your system, you can assign all files to disk files and then reassign them to $RECEIVE when testing is completed.

Example 32-1 Requester Code

```
       ...
    SELECT TRANSACTIONS ASSIGN TO "$BAL"
      ORGANIZATION IS SEQUENTIAL
      ACCESS MODE IS SEQUENTIAL....
  FD  TRANSACTIONS
      LABEL RECORDS ARE OMITTED
*               Variable-length records
      RECORD CONTAINS 1 TO 8 CHARACTERS.
  01  BALANCE              PICTURE 9(8).
  01  ERROR-ON-TASK.
      05  ERROR-CODE      PICTURE 9.
      05  ERR-MSG         PICTURE X(7).
  WORKING-STORAGE SECTION.
  01  TRAN-REQUEST.
      05  TRAN-CODE           PICTURE 9.
      05  TRAN-ACCOUNT-NUMBER PICTURE 9(6).
       ...
  PROCEDURE DIVISION....
      OPEN I-O TRANSACTIONS SYNCDEPTH 2....
      MOVE TASK-CODE  TO TRAN-CODE.
      MOVE ACCOUNT-IN TO TRAN-ACCOUNT-NUMBER.
      READ TRANSACTIONS RECORD
           WITH PROMPT TRAN-REQUEST...
```

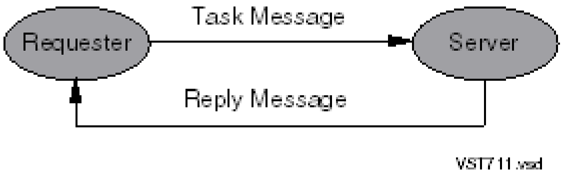
Example 32-2 Server Code

```
...
     SELECT TASKS-IN
       ASSIGN TO "$RECEIVE"
       ORGANIZATION IS SEQUENTIAL
       ACCESS MODE IS SEQUENTIAL.
     SELECT RESPONSE-OUT
       ASSIGN TO "$RECEIVE"
       ORGANIZATION IS SEQUENTIAL
       ACCESS MODE IS SEQUENTIAL.
     SELECT ERROR-MSG
       ASSIGN TO "$RECEIVE"
       ORGANIZATION IS SEQUENTIAL
       ACCESS MODE IS SEQUENTIAL.
  RECEIVE-CONTROL.
      TABLE OCCURS 20 TIMES
      SYNCDEPTH LIMIT IS 2
      REPLY CONTAINS 8 CHARACTERS.
       ...
 DATA DIVISION.
  FILE SECTION.
  FD  TASKS-IN
      LABEL RECORDS ARE OMITTED
      RECORD CONTAINS 7 CHARACTERS.
  01  TASK.
      05  TCODE            PICTURE 9.
      05  ACCOUNT          PICTURE 9(6).
  FD  RESPONSE-OUT
      LABEL RECORDS ARE OMITTED
      RECORD CONTAINS 8 CHARACTERS.
  01  BALANCE              PICTURE 9(8).
  FD  ERROR-MSG
      LABEL RECORDS ARE OMITTED
      RECORD CONTAINS 8 CHARACTERS.
  01  ERROR-ON-TASK.
      05  ERROR-CODE      PICTURE 9.
      05  ERR-MSG         PICTURE X(7).
  PROCEDURE DIVISION....
     OPEN INPUT TASKS-IN.
     OPEN OUTPUT RESPONSE-OUT
                 ERROR-MSG....
  READ-A-TASK.
     READ TASKS-IN
        AT END CLOSE TASKS-IN
               OPEN INPUT TASKS-IN
               GO TO READ-A-TASK.
     IF TCODE = 1
         PERFORM ...
     IF WS-BALANCE > 0
         MOVE WS-BALANCE TO BALANCE
         WRITE BALANCE
     ELSE
        MOVE WS-ERR-CODE TO ERROR-CODE
        MOVE MESSAGE(WS-ERR-CODE) TO ERR-MSG
        WRITE ERROR-ON-TASK....
```

$RECEIVE as Input-Output File

In this case, $RECEIVE is opened in I-O mode to receive requests and reply to them through the same file. Each request is acted upon and paired with a reply message sent back in response to the task.

Figure 32-3 $RECEIVE as Input/Output File



The requesting process follows the outline in Example 32-3.

Example 32-3 Requester Code

```
     ...
SELECT TRANSACTIONS ASSIGN TO "$BAL"
   ORGANIZATION IS SEQUENTIAL
   ACCESS MODE IS SEQUENTIAL.

     ...
FD   TRANSACTIONS
     LABEL RECORDS ARE OMITTED
     RECORD CONTAINS 1 TO 8 CHARACTERS.

01   TRAN-RESPONSE.
     05   RESULT-CODE    PICTURE 9.
          88 RESULT-OK      VALUE IS 1.
          88 RESULT-ERROR   VALUE IS 2....
     05   NEW-BALANCE    PICTURE 9(7)....

WORKING-STORAGE SECTION....

01   TRAN-REQUEST.
     05   TRAN-CODE      PICTURE 9.
          88 TRAN-PMT    VALUE IS 1....
     05   ACCOUNT-NUMBER PICTURE 9(6)....

PROCEDURE DIVISION.
     ...
   OPEN I-O TRANSACTIONS SYNCDEPTH 1....
   MOVE TCODE TO TRAN-CODE.
   MOVE ACCOUNT-IN TO ACCOUNT-NUMBER.
   READ TRANSACTIONS WITH PROMPT TRAN-REQUEST....
```

The server process receives the account number and transaction code from $RECEIVE and sends a response back to the requester, as Example 32-4 shows.

Example 32-4 Server Code

```
INPUT-OUTPUT SECTION.
FILE-CONTROL.
    ...
    SELECT TASKS-IN ASSIGN TO "$RECEIVE"
      ORGANIZATION IS SEQUENTIAL
      ACCESS MODE IS SEQUENTIAL.
    ...
RECEIVE-CONTROL.
```

```
    TABLE OCCURS 20 TIMES
    SYNCDEPTH LIMIT IS 1
    REPLY CONTAINS 8 CHARACTERS....
DATA DIVISION.
    ...
FD  TASKS-IN
    LABEL RECORDS ARE OMITTED
    RECORD CONTAINS 1 TO 8 CHARACTERS.
  01  TRAN-REQUEST.
      05  TRAN-CODE      PICTURE 9.
          88 TRAN-PMT    VALUE IS 1.
    ...
      05  ACCOUNT-NUMBER PICTURE 9(6).
    ...
  01  TRAN-RESPONSE.
      05  RESULT-CODE    PICTURE 9.
          88 RESULT-OK      VALUE IS 1.
          88 RESULT-ERROR   VALUE IS 2....
      05  NEW-BALANCE    PICTURE 9(7)....
PROCEDURE DIVISION....
   OPEN I-O TASKS-IN....
   READ TASKS-IN.
   IF TRAN-PMT
      PERFORM ...
      IF SOME-ERROR
         MOVE 2 TO RESULT-CODE
      ELSE MOVE 1 TO RESULT-CODE
           MOVE WS-NEW-BALANCE TO NEW-BALANCE
           WRITE TRAN-RESPONSE
   ELSE...
```

Process Names

A process that expects to receive messages must have a name by which the sending process can designate it.

All processes running on an HP system, named and unnamed, have process file names. If a process is not named explicitly—with the RUNNAMED directive, the NAME option of the RUN command, or the `name : length` parameter of the PROCESS_CREATE_ procedure—the operating environment issues it with a process file name to an unnamed process.

A file name for an unnamed process consists of an optional node name, a dollar sign ($), a processor and PIN, and a process sequence number.

A file name for a named process consists of an optional node name, a process name (whose first character is a dollar sign ($)), an optional sequence number, and optional qualifiers.

To find the names of processes currently active, use the TACL command PPD. NonStop systems maintain the list of process names in a destination control table (DCT). The systems recognize the PPD command as a request for a tabulation of information about named processes, not unnamed processes.

Example 32-5 Report Produced by PPD Command

```
NAME     PID1     PID2          ANCESTOR
$Z000    00,022   01,021            00,021
$CRT2    00,021   01,022            $Z000
$NULL    07,004   00,024            00,025
$CMON    07,005   00,026            00,025
$IMON    00,031   01,025            06,010
$DM00    00,032                     $IMON
$DM01    01,024                     $IMON
...
```

The report in Example 32-5 shows that the process-pair named $Z000 consists of a primary process running in processor module 00 with the process identification number (PIN) 022 and a backup process running in processor 01 with the PIN 021. The processor and PIN of the process that started $Z000 (its ancestor) is 00,021. Notice that not all processes are executing as process pairs—$DM00 and $DM01 (parts of the Inspect symbolic debugger) are running as single processes.

Every disk volume name, device name, spooler collector name, and so on, begins with a dollar sign. Each time you read from or write to one of them, you are reading from or writing to the process controlling that entity.

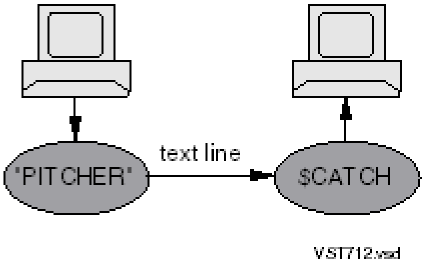
Example of Simple Interprocess Communication

This example illustrates the principles of interprocess communication using named processes and $RECEIVE (see Figure 32-4).

The program PITCHER accepts a line of text from its home terminal and transmits it to the process $CATCH (the program named CATCHER). When it receives a line beginning with the value “END,” PITCHER terminates after transmitting the line.

CATCHER displays each line on its home terminal. When the process that opened CATCHER terminates, an HP COBOL run-time routine translates the closure as an end of file (EOF) on $RECEIVE. The AT END phrase of the READ statement passes control around the DISPLAY statement, the PERFORM terminates because its UNTIL phrase is satisfied, and CATCHER terminates. (Use of an empty declarative would eliminate the need for the END-TRANS paragraph and the AT END GO TO construct.)

Figure 32-4 $RECEIVE From PITCHER to CATCHER



The **bold** text in Example 32-6 is significant. Note that, in PITCHER, the file for CATCHER is opened for I-O even though it is being used only for output. If you open a process with the OUTPUT or EXTEND attribute, HP COBOL handles the process as a printer device and tries to advance it to the next page—an operation that is meaningless (and fatal) to CATCHER.

Example 32-6 PITCHER Code

```
IDENTIFICATION DIVISION.
   PROGRAM-ID.    PITCHER.
   AUTHOR.        JO COBOL.
   INSTALLATION.  TRANSACTIONS ANONYMOUS.
   DATE-WRITTEN.  29 FEBRUARY 1988.
   DATE-COMPILED.

*************************************************************
* This program illustrates the transmission of messages to  *
* another COBOL program, CATCHER, whose process name is     *
* $CATCH.                                                   *
*                                                           *
* Records entered at this process's home terminal are sent  *
* $CATCH.                                                   *
*************************************************************

ENVIRONMENT DIVISION.
   CONFIGURATION SECTION.
     SOURCE-COMPUTER.  HP TXP.
     OBJECT-COMPUTER.  HP TXP.
   INPUT-OUTPUT SECTION.
     FILE-CONTROL.
       SELECT REPORT-OUT
              ASSIGN TO "$CATCH"
              ORGANIZATION IS SEQUENTIAL
              ACCESS MODE IS SEQUENTIAL.

 DATA DIVISION.
   FILE SECTION.
     FD REPORT-OUT
        LABEL RECORDS ARE OMITTED.
     01 REPORT-LINE.
        05 REPORT-END     PICTURE X(3).
           88 LAST-LINE-ARRIVED VALUE "END".
        05 REPORT-REST    PICTURE X(77).
 PROCEDURE DIVISION.
 A.
    OPEN I-O REPORT-OUT.
    PERFORM TRANSPUT UNTIL LAST-LINE-ARRIVED.
    STOP RUN.
 TRANSPUT.
    ACCEPT REPORT-LINE.
    WRITE REPORT-LINE.
```

Example 32-7 CATCHER Code

```
 IDENTIFICATION DIVISION.
   PROGRAM-ID.    CATCHER.
   AUTHOR.        SANDY COBOL.
   INSTALLATION.  TRANSACTIONS ANONYMOUS.
   DATE-WRITTEN.  29 FEBRUARY 1988.
   DATE-COMPILED.

*********************************************************

* This program illustrates the receipt of messages from *
* another COBOL program, PITCHER.                       *
*                                                       *
* This program must be run as a process named $CATCH.   *
*                                                       *
* Records received are displayed on this program's home *
* terminal.                                             *
*********************************************************

 ENVIRONMENT DIVISION.
   CONFIGURATION SECTION.
     SOURCE-COMPUTER.  HP TXP.
     OBJECT-COMPUTER.  HP TXP.
```

```
  INPUT-OUTPUT SECTION.
   FILE-CONTROL.
    SELECT PRINT-LINES-IN
           ASSIGN TO "$RECEIVE"
           ORGANIZATION IS SEQUENTIAL
           ACCESS MODE IS SEQUENTIAL
           FILE STATUS IS PRINT-STATUS.

 DATA DIVISION.
  FILE SECTION.
  FD PRINT-LINES-IN
     LABEL RECORDS ARE OMITTED.
  01 LINE-IN           PICTURE X(80).
 WORKING-STORAGE SECTION.
  01 PRINT-STATUS      PICTURE X(2).
     88 SENDER-DISAPPEARED VALUE "10".

 PROCEDURE DIVISION.
 A.
    OPEN INPUT PRINT-LINES-IN.
    PERFORM TRANSPUT THROUGH END-TRANS
            UNTIL SENDER-DISAPPEARED.
    STOP RUN.
 TRANSPUT.
    READ PRINT-LINES-IN
         AT END GO TO END-TRANS.
    DISPLAY LINE-IN.
 END-TRANS.
    EXIT.
```

RECEIVE-CONTROL Paragraph

The RECEIVE-CONTROL paragraph of the Environment Division is an HP extension to COBOL. The RECEIVE-CONTROL paragraph in process $XX serves these purposes:

- It defines the receive-control table for process $XX. This specifies the maximum number of other processes that can have process $XX open concurrently. When all such other processes close process $XX, an HP COBOL run-time routine reports an end of file (EOF) on $RECEIVE.
- It defines the reply table for process $XX. This specifies the number of replies (and the length of a reply) to be saved for each requesting process. A process with a fault-tolerant requester must allow for replies to be saved so that the fault-tolerant facility can retransmit them to restore synchronization in the case of a takeover by the requester's backup process. (If the requester is not fault-tolerant, the reply table is useless and wastes space.)
- It designates a data item to contain an error code to be returned to a requesting process that is handling process $XX as a device. Every device in the system returns an error code, a condition code, and a reply-message text.
- It designates a data item to contain the message source descriptor. This fixed-format item is defined in RECEIVE-CONTROL Paragraph (page 155). When the HP COBOL run-time routines complete a successful READ on $RECEIVE, they update this data item to report:
  - — The message source (the operating environment or another user process)
  - — The entry number in the receive-control table
  - — The process ID of the requesting process (name, processor number, and number of process in that processor)

NOTE: If the process is running at a high PIN, the process ID that the operating environment returns is synthetic. For information about synthetic IDs of high-PIN processes, see the *Guardian Application Conversion Guide.*

- It specifies which classes of operating environment messages are to be passed to process $XX.

An ordinary Pathway server uses only three entries of the RECEIVE-CONTROL paragraph:

- TABLE OCCURS

  Specifies how many requesters can have the server open at one time. Any additional requesters attempting to open the server are refused. This value must be greater than or equal to the MAXLINKS value in the PATHCOM command file that created the PATHMON environment. A large value is recommended.

- SYNCDEPTH

  Usually set to 1 for Pathway servers unless they are multithreaded.

- REPLY CONTAINS

  Specifies the length of the reply that the server sends back to the requester. It specifies the name of the file containing the longest record that is used as a reply or an explicit number of characters.

At-End Condition

An entry is made in the receive-control table when a requester executes an OPEN for a file assigned to the server process. A requester can have more than one OPEN issued to a server process at any given time. The receive-control table has a separate entry for each of these OPENs.

An entry is deleted from the receive-control table whenever the requester issues a CLOSE for the file assigned to the server process. A server continues to receive requester messages through $RECEIVE if there is an entry in the table. When the last entry is deleted, an at-end condition arises for the server's READ statement.

This at-end condition is handled as an end of file for $RECEIVE. If a file status code data item is defined, it is set to "10" (EOF). Control passes to the statement in the AT END phrase or to a USE procedure if no AT END phrase is present. If the server is a Pathway server, it must stop itself when it detects an at-end condition.

An attempt to read a file when it is at end of file causes a permanent error, setting the file status code to "30." Ordinarily, the program must close and reopen the file before any further activity is possible on that file. For $RECEIVE, however, the run-time routines simulate a close/open sequence, relieving the program of this responsibility; however, such a READ after end of file should include a TIME LIMIT phrase, or the READ could wait indefinitely. See READ for Sequential or Dynamic Access (page 414).

When a server process has just begun execution and no entries are yet in its receive-control table, an at-end condition will never occur before the first OPEN message is received.

Summary of $RECEIVE Rules

These rules apply to programs that use $RECEIVE:

- Files assigned to $RECEIVE must be sequentially organized and not described with alternate keys or LINAGE clauses.
- A RECEIVE-CONTROL paragraph in the Input-Output Section of the Environment Division is necessary to define two internal tables essential to the function of $RECEIVE. (Although the RECEIVE-CONTROL paragraph is optional, the default tables permit only the most limited use of $RECEIVE.)

- The OPEN SYNCDEPTH value in the requester process that opens the server process must not exceed the sync value in the RECEIVE-CONTROL paragraph of the server process.
- Only one reply WRITE to each READ on $RECEIVE is allowed. If you fail to reply to a message before you read the next one, the HP COBOL run-time routines send a default reply. In this situation, you cannot reply to that earlier message.
- A program can use $RECEIVE whether or not the program runs as a process pair.
- The contents of the record area associated with $RECEIVE are subject to modification by the HP COBOL run-time routines. Even if a process has not requested delivery of system messages in the RECEIVE-CONTROL paragraph, the run-time routines handle system messages in the $RECEIVE record area. These unrequested messages are not delivered to the process. If a process modifies the record area and then performs a read operation on $RECEIVE, the portion of the record area beyond the actual received message can include residue of such system messages.
- If you want to know the length of the record you read from $RECEIVE, identify them with RECORD IS VARYING... DEPENDING, which enables you to determine the length of the message that was sent to $RECEIVE by the requester.
- If the ASSIGN clause specifies $RECEIVE, the upper limit of the RECORD CONTAINS clause is 2MB and the maximum size of a logical record for the file is 2MB. If the ASSIGN clause specifies a define name, even a define that translates to $RECEIVE, the maximum record size is limited to 32KB. If the file is not associated with $RECEIVE until runtime (for example, by a TACL ASSIGN command or by a call to COBOL_ASSIGN_), the maximum record size of the file is 32KB only.

Simple Server Example

The simple server in Example 32-8 looks up items in a telephone directory.

Example 32-8 Simple Server

```
 IDENTIFICATION DIVISION.
   PROGRAM-ID.   BRIEF-EXAMPLE.
   AUTHOR.       ZANE COBOL.
   DATE-WRITTEN. 29 February, 1988
   DATE-COMPILED.

**************************************************************************
* This simple server performs telephone-book lookups.                    *
* Given a name, it reports the name and a number.                        *
* Given a number, it reports the number and a name.                      *
* It assumes that no two persons have the same number or the same name.  *
**************************************************************************

 ENVIRONMENT DIVISION.
   CONFIGURATION SECTION.
     SOURCE-COMPUTER. HP TXP System.
     OBJECT-COMPUTER. HP TXP System.

   INPUT-OUTPUT SECTION.
     FILE-CONTROL.

*      Declaring MESSAGE-IN and MESSAGE-OUT separately allows us to
*      redirect either or both for debugging during development.

       SELECT MESSAGE-IN
          ASSIGN TO $RECEIVE
       FILE STATUS IS RECEIVE-FILE-STATUS.

       SELECT MESSAGE-OUT
       ASSIGN TO $RECEIVE
       FILE STATUS IS RECEIVE-FILE-STATUS.
       SELECT PHONE-BOOK
         ASSIGN TO "$AA.BB.PHONE"
         ORGANIZATION         IS INDEXED
         ACCESS MODE          IS DYNAMIC
         RECORD KEY           IS PH-EMPLOYEE-NUM
         ALTERNATE RECORD KEY IS PH-EMPLOYEE-NAME
         ALTERNATE RECORD KEY IS PH-EMPLOYEE-PHONE-NUM
          FILE STATUS          IS PHONE-FILE-STATUS.

    RECEIVE-CONTROL.

*     Up to five requesters can have this server open

      TABLE OCCURS 5 TIMES
      SYNCDEPTH LIMIT IS 1
      REPLY CONTAINS MESSAGE-OUT RECORD.

 DATA DIVISION.
 FILE SECTION.
 FD  MESSAGE-IN

     LABEL RECORDS ARE OMITTED.
 01  PHONE-QUERY-MSG.
     05 FILLER                    PIC S9(4) COMP.
     05 LOOKUP-CODE               PIC X.
        88 NAME-LOOKUP                      VALUE "N".
        88 PHONE-LOOKUP                     VALUE "#".
     05 LOOKUP-NUM                PIC X(10).
     05 LOOKUP-NAME               PIC X(35).

 FD  MESSAGE-OUT
     LABEL RECORDS ARE OMITTED
*     record is variable length.
     RECORD CONTAINS 1 TO 200 CHARACTERS.

 01  PHONE-REPLY-MSG.
     05 REPLY-CODE                PIC S9(4) COMP.
*           000 = OK, reply contains a value
*           900 = no match in phone book
     05 LOOKUP-CODE               PIC X.
        88 NAME-LOOKUP                      VALUE "N".
        88 PHONE-LOOKUP                     VALUE "#".
     05 LOOKUP-NUM                PIC X(10).
     05 LOOKUP-NAME               PIC X(35).

 01  BAD-PHONE-REPLY-MSG.
     05 BAD-REPLY-CODE            PIC S9(4) COMP.
*           001 = bad request, no code given
*           002 = start of lookup failed

  FD PHONE-BOOK
     LABEL RECORDS ARE OMITTED.
  01 PHONE-REC.
     05 PH-EMPLOYEE-NUM           PIC X(5).
     05 PH-EMPLOYEE-NAME.
        10 PH-EMPLOYEE-NAME-LAST  PIC X(15).
        10 PH-EMPLOYEE-NAME-REST  PIC X(20).
     05 PH-EMPLOYEE-PHONE-NUM     PIC X(10).

 WORKING-STORAGE SECTION.

 01 REPLY-CODE-VALUES.
    05 OK-REPLY           PIC 999 VALUE ZERO.
    05 BAD-REQUEST-REPLY  PIC 999 VALUE 1.
    05 START-FAILED-REPLY PIC 999 VALUE 2.
    05 NONE-FOUND-REPLY   PIC 999 VALUE 900.
 01 FILE-STATUSES.
    05 RECEIVE-FILE-STATUS.
       88 RECEIVE-FILE-OK                VALUE IS ZEROS.
       88 CLOSE-FROM-REQUESTER           VALUE IS "10".
       10 R-STAT-1                PIC X.
       10 R-STAT-2                PIC X.
    05 PHONE-FILE-STATUS.
       10 P-STAT-1                PIC X.
       10 P-STAT-2                PIC X.
    88 PHONE-FILE-OK                     VALUE IS ZEROS.

PROCEDURE DIVISION.

 DECLARATIVES.

*****************************************************************
   UA-MESSAGE-IN  SECTION.
      USE AFTER ERROR PROCEDURE ON MESSAGE-IN.
   UA-MESSAGE-IN-PROC.
      IF R-STAT-1 > 1
         DISPLAY " ERROR ON MESSAGE-IN FILE "
                 " STAT-1 = " R-STAT-1
                 " STAT-2 = " R-STAT-2.
*****************************************************************
   UA-PHONE-BOOK  SECTION.
      USE AFTER ERROR PROCEDURE ON PHONE-BOOK.
   UA-PHONE-BOOK-PROC.
*  Empty declarative--intercepts error conditions and allows
*  PHONE-FILE-STATUS to reflect the success of a START or READ
*****************************************************************
   UA-MESSAGE-OUT  SECTION.
      USE AFTER ERROR PROCEDURE ON MESSAGE-OUT.
   UA-MESSAGE-OUT-PROC.
      IF R-STAT-1 > 1
         DISPLAY " ERROR ON MESSAGE-OUT FILE "
                 " STAT-1 = " R-STAT-1

                 " STAT-2 = " R-STAT-2.
*****************************************************************
 END DECLARATIVES.
*****************************************************************
MAIN SECTION.
************************************
*   M  A  I  N      L  O  G  I  C  *
************************************

 BEGIN-COBOL-SERVER.
    PERFORM A-INIT.
    PERFORM B-TRANS
            UNTIL CLOSE-FROM-REQUESTER.
    PERFORM C-EOJ.
    STOP RUN.
****************************************************************
 A-INIT.
    OPEN INPUT  MESSAGE-IN          SYNCDEPTH 1
         OUTPUT MESSAGE-OUT
         INPUT  PHONE-BOOK  SHARED.
****************************************************************
 C-EOJ.
    CLOSE MESSAGE-IN
          MESSAGE-OUT
          PHONE-BOOK.
 B-TRANS.
    READ MESSAGE-IN.
*   Declarative handles EOF
    IF RECEIVE-FILE-STATUS = "00"
       IF NAME-LOOKUP OF MESSAGE-IN
          PERFORM B-01-NAME-LOOKUP-PROC
       ELSE
          IF PHONE-LOOKUP OF MESSAGE-IN
             PERFORM B-01-PHONE-LOOKUP-PROC
          ELSE
             MOVE SPACES TO PHONE-REPLY-MSG
             MOVE BAD-REQUEST-REPLY TO BAD-REPLY-CODE OF MESSAGE-OUT
    ...
    IF RECEIVE-FILE-STATUS = "00"
       WRITE PHONE-REPLY-MSG
    ELSE
       WRITE BAD-REPLY-MSG.
 B-01-NAME-LOOKUP-PROC.
    MOVE LOOKUP-NAME OF MESSAGE-IN TO PH-EMPLOYEE-NAME
    START PHONE-BOOK KEY IS = PH-EMPLOYEE-NAME.
    IF NOT PHONE-FILE-OK
       MOVE START-FAILED-REPLY TO REPLY-CODE OF MESSAGE-OUT
    ELSE
       READ PHONE-BOOK
       IF PHONE-FILE-OK
          PERFORM B-02-COPY-TO-REPLY
       ELSE
          MOVE NONE-FOUND-REPLY TO REPLY-CODE OF MESSAGE-OUT
    ...
 B-01-PHONE-LOOKUP-PROC.

    MOVE LOOKUP-NUM OF MESSAGE-IN TO PH-EMPLOYEE-PHONE-NUM.
    START PHONE-BOOK KEY IS = PH-EMPLOYEE-PHONE-NUM.
    IF NOT PHONE-FILE-OK
       MOVE NONE-FOUND-REPLY TO REPLY-CODE OF MESSAGE-OUT
    ELSE
       READ PHONE-BOOK
       IF PHONE-FILE-OK
          PERFORM B-02-COPY-TO-REPLY
       ELSE
          MOVE NONE-FOUND-REPLY TO REPLY-CODE OF MESSAGE-OUT
    ...
 B-02-COPY-TO-REPLY.
    MOVE PH-EMPLOYEE-PHONE-NUM TO LOOKUP-NUM OF MESSAGE-OUT.
    MOVE PH-EMPLOYEE-NAME      TO LOOKUP-NAME OF MESSAGE-OUT.
    MOVE OK-REPLY              TO REPLY-CODE OF MESSAGE-OUT.
```

More information:

Topics	Sources
Programming servers and Pathsend requesters in the Pathway environment	*TS/MP Pathsend and Server Programming Manual*
Programming SCREEN COBOL requesters in the Pathway environment	*Pathway/TS TCP and Terminal Programming Guide*
Configuring and managing servers in the Pathway environment	*TS/MP System Management Manual*

Managing a Process

Besides creating processes and communicating with processes, a process might need to obtain information about itself or about some other process in the same system or another system on the same Expand network.

Determining the Process Handle

If a process has a record of its process handle—process name, processor number, and number of the process within that processor (also called process identification number or PIN)—it can:

- Generate more useful diagnostic messages by including the process handle information and the time and date
- Use its knowledge of its location (processor number) in creating other processes with which it must communicate
- Obtain information from the operating system routines that require the processor number and PIN as parameters

A process can obtain its process ID by passing its process number to the PROCESSHANDLE_DECOMPOSE_ procedure, which returns the processor and PIN values as separate integer values. If you do not know the process number, you can get it by calling the PROCESSHANDLE_GETMINE_ procedure.

Example 32-9 PROCESSHANDLE_GETMINE_ and PROCESSHANDLE_DECOMPOSE_ Procedures

```
WORKING-STORAGE SECTION.
01   PROCESS-HANDLE       PIC X(20).
01   CPU-PIN.
    05   CPU              PIC S9(2) COMPUTATIONAL.
    05   PIN              NATIVE-2.
01   ERROR-NUMBER         PIC S9(5) COMPUTATIONAL.
01   NULL-PH              PIC X(20) VALUE ALL HIGH-VALUES....
PROCEDURE DIVISION....
ENTER TAL "PROCESSHANDLE_GETMINE_"
                      USING PROCESS-HANDLE
                      GIVING ERROR-NUMBER
ENTER TAL "PROCESSHANDLE_DECOMPOSE_"
                      USING PROCESS-HANDLE
                      CPU
                      PIN
                      GIVING ERROR-NUMBER
```

Determining the Node (System Number)

When NonStop systems are linked together through Expand to constitute a network, each system (or node) of the network has a system number. If copies of a process are running on different nodes, the process might need to determine the system number of the system on which it is running. The process might report diagnostic messages to a central log and include its system name, number, or both.

A process can obtain the system number of the system on which it is running by calling the routines PROCESSHANDLE_GETMINE_, PROCESSHANDLE_DECOMPOSE_, and NODENUMBER_TO_NODENAME_, as Example 32-10 shows.

Example 32-10 Determining a Process's Node (System Number)

```
 IDENTIFICATION DIVISION.
   PROGRAM-ID.     WHERE-AM-I.
   AUTHOR.         BECK COBOL.
   INSTALLATION.   TRANSACTIONS ANONYMOUS.
   DATE-WRITTEN.   29 FEBRUARY 1988.
   DATE-COMPILED.
```

```
*************************************************************
*  This program obtains and reports its own system number   *
*  and system name.                                         *
*************************************************************
 ENVIRONMENT DIVISION.
   CONFIGURATION SECTION.
     SOURCE-COMPUTER.  HP TXP.
     OBJECT-COMPUTER.  HP TXP.
 DATA DIVISION.
   WORKING-STORAGE SECTION.
     01 PROCESS-HANDLE       PIC X(20).
     01 CPU-PIN.
        05 CPU               NATIVE-2.
        05 PIN               NATIVE-2.
     01 ERROR-NUMBER         PIC S9(5) COMP.
     01 SYSTEM-NAME          PIC X(8).
     01 SYSTEM-NUMBER        NATIVE-2.
     01 SYSTEM-NAME-LENGTH   NATIVE-2.
 PROCEDURE DIVISION.
 WHERE-AM-I.
    ENTER TAL "PROCESSHANDLE_GETMINE_"
          USING PROCESS-HANDLE
          GIVING ERROR-NUMBER.
    ENTER TAL "PROCESSHANDLE_DECOMPOSE_"
               USING PROCESS-HANDLE
                     CPU
                     PIN
               GIVING ERROR-NUMBER.
    ENTER TAL "NODENUMBER_TO_NODENAME_"
               USING OMITTED
                     SYSTEM-NAME
                     SYSTEM-NAME-LENGTH
               GIVING ERROR-NUMBER.
    DISPLAY "I'm executing on system #"
            SYSTEM-NUMBER
            ", named """
            SYSTEM-NAME
            """".
    STOP RUN.
```

Identifying the Message Source

To enable your process to determine the source of any message it receives, include a MESSAGE SOURCE clause in the program's RECEIVE-CONTROL paragraph. This causes the HP COBOL run-time routines to automatically update the data item designated in the MESSAGE SOURCE clause at every successful READ on $RECEIVE.

The capability of identifying the source of a message allows a process to respond differently to the same request coming from different processes or even to reject certain requests from certain processes.

Example 32-11 uses the MESSAGE SOURCE clause to acquire the message-source record. It reads messages on $RECEIVE and displays the processor/PIN, process name, and message code of the message, then displays the message. WHO-SENT-THAT detects whether the sender has an explicit process name. If the sender had an explicit process name, the program determines the processor number and PIN and displays them and the process name.

Example 32-11 MESSAGE-SOURCE Clause

```
DATA DIVISION.
FILE SECTION.
   FD MESSAGE-IN-FILE
      RECORD CONTAINS 1 TO 82 CHARACTERS
      LABEL RECORDS ARE OMITTED.
```

```
01 MESSAGE-IN.
   05 SYS-MSG-CODE      PIC S9(4) COMP.
   05 SYS-MSG-TEXT      PIC X(80).
WORKING-STORAGE SECTION.
01 MESSAGE-SOURCE-REC.
   05 SYSTEM-FLAG       PIC S9  COMP.
   05 ENTRY-NUMBER      PIC 999 COMP.
   05 FILLER            PIC X(4).
   05 PROCESS-HANDLE    PIC X(20).
   05 CPU-PIN.
      06  CPU          PIC S9(2) COMPUTATIONAL.
      06  PIN              NATIVE-2.
   05 ERROR-NUMBER      PIC S9(5) COMPUTATIONAL.
   05 NULL-PH           PIC X(20) VALUE ALL HIGH-VALUES.
01 FILE-DATA.
   05 RECEIVE-FILE-STATUS PIC XX.
      88 RECEIVE-FILE-OK          VALUE "00".
      88 RECEIVE-FILE-EOF         VALUE "10".
01 PROCESS-NAME       PIC X(8).
01 PROCESS-NAME-LEN   NATIVE-2.
 PROCEDURE DIVISION.
 DECLARATIVES.
*  --Declaratives are a more powerful way to handle file
*  --errors than a simple AT END phrase, and one or the other
*  --is required to be present.
    HANDLE-INFILE-ERRORS SECTION.
       USE AFTER STANDARD ERROR PROCEDURE ON MESSAGE-IN-FILE.
    INFILE-ERROR.
       IF NOT RECEIVE-FILE-EOF
          DISPLAY "RECEIVE FILE ERROR STATUS = " RECEIVE-FILE-STATUS .
 END DECLARATIVES.
AA SECTION.
 AA-1.
    OPEN INPUT MESSAGE-IN-FILE.
    MOVE ZERO TO SYS-MSG-CODE
                 RECEIVE-FILE-STATUS.
    PERFORM WATCH
       UNTIL NOT RECEIVE-FILE-OK.
    STOP RUN.
 WATCH.
    READ MESSAGE-IN-FILE.
*    The READ causes the MESSAGE-SOURCE-REC to be set.
    PERFORM CHECK-STATUS.
 CHECK-STATUS.
    MOVE SPACES TO PROCESS-NAME.
    ENTER TAL "PROCESSHANDLE_DECOMPOSE_"
                          USING PROCESS-HANDLE
                          CPU
                          PIN
* No node number
                          OMITTED
* No node name
                          OMITTED
* No nodename length
                          OMITTED
                          PROCESS-NAME
                          PROCESS-NAME-LEN
                     GIVING ERROR-NUMBER.
    IF ERROR-NUMBER EQUAL TO 0
       PERFORM DISPLAY-PROCESS-NAME
    ELSE
       DISPLAY "ERROR " ERROR-NUMBER.
 DISPLAY-PROCESS-NAME.
    DISPLAY "MESSAGE WAS FROM CPU "  CPU  ", PIN " PIN.
    IF PROCESS-NAME-LEN > 0
```

```
       DISPLAY "PROCESS " PROCESS-NAME.
    DISPLAY "MESSAGE CODE " SYS-MSG-CODE.
    DISPLAY "MESSAGE CONTENTS: " SYS-MSG-TEXT.
```

Determining the Status

Using the ENTER verb, a process can call the PROCESS_GETINFO_ routine to obtain information about any process. The PROCESS_GETINFO_ routine can return any or all of:

- The process name, processor number, and number of the process within that processor
- The creator accessor ID (group ID, user ID) of the process
- The process accessor ID (group ID, user ID) of the process
- The execution priority at which the process is running
- The name of the loadfile from which the process was loaded
- The device name of the process's home terminal
- The system number (in a network of systems) on which the process is running
- An error value, indicating the success of (or the nature of the failure of) the request

Accessor IDs

The creator accessor ID and process accessor ID are elements of the security system of the operating environment. Every user has a unique user name and a corresponding unique user ID. A user name is of the form:

groupname.username

where *groupname* is the name of the group to which the user belongs, and *username* is a name identifying the individual user within the group. This is the same user name you use when you log on. A user ID is the numeric equivalent of the user name and is of the form:

group-id, user-id

where *group-id* and *user-id* are nonnegative integers in the range 0 through 255. These are the numeric values reported by the WHO command of the command interpreter or in the OWNER column of a FUP INFO report.

Every process on an HP system has two accessor IDs:

Accessor ID	Description
Creator accessor ID (CAID)	Identifies the user who initiated the creation of the process
Process accessor ID (PAID)	Authority of the process to make requests to the operating environment, such as to open a file or stop another process

When you log on to a NonStop system, the operating environment gives your command interpreter your user ID as its process accessor ID. Only a process having the appropriate process accessor ID can read, write, execute, or purge a given file.

All processes that you start (using the explicit or implicit RUN command) ordinarily inherit your process accessor ID as their process accessor ID and as their creator accessor ID. If, however, the loadfile from which you created the process has been designated appropriately (using the PROGID option of the FUP SECURE command or using a SETMODENOWAIT or SETMODE call to set the file security), then the new process adopts as its process accessor ID the loadfile's owner ID, not your process accessor ID.

This enables you to establish a loadfile, executable by other users, that can:

- Gain access to a file to which those users themselves cannot gain access
- Initiate processes which those users do not own

You can therefore provide a mechanism for passing nonsensitive data from a file containing both sensitive and nonsensitive data to a class of users who should not receive the sensitive data. For more information about these security features, see the operating environment user's guide for your system.

In Example 32-12, an HP COBOL program uses the PROCESS_GETINFO_ routine to discover its environment and then reports the information to its home terminal.

Example 32-12 PROCESS_GETINFO_ Routine

```
IDENTIFICATION DIVISION.
 PROGRAM-ID.    PROBE.
    AUTHOR.        TRACY COBOL.
    INSTALLATION.  TRANSACTIONS ANONYMOUS.
    DATE-WRITTEN.  03 FEBRUARY 2000.
    DATE-COMPILED. 03 FEBRUARY 2000.
******************************************************************
*  This program obtains and reports its own process information  *
*  from PROCESSINFO.                                             *
******************************************************************
 ENVIRONMENT DIVISION.
 CONFIGURATION SECTION.
    SOURCE-COMPUTER.  HP TXP.
    OBJECT-COMPUTER.  HP TXP.
 DATA DIVISION.
 WORKING-STORAGE SECTION.
 01  PROCESS-HANDLE       PIC X(20).
 01  CPU-PIN.
    05  CPU              PIC S9(2) COMPUTATIONAL.
    05  PIN              NATIVE-2.
    05 FILLER                  PIC X(4).
 01 BYTE-PAIR              PIC S9(4) COMP.
 01 CONSECUTIVE-BYTES      REDEFINES BYTE-PAIR.
    05 LEFT-BYTE           PIC X.
    05 RIGHT-BYTE          PIC X.
 01 BYTE-AS-NUMBER   PIC S9(4) COMP.
 01 BYTE-TO-NUMERIC REDEFINES BYTE-AS-NUMBER.
    05 NUMERIC-LEFT-BYTE   PIC X.
    05 NUMERIC-RIGHT-BYTE    PIC X.
 01 CREATOR-ACCESSOR-ID    PIC S9(4) COMP.
 01 PROCESS-ACCESSOR-ID    PIC S9(4) COMP.
 01 CREATOR-EDITED.
    05 FILLER              PIC X(6) VALUE "Group ".
    05 CREATOR-GROUP       PIC ZZ9.
    05 FILLER              PIC X(9) VALUE ", Member ".
    05 CREATOR-MEMBER      PIC ZZ9.
 01 PROCESS-EDITED.
    05 FILLER              PIC X(6) VALUE "Group ".
    05 PROCESS-GROUP       PIC ZZ9.
    05 FILLER              PIC X(9) VALUE ", Member ".
    05 PROCESS-MEMBER      PIC ZZ9.
 01 PRIORITY               PIC S9(3) COMP.
 01 PROGRAM-FILE-NAME      PIC X(36).
 01 PROGRAM-FILE-NAME-LEN  NATIVE-2.
 01 HOME-TERMINAL          PIC X(24).
 01 HOME-TERMINAL-LEN      NATIVE-2.
 01 ERROR-RETURN           PIC S9(2) VALUE ZERO.
 01 PROCESS-NAME       PIC X(8).
 01 PROCESS-NAME-LEN   NATIVE-2.
 PROCEDURE DIVISION.
 WHO.
    MOVE ZERO       TO  BYTE-AS-NUMBER.
    ENTER TAL "PROCESSHANDLE_GETMINE_"
               USING PROCESS-HANDLE
               GIVING ERROR-RETURN.
    ENTER TAL "PROCESS_GETINFO_"
* processhandle
```

```
          USING  PROCESS-HANDLE
* proc-fname
                 OMITTED
* proc-fname-len
                 OMITTED
* priority
                 PRIORITY
* moms-processhandle
                 OMITTED
* hometerm
                 HOME-TERMINAL
* hometerm-len
                 HOME-TERMINAL-LEN
* process-time
                 OMITTED
* creator-access-id
                 CREATOR-ACCESSOR-ID
* process-access-id
                 PROCESS-ACCESSOR-ID
* gmoms-processhandle
                 OMITTED
* jobid
                 OMITTED
* program-file
                 PROGRAM-FILE-NAME
* program-len
                 PROGRAM-FILE-NAME-LEN
* ... rest of params not used in this example
                 GIVING ERROR-RETURN.
    IF ERROR-RETURN = 0
       PERFORM EXPLAIN-MYSELF
    ELSE
       DISPLAY "PROCESS_GETINFO_ returned an error code of " ERROR-RETURN
    .
    STOP RUN.
 EXPLAIN-MYSELF.
    MOVE SPACES TO PROCESS-NAME.
    ENTER TAL "PROCESSHANDLE_DECOMPOSE_"
                          USING PROCESS-HANDLE
                          CPU
                          PIN
* No node number
                          OMITTED
* No node name
                          OMITTED
* No nodename length
                          OMITTED
                          PROCESS-NAME
                          PROCESS-NAME-LEN
                GIVING ERROR-RETURN.
    DISPLAY "I am process (" CPU "," PIN "), named " PROCESS-NAME.
    PERFORM CAPTURE-ACCESSOR-IDS.
    DISPLAY "My creator accessor ID is " CREATOR-EDITED.
    DISPLAY "My process accessor ID is " PROCESS-EDITED.
    DISPLAY "My priority is "            PRIORITY.
    DISPLAY "My loadfile name is "   PROGRAM-FILE-NAME.
    DISPLAY "My home terminal is "       HOME-TERMINAL.
    STOP RUN.
CAPTURE-ACCESSOR-IDS.
    MOVE CREATOR-ACCESSOR-ID TO BYTE-PAIR.
    MOVE LEFT-BYTE OF CONSECUTIVE-BYTES TO NUMERIC-RIGHT-BYTE.
    MOVE BYTE-AS-NUMBER TO CREATOR-GROUP.
    MOVE RIGHT-BYTE OF CONSECUTIVE-BYTES TO NUMERIC-RIGHT-BYTE.
    MOVE BYTE-AS-NUMBER TO CREATOR-MEMBER.
    MOVE PROCESS-ACCESSOR-ID TO BYTE-PAIR.
    MOVE LEFT-BYTE OF CONSECUTIVE-BYTES TO NUMERIC-RIGHT-BYTE.
    MOVE BYTE-AS-NUMBER TO PROCESS-GROUP.
    MOVE RIGHT-BYTE OF CONSECUTIVE-BYTES TO NUMERIC-RIGHT-BYTE.
    MOVE BYTE-AS-NUMBER TO PROCESS-MEMBER.
```

Search Mode

The PROCESS_GETINFOLIST_ routine lets you search for a process that meets one or more of these criteria:

- Has a specified process ID
- Has a specified creator accessor ID
- Has a specified process accessor ID
- Has a specified home terminal
- Has a priority not greater than a specified priority
- Came from a specified loadfile name

For example, you can search for a process that is running at a priority of less than 40, has $THIRTY as its home terminal, and was loaded from loadfile $MEDIUM.PEPPER.STEAK.

Using search mode, you can implement processes for such tasks as determining whether a certain named process is running, identifying processes that are running at inappropriate priorities, or determining whether a process is using a certain terminal.

The only required parameter is the 2-byte binary item containing a processor and PIN value. The routine uses this to determine where to start its search. The remaining parameters can be present or specified as OMITTED. The values to be matched in the search are specified by the same parameters the operating environment is to use to return the values appropriate to the process it finds.

In Example 32-13, an HP COBOL program finds and reports all processes on its system that have the same process accessor ID as the program.

Example 32-13 Reporting Processes With Program's Accessor ID

```
 IDENTIFICATION DIVISION.
   PROGRAM-ID.    WHAT-AM-I-DOING.
   AUTHOR.        BRINDLEY COBOL.
   INSTALLATION.  TRANSACTIONS ANONYMOUS.
   DATE-WRITTEN.  29 FEBRUARY 1988.
   DATE-COMPILED.
*****************************************************************
*  This program hunts out and reports all processes owned by    *
*  the current user.                                            *
*****************************************************************
 ENVIRONMENT DIVISION.
   CONFIGURATION SECTION.
     SOURCE-COMPUTER.  HP TXP.
     OBJECT-COMPUTER.  HP TXP.
 DATA DIVISION.
   WORKING-STORAGE SECTION.
     01  PROCESS-HANDLE       PIC X(20).
     01  CPU-PIN.
         05  CPU              PIC S9(2) COMPUTATIONAL.
         05  PIN              NATIVE-2.
     01  ERROR-NUMBER         PIC S9(5) COMPUTATIONAL.
     01  NULL-PH              PIC X(20) VALUE ALL HIGH-VALUES.
     01  NUMERIC-CPU           PIC S9(2) COMPUTATIONAL.
     01  NUMERIC-PIN           NATIVE-2.
     01 BYTE-PAIR              PIC S9(4) COMP.
     01 CONSECUTIVE-BYTES      REDEFINES BYTE-PAIR.
        05 LEFT-BYTE           PIC X.
        05 RIGHT-BYTE          PIC X.
     01 SEPARATED-BYTES.
        05 LEFT-BYTE.
           10 LEFT-HIGH-BYTE   PIC X.
           10 LEFT-LOW-BYTE    PIC X.
        05 NUMERIC-LEFT-BYTE   PIC S9(2) COMP  REDEFINES LEFT-BYTE.
        05 RIGHT-BYTE.
           10 RIGHT-HIGH-BYTE  PIC X.
           10 RIGHT-LOW-BYTE   PIC X.
        05 NUMERIC-RIGHT-BYTE  PIC S9(2) COMP  REDEFINES RIGHT-BYTE.
     01 PROCESS-ACCESSOR-ID    PIC S9(4) COMP.
     01 PROCESS-EDITED.
        05 PROCESS-GROUP       PIC ZZ9.
        05 FILLER              PIC X(1) VALUE ",".
        05 PROCESS-MEMBER      PIC 999.
     01 CREATOR-ACCESSOR-ID    PIC S9(4) COMP.
     01 PRIORITY               PIC S9(3) COMP.
     01 PROGRAM-FILE-NAME      PIC X(34).
     01 HOME-TERMINAL          PIC X(34).
     01 ERROR-RETURN           PIC S9(2) VALUE ZERO.
     01 MASQUE                 PIC S9(4) COMP VALUE 8192.
*                  Bits: 0010 0000 0000 0000
*                  to request match on process accessor id.
     01 CURRENT-CPU            PIC S9(4) COMP VALUE 0.
     01 BINARY-WORKTABLE       PIC S9(4) COMP.
     01 INTERNAL-NAME          PIC X(24).
     01 EXTERNAL-NAME          PIC X(34).
 PROCEDURE DIVISION.
 WHATS-HAPPENING.
* Start with cpu-PIN of 0,0
    MOVE LOW-VALUES TO CPU-PIN.
* Get owner's process accessor id
    ENTER TAL "PROCESS_GETINFO_"
          USING  PROCESS-HANDLE
                 OMITTED
                 OMITTED
                 PRIORITY
                 OMITTED
                 HOME-TERMINAL
                 OMITTED
                 OMITTED
                 CREATOR-ACCESSOR-ID
                 PROCESS-ACCESSOR-ID
                 OMITTED
                 OMITTED
                 PROGRAM-FILE-NAME
                 OMITTED
                 OMITTED
                 GIVING ERROR-RETURN.
    PERFORM EDIT-ACCESSOR-ID.
    DISPLAY "CPU,PIN   GRP,USR  PRI  loadfile             "
            "HOMETERM".
* Hunt through all CPUs in system
    PERFORM INVESTIGATE-A-CPU
            UNTIL CURRENT-CPU > 16.
    STOP RUN.
 INVESTIGATE-A-CPU.
    ENTER TAL "PROCESS_GETINFO_"
          USING  PROCESS-HANDLE
                 OMITTED
                 OMITTED
                 PRIORITY
                 OMITTED
                 HOME-TERMINAL
                 OMITTED
                 OMITTED
                 CREATOR-ACCESSOR-ID
                 PROCESS-ACCESSOR-ID
                 OMITTED
                 OMITTED
                 PROGRAM-FILE-NAME
                 OMITTED
                 OMITTED
                 GIVING ERROR-RETURN.
    IF ERROR-RETURN < 2
*      Found a match
       PERFORM REPORT-A-MATCH
```

```
        ADD 1 TO CPU-PIN
    ELSE
        IF ERROR-RETURN = 2
*          No more on current CPU
           PERFORM INCREMENT-CPU-OR-QUIT
        ELSE
             IF ERROR-RETURN < 99
*               CPU not configured or incommunicado
                PERFORM INCREMENT-CPU-OR-QUIT
             ELSE
                DISPLAY "INTERNAL ERROR IN PARAMETERS"
                STOP RUN.
REPORT-A-MATCH.
    PERFORM EDIT-CPU-PIN.
    DISPLAY "(" NUMERIC-CPU "," NUMERIC-PIN ")   "
            PROCESS-EDITED "   "
            PRIORITY       "   ".
    DISPLAY PROGRAM-FILE-NAME "   "
            HOME-TERMINAL  "   ".
 EDIT-CPU-PIN.
    MOVE CPU-PIN TO BYTE-PAIR.
    PERFORM SEPARATE-BYTE-PAIR.
    MOVE NUMERIC-LEFT-BYTE  TO NUMERIC-CPU.
    MOVE NUMERIC-RIGHT-BYTE TO NUMERIC-PIN.
 EDIT-ACCESSOR-ID.
    MOVE PROCESS-ACCESSOR-ID TO BYTE-PAIR.
    PERFORM SEPARATE-BYTE-PAIR.
    MOVE NUMERIC-LEFT-BYTE   TO PROCESS-GROUP.
    MOVE NUMERIC-RIGHT-BYTE  TO PROCESS-MEMBER.
 SEPARATE-BYTE-PAIR.
    MOVE LOW-VALUES TO SEPARATED-BYTES.
    MOVE LEFT-BYTE  OF CONSECUTIVE-BYTES TO LEFT-LOW-BYTE  OF SEPARATED-BYTES.
    MOVE RIGHT-BYTE OF CONSECUTIVE-BYTES TO RIGHT-LOW-BYTE OF SEPARATED-BYTES.
 INCREMENT-CPU-OR-QUIT.
    IF CURRENT-CPU < 15
       ADD 1 TO CURRENT-CPU
       MOVE CURRENT-CPU TO BINARY-WORKTABLE
       MULTIPLY 256 BY BINARY-WORKTABLE
       MOVE BINARY-WORKTABLE TO CPU-PIN
    ELSE
       DISPLAY "-----------------------------"
       STOP RUN.
```

Identifying the Creator

A process can use the PROCESS_GETINFO_ routine to determine the identity of the process that created it, which enables the process to be selective. It can terminate if it was created by an inappropriate agent. It can open different files depending on the identity of its creator. It can send messages to its creator explicitly. It can even suspend or stop its creator if its process accessor ID matches its creator's creator accessor ID.

When you issue a RUN command, such as RUN X, your TACL process initiates the specified process. The process produced from loadfile X has the TACL process as its creator.

In addition, suppose your process uses CLU_PROCESS_CREATE_ to initiate a new process. The child process can discover the identity of its parent process, if the child is not running as a process pair (in that case, the parent of the primary process is the backup process).

Example 32-14 uses the PROCESS_GETINFO_ routine to report the identity of its creator to its home terminal. To handle the possibility of an unnamed process, the example uses the same mechanism to decipher its creator's process name that is used in WHO-SENT-THAT in Identifying the Message Source (page 948). After identifying its creator process, the example determines whether it can stop its creator; if it can, it does.

Example 32-14 Identifying a Process's Creator

```
 IDENTIFICATION DIVISION.
   PROGRAM-ID.    WHOA-MOM.
   AUTHOR.        KRIS COBOL.
   INSTALLATION.  TRANSACTIONS ANONYMOUS.
   DATE-WRITTEN.  29 FEBRUARY 1988.
   DATE-COMPILED.
*********************************************************************
*  This program obtains and reports the process ID of its creator,  *
*  then attempts to stop its creator.                               *
*********************************************************************
 ENVIRONMENT DIVISION.
   CONFIGURATION SECTION.
     SOURCE-COMPUTER.  HP TXP.
     OBJECT-COMPUTER.  HP TXP.
 DATA DIVISION.
   WORKING-STORAGE SECTION.
     01 MOMS-PROCESS-ID.
        05 MOMS-PROCESS-NAME   PIC X(6).
        05 MOMS-CPU-PIN        PIC S9(4) COMP.
        05 CPU-PIN-REDEF   REDEFINES MOMS-CPU-PIN.
           10 CPU-PART    PIC X.
           10 PIN-PART    PIC X.
     01 PNAME.
        05 PNAME-BYTES-1-2   PIC XX.
        05 PNAME-FIRST-WORD-NUM REDEFINES PNAME-BYTES-1-2
                             PIC S9(3) COMP.
        05 FILLER            PIC X(4).
     01 CPU-PIN-REWORK.
        05 ALPHA-CPU.
           10 CPU-HIGH-BYTE     PIC X.
           10 CPU-LOW-BYTE      PIC X.
        05 NUMERIC-CPU          REDEFINES ALPHA-CPU
                                PIC S9(2) COMP.
        05 ALPHA-PIN.
           10 PIN-HIGH-BYTE     PIC X.
           10 PIN-LOW-BYTE      PIC X.
        05 NUMERIC-PIN          REDEFINES ALPHA-PIN
                                PIC S9(3) COMP.
     01 MY-CPU-PIN              PIC S9(4) COMP.
     01 MY-PAID                 PIC S9(4) COMP.
     01 MOMS-CAID               PIC S9(4) COMP.
     01 INTERNAL-NAME           PIC X(24).
     01 EXTERNAL-NAME           PIC X(34).
     01 ERROR-RETURN            PIC S9(2) VALUE ZERO.
     01 DISPLAY-BUFFER          PIC X(79).
 PROCEDURE DIVISION.
 WHO.
* -- PROCESS_GETINFO_ delivers 4-word process-id to MOMS-PROCESS-ID
    ENTER TAL "PROCESS_GETINFO_"
          USING MOMS-PROCESS-ID.
* -- Report MOM's process name to my home terminal
    PERFORM DISPLAY-PROCESS-NAME.
* -- Take only the CPU-PIN from MOM, and ask for MOM's
* -- creator accessor id.
    ENTER TAL "PROCESS_GETINFO_"
          USING  MOMS-CPU-PIN
                 OMITTED
                 MOMS-CAID
          GIVING ERROR-RETURN.
    IF ERROR-RETURN NOT = 0
       DISPLAY "Call to PROCESS_GETINFO_ returned an error code of "
               ERROR-RETURN
       STOP RUN.
* -- Get my process accessor id
    ENTER TAL "PROCESS_GETINFO_"
          USING PROCESS-HANDLE
                OMITTED
                OMITTED
                OMITTED
                OMITTED
                OMITTED
                OMITTED
                OMITTED
                MOMS-CAID
                MY-PAID
                OMITTED
                OMITTED
                OMITTED
                OMITTED
                OMITTED
                OMITTED
                GIVING ERROR-RETURN.
    IF ERROR-RETURN NOT = 0
       DISPLAY "Call to PROCESS_GETINFO_ returned an error code of "
               ERROR-RETURN
       STOP RUN.
* -- If my process accessor id matches MOM's creator accessor id, I can
* -- stop MOM before continuing
    IF MY-PAID = MOMS-CAID
       ENTER TAL "PROCESS_STOP_" USING MOMS-PROCESS-ID.
*   ...
    STOP RUN.
 DISPLAY-PROCESS-NAME.
    MOVE MOMS-PROCESS-NAME TO PNAME.
    IF PNAME-FIRST-WORD-NUM < 0
       MOVE MOMS-PROCESS-ID TO INTERNAL-NAME
       DISPLAY "My creator is ("
               NUMERIC-CPU "," NUMERIC-PIN ")-----"
               MOMS-PROCESS-NAME
...
 CAPTURE-CPU-PIN.
    MOVE CPU-PART   TO CPU-LOW-BYTE.
    MOVE LOW-VALUES TO CPU-HIGH-BYTE.
    MOVE PIN-PART   TO PIN-LOW-BYTE.
    MOVE LOW-VALUES TO PIN-HIGH-BYTE.
```

Monitoring Descendants’ Completion

The operating environment always sends messages about descendant processes to parent processes. If your process starts another process, it might need to determine whether the process ran to normal completion or terminated abnormally.

The HP COBOL run-time routines report the receipt of system messages only when the RECEIVE-CONTROL paragraph specifies it. If you want a process that creates other processes to be notified about termination or failure of these descendant processes:

- Include a REPORT SYSTEM MESSAGES entry in the RECEIVE-CONTROL paragraph of the parent process.
- Have the parent process monitor $RECEIVE and watch for ABEND and STOP messages.

A process other than the actual creator can intercept the STOP and ABEND messages by using the PROCESS_SETINFO_ routine to become the creator of record. See Changing the Creator ID.

Example 32-15 creates a FUP process and then watches for its termination.

Example 32-15 Monitoring Completion of Descendant Processes

```
?ENV COMMON
?SAVE ALL
?SEARCH $SYSTEM.SYSTEM.COBOLLIB
 IDENTIFICATION DIVISION.
   PROGRAM-ID.    FUPPERWARE.
   AUTHOR.        TERRY COBOL.
   INSTALLATION.  TRANSACTIONS ANONYMOUS.
   DATE-WRITTEN.  29 FEBRUARY 1988.
   DATE-COMPILED.
*************************************************************************
*  This program creates a FUP process and watches for its termination.  *
*************************************************************************
 ENVIRONMENT DIVISION.
   CONFIGURATION SECTION.
     SOURCE-COMPUTER.  HP TXP.
     OBJECT-COMPUTER.  HP TXP.
     SPECIAL-NAMES.
   INPUT-OUTPUT SECTION.
     FILE-CONTROL.
        SELECT MESSAGE-IN-FILE
               ASSIGN TO "$RECEIVE"
               FILE STATUS IS RECEIVE-FILE-STATUS.
     I-O-CONTROL.
     RECEIVE-CONTROL.
       TABLE OCCURS 1 TIMES
       SYNCDEPTH LIMIT IS 1
       MESSAGE SOURCE IS MESSAGE-SOURCE-REC
       REPORT SYSTEM MESSAGES.
DATA DIVISION.
   FILE SECTION.
     FD MESSAGE-IN-FILE
        LABEL RECORDS ARE OMITTED.
     01 MESSAGE-IN.
        05 SYS-MSG-CODE                  PIC S9(4) COMP.
           88 SYS-MSG-STOP               VALUE -101.
                   05 SYS-MSG-PROCNAME  PIC X(6).
        05 FILLER                        PIC X(66).
        05 SYS-MSG-STOP-FLAG             NATIVE-2.
        05 FILLER                        PIC X(948).
    WORKING-STORAGE SECTION.
     01 MESSAGE-SOURCE-REC.
        05 SYSTEM-FLAG        PIC S9  COMP.
        05 ENTRY-NUMBER       PIC 999 COMP.
        05 FILLER             PIC X(4).
        05 PROCESS-HANDLE
           10 CPU-PIN.
              15 CPU          PIC S9(2) COMP.
              15 PIN          NATIVE-2.
        05 FILLER             PIC X(16).
     01 CPU-PIN-REDEF.
        05 ALPHA-CPU.
           10 CPU-HIGH-BYTE      PIC X.
           10 CPU-LOW-BYTE       PIC X.
        05 NUMERIC-CPU           REDEFINES ALPHA-CPU
                                 PIC S9999 COMP.
        05 ALPHA-PIN.
           10 PIN-HIGH-BYTE      PIC X.
           10 PIN-LOW-BYTE       PIC X.
        05 NUMERIC-PIN           REDEFINES ALPHA-PIN
                                 PIC S9999 COMP.
    01 FILE-DATA.
       05 RECEIVE-FILE-STATUS.
          10 STAT-1                     PIC 9.
             88  CLOSE-FROM-REQUESTOR VALUE 1 THRU 3.
          10 STAT-2                     PIC 9.
    01  SAVE-MESSAGE-STUFF.
        05 FUP                   PIC X(21) VALUE "$SYSTEM.SYSTEM.FUP".
        05 SU-ERROR              PIC S9(4) VALUE ZERO     COMP.
        05 NEWPROCESS-ERR-LEFT  PIC 9(4).
        05 NEWPROCESS-ERR-RIGHT PIC 9(4).
        05 FUP-FAILED           PIC X(19) VALUE "Failed to start FUP".
        05 STRING-PORTION       PIC X(6)  VALUE "STRING".
        05 INFO-COMMAND         PIC X(6)  VALUE "INFO *".
        05 STARTUP-RESULT       PIC S9(4) VALUE ZERO     COMP.
        05 NULL-CPLIST          PIC S9(9) VALUE ZERO     COMP.
 PROCEDURE DIVISION.
 DECLARATIVES.
   HANDLE-INFILE-ERRORS SECTION.
     USE AFTER STANDARD ERROR PROCEDURE ON MESSAGE-IN-FILE.
   INFILE-ERROR.
     IF STAT-1 = 1
        DISPLAY "EOF on $RECEIVE"
     ELSE
        DISPLAY "RECEIVE FILE ERROR STATUS = "
                RECEIVE-FILE-STATUS
     ...
 END DECLARATIVES.
 AA SECTION.
 AA-1.
    OPEN INPUT MESSAGE-IN-FILE.
    MOVE ZERO TO SU-ERROR
                 SYS-MSG-CODE.
* Inject INFO command into startup message to pass to FUP
    ENTER "PUTSTARTUPTEXT"
          USING STRING-PORTION,
                INFO-COMMAND,
                NULL-CPLIST
          GIVING STARTUP-RESULT.
* Start FUP
    ENTER CLU_PROCESS_CREATE_"

          USING FUP
          GIVING SU-ERROR.
* Await termination of FUP, or report it never started
    IF SU-ERROR = 0
       PERFORM WATCH
         UNTIL SYS-MSG-STOP
       IF FUNCTION MOD (SYS-MSG-STOP-FLAG, 2) = 0
          DISPLAY "FUP terminated normally"
       ELSE
          DISPLAY "FUP aborted"
       END-IF
    ELSE
       PERFORM DISPLAY-STARTUP-FAILURE.
    STOP RUN.
 DISPLAY-STARTUP-FAILURE.
    EVALUATE SU-ERROR
    WHEN 1
       DISPLAY FUP-FAILED
         " -- REQUIRED PARAMETER MISSING OR ILLEGAL"
    WHEN 2
       DISPLAY FUP-FAILED
         " -- ILLEGAL loadfile NAME ("
                    FUP ")"
    WHEN 3
       DISPLAY FUP-FAILED
         " -- INFILE, OUTFILE, OR DEFAULT VOLUME"
       DISPLAY " NAME CANNOT BE CONVERTED TO NETWORK FORM"
    WHEN OTHER
      IF SU-ERROR < 256
            DISPLAY FUP-FAILED
              " -- File management error #"
              SU-ERROR
      ELSE
*        -- Received raw error from NEWPROCESS system procedure.
*        -- Decompose it into left byte and right byte values.
         DIVIDE        SU-ERROR
             BY        256
             GIVING    NEWPROCESS-ERR-LEFT
             REMAINDER NEWPROCESS-ERR-RIGHT.

         DISPLAY FUP-FAILED
                 " -- NEWPROCESS error # = ("
                 NEWPROCESS-ERR-LEFT
                 ","
                 NEWPROCESS-ERR-RIGHT
                 ")"
      END-IF
    END-EVALUATE
...
 WATCH.
    READ MESSAGE-IN-FILE.
```

Changing the Creator ID

One reason the operating environment keeps track of the creator of a process is to be able to notify a creator when a descendant process terminates. If process $PARENT starts process $CHILD, the operating environment notifies $PARENT through the $RECEIVE mechanism when $CHILD terminates.

It is possible for a third process, $THIRD, to instruct the operating environment to alter the system tables to make $THIRD the recorded parent of $CHILD. $THIRD does this by calling the PROCESS_SETINFO_ procedure. This operation is legal only if $THIRD has the same process accessor ID as $CHILD; the $THIRD process has no explicit control other than that. The fault-tolerant facility uses PROCESS_SETINFO_ to enable each member of a process pair to be notified if the other terminates.

After $THIRD has called PROCESS_SETINFO_, any notification of $CHILD's termination is sent to $THIRD, rather than to $PARENT. Also, if $CHILD calls PROCESS_GETINFO_, the operating environment reports back to $CHILD that $THIRD is $CHILD's creator. For more information about PROCESS_GETINFO_ or PROCESS_SETINFO_, see the *Guardian Procedure Calls Reference Manual*.

Suspending a Process

A process might need to automatically suspend its execution. It might be a sampling routine that wakes up, takes and records some measurements, and suspends itself for a time. It might be a monitoring routine that wakes up every 10 minutes and checks whether a specified terminal or a certain program is in use.

By calling the DELAY routine, a process can suspend its execution for a number of hundredths of a second.

Example 32-16 uses suspension to watch for a remote system to become available so it can send a message to a routine on the remote system.

Example 32-16 Suspending a Process

```
 IDENTIFICATION DIVISION.
   PROGRAM-ID.    DOZER.
   AUTHOR.        BUFFY COBOL.
   INSTALLATION.  TRANSACTIONS ANONYMOUS.
   DATE-WRITTEN.  29 FEBRUARY 1988.
   DATE-COMPILED.
**************************************************************************
*  This program sends one message to $BOSS on system \HQ.  If that       *
*  process cannot be opened, the program tries again every five minutes. *
**************************************************************************
```

```
 ENVIRONMENT DIVISION.
   CONFIGURATION SECTION.
     SOURCE-COMPUTER.  HP TXP.
     OBJECT-COMPUTER.  HP TXP.
   INPUT-OUTPUT SECTION.
     FILE-CONTROL.
       SELECT REPORT-OUT
              ASSIGN TO "\HQ.$BOSS"
              ORGANIZATION IS SEQUENTIAL
              ACCESS MODE IS SEQUENTIAL
              FILE STATUS IS REPORT-STATUS.
 DATA DIVISION.
   FILE SECTION.
     FD REPORT-OUT
        LABEL RECORDS ARE OMITTED.
     01 REPORT-LINE         PIC X(80).
   WORKING-STORAGE SECTION.
     01 REPORT-STATUS       PIC XX     VALUE "  ".
     01 REPEAT-INTERVAL     PIC S9(10) VALUE 30000.
*                                      = 5 min * 60 sec * 100
 PROCEDURE DIVISION.
 DECLARATIVES.
   HANDLE-REPORT-ERRORS SECTION.
     USE AFTER STANDARD ERROR PROCEDURE ON REPORT-OUT.
   REPORT-ERROR.
* An empty declarative can be used to intercept error conditions
* and set the file status data item.
* Presence of a declarative does not prevent the run-time routine
* from delivering an error 019 to the home terminal of this
* process.
 END DECLARATIVES.
 PULLMAN SECTION.
 ZZZZ.
   PERFORM OPEN-IT
     UNTIL REPORT-STATUS = "00".
   MOVE "Having a wonderful time!  Wish you were here."
     TO REPORT-LINE.
   WRITE REPORT-LINE.
   CLOSE REPORT-OUT.
   STOP RUN.
 OPEN-IT.
   OPEN OUTPUT REPORT-OUT.
   IF REPORT-STATUS NOT = "00"
*                      That is, error on file open.
      ENTER TAL "DELAY"
            USING REPEAT-INTERVAL.
```

33 Fault-Tolerant Processes

A process is running in a fault-tolerant manner when no single point of failure can stop the process or corrupt its data or the files it is manipulating. Processes are not automatically fault tolerant—they must be designed and implemented to be fault tolerant.

How might a single point of failure affect a process? Suppose a process is operating an automated teller machine (ATM) for a financial institution. If you, as a customer, come to the ATM and request $20 from your account:

- You want the ATM to service your request, not terminate before completing the transaction.
- You want the ATM to record at most one debit of $20 from your account; the institution wants the ATM to record at least one debit of $20 from your account.
- You want the ATM to dispense at least $20 to you; the institution wants the ATM to dispense at most $20 to you.

If the process is not fully fault tolerant, a number of possible failures can interfere with the preceding desires:

- A process might be running on a processor that fails (blows a fuse, is accidentally unplugged, is stopped by an operator, or whatever), so the transaction does not complete.
- A process might get part way through your withdrawal transaction, deducting the $20 from your balance but not yet reaching the point of dispensing the cash to you. If the process is automatically retried, it might deduct the $20 from your account again and dispense you the cash; but the balance in your account would then be down by $40.
- A process might fail during the transaction after disbursing the $20 to you, but before recording the fact, and resume at the point of asking what you want. If you again asked for $20, the process could disburse another $20 (total = $40) and record only a $20 withdrawal.
- A process might disburse the $20 cash to you and fail before making a permanent record of the transaction.

NonStop Operating System

The NonStop operating system architecture is the underlying mechanism that enables you to write fault-tolerant processes. The full redundancy of processors, devices, controllers, and paths among them is the basis for the NonStop operating system's fault tolerance. But given that base, there are still two ways a process (particularly a Pathway server process operating under TS/MP) can be designed to be fault tolerant: by using the fault-tolerant facility or by using TMF.

When you have decided which of the two mechanisms to use, you can read more about it in Fault-Tolerant Facility or TMF.

Introduction to the Fault-Tolerant Facility

NOTE: This topic does not apply to the OSS environment.

To use the fault-tolerant facility, you must include the NONSTOP compiler directive in your compilation and embed one STARTBACKUP statement and one or more CHECKPOINT statements at strategic points in your program.

At the beginning of its execution, after opening its files, your process executes a STARTBACKUP statement to instruct the operating environment to produce a backup process in a different processor and to open the same files in the backup process. The backup process is loaded from the same loadfile as the original (primary) process, but the operating environment does not actually start it running.

At certain points in your program, you code CHECKPOINT statements, which instruct the operating environment to copy a specific list of one or more of your data elements into the backup process's data space. If the primary process fails, the operating environment activates the backup process. The backup process starts executing at the code location following the last CHECKPOINT statement executed. At this point, the backup process's data space contains a copy of the part of the primary process's data space that was checkpointed at the time of the last CHECKPOINT statement.

Determining the proper sites and data-element lists for CHECKPOINT statements can be a painstaking operation. One recommendation is that you checkpoint everything involved in writing a record just before executing the WRITE statement.

Introduction to TMF

The HP NonStop™ Transaction Management Facility (TMF) relies on the principle of archiving and backing out transactions. Every transaction has a beginning and an end. If the entire transaction cannot be completed (due to some single point of failure), any changes made to the database must be backed out. This means that other processes must be kept from using any of the records that a transaction has modified until the transaction has been completed.

If you are writing a Pathway server in HP COBOL, and the Pathway application will use TMF, your design is much simpler than it would be if you used the fault-tolerant facility. All an HP COBOL server needs to do to be fault tolerant under these conditions is to lock every record it is going to change before changing the record, and lock any record it reads that affects the course of the transaction. All other coding for fault tolerance is at the requester level—the SCREEN COBOL program includes BEGIN TRANSACTION, END TRANSACTION, and ABORT TRANSACTION statements.

If you want an HP COBOL program that is not a Pathway server to run under TMF, you must include ENTER statements to call the routines BEGINTRANSACTION, ENDTRANSACTION, and so on.

Choosing the Fault-Tolerant Facility or TMF

If your program is updating unaudited files and you want it to be fault tolerant, you must code it to use the fault-tolerant facility; otherwise, whether to use the fault-tolerant facility or TMF is primarily a question of efficiency versus maintenance.

TMF is implemented with process pairs, so when you use TMF you are using the fault-tolerant facility indirectly. Using the fault-tolerant facility directly makes the program run more efficiently; however, maintenance is easier with TMF, because the programming and debugging of the process pairs has already been done for you.

It is easier to configure the system for TMF and to design, implement, and maintain servers that use TMF than it is to design, implement, and maintain process pairs. The auditing operation takes processing time. The audit trails take disk space and require tape operations. But just as Pathway/TS frees the application programmers from having to produce and support multithreaded requesters, TMF frees them from having to produce and maintain process pairs.

Fault-Tolerant Facility

The fault-tolerant facility is software that HP provides but that you must explicitly call to cause a loadfile to be executed as a process pair.

You must include a NONSTOP directive in your compilation before the PROCEDURE DIVISION header. You must include one STARTBACKUP statement to create the backup process, and a number of CHECKPOINT statements to transmit restart information to the backup process. Finally, you must run the program as a named process (to learn how, see Process Names (page 938)).

After you have done these things, the fault-tolerant facility does the remainder of the work to assure fault-tolerant operation.

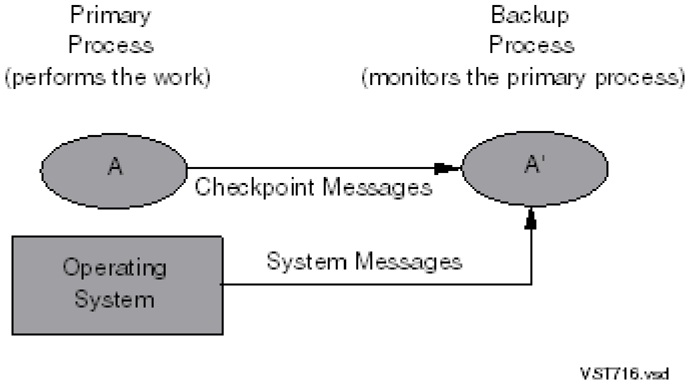
The information in these topics is general. For specific details on checkpointing, see the *Guardian Programmer's Guide*. For details on the STARTBACKUP and CHECKPOINT statements, see STARTBACKUP (page 467) and CHECKPOINT (page 314).

Process Pairs

A process is the basic executable unit known to the operating environment—the execution (in a processor) of a program. Specifically, the term program indicates a group of instruction codes and initialized data—an HP COBOL run unit; the term process denotes the changing states of an executing program. The same loadfile can be executing concurrently a number of times, but each execution is a separate process.

An application process can be designed to recover from any type of hardware failure except one—a failure of the processor in which it is executing. One way to provide fault tolerance is to establish the process as a process pair. A process pair consists of two executions of the same loadfile: the primary process executes in one processor; the backup process executes in another. Control in the program indicates whether the process is executing in the primary mode to perform its task or in the backup mode to monitor the primary process.

Figure 33-1 Process Pair



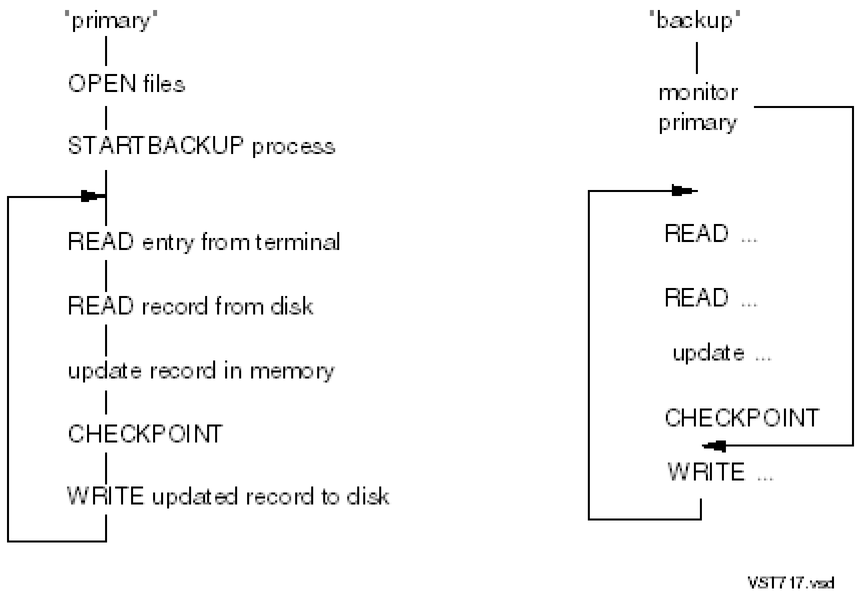
In this primary-plus-backup structure, the fault-tolerant facility (as directed by the primary process) keeps the backup process informed of the executing state of the primary process. At critical points in the processing, the primary process sends checkpoint messages to the backup process to pass the current state of the data, the file buffers, and the files to the backup process. When the backup process learns of the failure of its primary process (by the receipt of a process-failure or processor-failure system message through $RECEIVE), the backup process becomes the primary process and continues with the application's work (possibly starting a new backup process for itself).

The fault-tolerant facility provides the means of writing application programs that can recover from a processor module failure. When the primary process executes a STARTBACKUP statement, a fault-tolerant facility routine in the primary process directs the operating environment to start the backup process.

When the primary process executes a CHECKPOINT statement, a fault-tolerant facility routine transmits pertinent data to the backup process. While the primary process is operating, a fault-tolerant facility routine in the backup process automatically monitors and accepts checkpoint information from the primary process. If the backup process is notified of the failure of its primary process, the fault-tolerant facility causes the backup process to begin executing at the statement following the latest CHECKPOINT statement. (The notification to the backup process of the failure of the primary process comes in the form of a processor-down, stop, or abend message delivered through $RECEIVE and handled automatically by the HP COBOL fault-tolerant facility.)

Figure 33-2 illustrates the activity of a process pair. The backup process stays in monitor state while the primary process is operating. If the primary fails, the backup leaves the monitor state and begins executing at the point indicated by the last call to CHECKPOINT by the primary.

Figure 33-2 Activity of a Process Pair



This sequence of actions occurs when a process pair runs:

1. The primary process opens any files required for its execution.
2. The primary process starts its backup process in another processor module by executing a STARTBACKUP verb.

   This action also opens the files for the backup process and checkpoints the state of the primary process to the backup process. A process pair opens files in a manner that permits both members of the pair to have a file open while retaining the ability to exclude other processes from accessing a file. When a disk file has been opened by a process pair in this manner, a record or file lock by the primary process is also an equivalent lock by the backup process.
3. The backup process, at the beginning of its execution, automatically begins monitoring the primary process. This is the extent to which the backup process executes unless a failure of the primary process occurs.
4. The primary process begins executing its main processing loop. At critical points through the execution loop, typically before each write to a disk file, the primary process executes a CHECKPOINT statement to copy part of its environment and pertinent file control information to the backup process (marking a restart point for the backup process). Typically, a program contains several CHECKPOINT statements, each of which checkpoints only a portion of the primary process's environment.
5. If the primary process fails, the backup process begins executing at the restart point indicated by the latest execution of a CHECKPOINT statement. The backup process is then considered to be the primary process.
6. If the reason the primary process failed was a processor failure (that is, the backup process received a processor-down message), the fault-tolerant facility in the new primary (former backup) process automatically starts a new backup process when the failed processor has been repaired and brought back on line. This new backup process is then ready to take over if the primary process fails.

Checkpointing

When the primary process executes a CHECKPOINT statement, one of its fault-tolerant facility routines formats a message containing the information to be checkpointed and sends it to the backup process in the form of an interprocess message. A fault-tolerant facility routine in the backup process receives and acts upon the message.

The two types of information you must usually checkpoint are data items and sync blocks.

Data Items

These are usually file record areas but can be any desired data items in the File Section, the Working-Storage Section, or the Extended-Storage Section of the Data Division. You must checkpoint any data items that are part of the program's state—specifically the disk record that is about to be written, the terminal or tape record that was just read, and any data that is necessary to resume processing at the site of the checkpoint statement.

The reason for checkpointing data items is to give the backup process all the information it needs to reexecute an I-O request if the primary process fails. Usually, you checkpoint a data item just before writing the data to disk. You can also use data-item checkpointing to eliminate the need for the backup process to reexecute an I-O request. An example of this is an entry received from a terminal. You checkpoint the data item received from a terminal by a READ statement immediately after executing the READ statement to minimize the possibility that the operator has to reenter data.

Sync Blocks

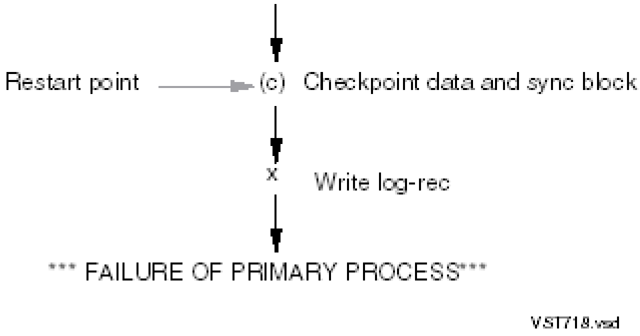
A sync block contains control information about the current state of a disk file (such as the current value of the file pointers).

The purpose of checkpointing the sync block is twofold:

- To ensure that a write operation is not duplicated when a backup process takes over from its primary process
- To pass the current file pointers' values to the file system of the backup processor

When a process executes a checkpoint of a sync block, the operating environment passes the information in the sync block to the file system of the backup processor. The reason for preventing duplicate operations is illustrated in Figure 33-3. In Figure 33-3, a primary process completes a sequential write operation (that is, append to end of file) successfully, but fails before a subsequent checkpoint to its backup process. On the takeover from the primary process, the backup process reexecutes the operations just completed by the primary process. If the write operation was performed as requested, it duplicates the record, but at the new end-of-file location.

Figure 33-3 Duplication in Takeover


Restart point
(c)
Checkpoint data and sync block
x
Write log-rec
*** FAILURE OF PRIMARY PROCESS***
VST718.vsd

To prevent such duplicate write operations by the backup process, you must specify a nonzero SYNCDEPTH parameter in the OPEN statement. This action allows the file system to record the completion status of each input-output operation. If the backup process requests an operation already completed by the primary process, the file system recognizes this condition. Then, instead of performing the operation, the file system returns the recorded completion status of the operation to the backup process. If, however, the requested operation has not been performed, it is performed and the completion status is returned to the backup process. The course of action taken by the file system is completely invisible to the backup process.

The file system can save the completion status of the latest 15 disk-file operations and relate those completions with up to 15 operations requested by a backup process upon takeover from a failed primary process. (For $RECEIVE or other processes handled as files, the number of saved completion status values is 255.) Use the SYNCDEPTH parameter to specify the maximum number of completion status values that the file system is to save. The SYNCDEPTH value is typically the same as the maximum number of operations that write to a file without an intervening checkpoint of the file's sync block. The default SYNCDEPTH value is 1. In a Pathway application, the SYNCDEPTH value must be 1.

General Rules

You can checkpoint the entire data area of the program and checkpoint each file after the execution of each statement in your program, but this places a tremendous burden on the interprocess message system and degrades the performance of your process and all processes executing on the same processors as your process pair.

Minimize the number of checkpoints in a processing loop and the amount of data checkpointed for each CHECKPOINT statement. One approach is to checkpoint only the items necessary to establish a restart point. Verify that you have captured all necessary data items (you might want to group them in the Data Division). Also, if you checkpoint the contents of a record area, verify that the remainder of the data storage is still valid in case of a takeover by the backup process.

When your program performs a series of updates to one or more disk files, you can combine checkpoints to reduce system overhead. Structure your program so that the series of write operations necessary to update a file are performed in a group. For each file to be checkpointed in this manner, the sync depth must equal the maximum number of write operations that occur between checkpoint operations (CHECKPOINT, OPEN, or CLOSE statements). Then, when a file is about to be updated by performing that collection of write operations, the file's sync block and its record areas must be checkpointed. In any case, ensure the integrity of all data by checkpointing.

Rules for Servers to Follow With Fault-Tolerant Requesters

If a server is handling requests from a requester that is running as a process pair, there is an important rule for the server to follow for checkpointing between a $RECEIVE READ-WRITE pair for a nonretryable request. A nonretryable request is a task that must not be automatically reexecuted if the requester's backup process takes over. For example, if a failure occurs after a request to deduct taxes from a paycheck is executed, reexecution would cause double deduction. This operation must be done only once for each calculation of the taxes; thus, it is a nonretryable request.

The rule for this case is: Have at least one of the CHECKPOINT statements include the *`file-name`* (from name of the file description entry) of the output file assigned to $RECEIVE—the *`file-name`* with which the write operation is associated.

Using the Fault-Tolerant Facility

To use the fault-tolerant facility, your source program must include a NONSTOP directive and STARTBACKUP and CHECKPOINT statements and its process must have a name.

NONSTOP Directive

To enable the fault-tolerant facility, you must specify the NONSTOP directive before the Procedure Division of the first program unit in the source text.

STARTBACKUP and CHECKPOINT Statements

Normally, the STARTBACKUP statement is executed once at the beginning of run-unit execution (after the opening of the files) to establish a process pair. A CHECKPOINT statement is then executed at critical points during processing to pass (that is, checkpoint) information to the backup process.

OPEN and CLOSE Statements

If a process is executing as a process pair (that is, the primary process has executed a STARTBACKUP statement), checkpointing occurs automatically when any OPEN or CLOSE statement executes and when the backup process is established. The execution of STARTBACKUP and CHECKPOINT statements sets the PROGRAM-STATUS variable to indicate the outcome of the checkpointing operation. The execution of OPEN and CLOSE statements (because they establish takeover points) also sets the PROGRAM-STATUS variable.

Following an OPEN or CLOSE of a file by the primary process, the sync blocks of all other files in the backup process become invalid. If the SYNCDEPTH exceeds 1, you must checkpoint both the record and the sync block of each file other than the one that was opened or closed before executing the next WRITE statement.

PROGRAM-STATUS Data Item

Each HP COBOL run unit contains a four-digit data item called PROGRAM-STATUS, shown in Example 33-1. When a process is running as a process pair, this item is set by the HP COBOL fault-tolerant facility to indicate the outcome of the execution of a STARTBACKUP, OPEN, CLOSE, or CHECKPOINT statement.

Example 33-1 PROGRAM-STATUS Data Item

```
01 PROGRAM-STATUS.
    05 PROGRAM-STATUS-1  PIC X.
    05 PROGRAM-STATUS-2  PIC X(3).
```

If one of options 0 through 2 of the STARTBACKUP statement is used (that is, if the HP COBOL fault-tolerant facility monitors the backup process), then the possible values of PROGRAM-STATUS for any execution of a STARTBACKUP, CHECKPOINT, OPEN, or CLOSE statement and their meanings are as shown in Table 33-1.

Table 33-1 Values for PROGRAM-STATUS When STARTBACKUP Has Option 0, 1, or 2

PROGRAM-STATUS Value	Meaning
0000	No error
0100	Takeover by backup, primary stopped
0101	Takeover by backup, primary aborted
0102	Takeover by backup, primary processor module failed
0103	Takeover by backup, primary called CHECKSWITCH
1000	Backup processor down
2*nnn*	Communication error; *nnn* = file system error number
3*nnn*	CHECKOPEN failure; *nnn* = file system error number
4*nnn*	Indicates a PROCESS_CREATE_ failure. For more information, see the *Guardian Procedure Calls Reference Manual*.
5000	Too many failure-restart cycles (takeovers by backup) or a logic error in program
6000	Parameter or logic error

If option 3 of the STARTBACKUP statement is specified (that is, if your program monitors the backup process), then the possible values of PROGRAM-STATUS are as shown in Table 33-2.

Table 33-2 Values for PROGRAM-STATUS When STARTBACKUP Has Option 3

	First Digit of PROGRAM-STATUS						
Statement	**0**	**1**	**2**	**3**	**4**	**5**	**6**
STARTBACKUP	Yes	Yes	Yes	Yes	Yes	No	Yes
CHECKPOINT	Yes	Yes	Yes	No	No	No	Yes
OPEN	Yes	Yes	Yes	Yes	No	No	No
CLOSE	Yes	Yes	Yes	No	No	No	No

If the primary process fails and a proper takeover occurs, PROGRAM-STATUS is set to a take-over status values (such as "01*nn*") when the backup process begins executing. Successive checkpoints set PROGRAM-STATUS to other values (such as "1000").

The action your program should take in response to an exception status (a status other than "0000") depends upon your application and the location of the error in your HP COBOL program. A takeover might need special processing. Most of the other status values indicate a program coding problem or a system configuration problem. Report the latter problem to the appropriate authority for your system.

How far and whether processing should continue following a nontakeover status is a subject for careful analysis.

If a checkpoint failure occurs (that is, status "1000", "2*nnn*", or "3*nnn*"), the backup process might remain in an invalid state; therefore, the HP COBOL fault-tolerant facility stops the backup process if such a failure occurs. If the STARTBACKUP option 0, 1, or 2 was specified (signifying that the HP COBOL fault-tolerant facility is to monitor the backup process), the facility tries to restart the backup process at the next execution of the CHECKPOINT statement.

If the checkpoint failure persists for 10 restart cycles, the HP COBOL fault-tolerant facility stops reestablishing a backup process and sets PROGRAM-STATUS to "5000."

Example 33-2 Key Parts of Fault-Tolerant HP COBOL Program

```
?NONSTOP
?SAVE PARAM
*  Compiler directive specifies that the program is to be
*  compiled for execution as a process pair.
 FILE SECTION.
 FD TEST-FILE
    LABEL RECORD IS OMITTED
    RECORD CONTAINS 100 CHARACTERS
    DATA RECORD IS TEST-REC.
 01 TEST-REC.
    05  REC-NUM  PIC 9(4).
    05  ALTKEY-1 PIC X(5).
    05  ALTKEY-2 PIC X(5)....

 WORKING-STORAGE SECTION.
 01 CKPT-DATA.
 *   This defines pertinent control information that is
 *   checkpointed to the backup process
    05  KEYNUM         PIC 9          VALUE 0.
    05  CURR-REC       PIC 9(4).
    05  CR   REDEFINES CURR-REC.
        10  FILLER     PIC XX.
        10  CR-1       PIC 9.
        10  CR-2       PIC 9.
    05  OLD-BASE       PIC 9(4).
    05  NEW-BASE      PIC 9(4)       VALUE 0.
  77  BCPU   PIC S9(2) COMP VALUE -1.
  77  PARAM      PIC X(9)       VALUE SPACES.
  77  PARAM-RET  PIC X          VALUE SPACES.
  77  ERROR-CODE PIC S9(4) COMP VALUE 0.
  PROCEDURE DIVISION.
  CENTRAL SECTION.      ...
      OPEN I-O TEST-FILE SHARED SYNCDEPTH 1.
      PERFORM GET-CPUNUM.
      STARTBACKUP BCPU, 1.
      IF PROGRAM-STATUS NOT = "0000"
         DISPLAY "STARTBACKUP UNSUCCESSFUL, STATUS = ",
                 PROGRAM-STATUS.
    STOP RUN.
    PERFORM UPDATE-RECS
       UNTIL UPDATES-DONE.
   ...
GET-CPUNUM.
          MOVE "BACKUPCPU" TO PARAM.
   ENTER "SMU_Param_GetText_" USING PARAM PARAM-RET
                              GIVING ERROR-CODE.
   MOVE PARAM-RET TO BCPU.
   IF BCPU < 0 OR BCPU > 15
      DISPLAY "BACKUP CPU INVALID, RUNNING WITHOUT BACKUP".   UPDATE-RECS.
      ...
   READ TEST-FILE WITH LOCK.
      ...
   CHECKPOINT TEST-FILE,
              TEST-REC,
              CKPT-DATA.
   IF PROGRAM-STATUS > "0000"
   AND PROGRAM-STATUS < "1000"
      DISPLAY "Backup took over".
   REWRITE TEST-REC WITH UNLOCK.
      ...
```

After the program is compiled and debugged, it must be run as a named process (see Process Names (page 938)).

Designing Programs for the Fault-Tolerant Facility

You must design a program to use the fault-tolerant facility. It is very difficult, if not impossible, to retrofit the use of the fault-tolerant facility to an existing program.

A checkpoint is a location in the program at which processing resumes when a takeover occurs. All data from the primary process that is part of the program state must be checkpointed to the backup process if the takeover is to succeed.

You must checkpoint before any nonretryable operation, such as a write to disk. You can (for operator convenience) checkpoint after a read from a terminal. You can (for programming convenience) checkpoint after a read from tape.

If your SYNCDEPTH exceeds one, and you open or close a file during checkpoint-protected operation, you must checkpoint the sync blocks and record areas of all other files before the next nonretryable operation.

Debugging Programs That Use the Fault-Tolerant Facility

Debugging a program that uses the fault-tolerant facility requires at least as careful an analysis as designing the program. This list of suggestions is not exhaustive.

- Test the program thoroughly as a simple batch job. You might create files of transactions that the job must successfully process before you try testing the fault-tolerance mechanisms. You can have the program read the files instead of having the program read from $RECEIVE.
- You can compile your program with STARTBACKUP and CHECKPOINT statements in place, but disable them for batch testing by running the program with the run-time parameter NONSTOP OFF. With NONSTOP OFF, the program does not attempt to execute the STARTBACKUP and CHECKPOINT statements. To establish the run-time parameter, use the TACL command PARAM (or the SMU routine SMU_Param_PutText_ if you use a COBOL process to start your batch test programs).
- When you are convinced that the program works without being fault tolerant, start testing the fault tolerance. Execute the program as a process pair, and use a symbolic debugger to set breakpoints at which you stop the primary process. You can set different breakpoints in the primary and backup processes to determine whether the backup process takes over properly.
- When all controlled testing is complete, obtain the use of a dedicated system and try full integrated testing in which you run the application as a process pair and stop one of the two processors on which the application is running.

TMF

TMF simplifies the task of maintaining data consistency for distributed databases that are being updated by several independent processes.

TMF was developed to help solve two problems faced by designers of database processing applications:

- What happens if a transaction starts to change a database and fails to complete all of its changes?
- How can a database be recovered after a catastrophic, multicomponent system failure?

Before TMF was available, the answer to the first question was to write application code to back out partial transactions and restore the affected parts of the database to the condition they were in when the failed transaction began. Further, this application code had to be fault tolerant to overcome possible single points of failure.

The answer to question two was to stop the system periodically and make tape or disk backups. Then if a catastrophic failure occurred, you could restore the database from the tape or disk and redo the transactions that had been done since the backup. This required considerable design and coding effort, particularly if you had to do it for several applications.

TMF has packaged the solution to these two questions of backout and recovery. TMF automatically backs out incomplete transactions and provides a mechanism for periodically taking online dumps and a roll-forward procedure to automate the repetition of post-backup transactions.

Concepts

The basic concepts of TMF are:

- Transactions in General
- Multiple-File Transactions
- Multiple Changes to a Single File
- Defining Transactions
- Audit Trails

- Online Dump and Roll-Forward Recovery
- Record Locking

Transactions in General

A transaction is a multistep operation that affects the consistency of a database by transforming the database from one consistent state to a new consistent state. A transaction is also a unit of recovery. During a transaction, the database can be inconsistent. But either the entire transaction must happen, or none of it must happen.

Each transaction has a unique transaction identifier. The identifier includes information about the starting time of the transaction.

If your transaction involves more than one file or if the single file it involves is available to other transactions, you can introduce inconsistency into the database.

Multiple-File Transactions

Suppose your process is maintaining a corporate personnel system in which each department has its own file of information. You want to move an employee's record from one file to another. It does not matter whether the process writes the record in the new file before deleting it from the old file or the reverse. During the interval between the two events, the database is inconsistent. The employee record is either absent from the database or present in two places.

Multiple Changes to a Single File

Suppose that the corporate personnel system does not use multiple files, but a single file. If you want to move an employee's record from one department's portion of the file to another department's portion of the file, the same problem arises as that with multiple files. If the record is deleted from one place in the file and written to another place in the file, there is an interval during which the database is inconsistent.

Defining Transactions

Because you, as the application programmer, are in the best position to define what constitutes a transaction, TMF gives you the authority to specify the beginning and the end of the transaction.

In a Pathway application, the SCREEN COBOL terminal program uses the BEGIN-TRANSACTION and END-TRANSACTION statements for this purpose.

In pTAL and HP COBOL programs outside of the Pathway environment, you must call the operating system routines BEGINTRANSACTION and ENDTRANSACTION explicitly.

If a terminal program determines that a transaction cannot or must not complete, it uses the SCREEN COBOL ABORT-TRANSACTION statement to instruct TMF to back it out and not retry it. If a process written in pTAL or HP COBOL needs to abort a transaction, it calls the ABORTTRANSACTION routine.

If a terminal program determines that a transaction should be aborted and retried as a new transaction (because of a locking contention problem, for example), it can use the SCREEN COBOL RESTART-TRANSACTION statement to initiate the operation. The pTAL or HP COBOL process must explicitly use the ABORTTRANSACTION routine and handle its own restarting.

After TMF is informed of the boundaries of a transaction, it can assure that either the entire transaction is performed or none of it is performed. TMF does this by using audit trails.

Audit Trails

An audit trail is a set of disk files in which a record is kept of before-images and after-images of all records changed by each transaction that occurs in a certain file. A file is described as audited when it has an audit trail. In TMF, every disk file in which records are to be changed can be audited. If a disk is involved in transactions whose failure could leave the database in an inconsistent state, that disk file should be audited.

On NonStop systems, under DP2 (Disk Process 2), all disks permit TMF auditing. To instruct TMF to audit a file, use the attribute AUDITED, which you set with the FUP command SET or ALTER. The AUDITED attribute is an attribute of the disk file itself; an HP COBOL program is unaware of and cannot affect this attribute.

Every disk operation that changes the contents of a database protected by TMF is recorded in an audit trail. The before and after values of the information are both recorded, and the database is changed. If the transaction does not complete, TMF backs out the transaction—replaces the changed value with the previous value.

You can specify that some files in a transaction are audited and some are not audited; however, this practice is risky. If a server updates an unaudited file having alternate keys, a single-component failure can interrupt a transaction between the updating of the prime key and alternate keys. In this case, the prime and alternate keys are inconsistent. If you do not protect such a file with auditing, you must code the server as a process pair to protect the consistency of the keys.

The TMF subsystem requires audit trails for both transaction backout and roll-forward recovery. Eventually, the space required for audit trails can exceed available resources. For this reason, the audit trails must periodically be dumped to tape and their disk space thereby reclaimed. The system manager can configure TMF to automatically dump the audit trails.

Even after a transaction is committed and its audit trail is no longer needed for backout, you might need the audit trail for a roll-forward recovery; therefore, tapes containing the dumped audit trails should be stored until you are certain that they are no longer needed.

Online Dump and Roll-Forward Recovery

The TMF subsystem can perform an online dump of the database periodically. The system manager is responsible for configuring TMF to do this at appropriate intervals. In the event of a total system failure, the operator directs TMF to restore the database from dump files. Then the operator directs TMF to conduct a roll-forward recovery to repeat the transactions that have occurred since the last online dump. A roll-forward recovery is the action of repeating each transaction that occurred to the database between the time of the last online dump and the time of the system failure.

Record Locking

As a transaction progresses in TMF, these types of resources must be locked:

- Records that are changed or inserted into the database
- Records whose values affect the course of the transaction
- Keys of deleted records

If you are going to rewrite a record in an audited file, you must first lock it when you read it. The TMF subsystem automatically locks records that are inserted and keys of records that are deleted.

If a process attempts to rewrite a record in an audited file without that record being locked, the operation terminates with a file system error 80 (invalid operation attempted on audited file or non-audited disk volume) and file status code "30" (permanent error).

If the process executes an UNLOCK operation on a record that was locked and modified in an audited file, TMF prevents the record from actually being unlocked until the transaction is completed or aborted. The TMF subsystem automatically unlocks these locked records at the end of a transaction, whether the process explicitly unlocks them or not. If there is any chance that the same logic will be run on audited files at some times and on non-audited files at other times, it is best to specify explicit unlocking operations.

If you unlock a record in a nonaudited file in TMF or unlock a record that you locked but did not modify, the unlock operation occurs immediately.

Designing Programs for the TMF Subsystem

Like any other programming task, the most important part of designing for TMF is planning. Someone must plan the training, hardware, development, installation, testing, and operation.

Before development can proceed, the designers must be trained in TMF. Eventually, some training is necessary for everyone involved.

When the designers understand TMF, the project must be planned. Identify the transactions in the system. Do all transactions need to be protected by TMF? Are the transactions of appropriate size (number of SEND operations and number of screens involved)?

The entire installation must be scheduled, whether you are converting to TMF, adding a TMF project on a system that is already using TMF, or creating a new system.

Testing must be planned. Application code should be tested first without TMF, then with TMF. Operations and development groups must practice recovery procedures.

Daily operation must be planned. Determine the frequency of online dumps and whether the dump is to tape or disk. Determine how the archival tapes or disks are transported to storage and how they are retrieved. Have a disaster recovery script. Practice disaster recovery occasionally.

The TMF Subsystem and Requester Screen Transactions

Because the server's TMF protection is governed by the requester, you need to coordinate the designs of the two. If the terminal user needs to traverse more than one screen for each transaction, you must consider how to package the transaction.

Transactions that consist of several screens are easy and straightforward to program but can be expensive. If a transaction spans several screens, it can keep records locked for long periods; however, if you design for single-screen transactions, you might have some problems when a series of screens makes up a single conceptual transaction.

You recover from a failure during the series by adopting a strategy to protect records by allocating a field to mark the record "in use." Then other transactions can read the record but agree not to attempt to change it. This mechanism puts more responsibility on the designer and the maintenance staff. The "in-use" field identifies the terminal that is using the record so that, in the event of major failure and recovery, the restored system can resume properly.

You can also collect information from the database without locking records and pass the information back to the requester. Then, when the requester makes its final request to update the database, it can send the server the before and after versions of the data, and the server can read and lock the records. Then the server can verify that the records have not been changed and can rewrite the new records, or the server can respond to the requester with the new information and have the requester ask the terminal operator what to do with the record in its revised form. This strategy increases the context that is carried by the requester, which affects performance.

The TMF Subsystem for Batch Jobs

The TMF subsystem is appropriate for any batch job where you cannot afford the time to rerun it if it fails near its completion. Suppose your system must be available to process interactive transactions daily from 6 AM to 9 PM, and overnight a 5-hour batch job updates the database. There are only 9 hours between the suspension and resumption of interactive work. In that 9 hours, you must dump the database to tape and then update it. If the update fails, you must reload from the tape and repeat the update. There is not enough time to do this if the failure occurs a few hours into the batch update.

In such a batch update, the update cannot be one big transaction. Not only must the audit trail for each transaction remain on disk until the transaction is complete, but backing out the incomplete transaction takes about as long as the transaction had been running at the time of the failure; however, the I-O overhead of making each batch transaction a TMF transaction might be burdensome. It is best to cluster about 20 to 100 batch transactions as one TMF transaction.

Debugging Programs That Use the TMF Subsystem

There are no debugging tools specific to TMF application debugging. You can use a symbolic debugger to set breakpoints in both SCREEN COBOL terminal programs and in server programs. If the symbolic debugger that you use can accept its commands from an OBEY command file, you can create a script that interrupts transactions at various points in the requester and the server and tests the recovery mechanisms.

34 Migrating TNS/R Programs to TNS/E Programs

NOTE: This section applies to migrating TNS/R HP COBOL programs to TNS/E HP COBOL programs, and assumes that you are running RVU G06.20 or later. To migrate TNS HP COBOL programs to TNS/R or TNS/E HP COBOL programs, see the *COBOL Manual for TNS and TNS/R Programs*.

HP provides these native compilers for HP COBOL source programs:

Compiler	Objects Produced
NMCOBOL	TNS/R
ECOBOL	TNS/E

The ECOBOL compiler is available both on NonStop systems and on the PC. In this section, “ECOBOL compiler” refers to both of the ECOBOL compilers, unless otherwise noted.

Taking an HP COBOL source program that was compiled with the NMCOBOL compiler and recompiling it with the ECOBOL compiler is called “migrating from TNS/R to TNS/E.” Migrating to TNS/E might require source program changes.

You can write HP COBOL source programs that can be compiled with any version of the NMCOBOL or ECOBOL compiler.

Reason to Migrate

An HP COBOL program compiled with the NMCOBOL compiler will not run on a TNS/E system. If you want to run an HP COBOL program on a TNS/E system, you must compile it with the ECOBOL compiler.

Migrating HP COBOL Programs

The recommended procedure for migration from TNS/R HP COBOL to TNS/E HP COBOL is:

1. **If necessary, change your source program (see Source Program Changes).**

   To learn whether you must change your source program, compile it with the ECOBOL compiler, which will issue warnings when it encounters source constructs that it does not accept.

2. **If your program calls TNS/R programs, migrate them to TNS/E.**

   TNS/R and TNS/E programs cannot call each other. For general migration information, see the *H-Series Application Migration Guide*.

   For this list of languages, follow these instructions. The last one, Data Alignment, applies to several languages.

   - **HP C and HP C++**

     Recompile HP C and HP C++ programs with the TNS/E HP C and HP C++ compilers, respectively. Specify the `SYMBOLS` pragma when recompiling an HP C or HP C++ program that your HP COBOL program references in a CALL or ENTER statement (so that the native compiler can validate calls and parameters to the HP C or HP C++

module). For information on tools that can help you migrate HP C or HP C++ programs to TNS/E, see the *C/C++ Programmer's Guide*.

- **pTAL**

  Recompile pTAL programs with the EpTAL compiler, using the `SYMBOLS` directive if you want to reference a pTAL routine in a CALL or ENTER statement. For information about the EpTAL compiler, see the *pTAL Reference Manual*.

- **Data Alignment**

  The ECOBOL compiler aligns each level-01 item and each level-77 item on a physical 8-byte boundary, not on a physical 2-byte boundary as the NMCOBOL compiler does. Offsets from the containing level-01 or level-77 item are the same in both compilers. The difference in alignment of level-01 and level-77 items for the ECOBOL compiler will not affect a program unless the program depends on the relative placement in memory of distinct level-01 or level-77 items. If your program depends on such relative placement, correct it.

3. **(Optional) Put converted routines in a DLL.**

   If you want to put the routines that you converted to native mode in Step 2 in a DLL, follow the directions in Dynamic-Link Libraries (DLLs) (page 608)).

4. **Compile your source program with the ECOBOL compiler.**

   For the Guardian environment, see Running the Compiler (page 536). For the OSS environment, see Chapter 20: Using HP COBOL in the OSS Environment (page 721).

   The ECOBOL compiler needs more symbol table space than the NMCOBOL compiler does. If the ECOBOL compilation fails due to dictionary overflow, use the PARAM SYMBOL-BLOCKS (page 537) command to increase the space available for the symbol table and embedded SQL/MP statements and then recompile.

5. **Run the HP COBOL program that you compiled in Step 4.**
6. **If necessary, debug the program.**

Source Program Changes

Source program changes fall into these categories:

- General Migration Tasks
- Removal Required
- Possible Changes Required
- Removal Optional

General Migration Tasks

If your HP COBOL program calls obsolete or changed Guardian procedures, replace them. Change any calls to procedures affected by either the Kernel Managed Swap Facility (KMSF) or the native process architecture (for example, process creation calls). For more information on the obsolete or changed procedures, see the *H-Series Application Migration Guide*.

Removal Required

Replace any references to TNS/R libraries with references to the analogous TNS/E libraries:

Replace ...	With ...
NMCOBEX0	ECOBEX0
NMCOBEX1	ECOBEX1
NMCOBEXT	ECOBEXT

References to TNS /R libraries can appear in these contexts:

- CONSULT directive
- SEARCH directive
- File-mnemonic clause of the SPECIAL-NAMES paragraph
- OF or IN clause of the CALL or ENTER statement

Possible Changes Required

- Directives
- RENAMES Clause

Make any necessary changes to these items before compiling your HP COBOL source program with the ECOBOL compiler.

Directives

- **CONSULT Directive**

  For the NMCOBOL compiler, each object-name in a CONSULT directive must designate a TNS/R native object file (otherwise the compiler reports an error).

  For the ECOBOL compiler, each object-name in a CONSULT directive must designate a TNS/E native object file (otherwise the compiler reports an error).

  For a complete description of this directive, see CONSULT and NOCONSULT (page 552).

- **OPTIMIZE Directive**

  The NMCOBOL compiler handles OPTIMIZE 2 as if it were OPTIMIZE 1, because native COBOL does not support global optimization. The ECOBOL compiler handles OPTIMIZE 2 as the COBOL85 compiler does.

	Effect	
Level	**NMCOBOL Compiler**	**ECOBOL Compiler**
0	Code is not optimized. Provided in case other optimization levels cause errors. Supports symbolic debugging; data is always in memory.	Code is not optimized. Provided in case other optimization levels cause errors. Supports symbolic debugging; data is always in memory.
1 (default)	Code is optimized within statements and across statement boundaries. The resulting code is more efficient than that produced by lower levels of optimization.	Code is optimized within statements, but not across statement boundaries. Appropriate for application programs still being developed and tested. Supports symbolic debugging; data is not always in memory.
2	Uses level 1 instead.	Code is optimized within statements and across statement boundaries, and the resulting code is more efficient than code produced by lower levels.

  OPTIMIZE 0 (no optimization) is recommended when you are debugging a program using the ECOBOL compiler. OPTIMIZE 1 (most optimizations) is recommended for production.

  For a complete description of this directive, see OPTIMIZE (page 567).

- **SEARCH Directive**

  For the NMCOBOL compiler, each object-name in a SEARCH directive must designate a TNS/R native object file (otherwise the compiler reports an error).

  For the ECOBOL compiler, each object-name in a SEARCH directive must designate a TNS/E native object file (otherwise the compiler reports an error).

For a complete description of this directive, see SEARCH and NOSEARCH (page 573).

- **UL Directive**

  For the NMCOBOL compiler, the UL directive generates code for a user library. For the ECOBOL compiler, the UL directive generates code for a DLL (that is, it has the same effect as the SHARED directive).

  For a complete description of this directive, see UL (page 588).

RENAMES Clause

The RENAMES clause must not rename a level-01 data item. The NMCOBOL compiler does not detect this error, but the ECOBOL compiler does.

If your TNS/R HP COBOL program has a RENAMES clause that renames a level-01 data item, change your source code before compiling it with the ECOBOL compiler.

If the level-01 data item is elementary, change the RENAMES clause to a REDEFINES clause. For example, change:

```
01 CARD-COUNTER PIC 9(6).
66 ITEM-COUNT RENAMES CARD-COUNTER.
```

To:

```
01 CARD-COUNTER PIC 9(6).
66 ITEM-COUNT PIC 9(6) REDEFINES CARD-COUNTER.
```

If the level-01 data item is a structure, rename the first subordinate data item through the last subordinate data item . For example, change:

```
01 CARD-REC.
   05 REFERENCE-NUMBER PIC 9(6).
   05 CARD-CODES.
      10 STORE-CODE PIC 9.
      10 STATE-CODE PIC 9(4).
   05 ACCOUNT-NUMBER PIC 9(6).
   05 CHECK-DIGIT PIC 9.
   66 CARD-DATA RENAMES CARD-REC.
```

To:

```
01 CARD-REC.
   05 REFERENCE-NUMBER PIC 9(6).
   05 CARD-CODES.
      10 STORE-CODE PIC 9.
      10 STATE-CODE PIC 9(4).
   05 ACCOUNT-NUMBER PIC 9(6).
   05 CHECK-DIGIT PIC 9.
   66 CARD-DATA RENAMES REFERENCE-NUMBER THRU CHECK-DIGIT.
```

For a complete description of the RENAMES clause, see Descriptions That Rename Items (Level 66) (page 231). For a complete description of the REDEFINES clause, see REDEFINES Clause (page 198).

Removal Optional

The ECOBOL compiler ignores these items, so you can (but need not) remove them from your source program.

- **LARGEDATA**

  For the NMCOBOL compiler, the LARGEDATA directive determines whether individual data items are located in the small data area or the large data area. The default value for the LARGEDATA directive’s parameter is 64.

The ECOBOL compiler ignores the LARGEDATA directive and issues a warning.

- **LESS-CODE**

  For the NMCOBOL compiler, the LESS-CODE directive determines whether the program generates code that initializes the Working-Storage Section or Extended-Storage Section or uses a system call to initialize the Extended-Storage Section.

  The ECOBOL compiler ignores the LESS-CODE directive and issues no warning.

- **NON-SHARED Directive**

  For the NMCOBOL compiler, the NON-SHARED directive generates nonshared code (non-PIC). The ECOBOL compiler, which always generates PIC, ignores the NON-SHARED directive and issues a warning.

Maintaining Common Source Code

To maintain “common source code” that can be compiled with D44.01 and later RVUs of the NMCOBOL compiler and the ECOBOL compiler, follow these guidelines:

- Either do not use HP COBOL language features that are not supported by both compilers, or use conditional compilation to isolate those features.

  You can effect conditional compilation with these directives:

  — IF and IFNOT (page 561)

  — ENDIF (page 557)

  — SETTOG (page 575)

  — RESETTOG (page 570)

- Use separate build scripts to run TNS and native tools, including the NMCOBOL and ECOBOL compilers.
- For each compilation, use the appropriate file (or files) of dummy routines.

  Files of dummy routines enable the compilers to accept ENTER statements that are to be resolved at program load time (see Table 14-4: Files of Dummy Routines (page 617)).

Using the Inspect, Native Inspect, and Visual Inspect Debuggers

NOTE: You can debug TNS/E PIC files with either Visual Inspect or Native Inspect.

The Inspect and Visual Inspect interactive symbolic debuggers handle TNS and native HP COBOL programs differently. For a TNS/R native HP COBOL program, the Inspect debugger’s STEP command is line-oriented. If the current statement is on a single line, the STEP command steps to the next statement (that is, the debugger works as the command STEP 1 V works for a TNS HP COBOL program). If the current statement is on the same line as another statement or statements, the STEP command steps to the next line on which a new statement starts. The Inspect debugger cannot step one sentence in a native HP COBOL program (that is, the command STEP 1 S does not work).

The Native Inspect debugger replaces Inspect as the TNS/E command line debugger on the NonStop server. Native Inspect is based on the industry-standard GNU GDB debugger. It provides most of the same capabilities as Inspect, but has a different command syntax.

35 Native COBOL Cross Compiler on PC

The native COBOL cross compiler is an optional cross compiler that runs on the PC platforms listed in Table 35-1.

Table 35-1 ECOBOL Cross Compiler Platforms

			Operating System			
Platform	**Cross Compiler Name**	**Product Number**	**Windows 98**	**Windows NT 4.0**	**Windows 2000**	**Windows XP**
ETK[1]	NonStop COBOL for TNS/E	T0356	No	No	Yes	Yes
PC command line	`ecobol`	T0356	No	No	Yes	Yes

[1] HP Enterprise Toolkit—NonStop Edition

Cross Compiler Features

The native COBOL cross compiler allows you to:

- Write, compile, and link these kinds of NonStop Itanium-based server applications on the PC and transfer them to the OSS or Guardian platform for use in production:
  - Executable programs
  - Static libraries
  - User libraries
  - DLLs

  Object files built on the PC platform using the native COBOL compiler are compatible with object files built on the NonStop Itanium-based server platform using the ECOBOL compiler.

> **NOTE:** The native COBOL cross compiler version on your PC must be compatible with the COBOL run-time library on your NonStop Itanium-based server.

- Link ECOBOL, TNS/E C/C++, and EpTAL objects into a single object file.
- When multiple RVUs are installed, choose any installed RVUs of the cross compilers, tools, and libraries.
- On the ETK platform, enter ADD, MODIFY, SET, and DELETE statements into a TACL DEFINE file [see TACL DEFINE Tool (ETK) (page 989)].
- Compile SQL/MP or SQL/MX statements embedded in native COBOL source code.

  Your PC must be connected to the NonStop host to perform certain SQL/MP or SQL/MX compile-time operations and to run your applications.

The native COBOL cross compiler is delivered on a separate independent product CD using Scout for NonStop Servers, but it is not available on the site update tape (SUT) for the ECOBOL compiler.

NonStop COBOL for TNS/E (ETK)

The optional native COBOL cross compiler for use with the ETK is called NonStop COBOL for TNS/E. The ETK is a GUI-based extension package to Visual Studio.NET that provides full application development functions targeted for NonStop servers. From the GUI, you can choose native COBOL cross-compiler options. Development, editing, and building functions are very similar on Visual Studio.NET and the ETK.

NonStop COBOL for TNS/E components are:

Component Name	File
NonStop ECOBOL driver executable	`ecobol.exe`
NonStop ECOBOL front end	`ecobfe.exe`

The directory structure of NonStop COBOL for TNS/E is:

Directory	Files
`bin`	`ecobol.exe` `eld.exe`
`cmplr`	`ecobfe.exe`

`lib`	`ecobext.o`
	`libcob.so`
	libraries in `libc.obey`

You can embed SQL/MP or SQL/MX statements in NonStop COBOL for TNS/E source code. Valid NonStop COBOL for TNS/E source files have these extensions:

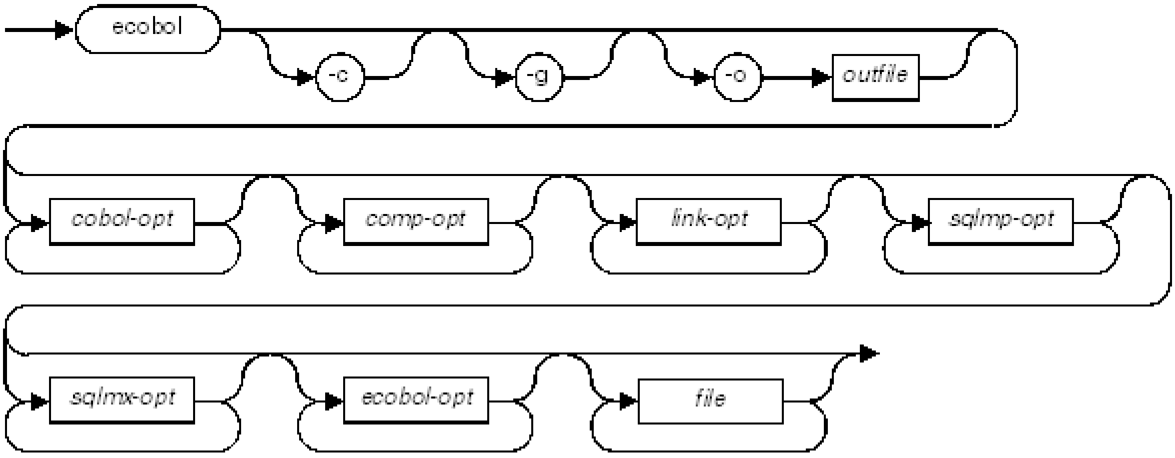
Extension	Meaning
`.cob` or `.cbl`	COBOL source code with or without embedded SQL/MP or SQL/MX statements
`.ecob` or `.ecbl`	COBOL source code with embedded SQL/MP or SQL/MX statements

Your PC must be connected to the NonStop host to perform certain SQL/MP or SQL/MX compile-time operations and to run your applications.

For PC and NonStop server hardware and software requirements, see the ETK online help. For instructions for accessing the online help, see Documentation.

ecobol (PC Command Line)

You can invoke the native COBOL cross compiler, `ecobol`, at the command line (DOS prompt) on your PC.



VST821.vsd

ecobol

must be typed as shown, in lowercase letters.

-c, -g, -o *outfile*

are as described in the *Open System Services Shell and Utilities Reference Manual*.

cobol-opt

is one of these ecobol flags:

cobol-opt	Sources
-Wcobol="*arguments*"	*Using the Command-Line Cross Compilers on Windows* (see Documentation)
-Wcopylib=*file*	*Open System Services Shell and Utilities Reference Manual*

comp-opt

is one of these ecobol flags:

- -I *directory*
- -O [0 | 1]
- -Wcall_shared
- -Wcodecov
- -Wcolumns=*N*
- -Werrors=*N*
- -Wglobalized
- -W[no]innerlist
- -W[no]list
- -W[no]map
- -Wmigration_check
- -Woptimize={0|1|2}
- -Wshared
- -Wstandard={1985|2002}
- -W[no]suppress
- -Wsyntax
- -Ww
- -Wsystype={guardian|oss}

-Wsystype={guardian|oss} determines whether ecobol creates loadfiles for the Guardian environment or for the OSS environment. The default is the Guardian environment.

For descriptions of the other flags, see the *Open System Services Shell and Utilities Reference Manual*.

link-opt

is one of these ecobol flags:

- -L
- -l
- -s
- -Wansistreams
- -WBdllsonly
- -WBdynamic
- -WBstatic
- -Weld
- -Weld_obey

- -Wheap
- -Whighpin
- -Whighrequesters
- -W[no]include_whole
- -W[no]inspect
- -Wnostdlib
- -W[no]optional_lib
- -Wr
- -W[no]reexport
- -Wrunnamed
- -W]no]saveabend
- -Wu
- -Wx

sqlmp-opt

is described in *Using the Command-Line Cross Compilers on Windows* (see Documentation).

sqlmx-opt

is described in *Using the Command-Line Cross Compilers on Windows* (see Documentation).

ecobol-opt

is one of these ecobol flags:

- -Wdryrun
- -Whelp
- -Wsavetemps
- -Wusage
- -Wv
- -Wverbose

file

is the source file name.

On the Guardian platform, you issue these commands (if desired) before you issue the compilation command:	On the PC, you put these arguments in the -Wcobol flag in the compilation command:
DEFINE[1]	DEFINE*name* [=] *value*
PARAM SYMBOL-BLOCKS *count*[2]	SYMBOL-BLOCKS [=] *count*

[1] For information about DEFINE, see DEFINEs (page 601)

[2] For information about PARAM SYMBOL-BLOCKS, see PARAM SYMBOL-BLOCKS (page 537).

You can embed SQL/MP or SQL/MX statements in Native COBOL Cross Compiler for TNS/E source code. Valid Native COBOL Cross Compiler for TNS/E source files have these extensions:

Extension	Meaning
.cob or .cbl	COBOL source code with or without embedded SQL/MP statements
.ecob or .ecbl	COBOL source code with embedded SQL/MX statements

To complete SQL/MP or SQL/MX compilation, you must specify the host and the user logon. Specifying the NonStop server location (Guardian subvolume or OSS directory) is optional. For more information, see *Using the Command-Line Cross Compilers on Windows*.

The command-line interface allows you to create batch scripts for use on multiple platforms.

Linking

Native COBOL cross compiler linking is performed using Windows `eld`. The ETK provides a GUI-based interface for you to select linking options. When you invoke `eld` through the command line, you must specify the run-time libraries to `eld`. For information about Windows `eld` and `eld` options, see the *eld Manual*.

Debugging

- ETK Platform
- Command-Line Platform
- RUNV

ETK Platform

On the ETK Platform, debug native COBOL source code using Visual Inspect. After it is installed on your workstation, you can configure Visual Inspect as an external tool (in the ETK).

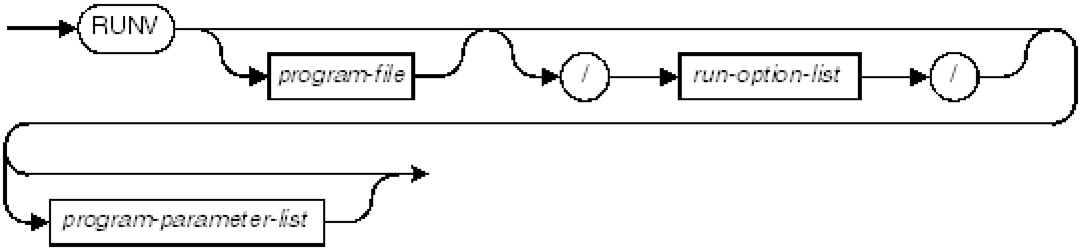
Command-Line Platform

On the command-line platform, debug loadfiles that were compiled with the native COBOL cross compiler either with Visual Inspect on Windows (some restrictions might apply for files containing SQL/MX statements) or by running NativeInspect on the NonStop server. To use Native Inspect, you must copy the loadfiles and the source files to the host (see PC-to-NonStop-Host Transfer Tools).

For information about Visual Inspect, see its online help. For information about Native Inspect, see the *Native Inspect Manual*.

RUNV

RUNV, which is a TACL macro in the Guardian environment and a script in the OSS environment, starts a program under the control of Visual Inspect.



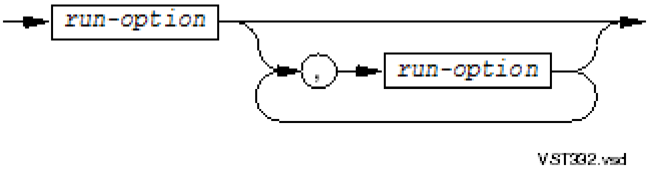
`RUNV`

without *`program-file`*, displays the RUNV help.

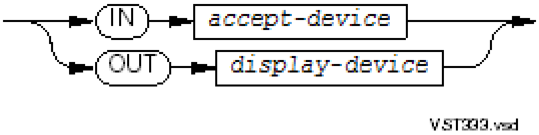
`program-file`

is the name of the loadfile to run. *`program-file`* can be represented as a file-system file name or (in the Guardian environment) a DEFINE name. *`program-file`* is added to the list in Visual Inspect's Application Control View. For information about Visual Inspect, see its online help.

run-option-list



run-option



is an option of the RUN command, described in the *TACL Reference Manual*. For COBOL programs, two of the run options, IN and OUT, have special significance:

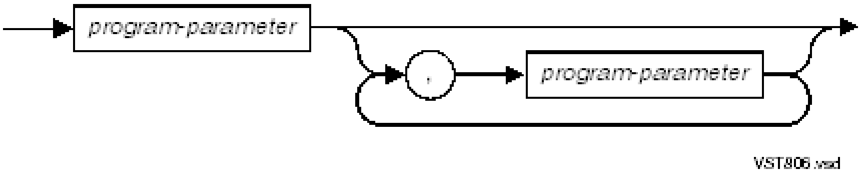
accept-device

has a value that designates a terminal or a process and can be a file-system file name or (in the Guardian environment) a DEFINE name. *accept-device* specifies the device from which an unqualified ACCEPT statement retrieves input. If you omit this option, unqualified ACCEPT statements retrieve input from the home terminal (unless PARAM EXECUTION-LOG specifies a different device).

display-device

can be a file-system file name or (in the Guardian environment) a DEFINE name. Its value designates a terminal, line printer, process, existing entry-sequenced disk file, or operator console to which unqualified DISPLAY statements are to deliver output. If you omit *display-device*, unqualified DISPLAY statements deliver output to the home terminal (unless PARAM EXECUTION-LOG specifies a different device).

program-parameter-list



program-parameter

is a parameter of the program that is to be run.

In the Guardian environment, the NonStop TCP/IP address of your workstation is automatically detected when you invoke RUNV.

In the OSS environment, you must either set the _TANDEM_VISUALINSPECT_WSADDR environment variable in your $HOME/.profile file or enter the workstation IP address as a command-line parameter. RUNV reads the IP address from the environment variable or from the command line. If you have set the environment variable but enter a different IP address on the RUNV command line, the command-line entry overrides the environment variable definition. You define the workstation IP address in your $HOME/.profile file only once unless the IP address changes.

Tools and Utilities

These tools and utilities allow you to use the native COBOL cross compiler more efficiently:

- NonStop ar Utility
- TACL DEFINE Tool (ETK)
- PC-to-NonStop-Host Transfer Tools

NonStop ar Utility

The NonStop `ar` utility creates and maintains archive libraries. The NonStop `ar` utility can process a set of arguments that you enter, or you can use an OBEY command file to supply arguments to the `ar` utility.

When an archive contains one or more TNS/E (native) object files, Windows `eld` can use the archive as an object file library, replacing most functions provided by the Binder SELECT SEARCH command. For information about Windows `eld`, see the *eld Manual*.

The Windows-hosted `ar` utility does not differentiate between OSS and Guardian target files. To avoid run-time errors, be sure that procedures in archives used for resolving references work on the target platform.

NOTE: The `ar` utility can also use archives created from TNS/E linkable files by the `eld` utility. For information about the `eld` utility, see the *eld Manual*.

TACL DEFINE Tool (ETK)

On the ETK platform, this GUI-based tool allows you to add ADD, MODIFY, SET, and DELETE statements to a DEFINE file. The TACL DEFINE tool automatically sets the first entry in the DEFINE command file to be SET DEFMODE ON. You can leave this default or change it to SET DEFMODE OFF. Files created by the TACL DEFINE tool have the extension `.tdf`.

PC-to-NonStop-Host Transfer Tools

The ETK allows you to transfer loadfiles and source files to the NonStop host using the Deploy command or the Transfer Tool. Deploy builds and copies each project in the active solution to the NonStop host. The Transfer Tool moves files to the NonStop host for execution and debugging. The Transfer Tool is better for transferring very large, complex applications to the NonStop host. For most applications, Deploy is more convenient.

From the PC command line, you can use any FTP application to transfer loadfiles and source files to the NonStop host.

Documentation

The ETK has online help that provides conceptual, reference, task-oriented, and error message information, as well as a quick-start tutorial. To access the online help, you can either:

- From the Help menu, choose Contents, Index, or Search.
- Click the Help in any ETK dialog box.

The command-line documentation, *Using the Command-Line Cross Compilers on Windows*, is available:

- On the native COBOL cross compiler CD
- In the ETK online help under References

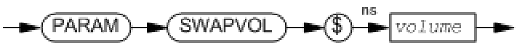
Syntax information for individual cross compilers is also available from the command-line:

Cross Compiler	Command
TNS/E C/C++	`c89 -Whelp`
ECOBOL	`ecobol -Whelp`
EpTAL	`eptal -Whelp`

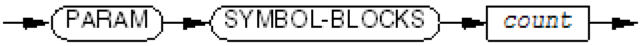
36 Commands

Guardian Environment

Before Running the Compiler (Optional)

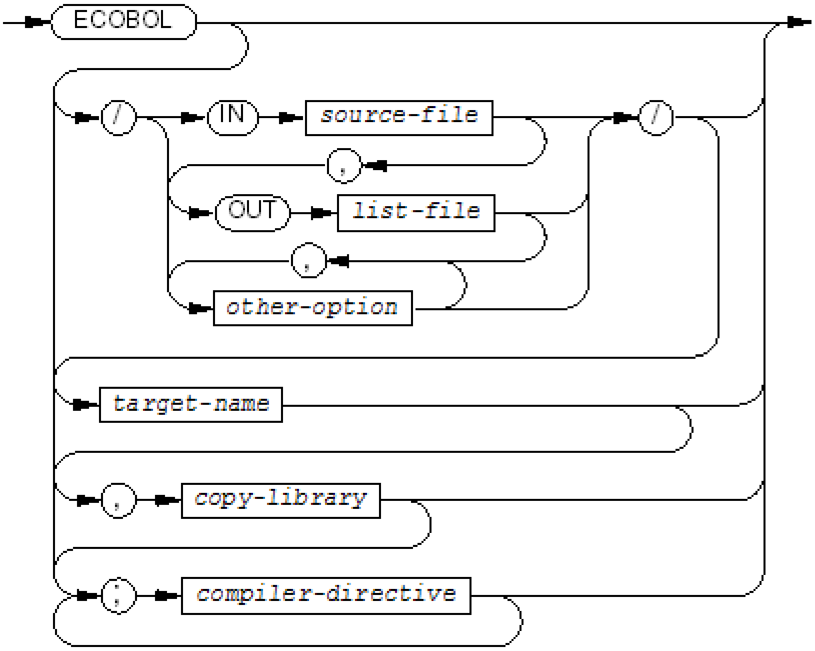


VST418.vsd



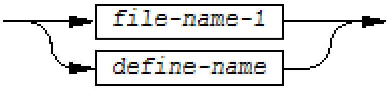
VST419.vsd

Running the ECOBOL Compiler



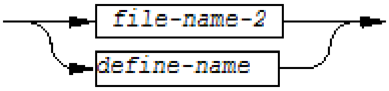
VST807.vsd

source-file



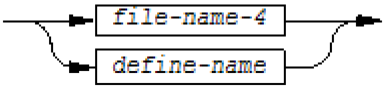
VST257.vsd

list-file



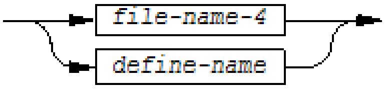
VST259.vsd

target-name



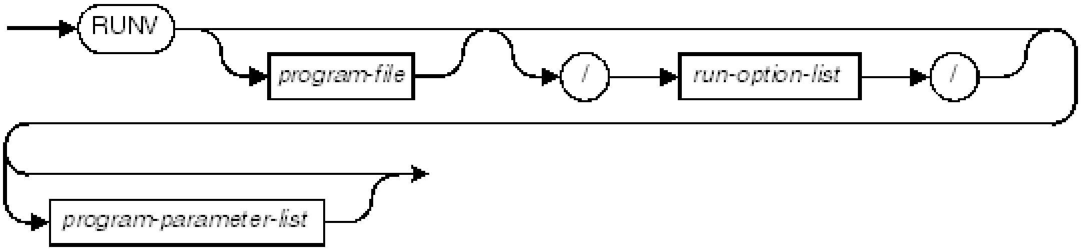
VST261.vsd

copy-library



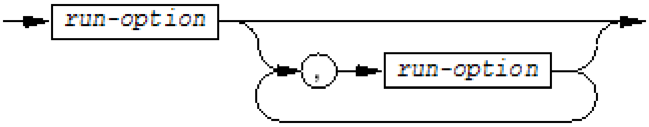
VST261.vsd

Running the Native Cross Compiler Under Visual Inspect



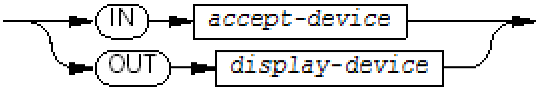
VST805.vsd

run-option-list



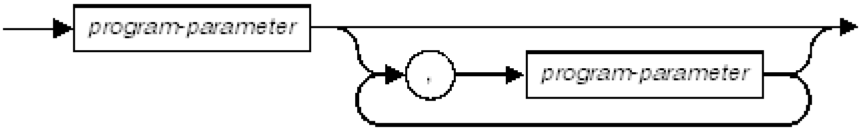
VST392.vsd

run-option



VST393.vsd

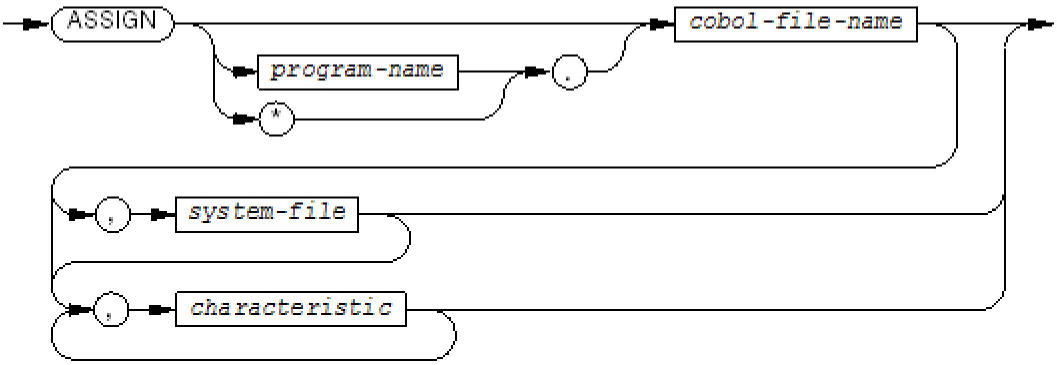
program-parameter-list



VST806.vsd

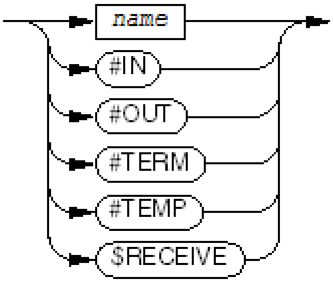
Before Running the Compiled Program (Optional)

ASSIGN



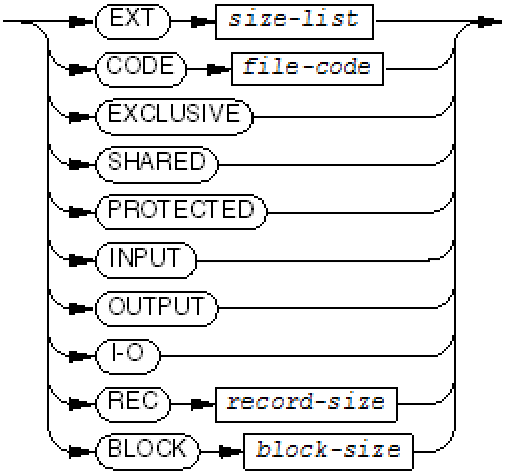
VST394.vsd

system-file



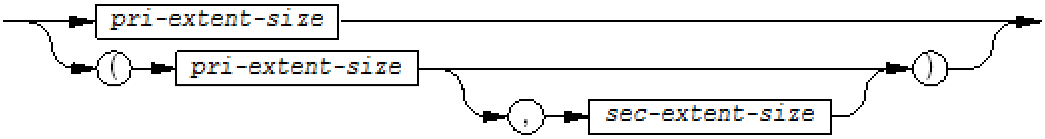
VST395.vsd

characteristic



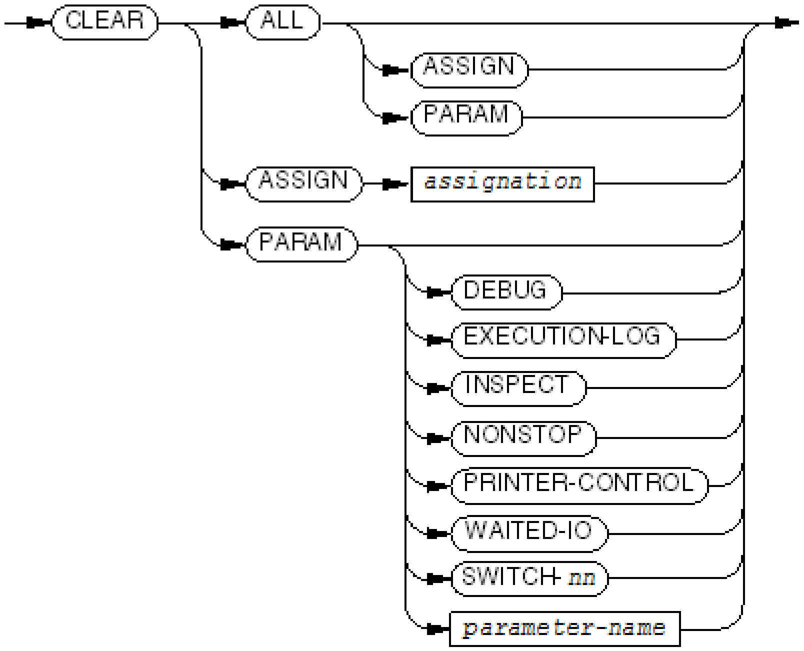
VST396.vsd

size-list



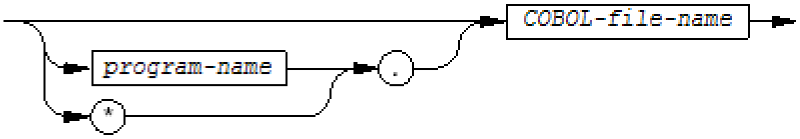
VST397.vsd

CLEAR



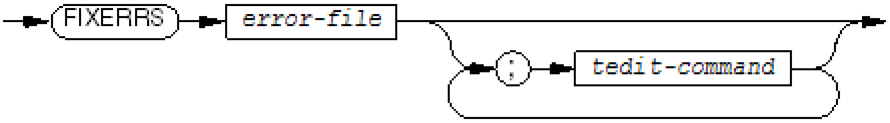
VST340.vsd

assignation



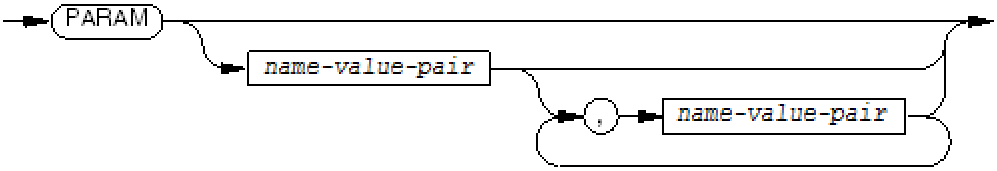
VST698.vsd

FIXERRS



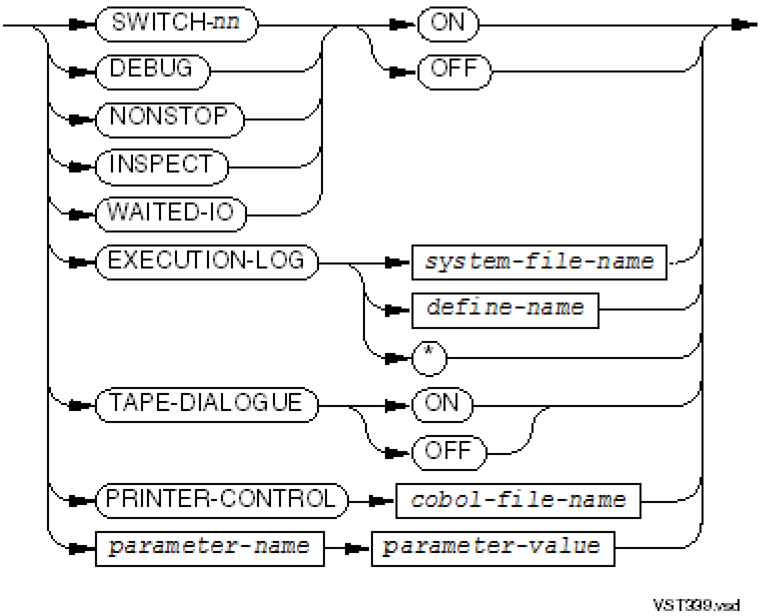
VST372.vsd

PARAM

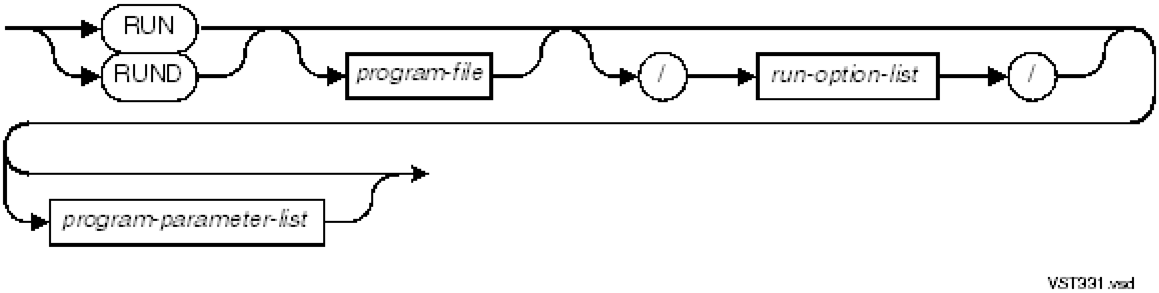


VST338.vsd

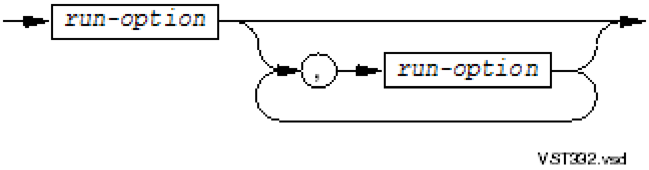
name-value-pair



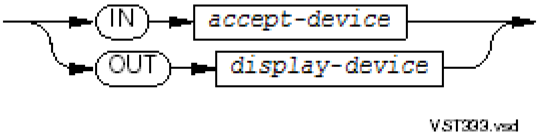
Running the Compiled Program



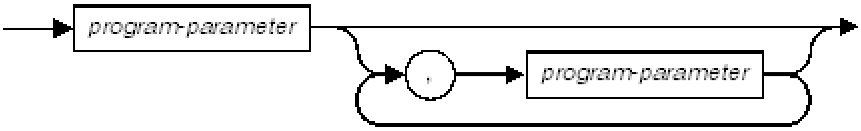
run-option-list



run-option



program-parameter-list



VST806.vsd

OSS Environment

Before Running the Compiler (Optional)

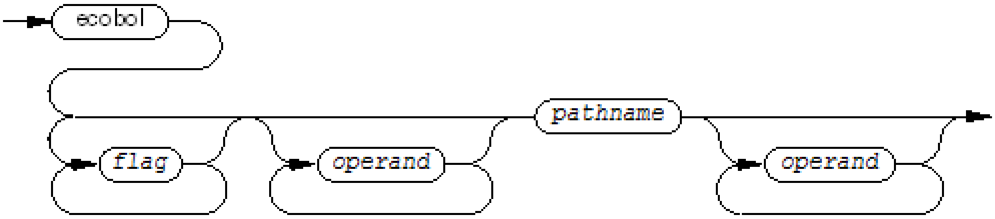
To change one or more of the defaults in Table 36-1 before executing the `ecobol` command, use the OSS export command.

To execute an `ecobol` command with a specified set of environment variables, use the OSS `env` function with the environment variables in Table 36-1.

Table 36-1 Environment Variables

Variable	Effect	Default
ECOBOL	Determines the pathname of the ECOBOL compiler that the `ecobol` utility calls	`/G/system/system/ecobfe`
SQLCOMP	Determines the pathname of the SQL/MP compiler that the `ecobol` utility calls	`/G/system/system/sqlcomp`
MXSQLCO	Determines the pathname of the alternate COBOL SQL/MX preprocessor that the `ecobol` utility calls	`/usr/tandem/sqlmx/bin/`
MXCMP	Determines the pathname of the alternate SQL/MX compiler that the `ecobol` utility calls	`/G/system/system/`
SQLCLIO	Determines the pathname of the object file that describes the SQL/MX API to the `ecobol` utility	`/usr/tandem/sqlmx/lib/esqlcli.o`

Running the ecobol Compiler

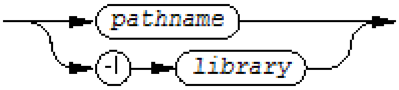


VST816.vsd

flag

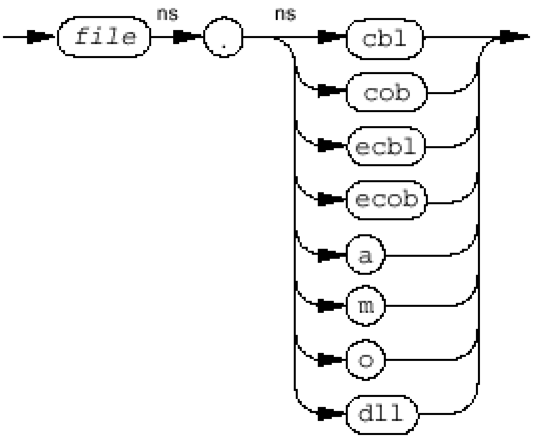
is as described in the *Open System Services Shell and Utilities Reference Manual*.

operand



VST622.vsd

pathname



VST817.vsd

NOTE: The file suffixes (`cbl`, `cob`, and so on) are not case-sensitive.

Running the Compiled Program

Type the name of the loadfile and press Return; for example:

```
a.out
```

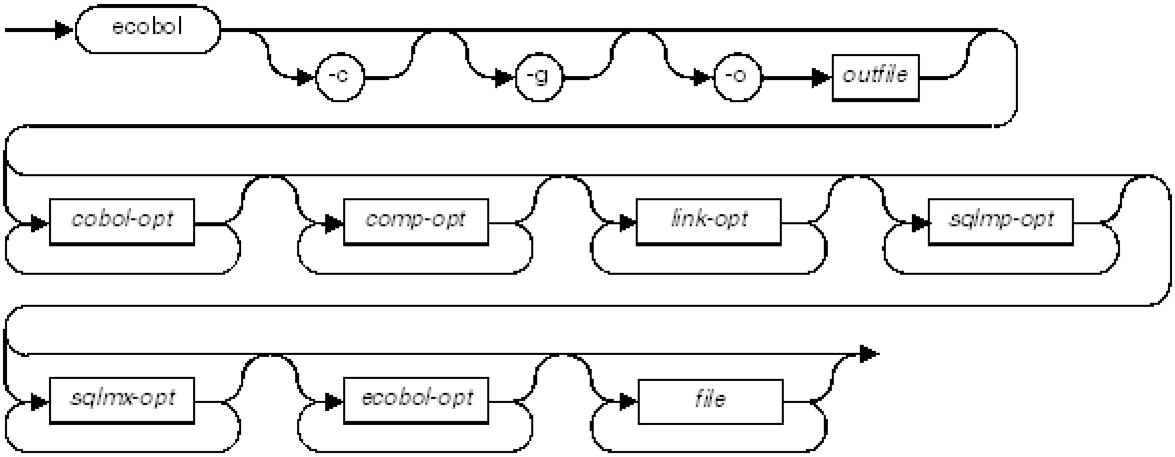
To run the program with the selected debugger, enter:

```
run -debug a.out
```

If the current directory is not in your search path, add it with this command:

```
export PATH=$PATH:.
```

ecobol (PC Command Line)



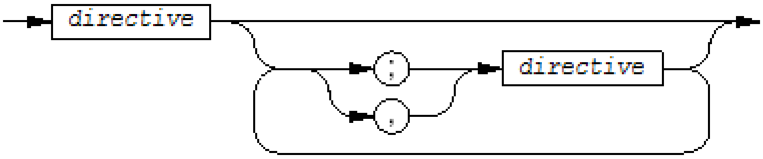
VST821.vsd

-c, -g, -o, `outfile`, `-cobol-opt`, `-comp-opt`, `-link-opt`, `-sqlmp-opt`, `-sqlmx-opt`, `-ecobol-opt`, `file`

See ecobol (PC Command Line) (page 984).

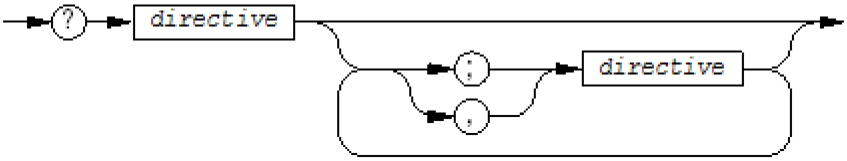
37 Compiler Directives

On the compiler command line:



VST263.vsd

In the program source text:

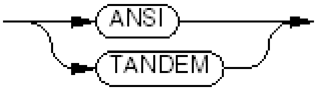


VST264.vsd

In the OSS environment, include compiler directives in the `ecobol` command line with the `-Wcobol` flag.

NOTE: "**Default:**" identifies the default for the compiler directive itself, not for its optional parameter(s). This default applies if a program does not contain the compiler directive at all.

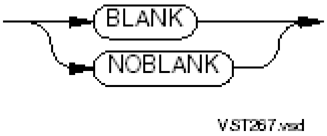
ANSI



VST266.vsd

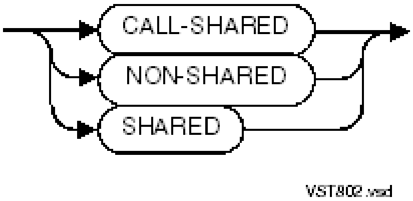
Default:	TANDEM
Placement:	Anywhere
Scope:	ANSI or TANDEM within a section of text obtained from a copy library or source library is effective only for the length of that text section. When the compiler reverts to the source file where it found the COPY verb or SOURCE directive, the previously active reference format applies.
Dependencies:	None
References:	TANDEM

BLANK and NOBLANK



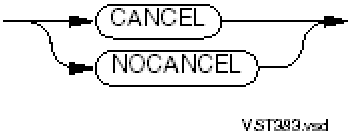
Default:	NOBLANK
Placement:	Anywhere
Scope:	The last BLANK or NOBLANK in the program applies.
Dependencies:	None

CALL-SHARED



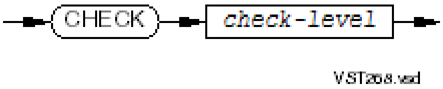
Default:	CALL-SHARED
Placement:	Anywhere
Scope:	The last CALL-SHARED or SHARED in the compilation unit applies to the entire compilation unit.
Dependencies:	• If RUNNABLE is active, CALL-SHARED uses the linker to create a PIC executable object file; otherwise, CALL-SHARED creates a PIC linkfile. • Do not use with UL.
References:	• SHARED • RUNNABLE • UL

CANCEL and NOCANCEL



Default:	CANCEL
Placement:	Anywhere
Scope:	Applies to the program that contains it. CANCEL and NOCANCEL do not apply to initial programs, which are initialized every time they are called.
Dependencies:	None

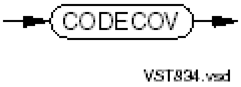
CHECK



Default:	CHECK 1
Placement:	Anywhere
Scope:	In each separately compiled program, the last CHECK in the program unit determines the check level for the code block being produced.
Dependencies:	None

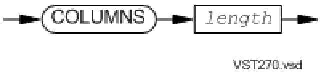
check-level	What is Checked	Comments
0	Nothing	CHECK 0 results in the fastest execution time.
1 (default)	Nothing	CHECK 1 might have a different meaning in future versions of HP COBOL for HP NonStop systems (“HP COBOL”). For the fastest execution (and no subscript checking) in current and future versions of HP COBOL, specify CHECK 0.
2	Validity of subscripts and indexes	
3	Validity of subscripts, indexes, and reference modifiers	
4	Validity of subscripts, indexes, reference modifiers, and certain data conversions.	Specifies the maximum level of checking that HP COBOL currently provides.
5-15	Validity of subscripts, indexes, reference modifiers, and certain data conversions.	5 - 15 might have different meanings in future versions of HP COBOL. For the maximum level of checking that HP COBOL currently provides, specify CHECK 4. For maximal checking, specify CHECK 15 (which could increase the program’s run-time overhead in future versions of HP COBOL if additional checking levels are implemented).

CODECOV



Default:	No code coverage instrumentation is included in the generated object file
Placement:	In the command line
Scope:	Applies to the compilation unit
Dependencies:	None

COLUMNS



Default: COLUMNS 132

Placement:

- On a directive line, COLUMNS must begin with a question mark (?) in column 1, regardless of any active ANSI directive.
- In a COPY library or a source library, COLUMNS must be the only directive on its line and must precede all SECTION directives in that library.
- The COBOL85 compiler accepts at most one COLUMNS for each source file (that is, one for the primary source file and one for each COPY library or source library called by the primary source file).

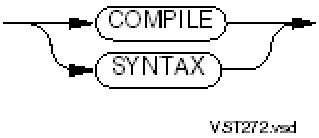
Scope: When the compiler shifts from reading the primary source file to reading a COPY library (in compliance with a COPY statement) or a source library (in compliance with a SOURCE directive), it saves the current (default or specified) logical length for source lines. That length applies to all source lines in the COPY library or source library, unless a COLUMNS directive occurs in the COPY library file (as mentioned in the preceding item). In this case, the compiler reverts to the saved logical length when it resumes reading text from the primary source file.

Dependencies: COLUMNS works only if TANDEM is active.

References:

- ANSI
- SOURCE
- TANDEM

COMPILE



Default: COMPILE

Placement: Outside the boundaries of a separately compiled program; that is, not between the Identification Division header of a separately compiled program and its end, which is marked by one of:

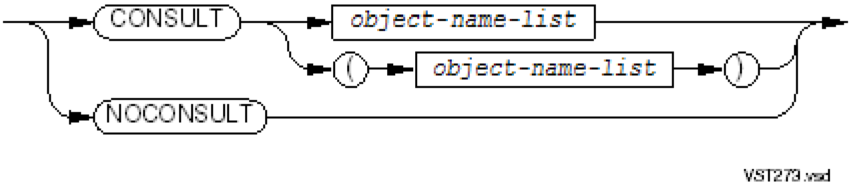
- The corresponding END PROGRAM statement
- ENDUNIT
- The end of the source file

Scope: The last COMPILE or SYNTAX in the compilation unit applies to the entire compilation unit.

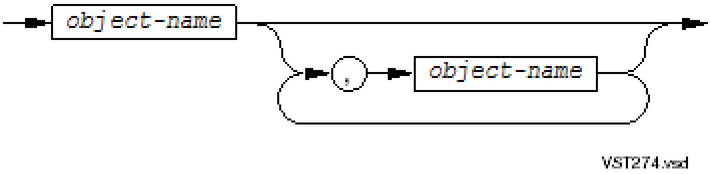
Dependencies: None

CONSULT and NOCONSULT

NOTE: The compiler ignores the NOCONSULT directive and issues a warning.

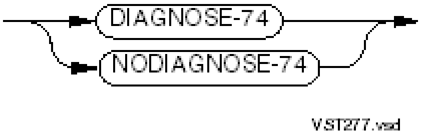


object-name-list



Default:	NOCONSULT
Placement:	Anywhere
Dependencies:	None

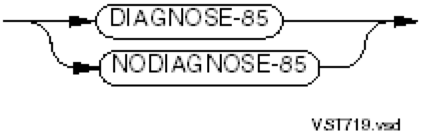
DIAGNOSE-74 and NODIAGNOSE-74



Default:	NODIAGNOSE-74
Placement:	Anywhere
Scope:	Applies until its opposite overrides it
Dependencies:	None

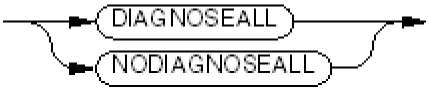
DIAGNOSE-85 and NODIAGNOSE-85

The COBOL85 compiler does not recognize these directives.



Default:	NODIAGNOSE-85
Placement:	Anywhere
Scope:	Applies until its opposite overrides it
Dependencies:	None

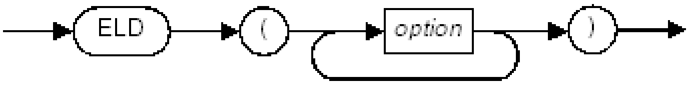
DIAGNOSEALL and NODIAGNOSEALL



VST470.vsd

Default:	NODIAGNOSEALL
Placement:	Anywhere
Scope:	Applies until its opposite overrides it
Dependencies:	None

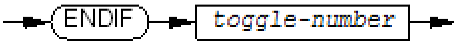
ELD



VST157.vsd

Default:	None
Placement:	In the command line
Scope:	Applies to the compilation unit
Dependencies:	None

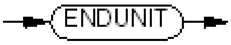
ENDIF



VST278.vsd

Default:	None
Placement:	Must be either on a directive line of its own or be the last of a sequence of directives
Dependencies:	Requires a preceding IF or IFNOT directive with the same `toggle-number`
References:	IF and IFNOT

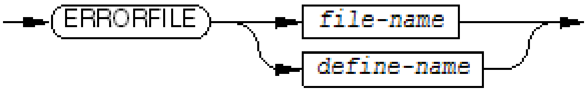
ENDUNIT



VST279.vsd

Default:	The compiler detects the end of a Procedure Division by encountering either an END PROGRAM statement or the end of the file.
Placement:	Must be either on a directive line of its own or be the last of a sequence of directives
Scope:	Applies to program unit
Dependencies:	None

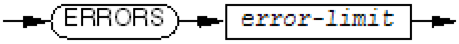
ERRORFILE



VST281.vsd

Default:	None
Placement:	Either on the compiler command line or in the source file before the Identification Division
Scope:	Applies to the compilation unit
Dependencies:	None

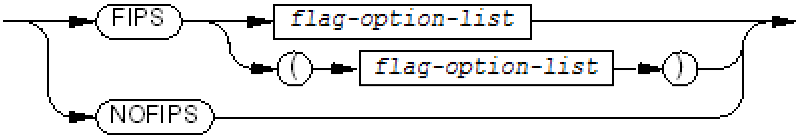
ERRORS



VST282.vsd

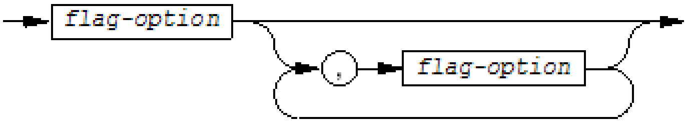
Default:	ERRORS 100
Placement:	Anywhere
Scope:	Applies to the compilation unit
Dependencies:	None

FIPS and NOFIPS



VST468.vsd

flag-option-list



VST469.vsd

flag-option

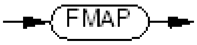
Value	Identified
OBSOLETE	Obsolete language elements
ABOVEMIN	Language elements above the minimum subset
ABOVEINTER	Language elements above the intermediate subset
LEVEL1COM *	Communication language elements
ABOVELEVEL1COM *	Communication language elements above level 1 of communication

Value	Identified
LEVEL1DEB	Debug language elements
ABOVELEVEL1DEB *	Debug language elements above level 1 of Debug
REPORTWRITER *	Report Writer language elements
LEVEL1SEG	Segmentation language elements
ABOVELEVEL1SEG	Segmentation language elements above level 1 of segmentation
NONSTANDARDEXT	Nonstandard extensions to COBOL

* This option does not affect compilation because the compiler does not support this module. If you use this option, the compiler issues a warning and ignores the option.

Default:	NOFIPS
Placement:	Anywhere
Scope:	Applies until its opposite overrides it
Dependencies:	FIPS is incompatible with STANDARD 2002

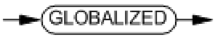
FMAP



VST731.vsd

Default:	The compiler does not produce a source file map.
Placement:	Anywhere
Scope:	Applies to the compilation unit
Dependencies:	NOLIST and SUPPRESS do not suppress the source file map that FMAP produces.
References:	• LIST and NOLIST • SUPPRESS and NOSUPPRESS

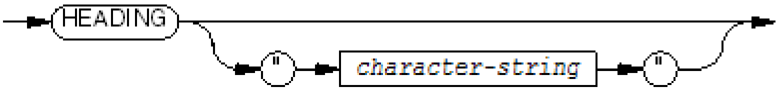
GLOBALIZED



VST835.vsd

Default:	Compiler generates nonpreemptable object code
Placement:	In the command line
Scope:	Applies to the compilation unit
Dependencies:	None

HEADING



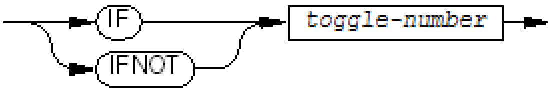
VST283.vsd

`character-string`

Default values is all spaces.

Default:	Standard top-of-page line
Placement:	Anywhere
Scope:	Applies until another HEADING overrides it
Dependencies:	None

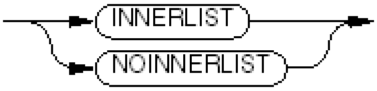
IF and IFNOT



VST287.vsd

Default:	None
Placement:	IF must be either on a directive line of its own or be the last of a sequence of directives.IFNOT can be anywhere.
Scope:	IF or IFNOT applies until matching ENDIF appears.
Dependencies:	Requires a preceding IF or IFNOT directive with the same `toggle-number`
References:	IF and IFNOT

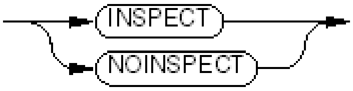
INNERLIST and NOINNERLIST



VST724.vsd

Default:	NOINNERLIST
Placement:	Anywhere
Scope:	Applies until its opposite overrides it
Dependencies:	INNERLIST works only if LIST is active and SUPPRESS is not.
References:	• LIST and NOLIST • SUPPRESS and NOSUPPRESS

INSPECT and NOINSPECT

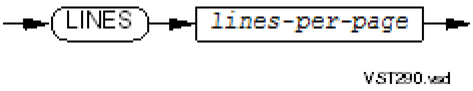


VST288.vsd

Default:	INSPECT
Placement:	Anywhere

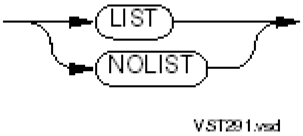
Scope:	The last INSPECT or NOINSPECT in the compilation unit applies to the compilation unit.
Dependencies:	NOINSPECT and SAVEABEND override each other (whichever is last is active).
References:	SAVEABEND and NOSAVEABEND

LINES



Default:	LINES 60
Placement:	Anywhere
Scope:	Applies until another LINES overrides it
Dependencies:	LINES works only if paging applies to the compilation list device.

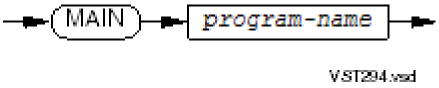
LIST and NOLIST



Default:	LIST
Placement:	Anywhere
Scope:	Applies until overridden by its opposite
Dependencies:	SUPPRESS overrides LIST (and therefore CODE, CROSSREF, ICODE, LMAP, MAP, SHOWCOPY, and SQL). NOLIST suppresses INNERLIST.
References:	• SUPPRESS and NOSUPPRESS • INNERLIST and NOINNERLISTcond text? • MAP and NOMAP • SHOWCOPY and NOSHOWCOPY • SQL and NOSQL

MAIN

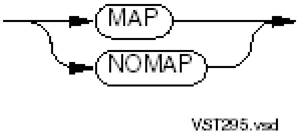
NOTE: Put this directive before the Identification Division header of the first program unit in the compilation unit.



Default:	Every program unit that does not have a Linkage Section is compiled as a main program. If more than one program unit of a compilation unit qualifies as a main program, the compiler reports this as an error, compiles the first qualifying program unit as the main program, and produces multiple object files.
Placement:	Before the Identification Division header of the first program unit in the compilation unit

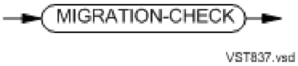
Scope:	Applies to the program that contains it
Dependencies:	None

MAP and NOMAP



Default:	NOMAP
Dependencies:	MAP works only if LIST is active and SUPPRESS is not.
Scope:	The last MAP or NOMAP applies.
Dependencies:	MAP works only if LIST is active and SUPPRESS is not.
References:	• LIST and NOLIST • SUPPRESS and NOSUPPRESS

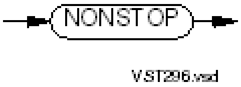
MIGRATION-CHECK



Default:	No warnings issued
Placement:	In the command line or before the Identification Division of the first program unit in the source text.
Dependencies:	Applies only when STANDARD 1985 is enabled.
Scope:	Applies to the compilation unit
References:	• STANDARD • DIAGNOSEALL and NODIAGNOSEALL

NONSTOP

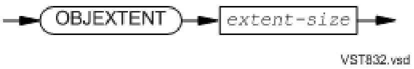
NOTE: Do not use this directive in the OSS environment.



Default:	The program cannot run as a process pair (even if it contains a PARAM NONSTOP ON command).
Placement:	Must appear before the Identification Division of the first program unit in the source text
Scope:	Applies to the program unit
Dependencies:	PARAM NONSTOP OFF command suppresses NONSTOP.
References:	PARAM Command

OBJEXTENT

NOTE: Do not use this directive in the OSS or Windows environment.

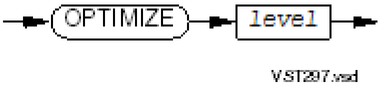


`extent-size`

specifies the number of pages to use for the primary and secondary extents of the generated object file. *`extent-size`* is an integer in the range 2 through 1070. Use this directive to increase the capacity of the generated object file. For more information about extents, see the Guardian Programmer's Guide.

Default:	OBJEXTENT 32
Placement:	On the Guardian command line
Scope:	Applies to the compilation unit
Dependencies:	You cannot specify OBJEXTENT and the linker option `-NS_extent_size` on the same command line

OPTIMIZE

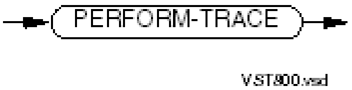


`level`

Level	Effect
0	Code is not optimized. Provided in case other optimization levels cause errors. Supports symbolic debugging; data is always in memory.
1 (default)	Code is optimized within statements and across statement boundaries. The resulting code is more efficient than that produced by lower levels of optimization. Supports symbolic debugging; data is not always in memory.
2	Code is optimized within statements and across statement boundaries, and the resulting code is more efficient than code produced by lower levels.

Default:	OPTIMIZE 1
Placement:	Outside the boundary of a separately compiled program
Scope:	The optimization level active at the beginning of a separately compiled program determines the level of optimization for that program and any programs it contains.
Dependencies:	None

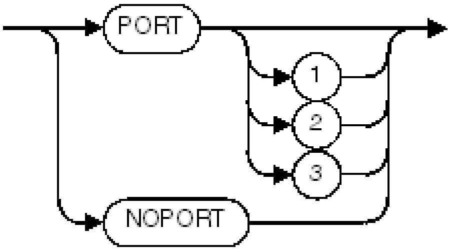
PERFORM-TRACE



Default:	The compiler does **not** provide additional information if run-time error 148 occurs.
Placement:	Anywhere

Scope:	Applies to the compilation unit
Dependencies:	None

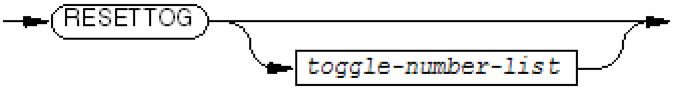
PORT and NOPORT



VST600.vsd

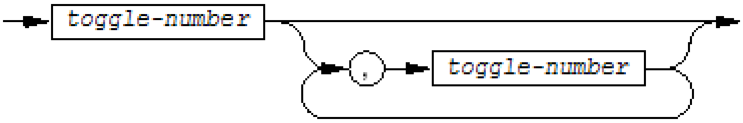
Default:	NOPORT
Placement:	Anywhere
Scope:	Applies to the compilation unit
Dependencies:	None

RESETTOG



VST298.vsd

`toggle-number-list`

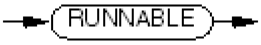


VST299.vsd

Default value is all toggles.

Default:	None
Placement:	Must be either on a directive line of its own or be the last of a sequence of directives
Scope:	Applies until SETTOG overrides it
Dependencies:	SETTOG overrides it.
References:	SETTOG

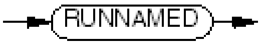
RUNNABLE



VST729.vsd

Default:	The compiler does **not** use the linker to produce a loadfile if there were no compilation errors.
Placement:	Anywhere
Scope:	Applies to the compilation unit
Dependencies:	SYNTAX overrides RUNNABLE.You must specify RUNNABLE to use RUNNAMED, SAVEABEND, NOSAVEABEND, or SUBTYPE in the compiler.With CALL-SHARED (the default), RUNNABLE uses the linker to produce a PIC loadfile.With SHARED, RUNNABLE uses the linker to produce a PIC library file (DLL).
References:	• SYNTAX • RUNNAMED • SAVEABEND and NOSAVEABEND • SUBTYPE • CALL-SHARED • SHARED

RUNNAMED

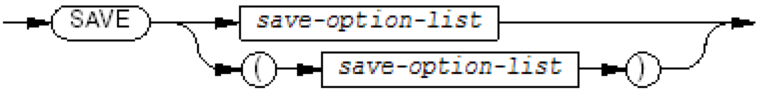


VST300.vsd

Default:	None
Placement:	Any of: • Anywhere in a source program • In a linker session (specify `-change`)
Scope:	Applies to the program in which or before which it appears, to all subsequent programs in the compilation unit, and to all target files
Dependencies:	RUNNAMED is appropriate only if the program was compiled with RUNNABLE (because a linkfile is not a run unit)..
References:	RUNNABLE

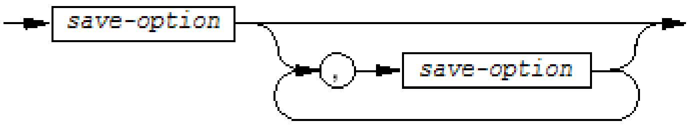
SAVE

NOTE: Do not use this directive in the OSS environment.



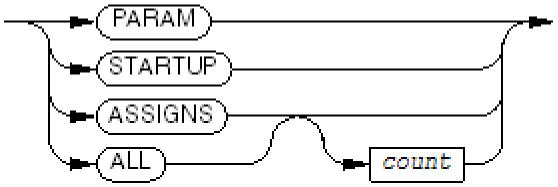
VST301.vsd

save-option-list



VST302.vsd

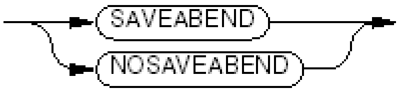
save-option



VST303.vsd

Default:	The compiler does not save initialization messages.
Placement:	Anywhere
Scope:	Applies to the compilation unit
Dependencies:	None

SAVEABEND and NOSAVEABEND

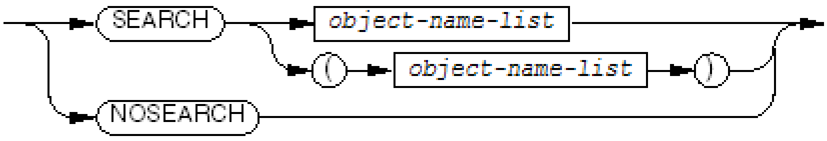


VST304.vsd

Default:	NOSAVEABEND
Placement:	Anywhere
Scope:	The last SAVEABEND or NOSAVEABEND in the compilation unit applies to the entire compilation unit.
Dependencies:	SAVEABEND and NOINSPECT override each other (whichever is last is active). SAVEABEND and NOSAVEABEND are appropriate only if the program was compiled with RUNNABLE.
References:	• INSPECT and NOINSPECT • RUNNABLE

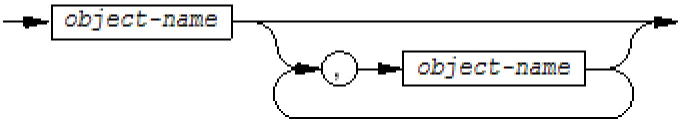
SEARCH and NOSEARCH

NOTE: The compiler ignores the NOSEARCH directive and issues a warning.



VST305.vsd

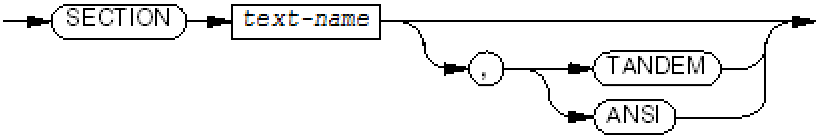
object-name-list



VST306.vsd

Default:	NOSEARCH
Placement:	Anywhere
Scope:	Applies to the compilation unit
Dependencies:	Must be used with either RUNNABLE or CONSULT if you are creating a PIC object file
References:	• RUNNABLE • CONSULT and NOCONSULT

SECTION



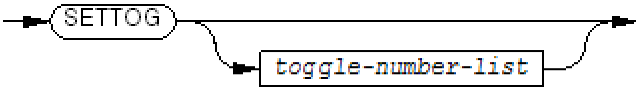
VST307.vsd

If you do not specify TANDEM or ANSI, the compiler assumes that the format of the text is the current source-text format.

Default:	None
Placement:	Must be on a directive line of its own. If it appears in the text of the compilation source file (the IN file), it is ignored.
Scope:	Applies only to the section of text that it specifies
Dependencies:	None

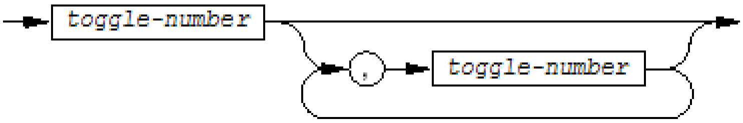
SETTOG

NOTE: The SETTOG directive must either be on a directive line of its own or be the last of a sequence of directives.



VST308.vsd

toggle-number-list

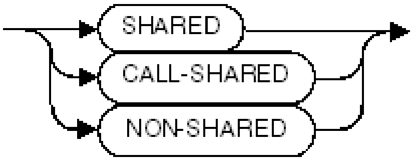


VST299.vsd

Default value is all toggles.

Default:	None
Placement:	Must either be on a directive line of its own or be the last of a sequence of directives.
Scope:	Applies until RESETTOG overrides it
Dependencies:	None
References:	RESETTOG

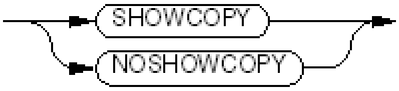
SHARED



VST804.vsd

Default:	CALL-SHARED
Placement:	Anywhere
Scope:	The last SHARED or CALL-SHARED in the compilation unit applies to the entire compilation unit.
Dependencies:	• If RUNNABLE is active, SHARED uses the linker to create a PIC library file (DLL); otherwise, SHARED creates a PIC linkfile. • Do not use with CALL-SHARED or UL.
References:	• CALL-SHARED • RUNNABLE • UL

SHOWCOPY and NOSHOWCOPY

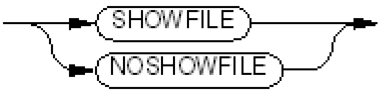


VST310.vsd

Default:	SHOWCOPY
Placement:	Anywhere
Scope:	Applies until its opposite overrides it

Dependencies:	SHOWCOPY works only if LIST is active and SUPPRESS is not.
References:	• LIST and NOLIST • SUPPRESS and NOSUPPRESS

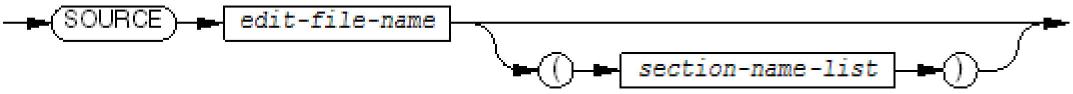
SHOWFILE and NOSHOWFILE



VST311.vsd

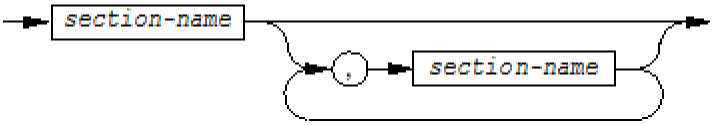
Default:	NOSHOWFILE
Placement:	Anywhere
Scope:	Applies until its opposite overrides it
Dependencies:	None

SOURCE



VST312.vsd

section-name-list

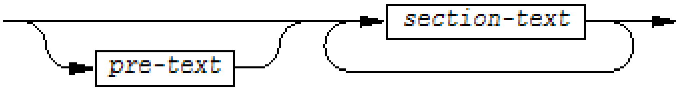


VST313.vsd

Default value is entire file.

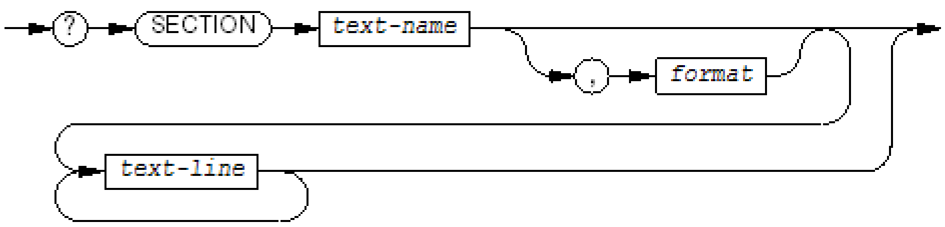
Default:	None
Placement:	Must be the last directive on its line.
Dependencies:	None

Format of a source library:



VST252.vsd

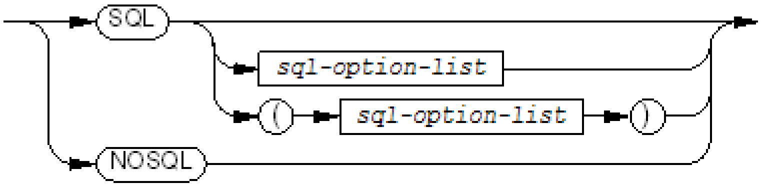
section-text



VST253.vsd

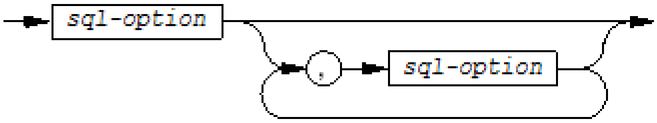
SQL and NOSQL

NOTE: The compiler ignores the NOSQL directive and issues a warning.



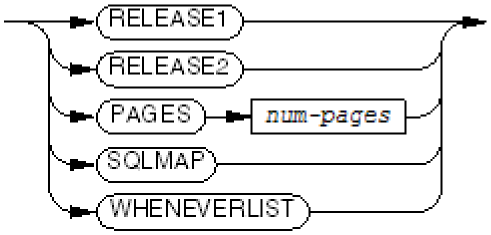
VST315.vsd

sql-option-list



VST316.vsd

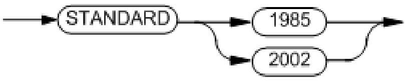
sql-option



VST317.vsd

Default:	None. If the program contains SQL/MP statements, the SQL directive is required; otherwise, it is unnecessary.
Placement:	In the compiler command line.
Scope:	Applies to the compilation unit
Dependencies:	SQL works only if LIST is active and SUPPRESS is not.
References:	• LIST and NOLIST • SUPPRESS and NOSUPPRESS

STANDARD



VST836.vsd

1985

Causes the compiler to apply the COBOL-1985 standard.

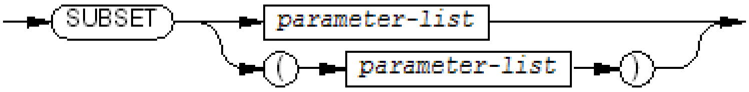
2002

Causes the compiler to apply the COBOL-2002 standard.

Default:	STANDARD 1985
Placement:	In the command line
Scope:	Applies to the compilation unit
Dependencies:	STANDARD 2002 is incompatible with the FIPS and SUBSET directives. Whichever directive is specified first takes effect; whichever is specified next causes the compiler to issue a warning and has no other effect.
References:	• FIPS and NOFIPS • SUBSET

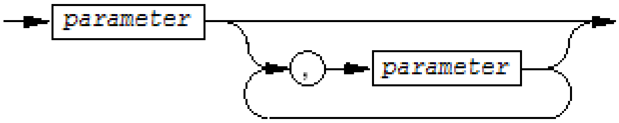
SUBSET

NOTE: Put this directive before the Identification Division header of the first program unit in the compilation unit.



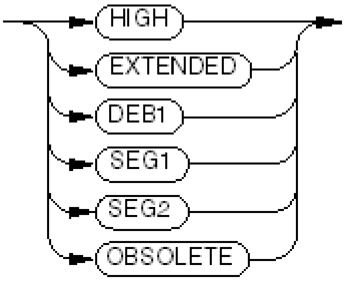
VST319.vsd

parameter-list



VST320.vsd

parameter

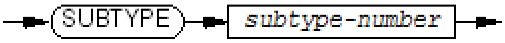


VST321.vsd

Default:	None
Placement:	Must appear before the first COBOL source program (that is, before the first Identification Division header).
Scope:	The last valid SUBSET in the program applies.
Dependencies:	SUBSET is incompatible with STANDARD 2002

SUBTYPE

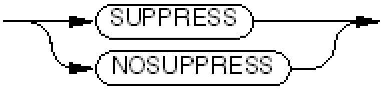
NOTE: Do not use this directive in the OSS environment.



VST323.vsd

Default:	SUBTYPE 0
Placement:	Anywhere
Scope:	The last SUBTYPE in the program applies.
Dependencies:	SUBTYPE is appropriate only if the program was compiled with RUNNABLE.
References:	RUNNABLE

SUPPRESS and NOSUPPRESS



VST324.vsd

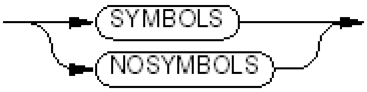
Default:	NOSUPPRESS
Placement:	Accepted anywhere, but to suppress the compiler listing without altering the source text, put SUPPRESS on the command line.
Scope:	Applies until its opposite overrides it

Dependencies: SUPPRESS overrides LIST (and therefore, it also overrides CODE, CROSSREF, ICODE, LMAP, MAP, SHOWCOPY, and SQL).

References:

- LIST and NOLIST
- MAP and NOMAP
- SHOWCOPY and NOSHOWCOPY
- SQL and NOSQL

SYMBOLS and NOSYMBOLS



VST325.vsd

Default: NOSYMBOLS

Placement: Accepted anywhere, but to affect a program other than the first program in a compilation unit, SYMBOLS must follow either an ENDUNIT directive or an END PROGRAM statement, which signals the end of the preceding program's Procedure Division.

If you want a symbol table for a given separately compiled program, put SYMBOLS before its Identification Division header. If the compiler encounters a subsequent SYMBOLS or NOSYMBOLS directive within the text of that separately compiled program, it issues a warning and ignores that directive.

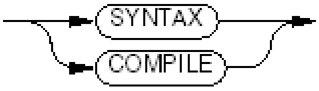
Scope: Applies until its opposite overrides it

Dependencies: SYMBOLS does not work when SYNTAX is active.For SYMBOLS to affect a program other than the first program in a compilation unit, it must be preceded by either an ENDUNIT directive or an END PROGRAM statement, which signals the end of the preceding program's Procedure Division.

References:

- ENDUNIT
- SYNTAX

SYNTAX



VST326.vsd

Default: COMPILE

Placement: Outside the boundaries of a separately compiled program; that is, not between the Identification Division header of a separately compiled program and its end, which is marked by one of:

- The corresponding END PROGRAM statement
- ENDUNIT
- The end of the source file

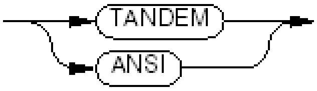
Scope: The last COMPILE or SYNTAX in the compilation unit applies to the entire compilation unit.

Dependencies: SYNTAX overrides RUNNABLE.

References:

- ENDUNIT
- RUNNABLE

TANDEM



VST327.vsd

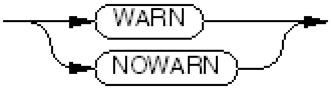
Default:	TANDEM
Placement:	Anywhere
Scope:	TANDEM or ANSI within a section of text obtained from a copy library or source library is effective only for the length of that text section. When the compiler reverts to the source file where it found the COPY verb or SOURCE directive, the previously active reference format applies.
Dependencies:	None
References:	ANSI

UL

VST735.vsd

Default:	None
Placement:	Before the first IDENTIFICATION DIVISION header of the first program in the compilation unit
Scope:	Applies to the compilation unit
Dependencies:	• If RUNNABLE is active, UL uses the linker to create a PIC library file (DLL); otherwise, UL creates a PIC linkfile. • Do not use with CALL-SHARED or SHARED.
References:	• CALL-SHARED • SHARED

WARN and NOWARN

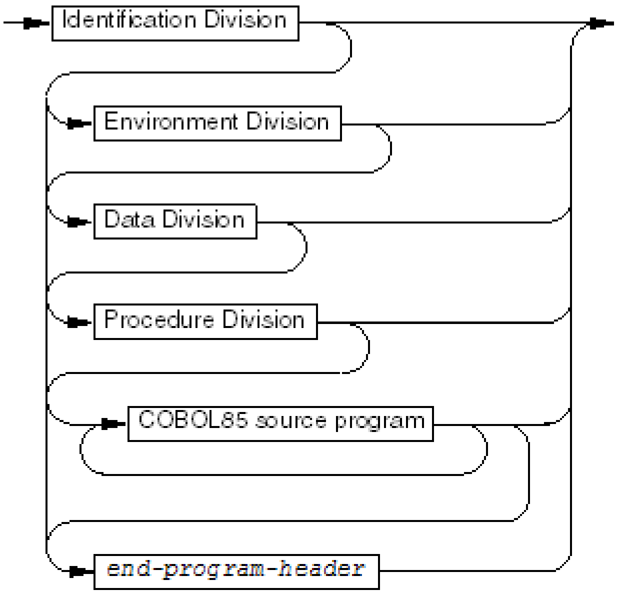


VST330.vsd

Default:	WARN
Placement:	Anywhere
Scope:	Applies to the compilation unit
Dependencies:	If LIST is not enabled, the last line of source text processed is also listed to provide a point of reference for each warning message.
References:	LIST and NOLIST

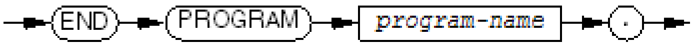
38 Source Program Organization and Format

Separately Compiled Source Program



VST373.vsd

end-program-header



VST374.vsd

Division syntax:

- Chapter 41: Identification Division (page 1041)
- Chapter 42: Environment Division (page 1043)
- Chapter 43: Data Division (page 1061)
- Chapter 44: Procedure Division (page 1073)

Reference Format

- TANDEM
- ANSI

TANDEM

Figure 38-1 Tandem Reference Format

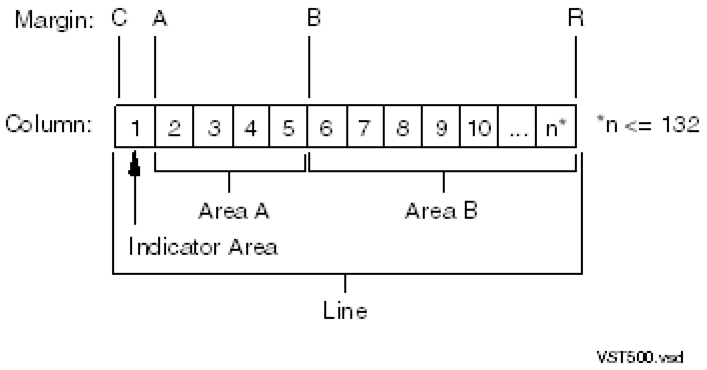


Table 38-1 Valid Indicator Field Characters (Tandem Reference Format)

Valid Character	Character Name	Means that the line is a ...
?	Question mark	Compiler directive
*	Asterisk	Ordinary comment
/	Slash	Comment to be printed at the top of the next page
D	Uppercase *D*	Debugging line
d	Lowercase *d*	Debugging line
-	Hyphen	Continuation line
	Space	Text line

ANSI

Figure 38-2 ANSI Reference Format

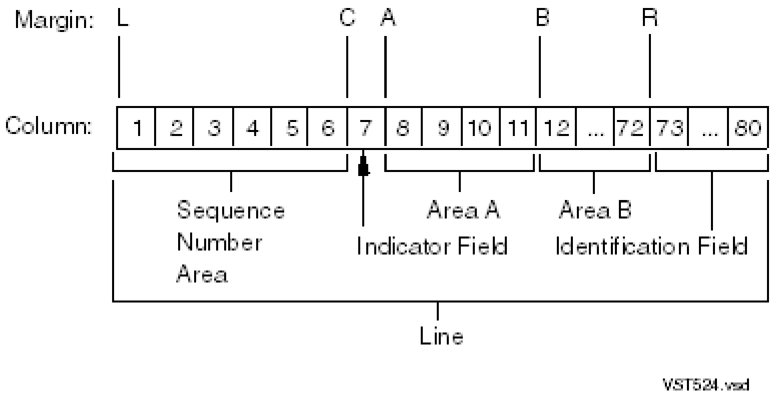


Table 38-2 Valid Indicator Area Characters (ANSI Reference Format)

Valid Character	Character Name	Indicates that the line is a ...
?	Question mark	Compiler directive
*	Asterisk	Ordinary comment
/	Slash	Comment to be printed at the top of a page
D	Uppercase *D*	Debugging line

Table 38-2 Valid Indicator Area Characters (ANSI Reference Format) *(continued)*

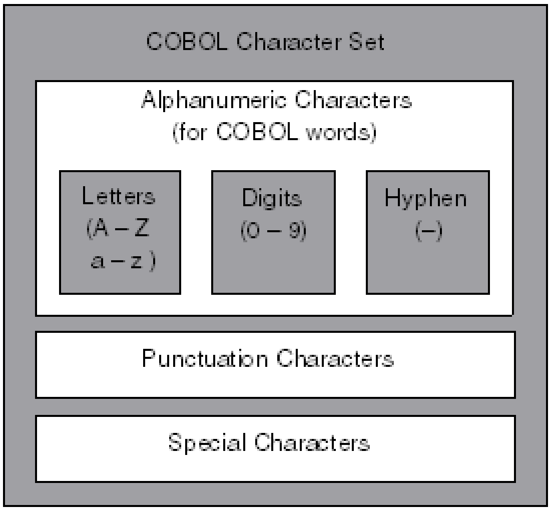
Valid Character	Character Name	Indicates that the line is a ...
d	Lowercase `d`	Debugging line
-	Hyphen	Continuation line

39 Language Elements and Expressions

See also Chapter 22: Reserved Words (page 743).

COBOL Character Set

Figure 39-1 COBOL Character Set



VST503.vsd

- Table 39-1: Alphanumeric Characters (for COBOL Words) (page 1027)
- Table 39-2: Punctuation Characters (page 1027)
- Table 39-3: Special Characters (page 1028)

Table 39-1 Alphanumeric Characters (for COBOL Words)

Characters	Name of Character Set
0 through 9	Digits
A through Z	Uppercase letters
a through z	Lowercase letters
-	Hyphen or minus sign

Table 39-2 Punctuation Characters

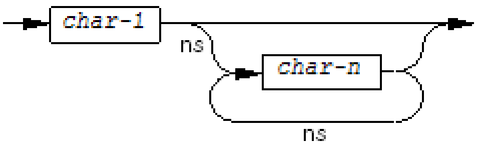
Character	Name of Character
	Space
,	Comma
;	Semicolon
:	Colon
.	Period
"	Quotation mark
(	Left parenthesis
)	Right parenthesis
=	Equal sign

Table 39-3 Special Characters

Character	Name of Character
+	Plus sign
-	Hyphen or minus sign
*	Asterisk
/	Stroke or slash
$	Dollar sign
>	Greater than sign
<	Less than sign

Character-Strings

COBOL Word



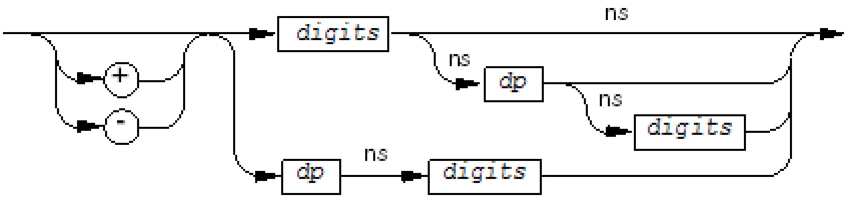
VST741.vsd

Maximum length: 30 characters.

Literals

- Numeric Literals
  - Decimal Numeric Literal
  - Hexadecimal Numeric Literal
- Nonnumeric Literals
  - Simple Nonnumeric Literal
  - Hexadecimal Nonnumeric Literal
- National Literal
- Figurative Constants

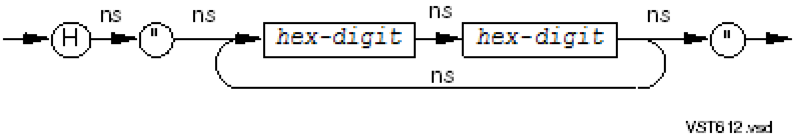
Decimal Numeric Literal



VST743.vsd

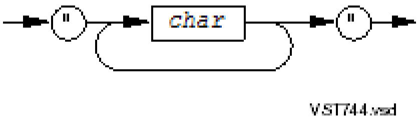
Maximum number of digits in one decimal numeric literal: 18.

Hexadecimal Numeric Literal



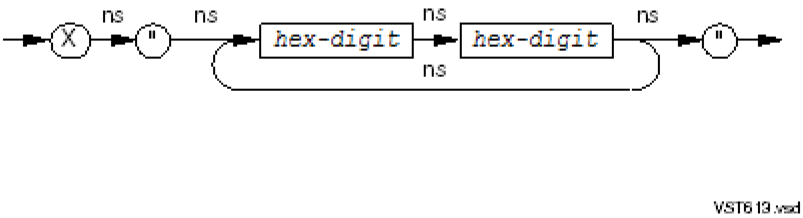
Maximum number of hexadecimal digits: 16 (8 pairs).

Simple Nonnumeric Literal



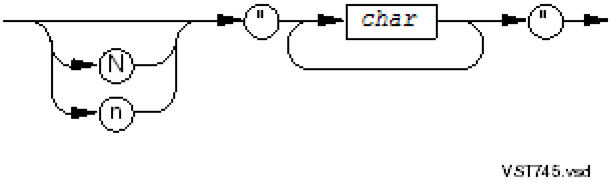
Maximum number of characters: 160.

Hexadecimal Nonnumeric Literal



Maximum number of hexadecimal digits: 320 (160 pairs).

National Literal



Maximum number of characters: 160.

Figurative Constants

Table 39-4 Figurative Constants

Figurative Constant *	What It Represents
ZERO ZEROS ZEROES	One or more of the character zero (0), depending on the context
SPACE SPACES	One or more spaces, depending on the context
HIGH-VALUE HIGH-VALUES	One or more of the character that has the highest position in the program collating sequence, except in the SPECIAL-NAMES paragraph, where it represents the character that has the highest position in the ASCII character set (the 256th character, which is all binary ones) or in the national character set (default is hexadecimal FFFF)

Table 39-4 Figurative Constants *(continued)*

Figurative Constant *	What It Represents
LOW-VALUE LOW-VALUES	One or more of the character that has the lowest position in the program collating sequence, except in the SPECIAL-NAMES paragraph, where it represents the character that has the lowest position in the ASCII character set (the first character, the NUL, which is all binary zeros) or in the national character set (default is hexadecimal 0000)
QUOTE QUOTES	One or more of the character quotation mark (") You cannot use either of these words instead of quotation marks to enclose a nonnumeric literal
symbolic-character	One or more of the character specified as the value of *symbolic-character* in the SYMBOLIC CHARACTERS clause of the SPECIAL-NAMES paragraph
ALL *literal*	The value of *literal* must contain one or more of: • A nonnumeric literal • A national literal • A symbolic-character • One of the other reserved words previously defined as figurative constants (except that you cannot precede an ALL literal form of figurative constant with another ALL) When *literal* is a nonnumeric literal or a national literal, this form implies repetition of the literal's value to the extent required by the context. For the other cases, the word ALL is redundant and is used for readability only.

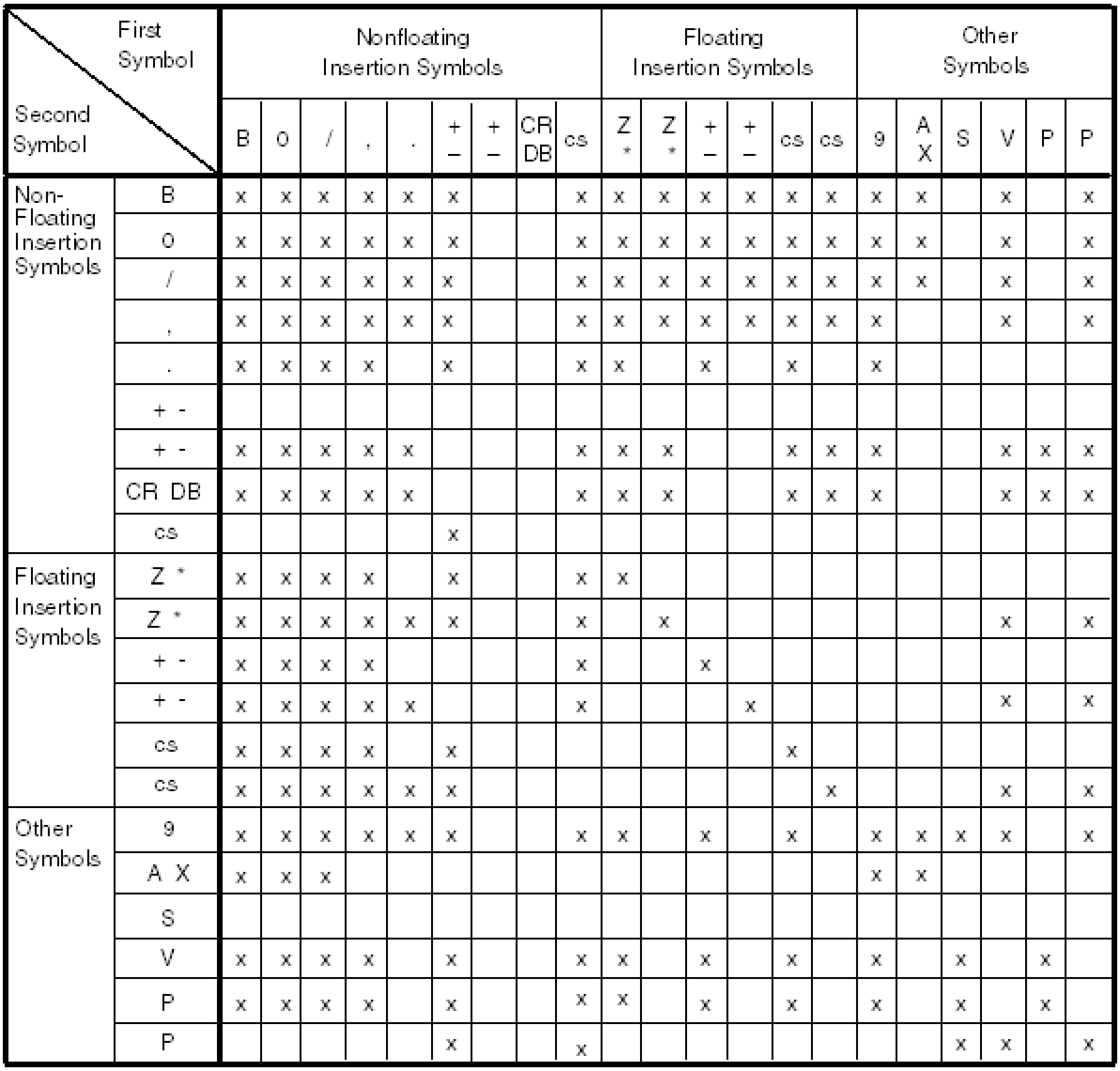
* Singular and plural forms are equivalent and can be used interchangeably.

PICTURE Character-Strings

Table 39-5 PICTURE Character-String Editing Characters

Character	Editing Operation
B	Space insertion
Z	Zero suppression
0	Zero
+	Plus
-	Minus
CR	Credit
DB	Debit
*	Check protect
$	Currency sign
,	Comma or decimal point
.	Period or decimal point
/	Slash

Figure 39-2 Precedence Rules for PICTURE Symbols

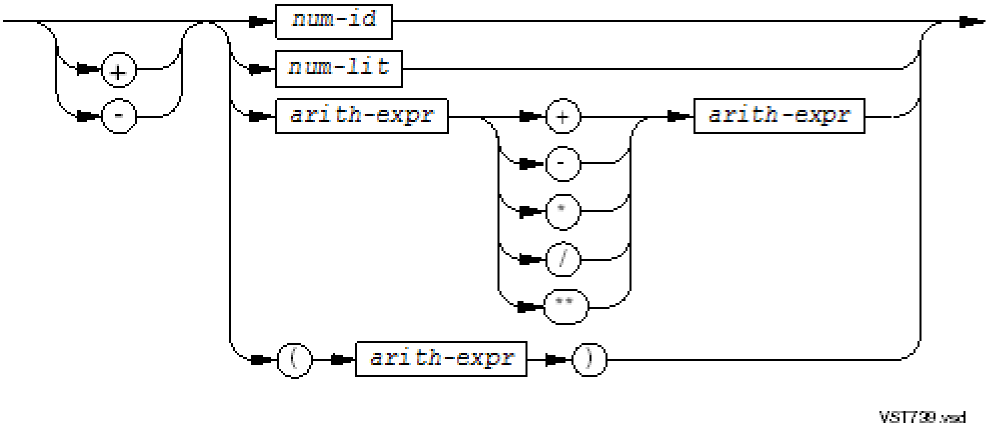


Second Symbol	First Symbol	Nonfloating Insertion Symbols									Floating Insertion Symbols						Other Symbols					
		B	0	/	,	.	+ –	+ –	CR DB	cs	Z *	Z *	+ –	+ –	cs	cs	9	A X	S	V	P	P
Non-Floating Insertion Symbols	B	x	x	x	x	x	x			x	x	x	x	x	x	x	x	x		x		x
	0	x	x	x	x	x	x			x	x	x	x	x	x	x	x	x		x		x
	/	x	x	x	x	x	x			x	x	x	x	x	x	x	x	x		x		x
	,	x	x	x	x	x	x			x	x	x	x	x	x	x	x			x		x
	.	x	x	x	x		x			x	x		x		x		x					
	+ -																					
	+ -	x	x	x	x	x				x	x	x			x	x	x			x	x	x
	CR DB	x	x	x	x	x				x	x	x			x	x	x			x	x	x
	cs						x															
Floating Insertion Symbols	Z *	x	x	x	x		x			x	x											
	Z *	x	x	x	x	x	x			x		x								x		x
	+ -	x	x	x	x					x			x									
	+ -	x	x	x	x	x				x				x						x		x
	cs	x	x	x	x		x								x							
	cs	x	x	x	x	x	x									x				x		x
Other Symbols	9	x	x	x	x	x	x			x	x		x		x		x	x	x	x		x
	A X	x	x	x													x	x				
	S																					
	V	x	x	x	x		x			x	x		x		x		x		x		x	
	P	x	x	x	x		x			x	x		x		x		x		x		x	
	P						x			x									x	x		x

cs = currency symbol

VST525.vsd

Arithmetic Expressions



- Table 39-6: Arithmetic Operators (page 1032)
- Table 39-7: Hierarchy of Operators (page 1032)
- Table 39-8: Precedence in Arithmetic Expressions (page 1032)
- Table 39-9: Operator-Operand Combinations (page 1033)

Table 39-6 Arithmetic Operators

Operator		
Symbol	**Kind**	**Meaning**
+	Unary	Multiplication by +1
	Binary	Addition
-	Unary	Multiplication by -1
	Binary	Subtraction
*	Binary	Multiplication
/	Binary	Division
**	Binary	Exponentiation

Table 39-7 Hierarchy of Operators

Hierarchy	Operators
1st	Unary plus and minus
2nd	Exponentiation
3rd	Multiplication and division
4th	Addition and subtraction

Table 39-8 Precedence in Arithmetic Expressions

Ambiguous	Interpretation
A / B * C	(A / B) * C
A / B / C	(A / B) / C
A ** B ** C	(A ** B) ** C
A + B / C + D ** E * F - G	((A + (B / C)) + ((D ** E) * F)) - G

Table 39-9 Operator-Operand Combinations

	Successor Element				
First Element	Variable	Binary Operator + - * / **	Unary Operator + or -	(	)
Identifier or literal	No	Yes	No	No	Yes
Binary Operator + - * / **	Yes	No	Yes	Yes	No
Unary Operator + or -	Yes	No	No	Yes	No
(	Yes	No	Yes	Yes	No
)	No	Yes	No	No	Yes

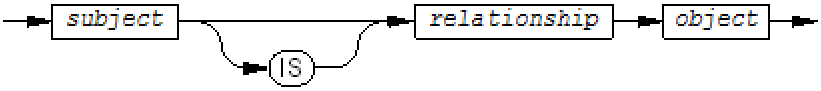
Conditional Expressions

- Simple Relation Condition
- Simple Class Condition
- Simple Sign Condition
- Negated Condition
- Combined Condition
- Abbreviated Combined Relation Condition

Simple Relation Condition

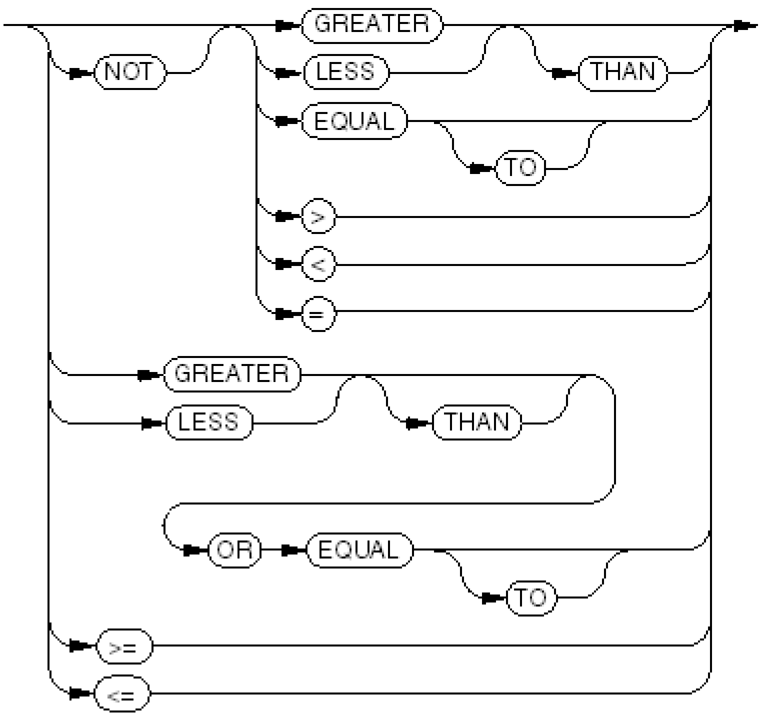
- Nonpointer Operands
- Pointer Operands

Nonpointer Operands



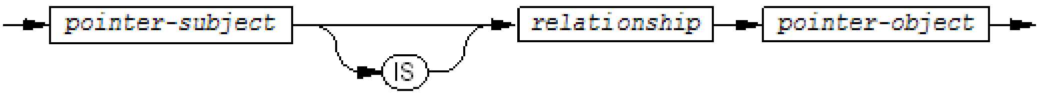
VST118.vsd

relationship



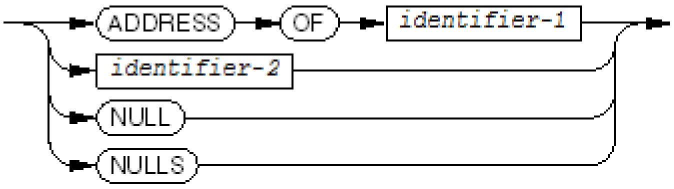
VST119.vsd

Pointer Operands



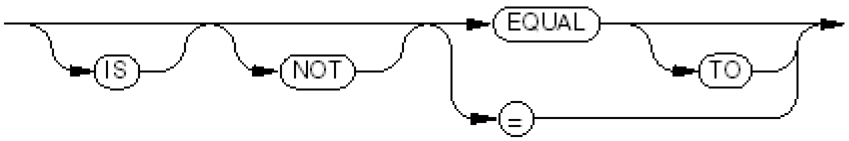
VST602.vsd

pointer-subject, pointer-object



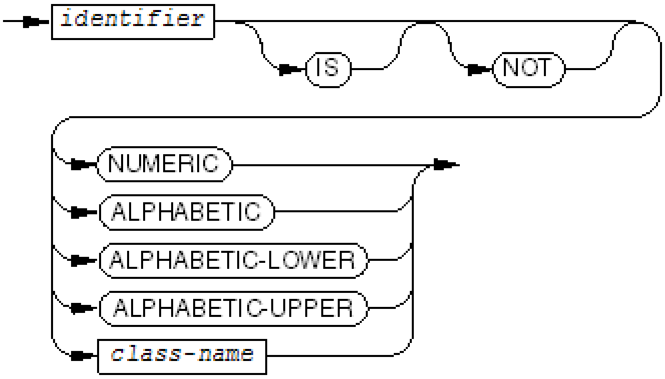
VST603.vsd

relationship



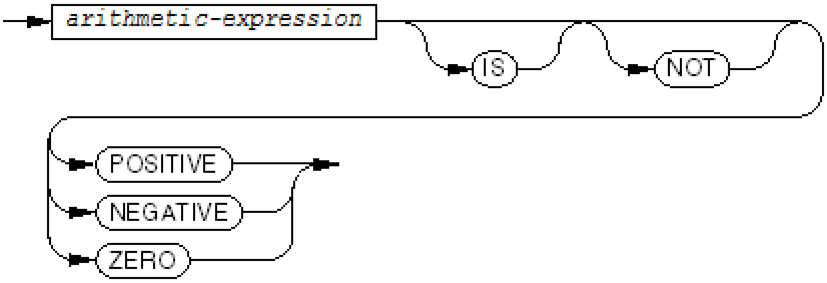
VST604.vsd

Simple Class Condition



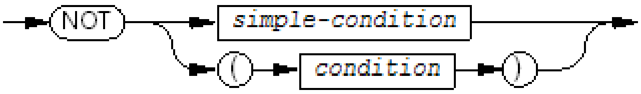
VST120.vsd

Simple Sign Condition



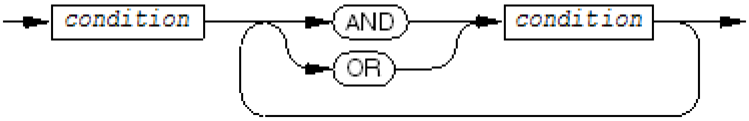
VST121.vsd

Negated Condition



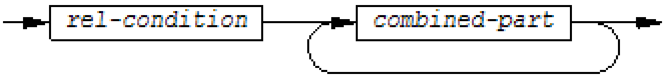
VST122.vsd

Combined Condition



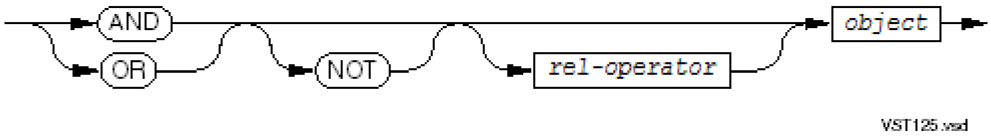
VST123.vsd

Abbreviated Combined Relation Condition



VST124.vsd

combined-part



Concatenation Expressions

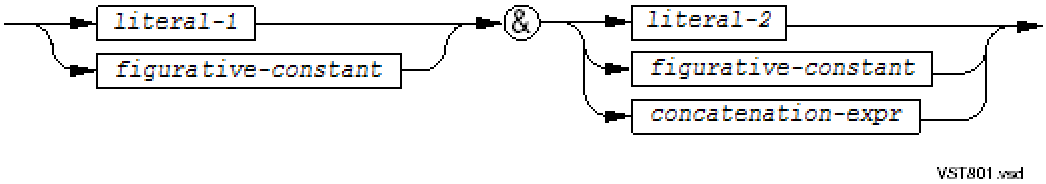


Table 39-10 Maximum Length of Result of Concatenation Expression

Class of Operands	Length of Result
Alphanumeric	160 alphanumeric character positions
National	80 national character positions

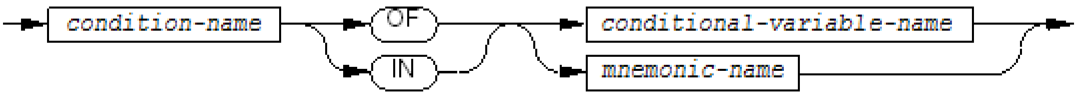
A figurative constant occupies one character position.

Table 39-11 Class of Result of Concatenation Expression

Number of Operands That Are Figurative Constants	Class of Result
0	Same class as the operands (neither of which is a figurative constant)
1	Same class as the other operand (which is not a figurative constant)
2	Alphanumeric

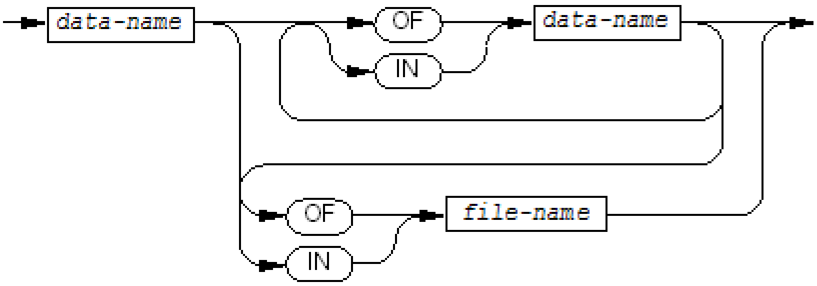
40 Data References

Qualified Condition-Name



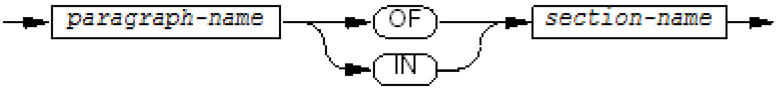
VST001.vsd

Qualified Data-Name



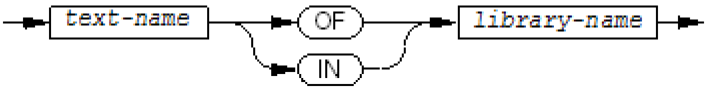
VST002.vsd

Qualified Paragraph-Name



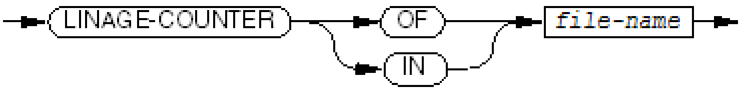
VST003.vsd

Qualified Text-Name



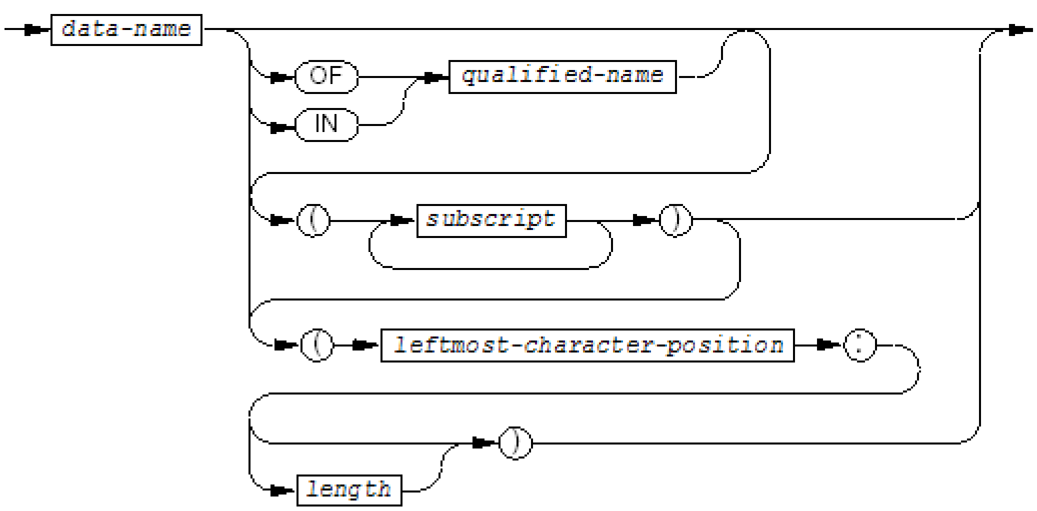
VST004.vsd

Qualified LINAGE-COUNTER



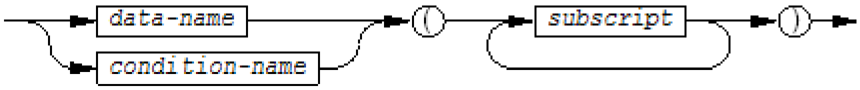
VST005.vsd

Qualified Identifier



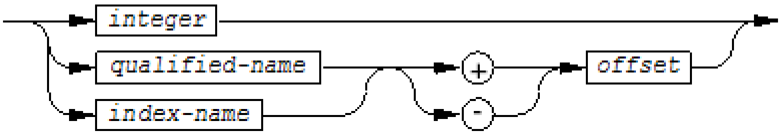
VST009.vsd

Subscripted Table Element



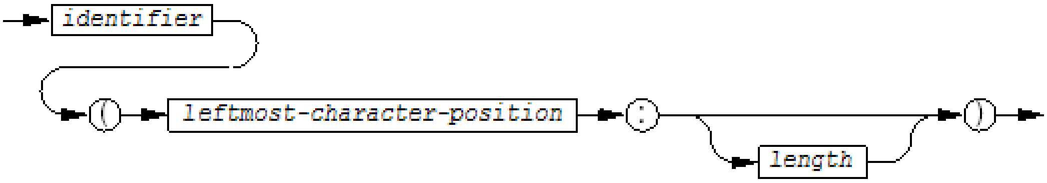
VST006.vsd

subscript



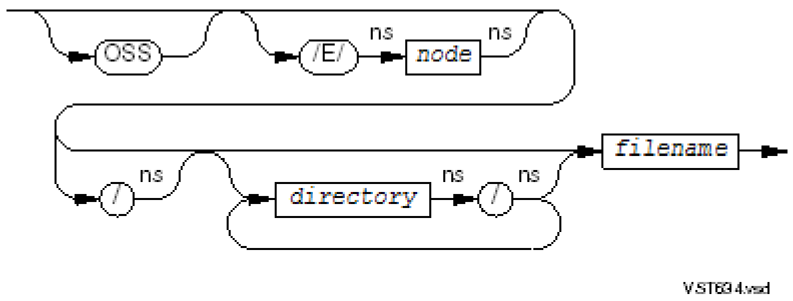
VST007.vsd

Reference Modifier

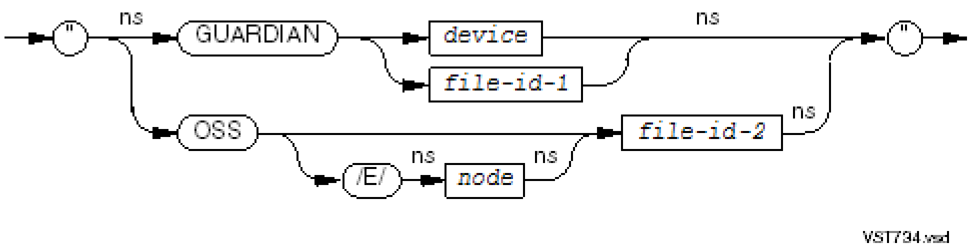


VST008.vsd

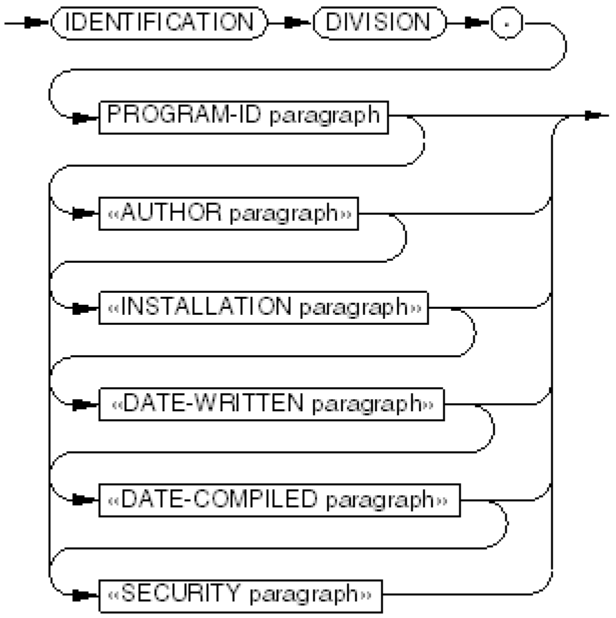
OSS Pathname for OSS File



OSS Pathname for Guardian File

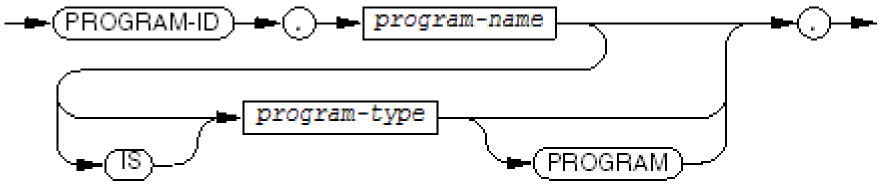


41 Identification Division



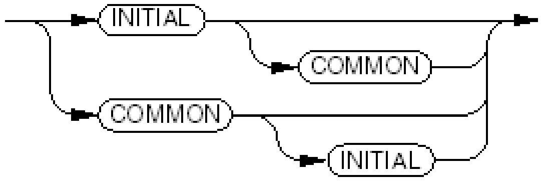
VST010.vsd

PROGRAM-ID paragraph



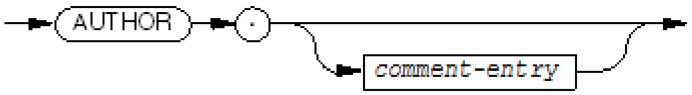
VST011.vsd

program-kind



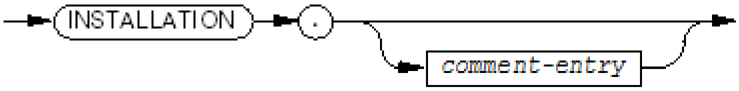
VST012.vsd

«AUTHOR paragraph»



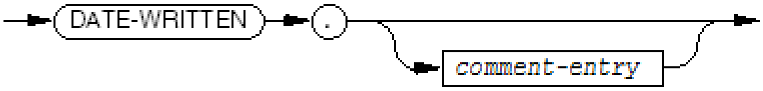
VST400.vsd

«INSTALLATION paragraph»



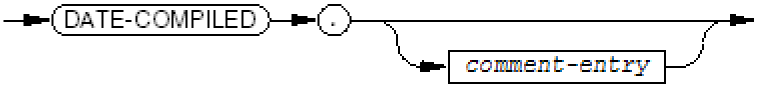
VST401.vsd

«DATE-WRITTEN paragraph»



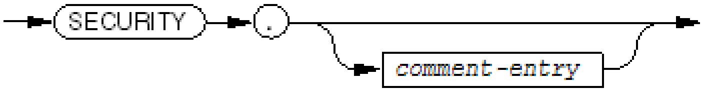
VST402.vsd

«DATE-COMPILED paragraph»



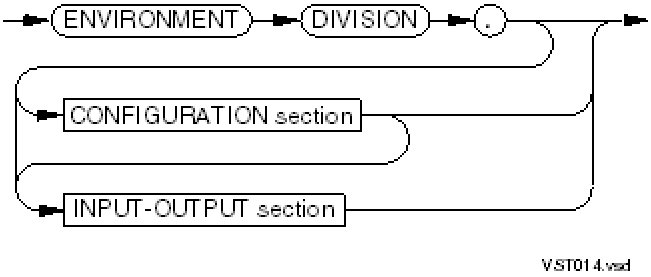
VST013.vsd

«SECURITY paragraph»

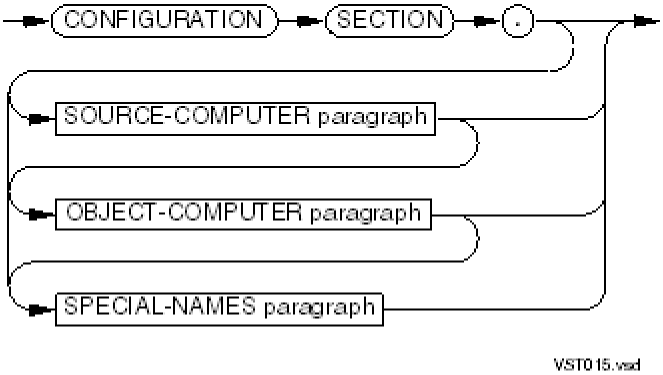


VST404.vsd

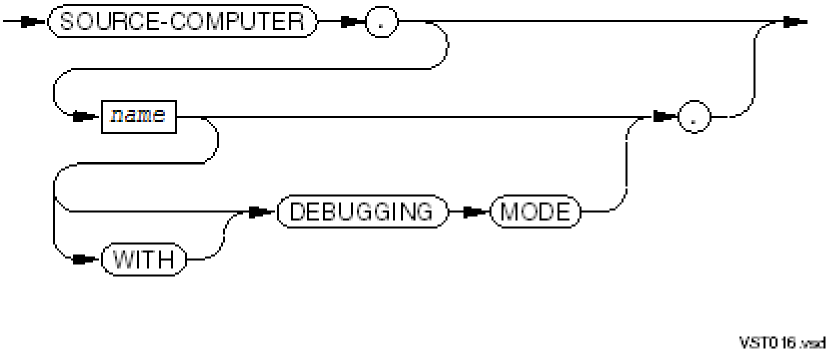
42 Environment Division



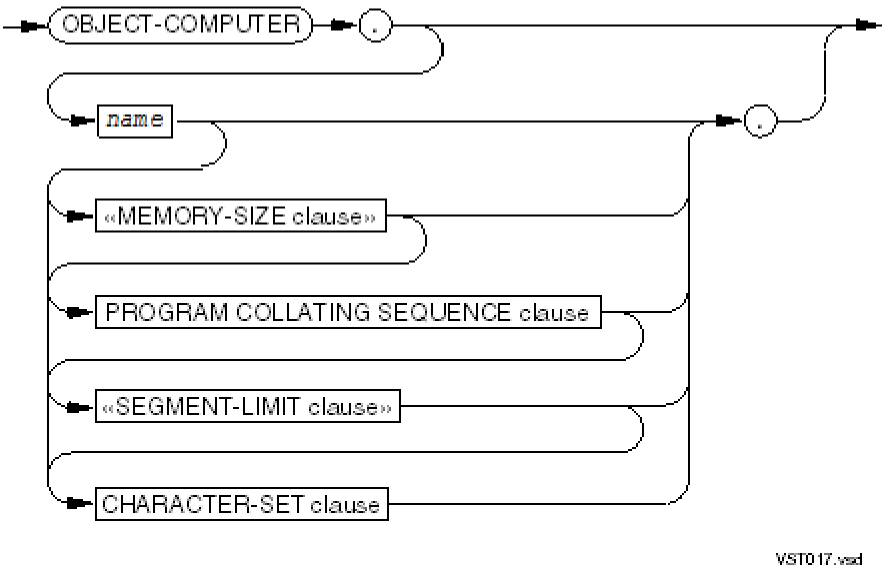
Configuration Section



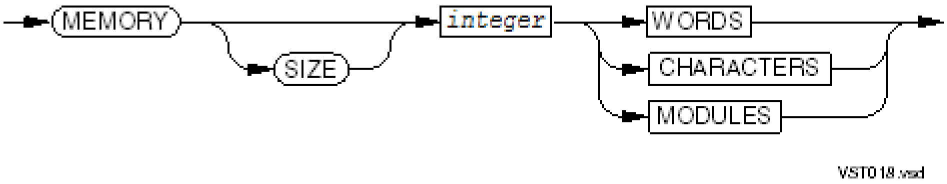
SOURCE-COMPUTER Paragraph



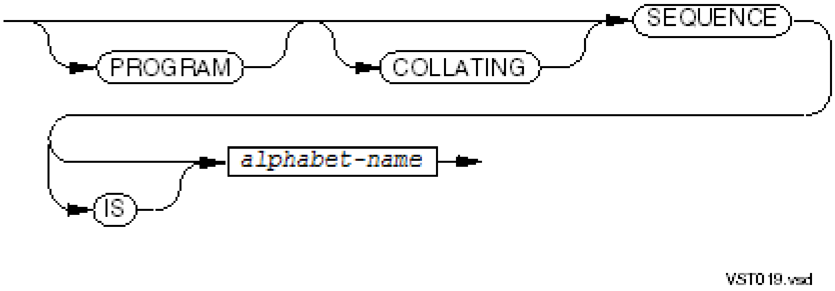
OBJECT-COMPUTER Paragraph



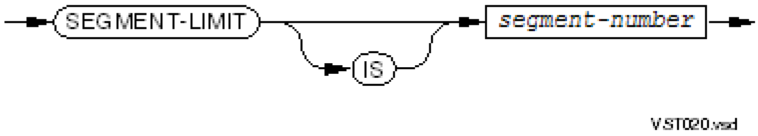
«MEMORY-SIZE clause»



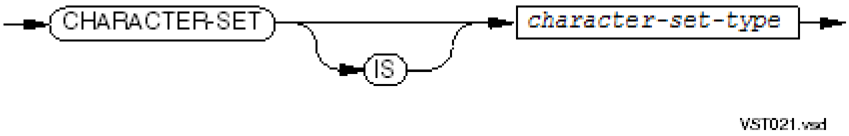
PROGRAM COLLATING SEQUENCE clause



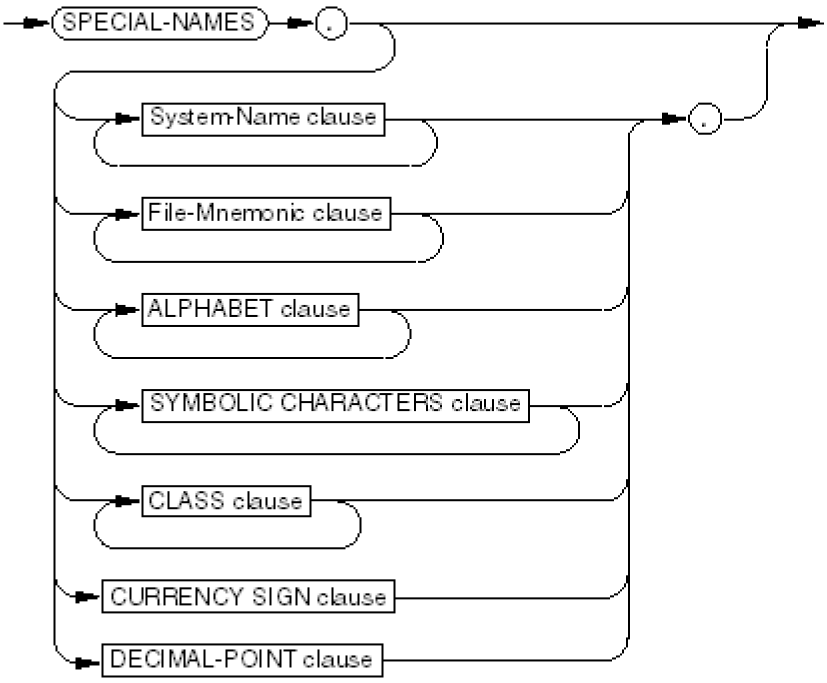
«SEGMENT-LIMIT clause»



CHARACTER-SET clause

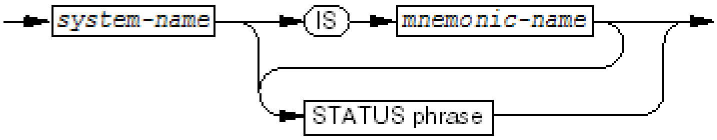


SPECIAL-NAMES Paragraph



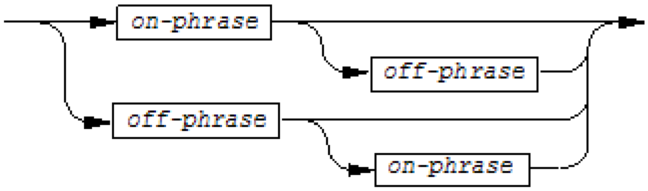
VST022.vsd

System-Name clause



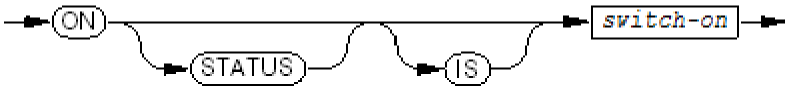
VST023.vsd

STATUS phrase



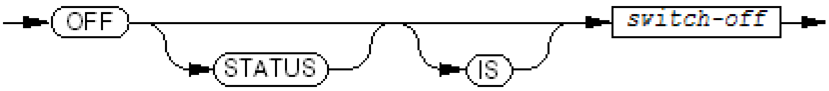
VST746.vsd

on-phrase



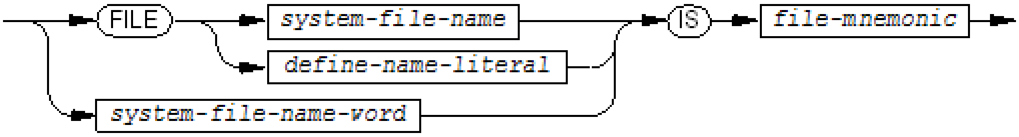
VST025.vsd

off-phrase



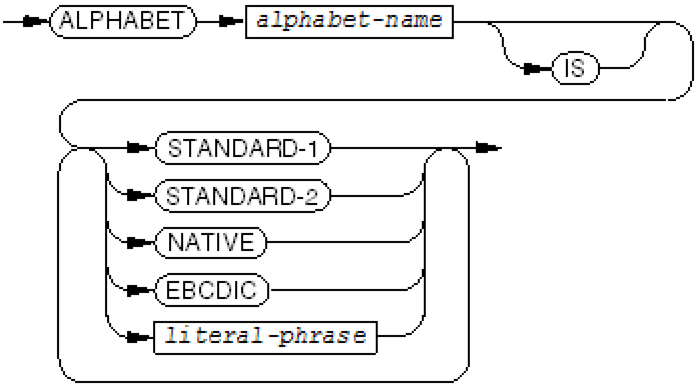
VST026.vsd

File-Mnemonic clause



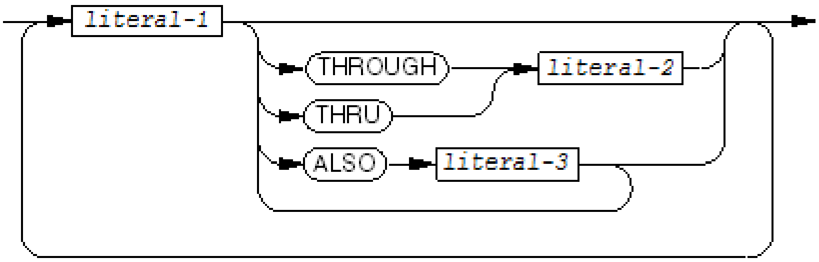
VST027.vsd

ALPHABET clause



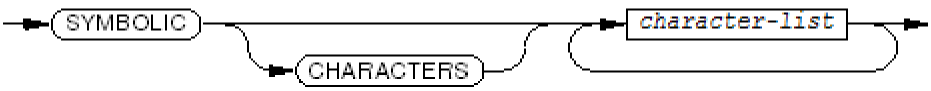
VST028.vsd

literal-phrase



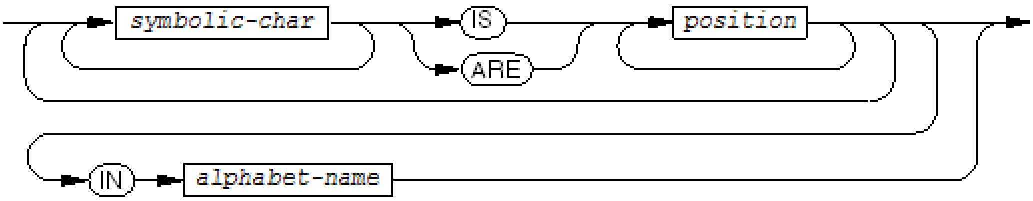
VST029.vsd

SYMBOLIC-CHARACTERS clause



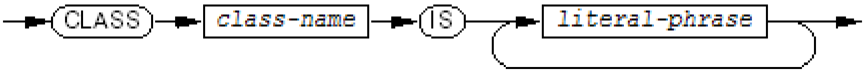
VST030.vsd

character-list



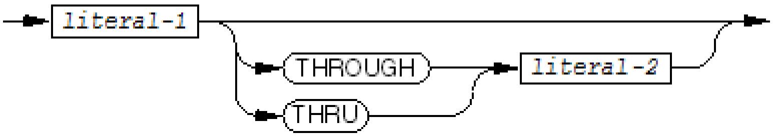
VST031.vsd

CLASS clause



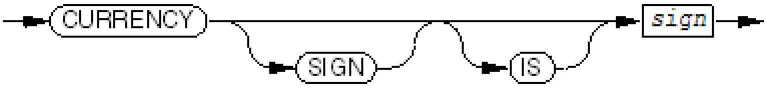
VST032.vsd

literal-phrase



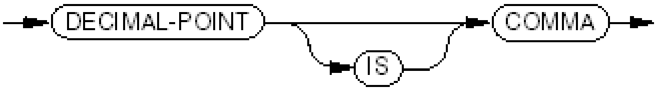
VST033.vsd

CURRENCY-SIGN clause



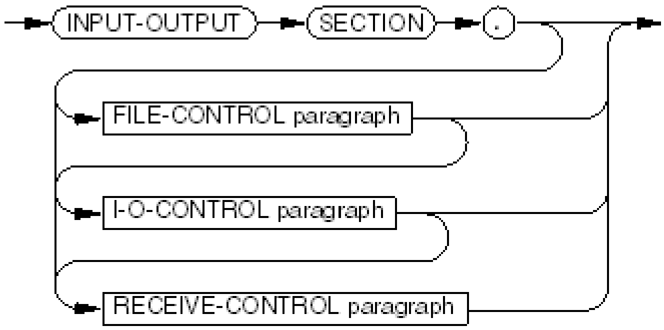
VST034.vsd

DECIMAL-POINT clause



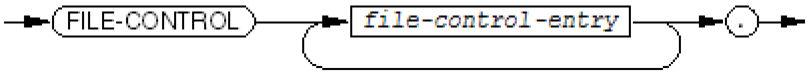
VST380.vsd

Input-Output Section



VST095.vsd

FILE-CONTROL Paragraph

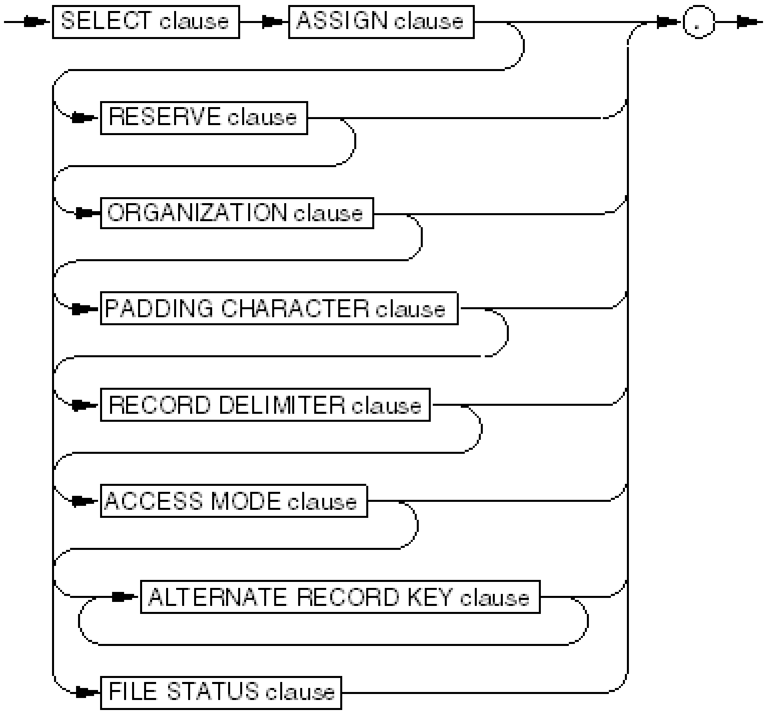


VST096.vsd

file-control-entry

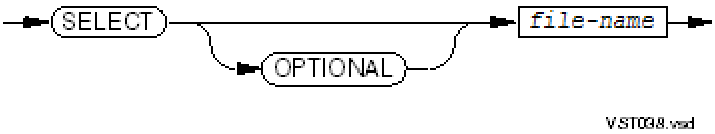
- Sequential File
- Line Sequential File
- Relative File
- Indexed File
- Queue File
- Sort-Merge File

Sequential File

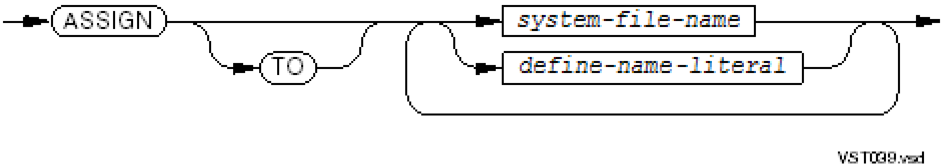


VST097.vsd

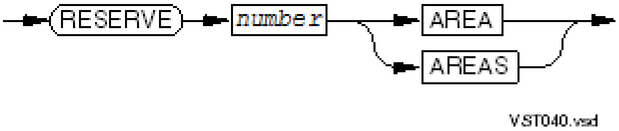
SELECT clause



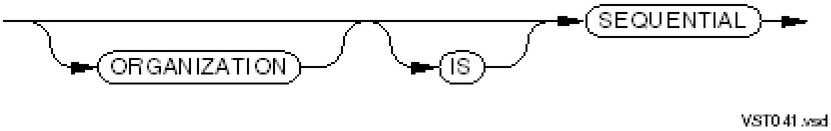
ASSIGN clause



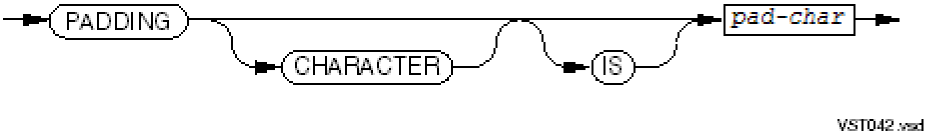
RESERVE clause



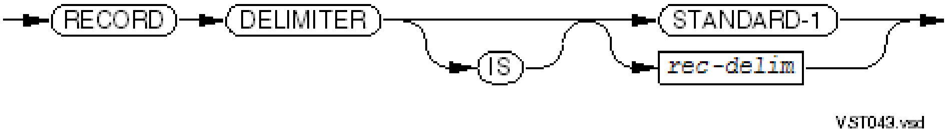
ORGANIZATION clause



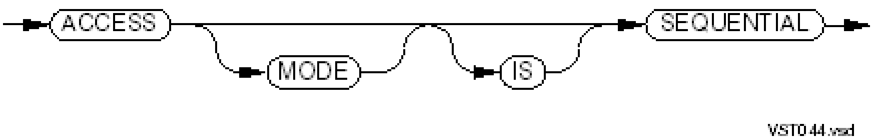
PADDING CHARACTER clause



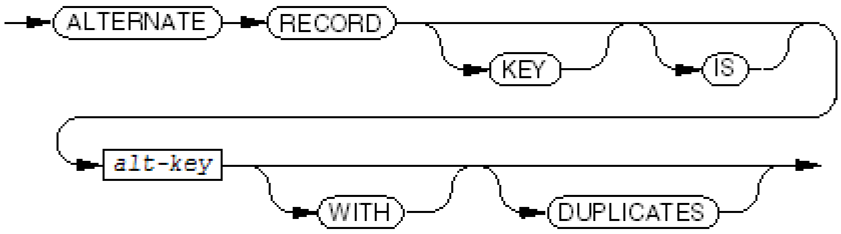
RECORD DELIMITER clause



ACCESS MODE clause

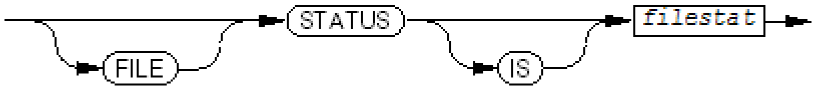


ALTERNATE RECORD KEY clause



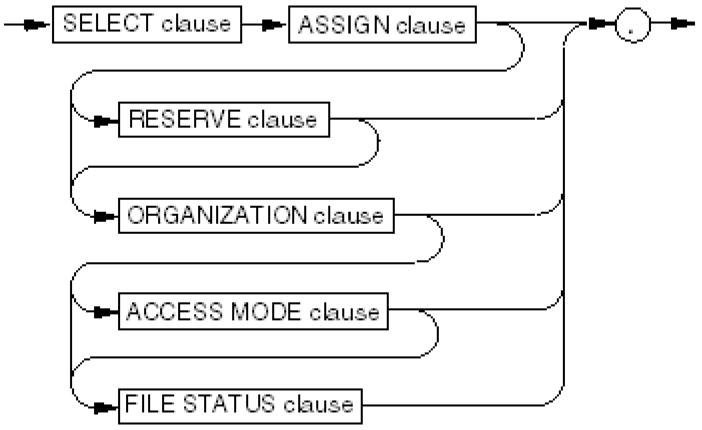
VST045.vsd

FILE STATUS clause



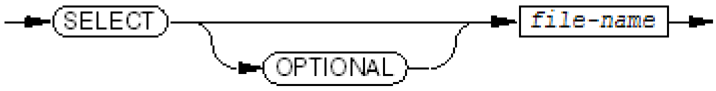
VST046.vsd

Line Sequential File



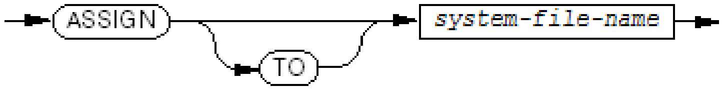
VST624.vsd

SELECT clause



VST098.vsd

ASSIGN clause

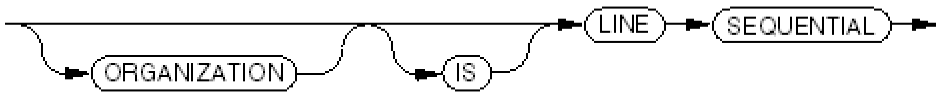


VST640.vsd

RESERVE clause

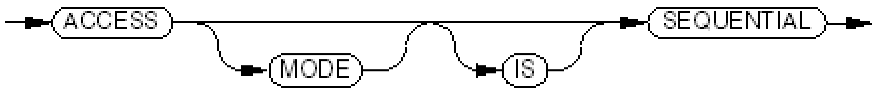
Ignored.

ORGANIZATION clause



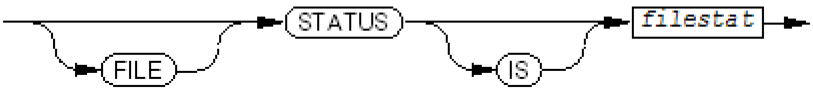
VST625.vsd

ACCESS MODE clause



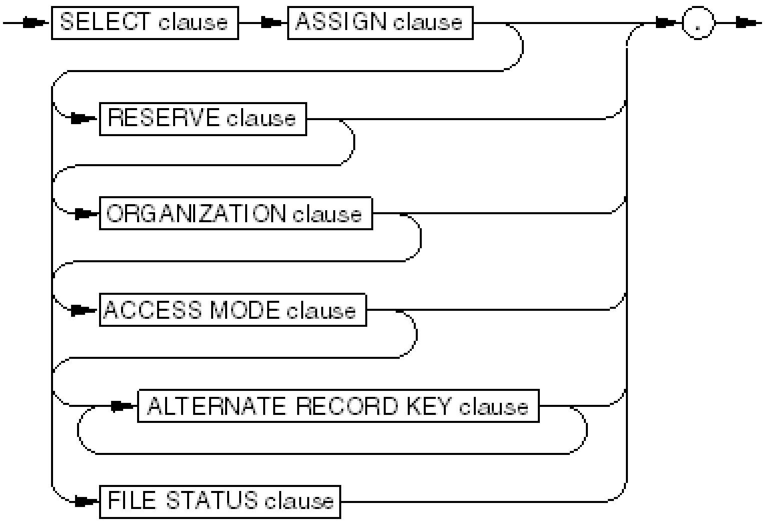
VST044.vsd

FILE STATUS clause



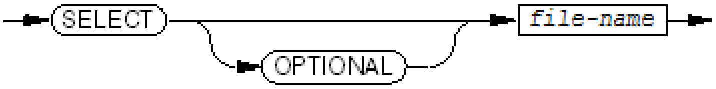
VST046.vsd

Relative File



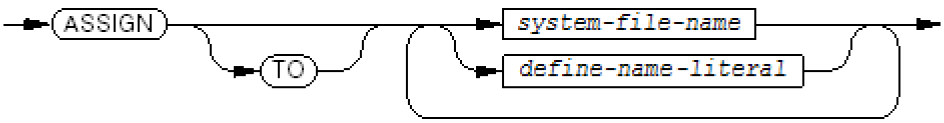
VST047.vsd

SELECT clause



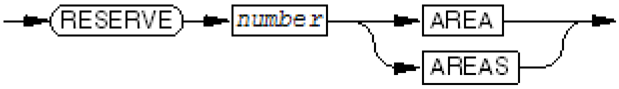
VST038.vsd

ASSIGN clause



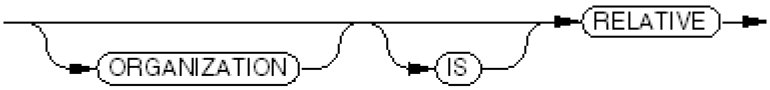
VST039.vsd

RESERVE clause



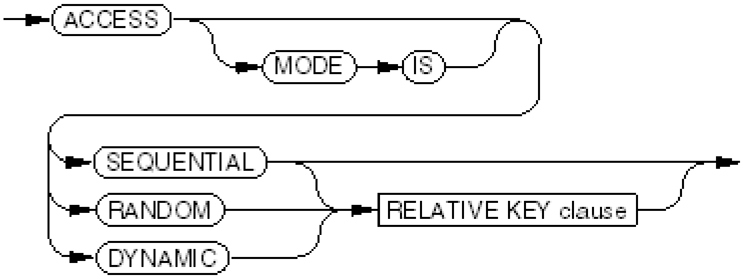
VST040.vsd

ORGANIZATION clause



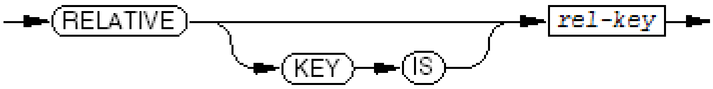
VST048.vsd

ACCESS MODE clause



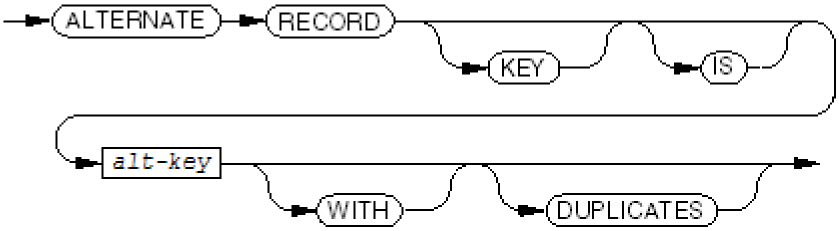
VST049.vsd

RELATIVE KEY clause



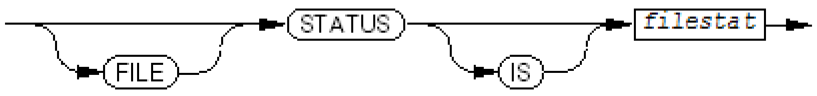
VST050.vsd

ALTERNATE RECORD KEY clause



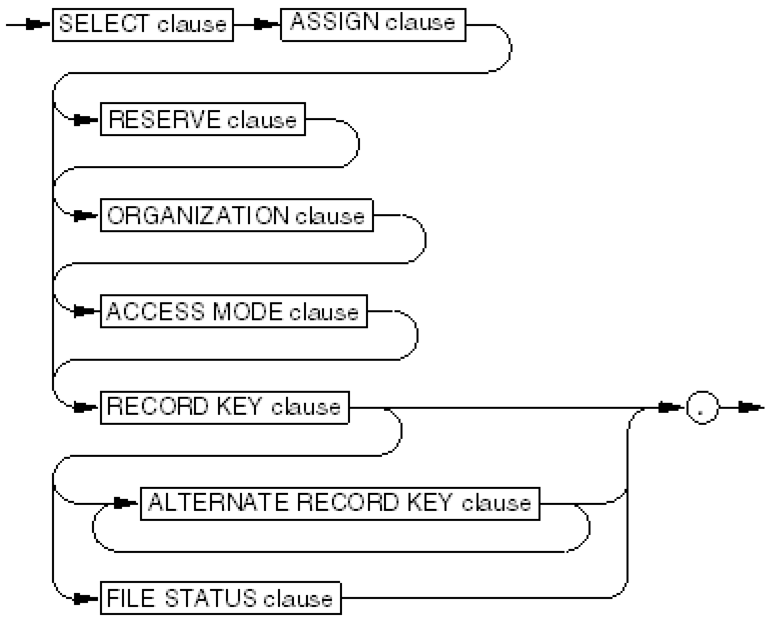
VST045.vsd

FILE STATUS clause



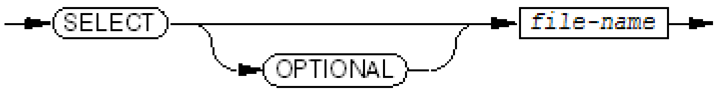
VST046.vsd

Indexed File



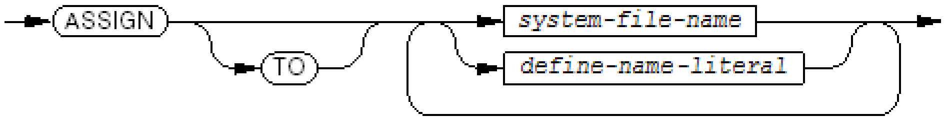
VST051.vsd

SELECT clause



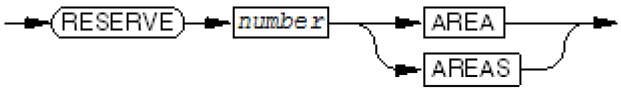
VST038.vsd

ASSIGN clause



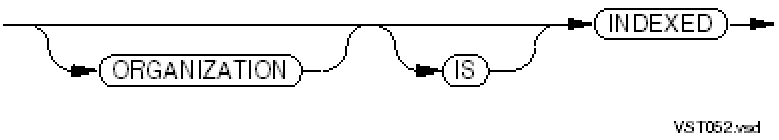
VST039.vsd

RESERVE clause

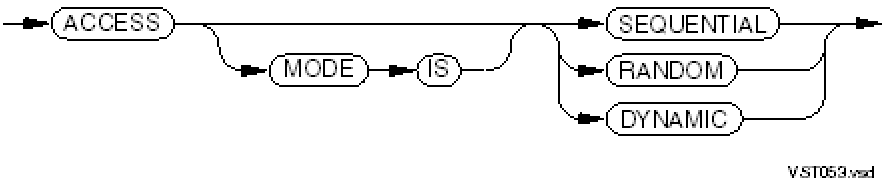


VST040.vsd

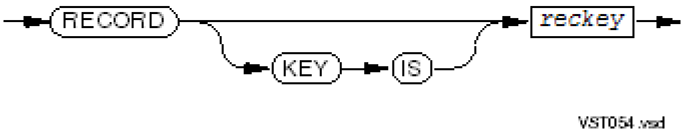
ORGANIZATION clause



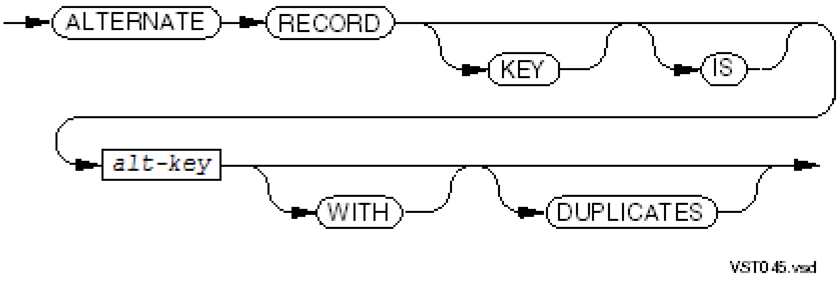
ACCESS MODE clause



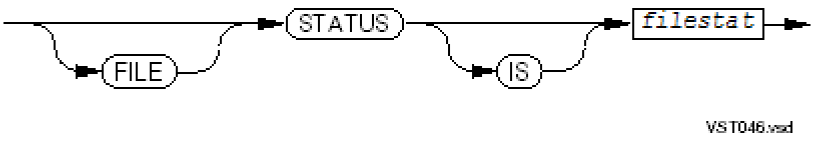
RECORD KEY clause



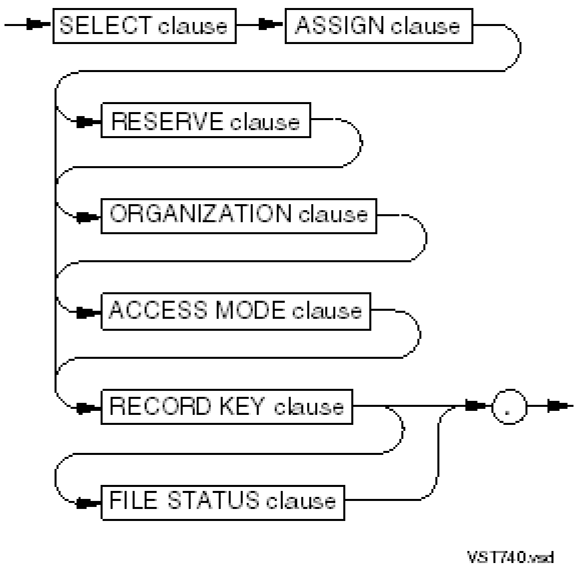
ALTERNATE RECORD KEY clause



FILE STATUS clause

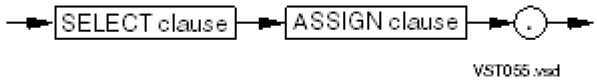


Queue File

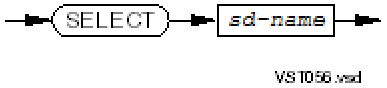


For descriptions of clauses, see Indexed File.

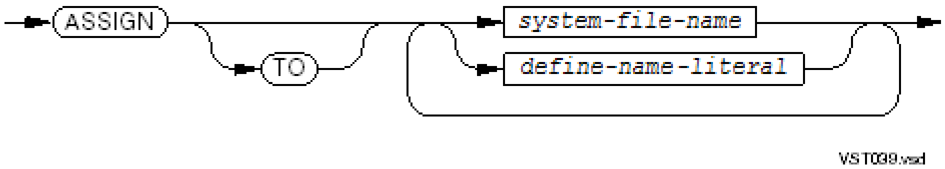
Sort-Merge File



SELECT clause



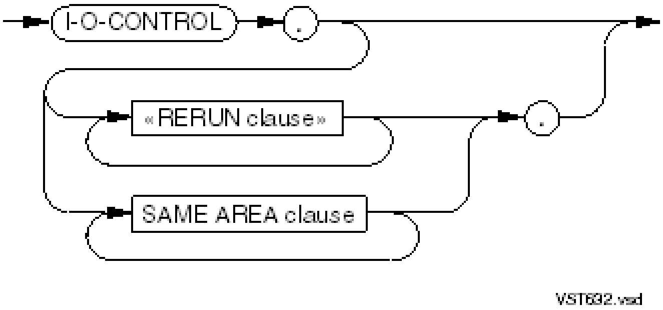
ASSIGN clause



I-O-CONTROL Paragraph

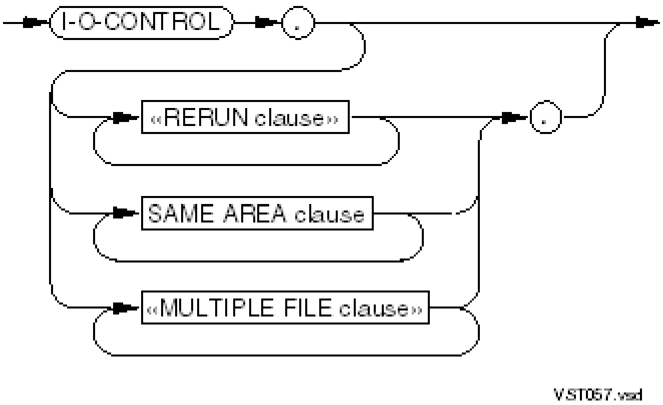
- Line Sequential File
- All Other File Types

Line Sequential File



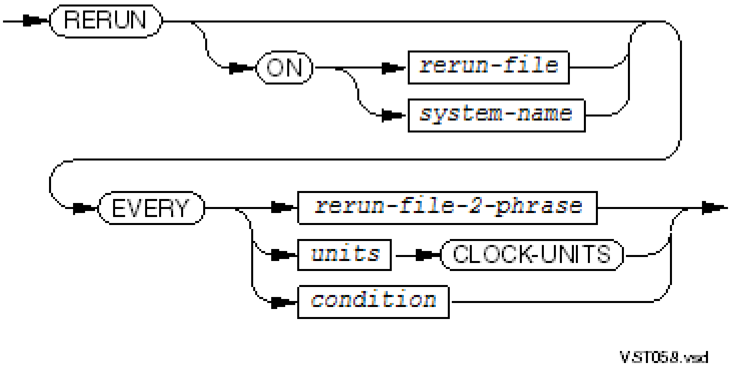
For descriptions of clauses, see All Other File Types.

All Other File Types

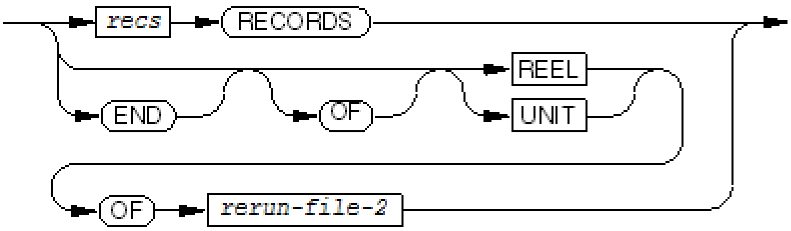


«RERUN clause»

NOTE: The 1985 COBOL standard classifies the RERUN clause as **obsolete**, so you are advised not to use it.

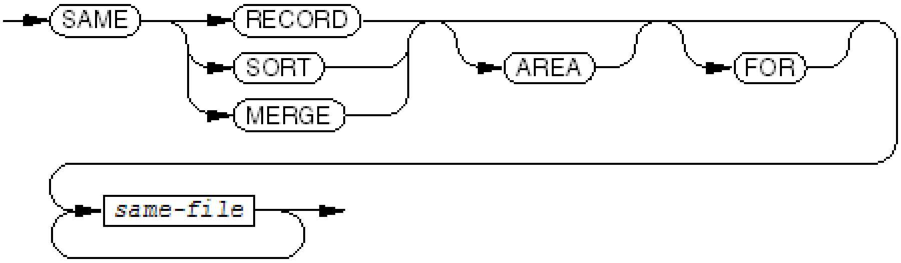


rerun-file-2-phrase



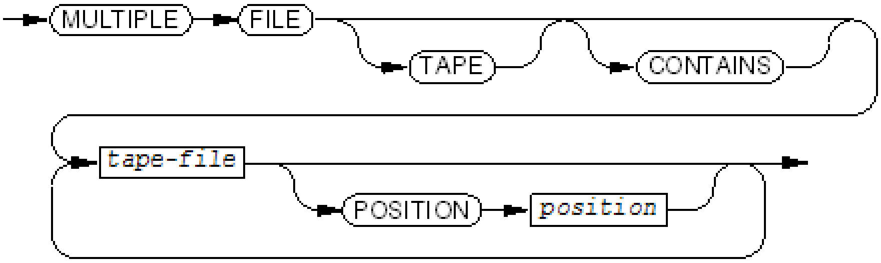
VST059.vsd

SAME AREA clause



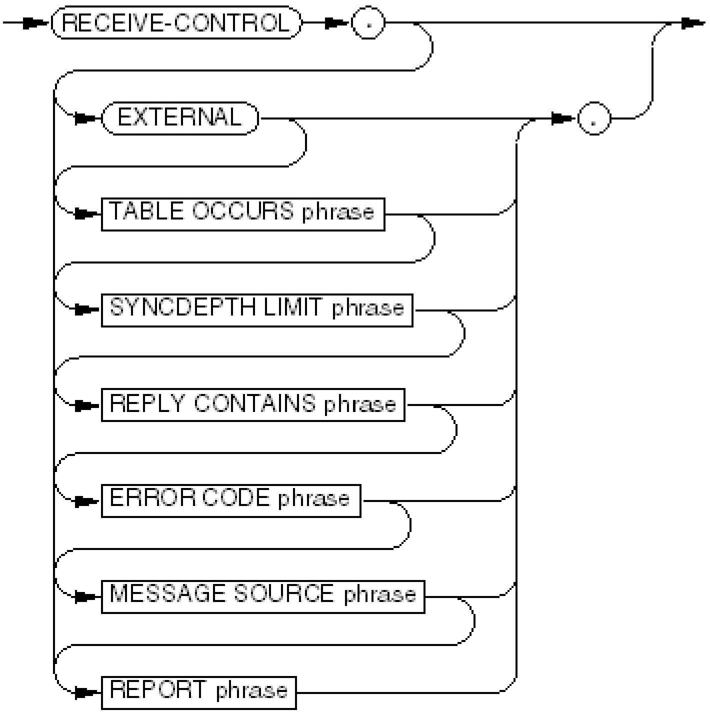
VST060.vsd

«MULTIPLE FILE clause»



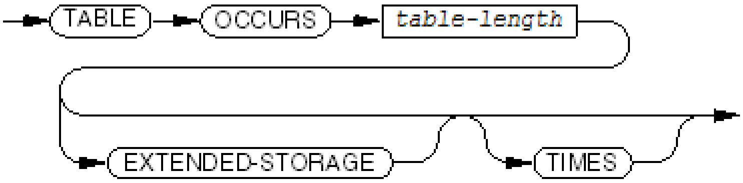
VST061.vsd

RECEIVE-CONTROL Paragraph



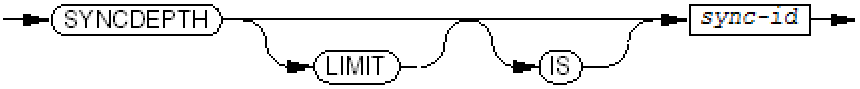
VST063.vsd

TABLE OCCURS phrase



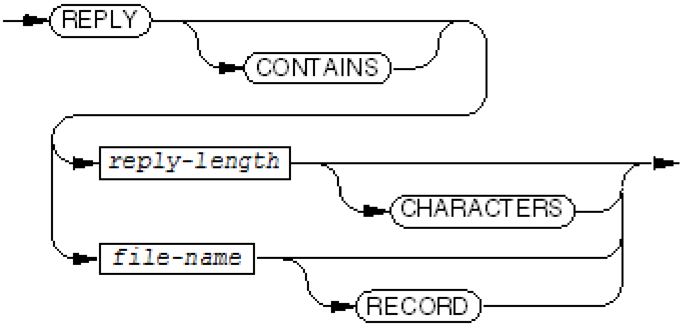
VST064.vsd

SYNCDEPTH LIMIT phrase



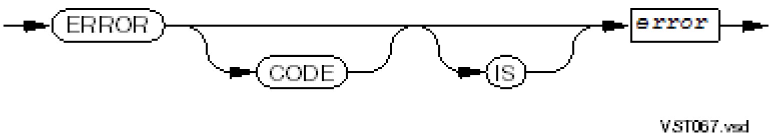
VST065.vsd

REPLY CONTAINS phrase

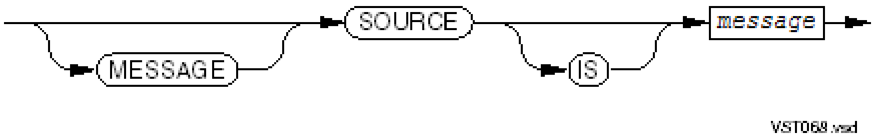


VST066.vsd

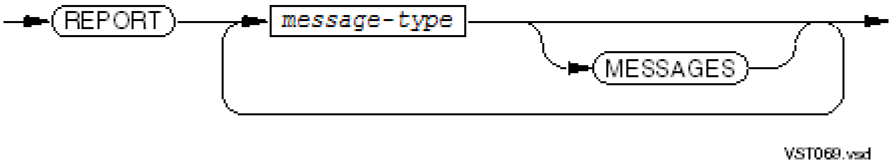
ERROR CODE phrase



MESSAGE SOURCE phrase



REPORT phrase



message-type

BREAK

CLOSE

CONTROL

CONTROLBUF

CPU-DOWN

CPU-UP

DEVICE-INFO

DEVICE-INFO-2-COMPLETION

FILE-GETINFOBYNAME-COMPLETION

FILE-FILENAME-COMPLETION

JOB-PROCESS-CREATION

LOGICAL-CLOSE

LOGICAL-OPEN

MEMORY-LOCK-COMPLETION

MEMORY-LOCK-FAILURE

MESSAGE-CANCELLED

MESSAGE-MISSED

NETWORK

NEWPROCESS-COMPLETION

NEWPROCESSNOWAIT-COMPLETION

NODE-DOWN

NODE-UP

OPEN

PATHSEND-DIALOG-ABORT

POWER-ON

PROCESS-CREATE-COMPLETION

PROCESS-DELETION

PROCESS-TIME-SIGNAL

REMOTE-CPU-DOWN

REMOTE-CPU-UP

RESETSYNC

SETMODE

SETPARAM

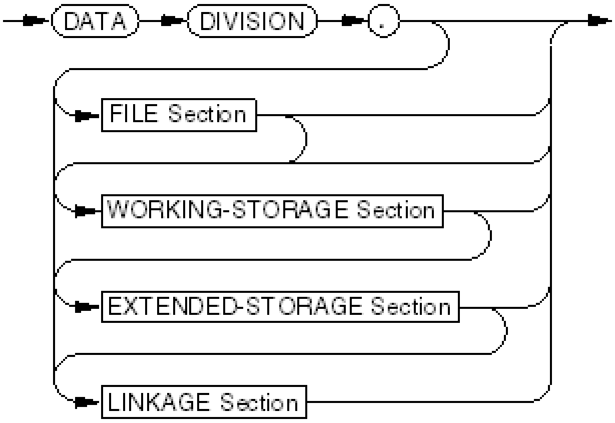
SETTIME

STATUS-3270

SYSTEM

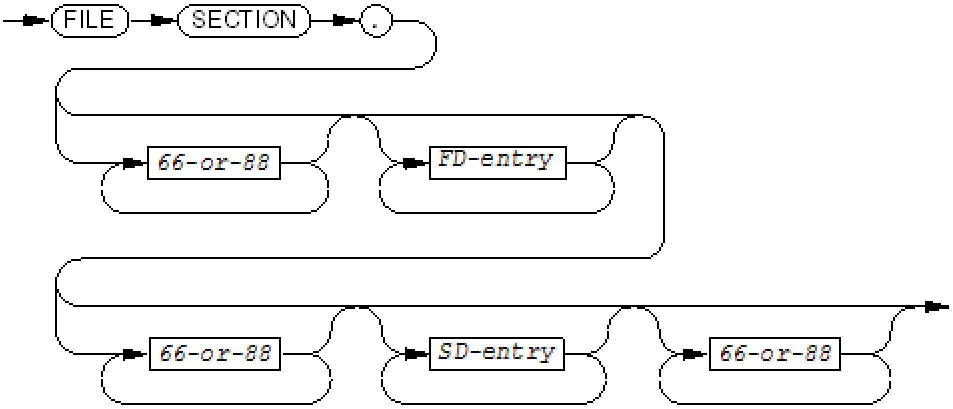
TIME-SIGNAL

43 Data Division



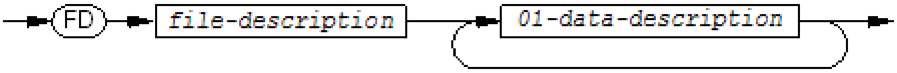
VST070.vsd

FILE Section



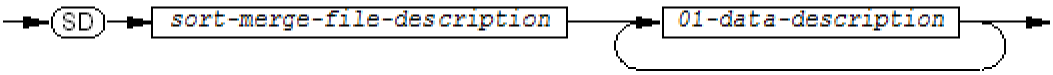
VST071.vsd

FD-entry



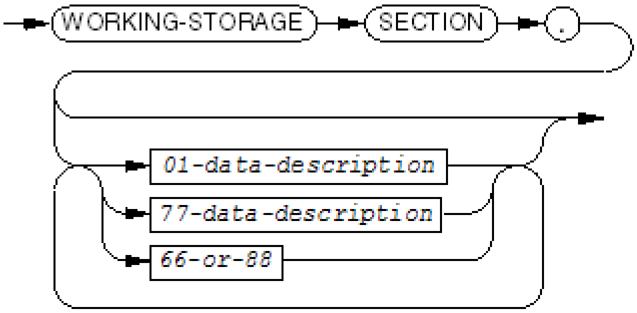
VST072.vsd

SD-entry



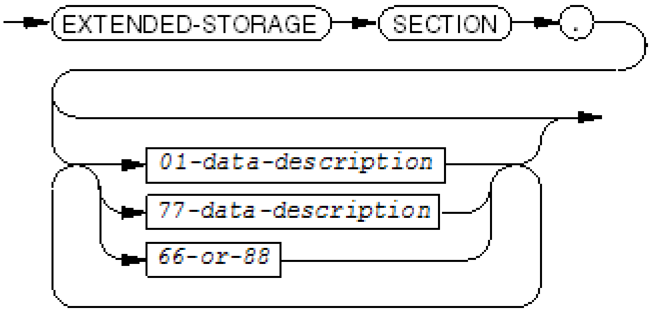
VST073.vsd

WORKING-STORAGE Section



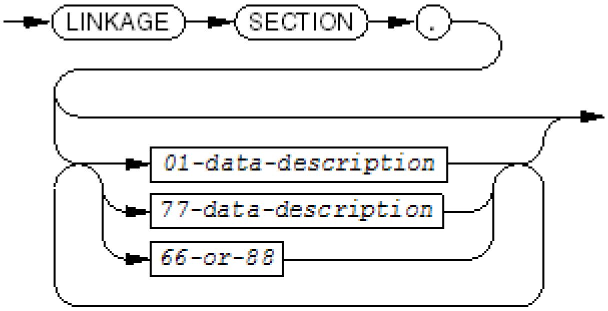
VST094.vsd

EXTENDED-STORAGE Section



VST095.vsd

LINKAGE Section

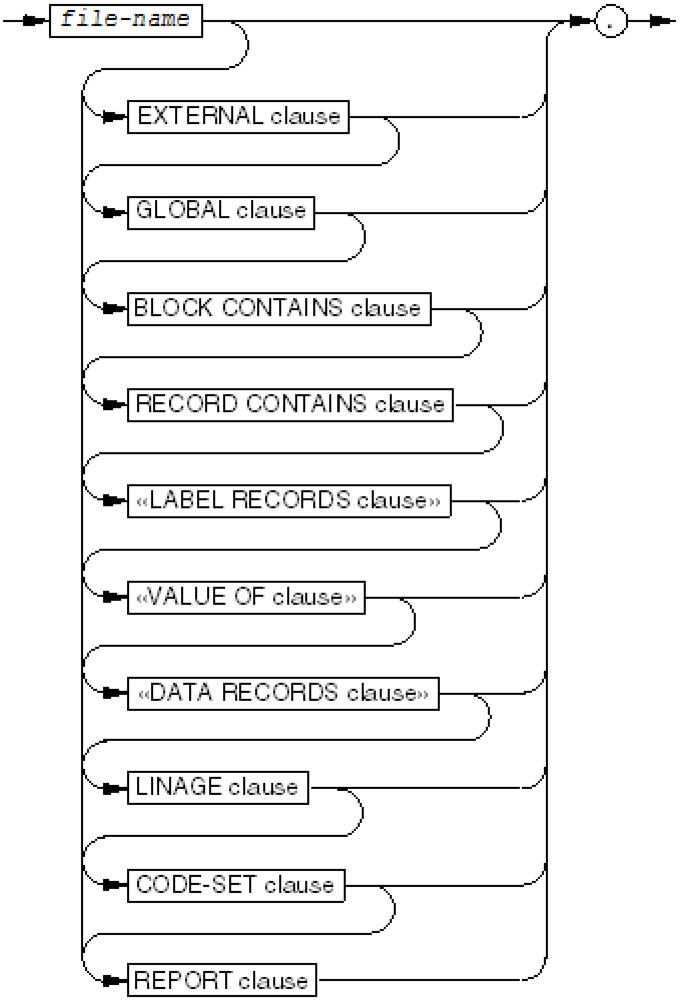


VST096.vsd

File Descriptions

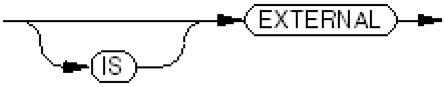
- Sequential File
- Line Sequential File
- Relative, Indexed, or Queue File
- Sort-Merge File

Sequential File



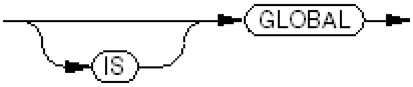
VST074.vsd

EXTERNAL clause



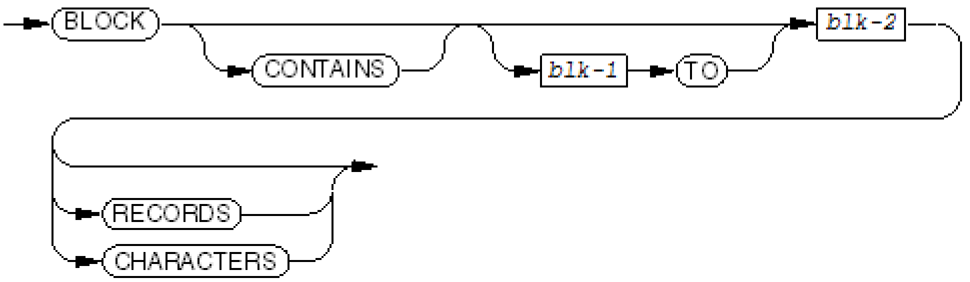
VST075.vsd

GLOBAL clause



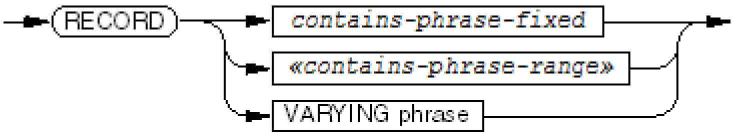
VST076.vsd

BLOCK CONTAINS clause



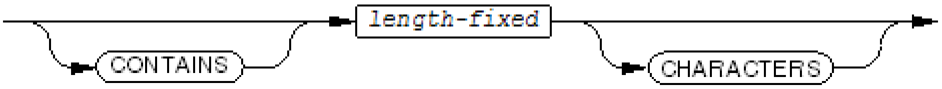
VST077.vsd

RECORD CONTAINS clause



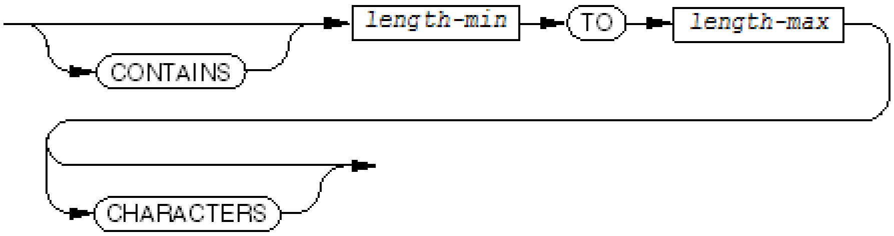
VST078.vsd

contains-fixed-phrase



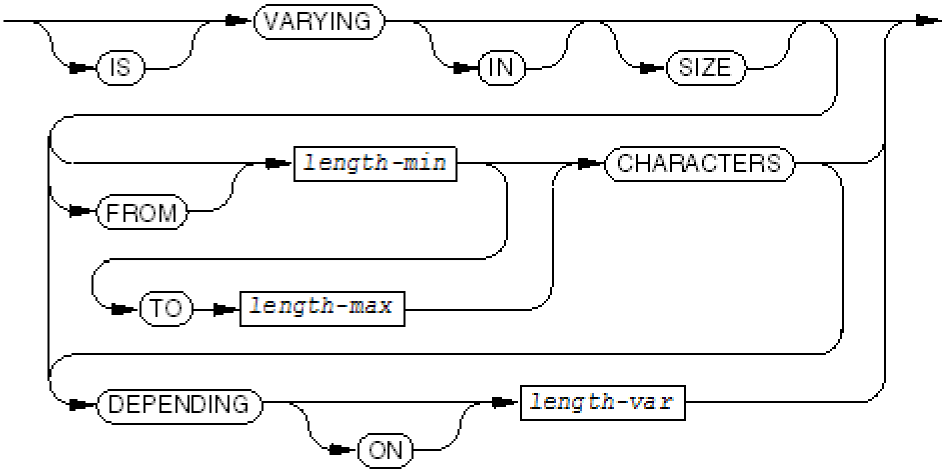
VST079.vsd

«contains-phrase-range»



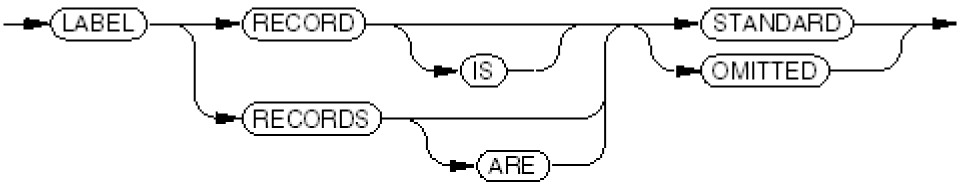
VST405.vsd

VARYING phrase



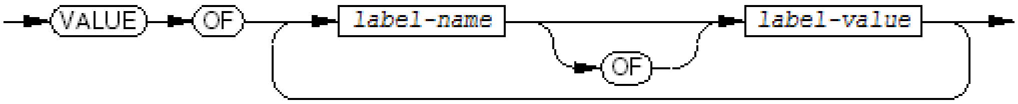
VST092.vsd

«LABEL RECORDS clause»



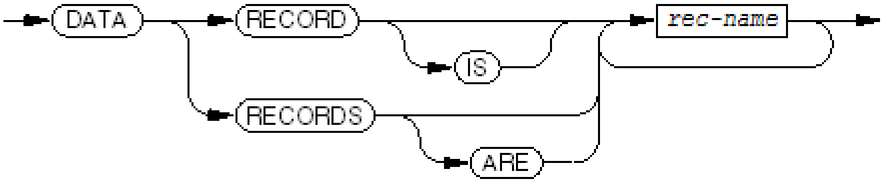
VST083.vsd

«VALUE OF clause»



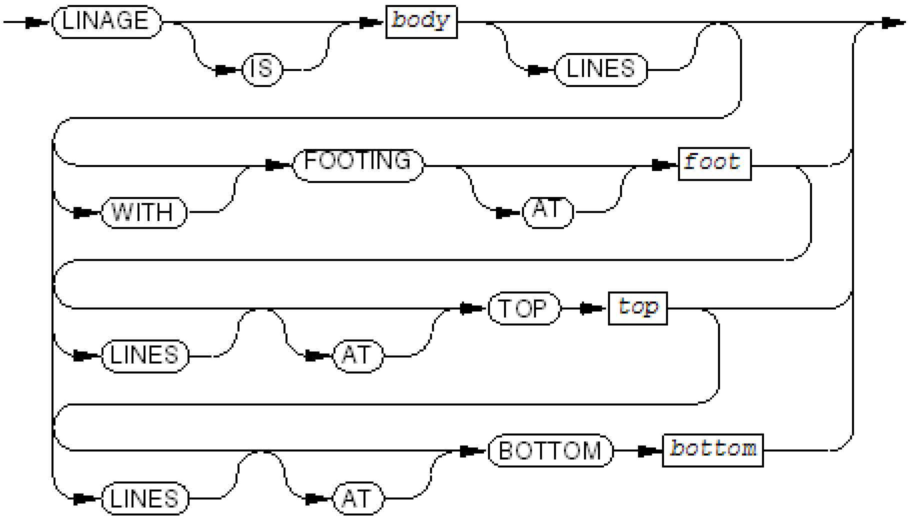
VST084.vsd

«DATA RECORDS clause»



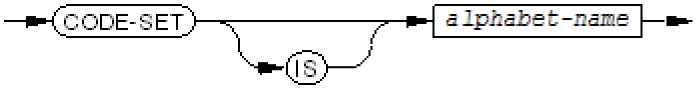
VST085.vsd

LINAGE clause



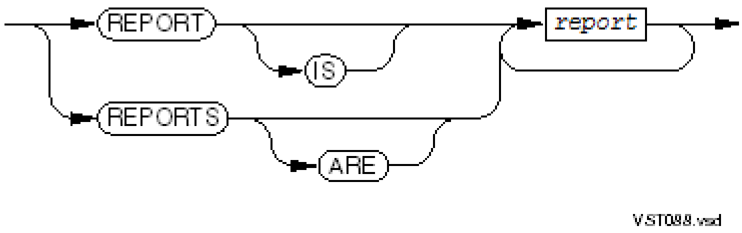
VST086.vsd

CODE-SET clause

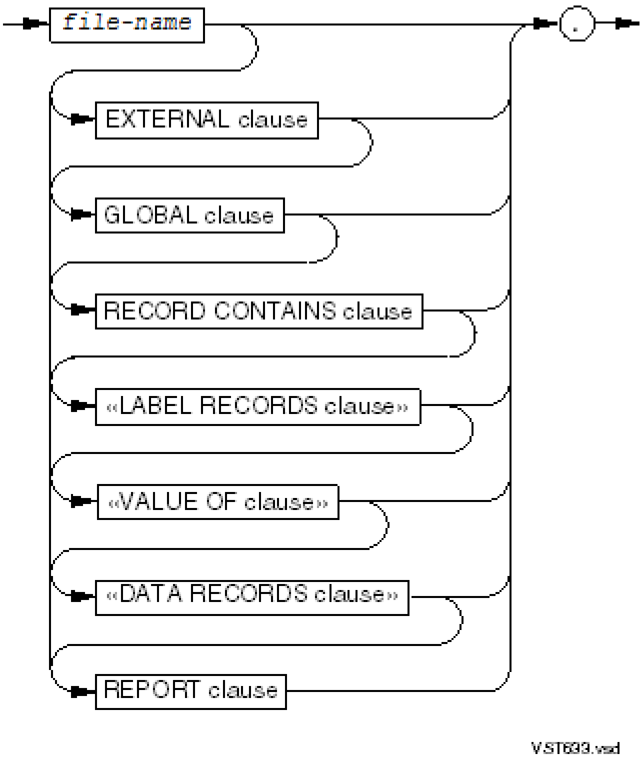


VST087.vsd

REPORT clause

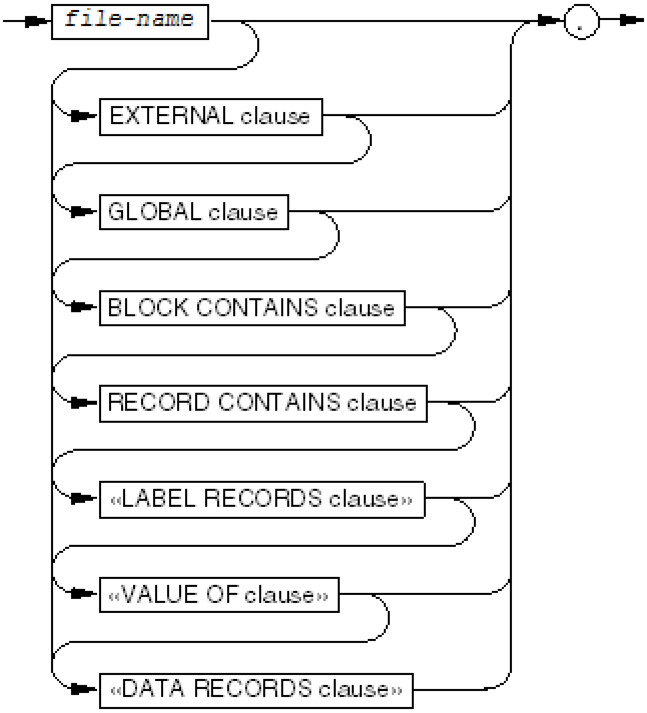


Line Sequential File



For descriptions of clauses, see Sequential File.

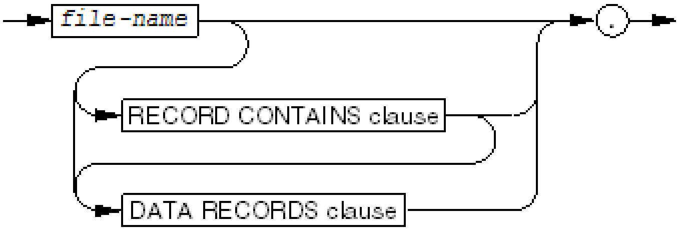
Relative, Indexed, or Queue File



VST697.vsd

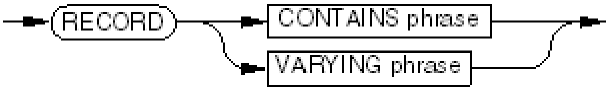
For descriptions of clauses, see Sequential File.

Sort-Merge File



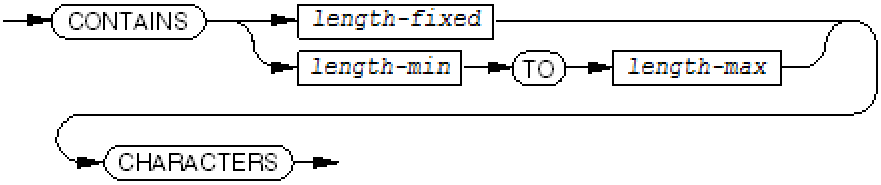
VST089.vsd

RECORD CONTAINS clause



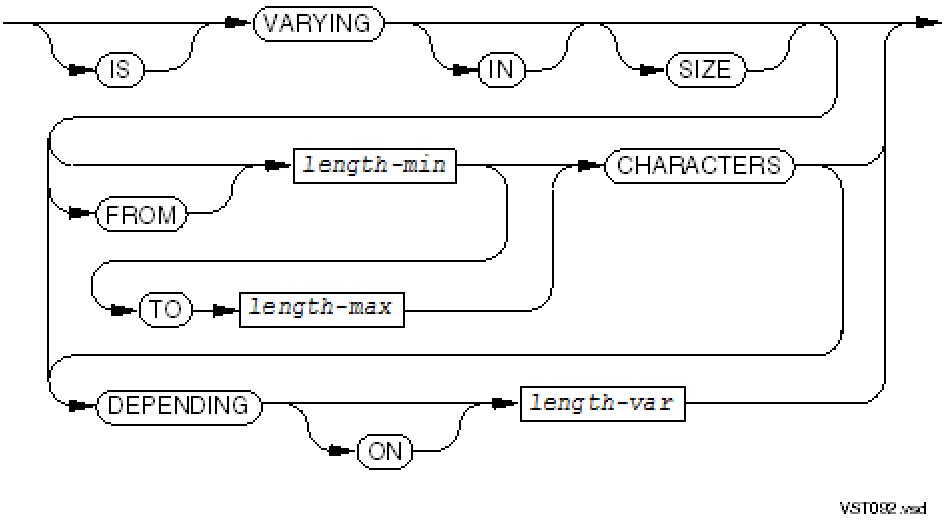
VST090.vsd

CONTAINS phrase

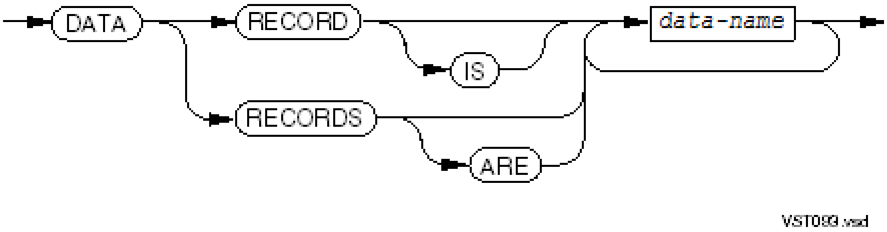


VST091.vsd

VARYING phrase



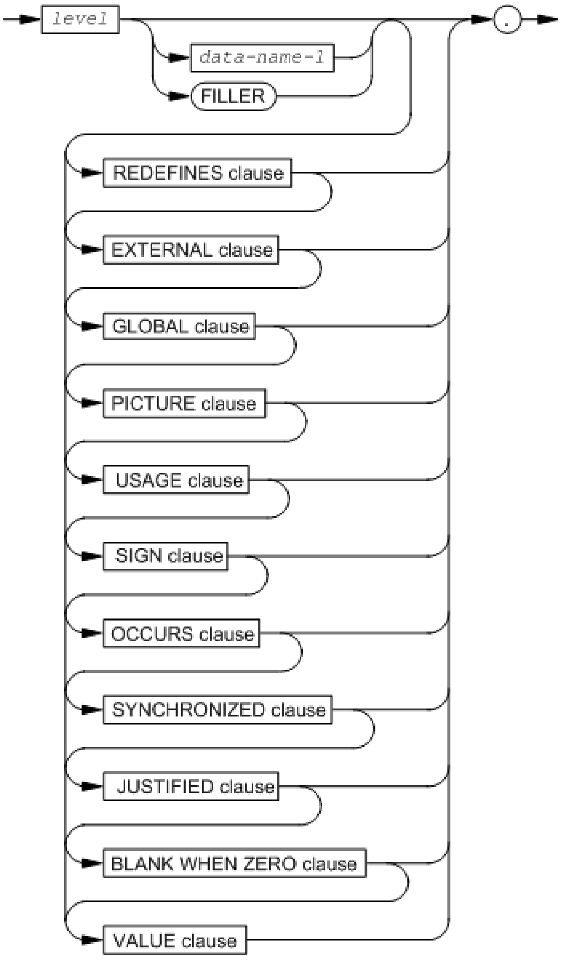
DATA RECORDS clause



Data Descriptions

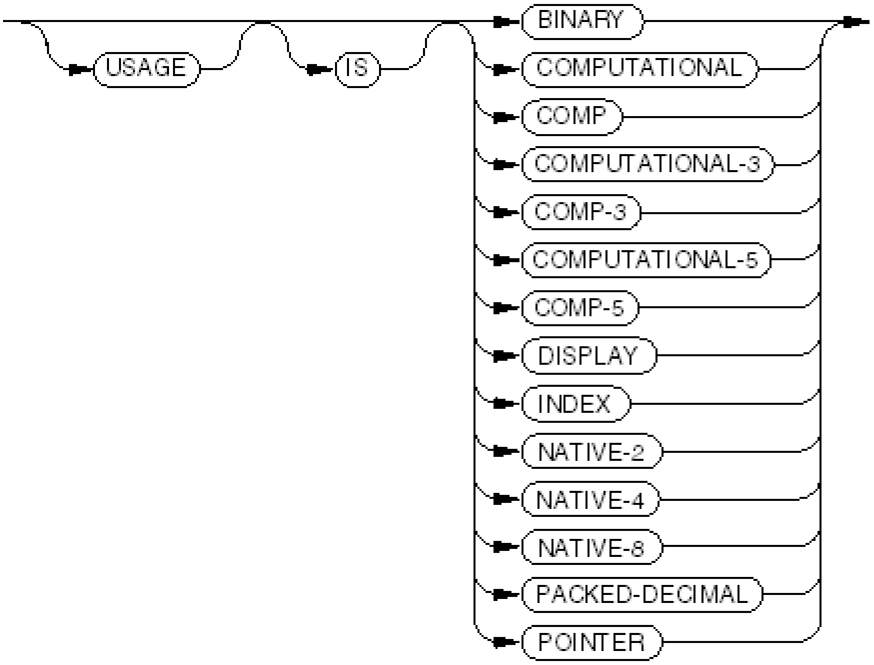
- Record Item (Levels 01 through 49)
- Level-66 Item
- Level-77 Item
- Level-88 Item
- Clause for Data Descriptions in the Linkage Section

Record Item (Levels 01 through 49)



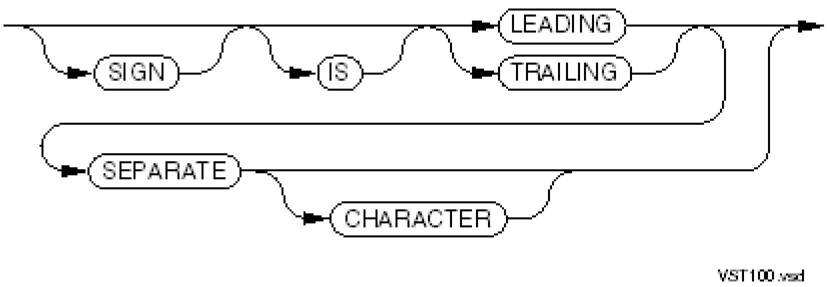
VST098.vsd

USAGE clause



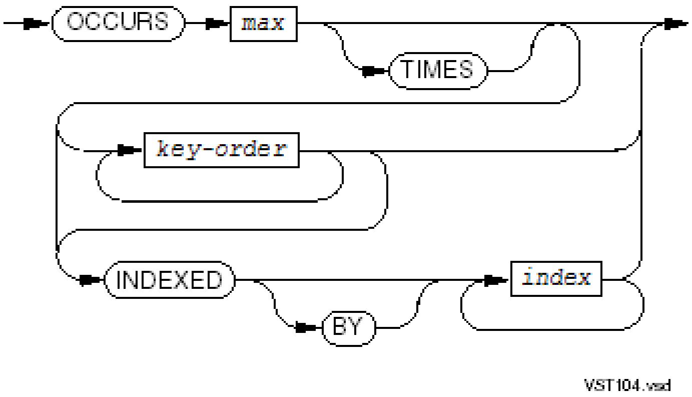
VST099.vsd

SIGN clause

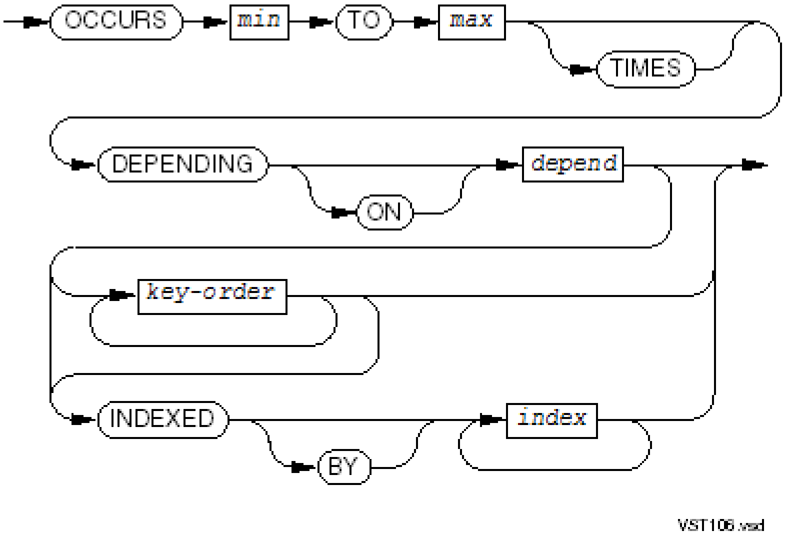


OCCURS clause

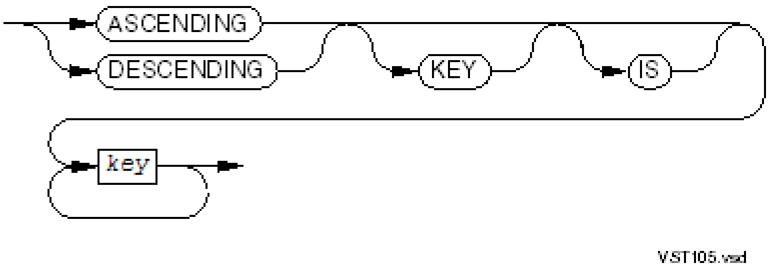
For a fixed-size table:



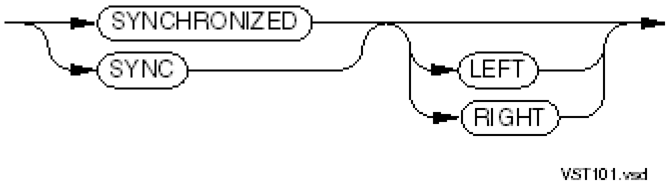
For a variable-size table:



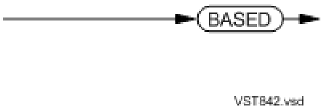
key-order



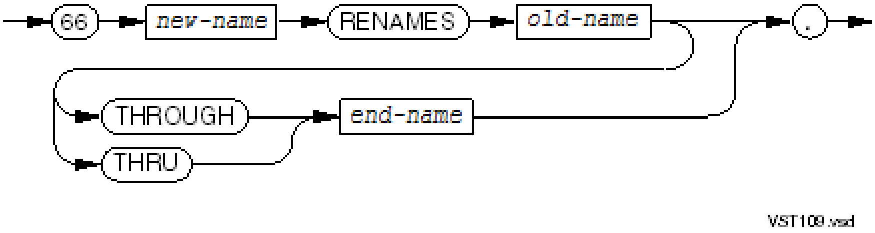
SYNCHRONIZED clause



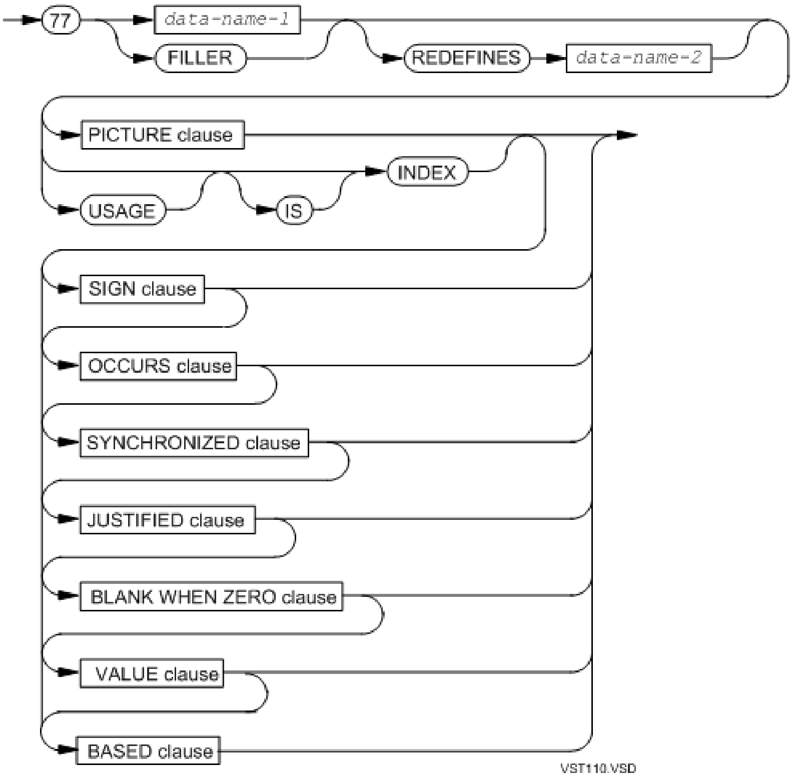
BASED clause



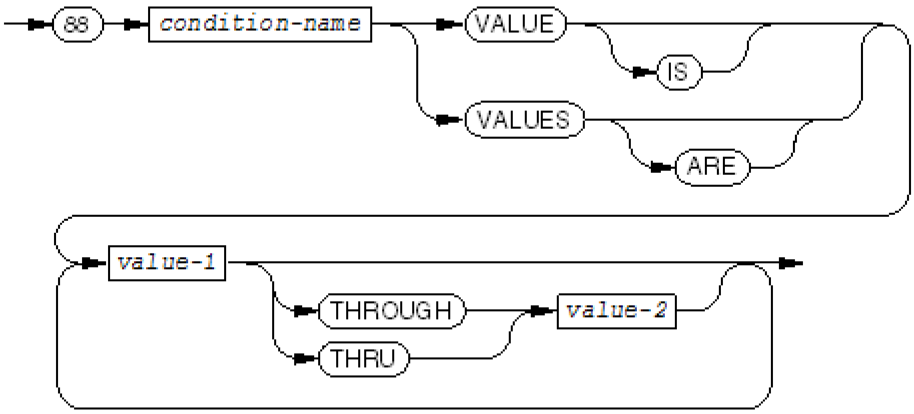
Level-66 Item



Level-77 Item

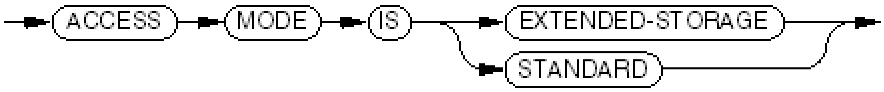


Level-88 Item



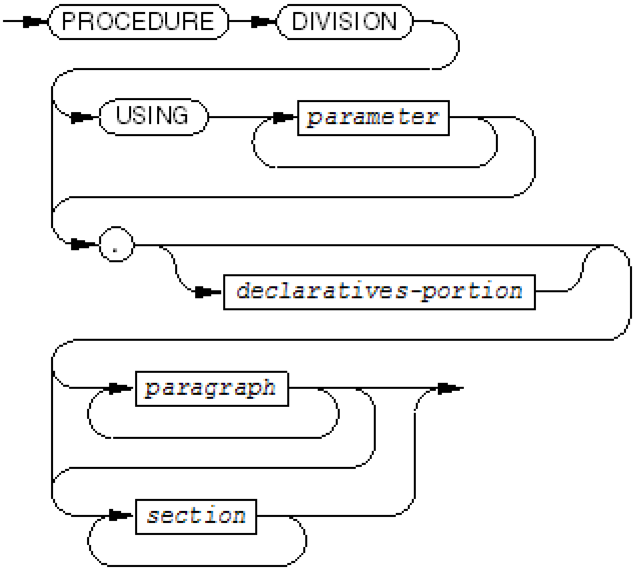
VST112.vsd

Clause for Data Descriptions in the Linkage Section



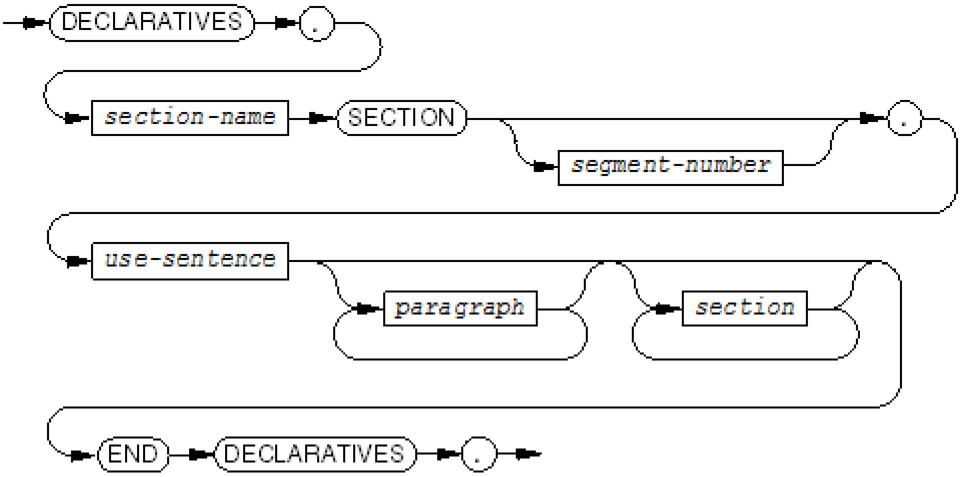
VST097.vsd

44 Procedure Division



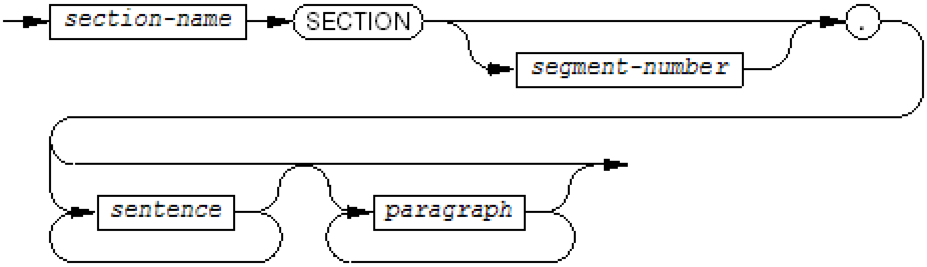
VST113.vsd

declaratives-portion



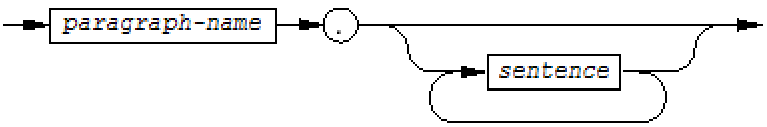
VST117.vsd

section



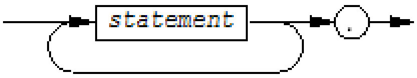
VST116.vsd

paragraph



VST115.vsd

sentence

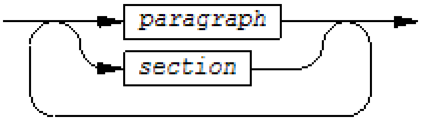


VST114.vsd

statement

See Statements.

procedure



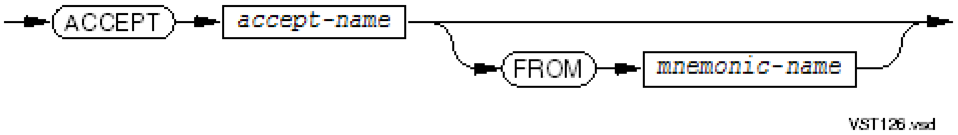
VST415.vsd

Statements:

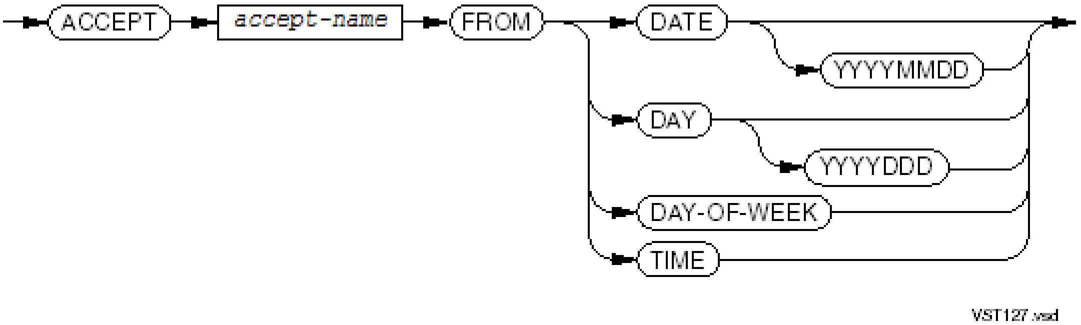
- ACCEPT With Mnemonic-Name (page 1076)
- ACCEPT With DATE, DAY, DAY-OF-WEEK, or TIME Phrase (page 1076)
- ADD TO (page 1077)
- ADD GIVING (page 1078)
- ADD CORRESPONDING (page 1079)
- ALLOCATE Bytes (page 1079)
- ALLOCATE Based Item (page 1079)
- ALTER (page 302)
- CALL (page 1080)
- CANCEL (page 1081)
- CHECKPOINT (page 1081)
- CLOSE for Sequential and Line Sequential Files (page 1081)
- CLOSE for Relative, Indexed, and Queue Files (page 1082)
- COMPUTE (page 1082)
- CONTINUE (page 1082)
- COPY (page 1083)
- DELETE (page 1084)
- DISPLAY (page 1085)
- DIVIDE INTO (page 1085)
- DIVIDE GIVING (page 1086)
- DIVIDE GIVING REMAINDER (page 1086)
- ENTER (page 1087)
- ENTER COBOL (page 1087)
- EVALUATE (page 1087)

- WRITE for Sequential Files (page 1114)
- WRITE for Line Sequential Files (page 1115)
- WRITE for Relative, Indexed, and Queue Files (page 1115)

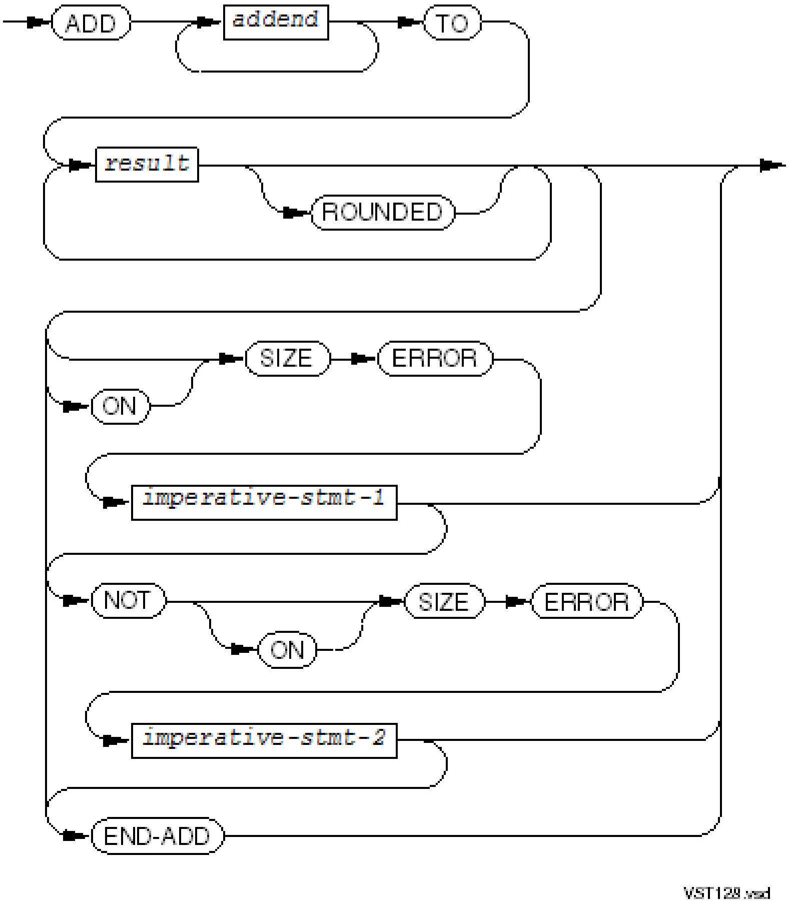
ACCEPT With Mnemonic-Name



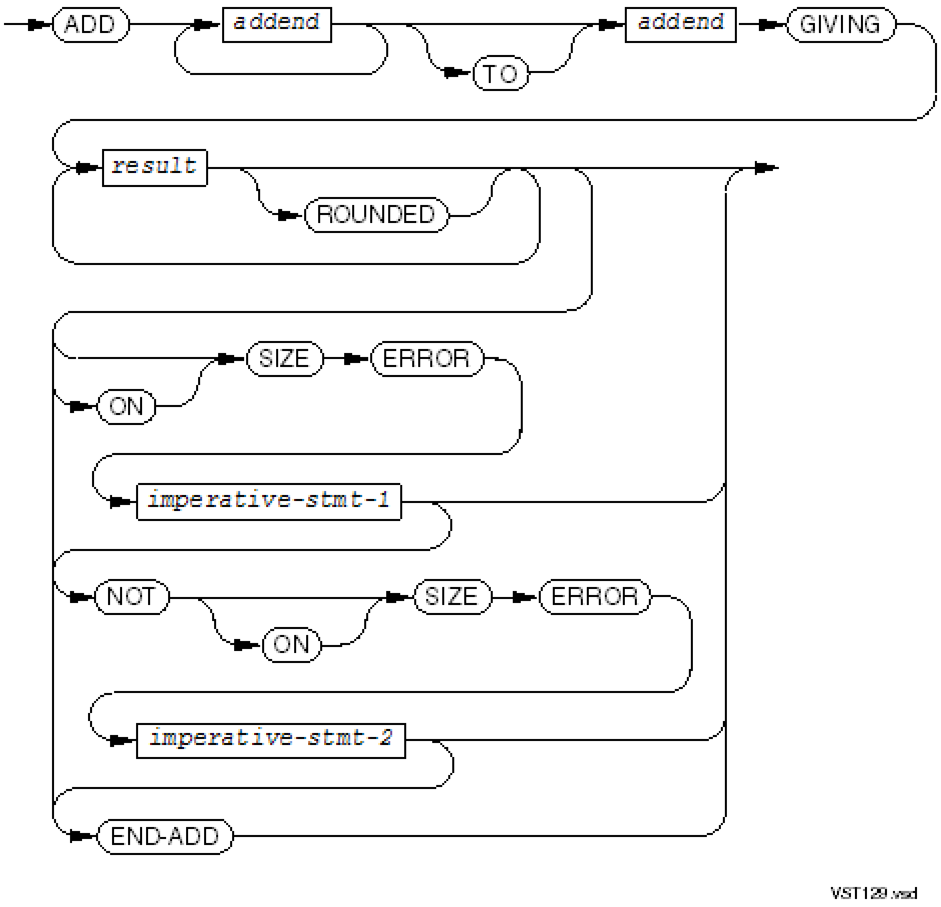
ACCEPT With DATE, DAY, DAY-OF-WEEK, or TIME Phrase



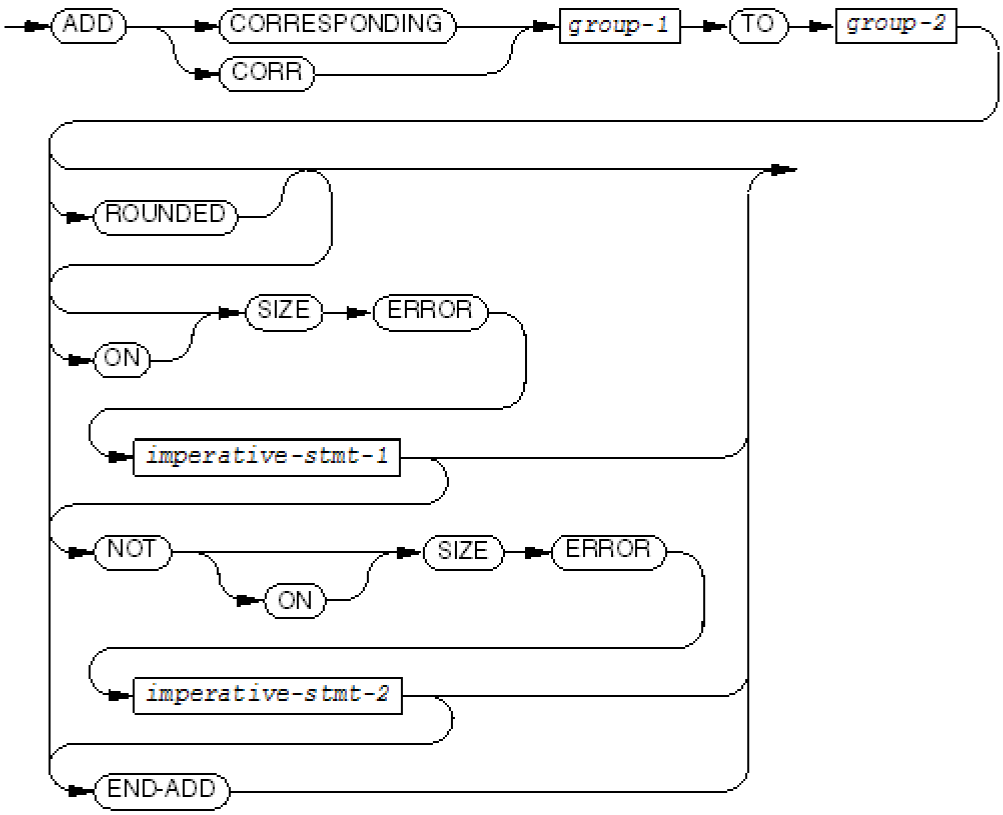
ADD TO


ADD
addend
TO
result
ROUNDED
ON
SIZE
ERROR
imperative-stmt-1
NOT
ON
SIZE
ERROR
imperative-stmt-2
END-ADD
VST128.vsd

ADD GIVING

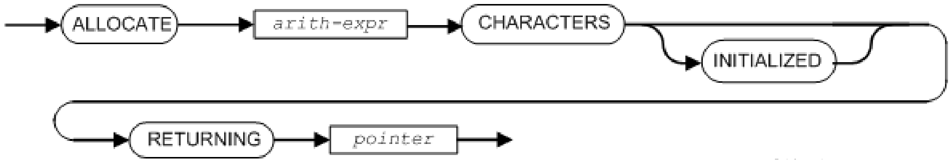

ADD
addend
TO
addend
GIVING
result
ROUNDED
ON
SIZE
ERROR
imperative-stmt-1
NOT
ON
SIZE
ERROR
imperative-stmt-2
END-ADD
VST129.vsd

ADD CORRESPONDING



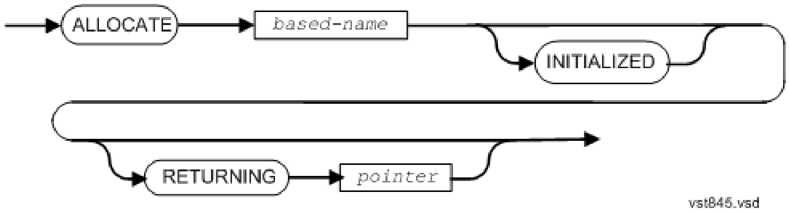
VST130.vsd

ALLOCATE Bytes



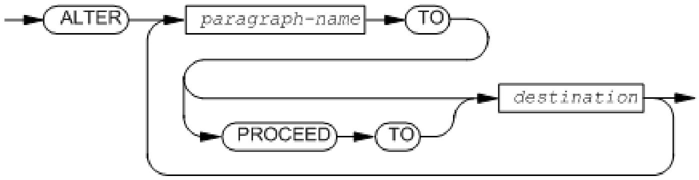
vst844.vsd

ALLOCATE Based Item



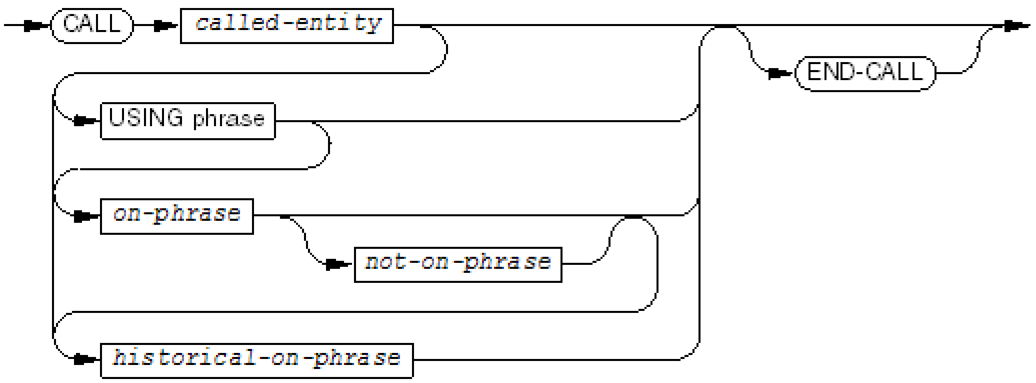
vst845.vsd

ALTER



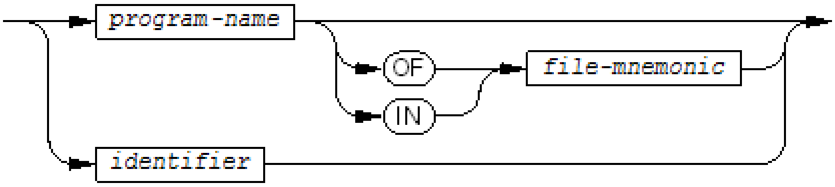
VST131.vsd

CALL



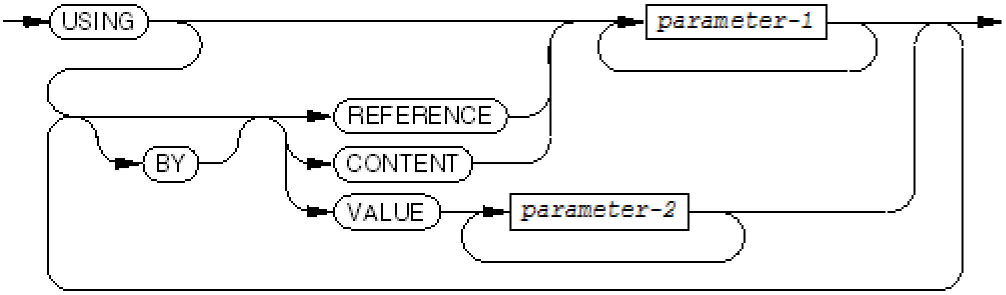
VST132.vsd

called-entity



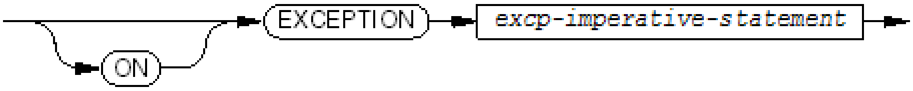
VST133.vsd

USING phrase



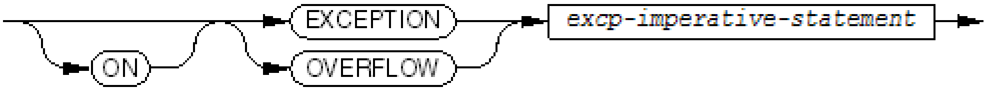
VST134.vsd

on-phrase



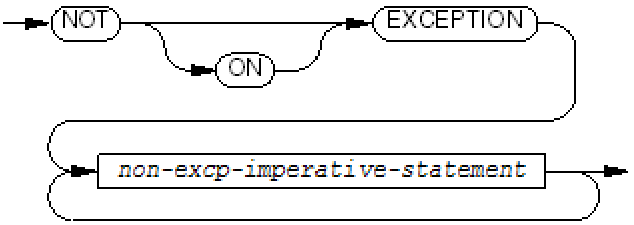
VST615.vsd

historical-on-phrase



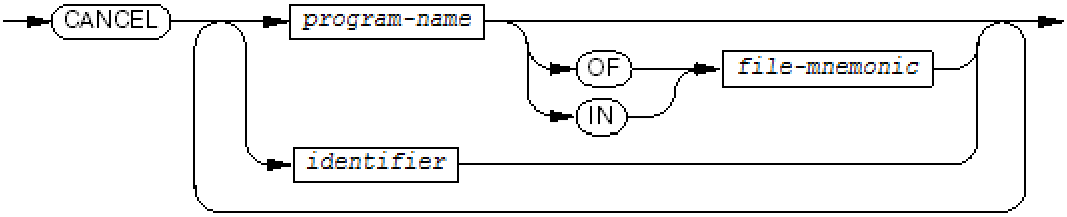
VST135.vsd

not-on-phrase



VST528.vsd

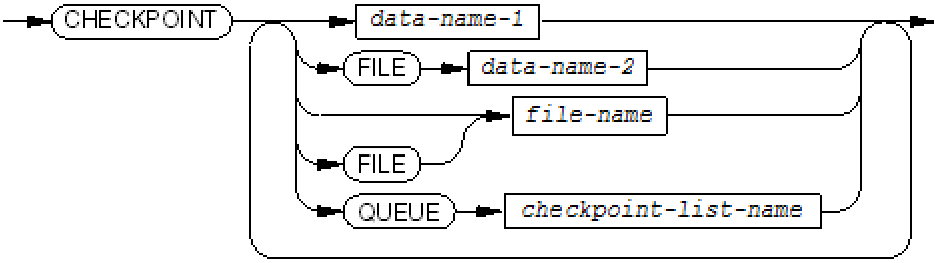
CANCEL



VST136.vsd

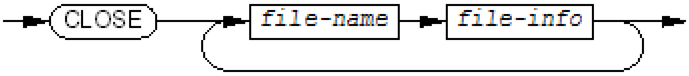
CHECKPOINT

NOTE: Do not use this directive in the OSS environment.



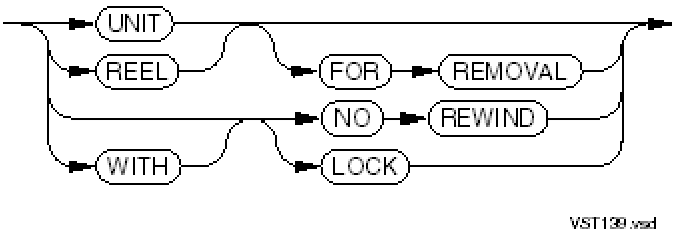
VST137.vsd

CLOSE for Sequential and Line Sequential Files

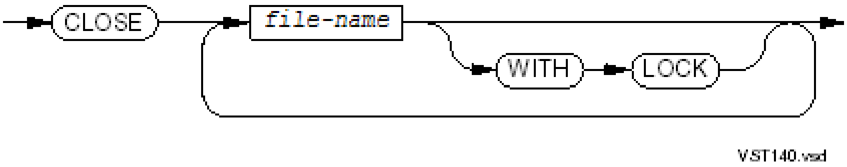


VST138.vsd

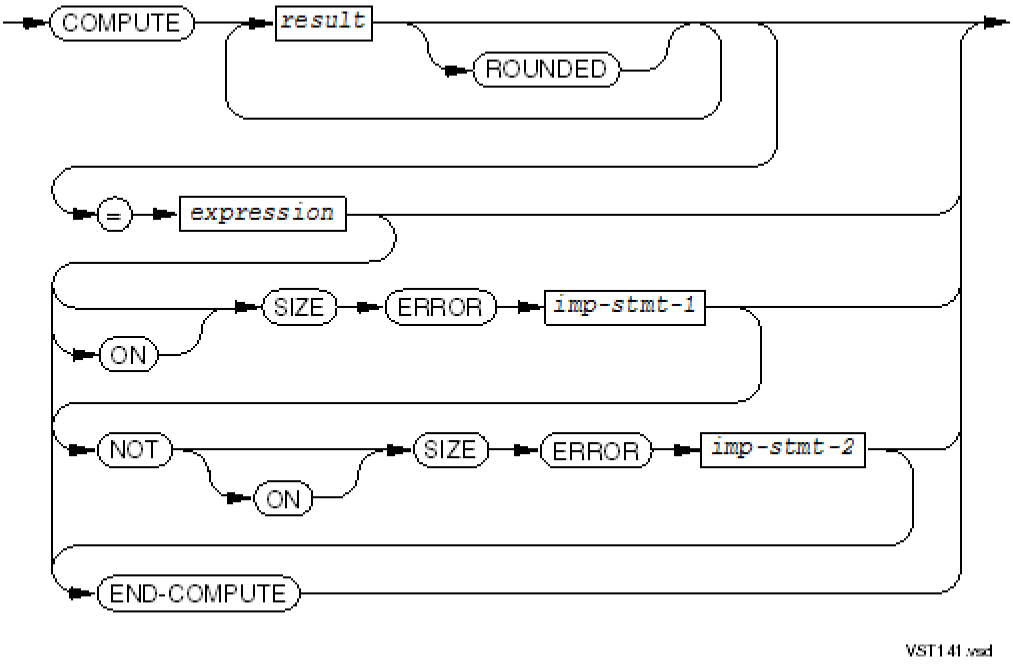
file-info



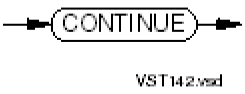
CLOSE for Relative, Indexed, and Queue Files



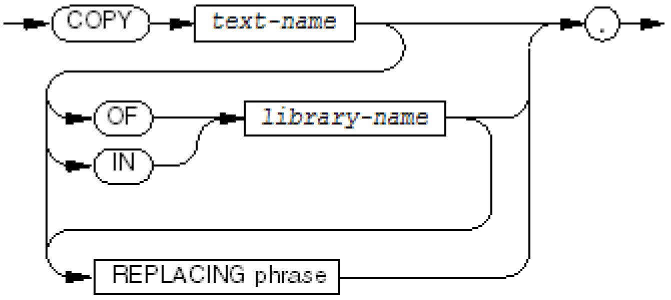
COMPUTE



CONTINUE

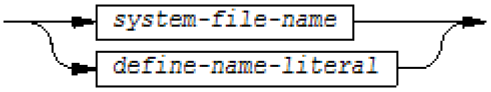


COPY



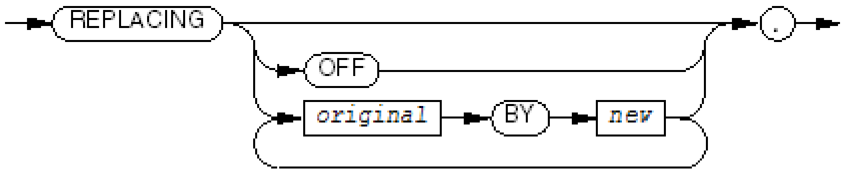
VST244.vsd

library-name



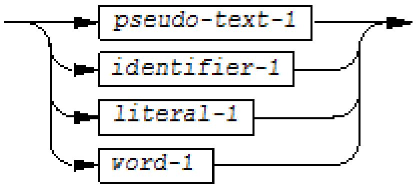
VST245.vsd

REPLACING phrase



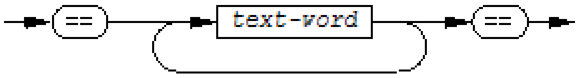
VST247.vsd

original



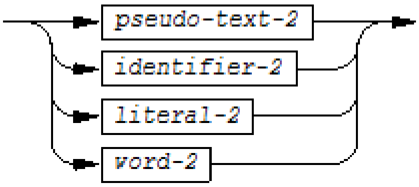
VST248.vsd

pseudo-text-1



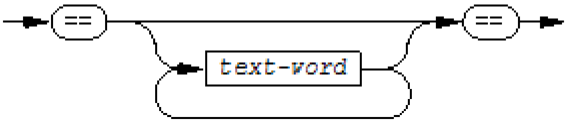
VST250.vsd

new



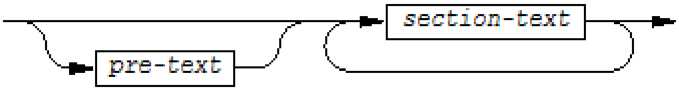
VST249.vsd

pseudo-text-2



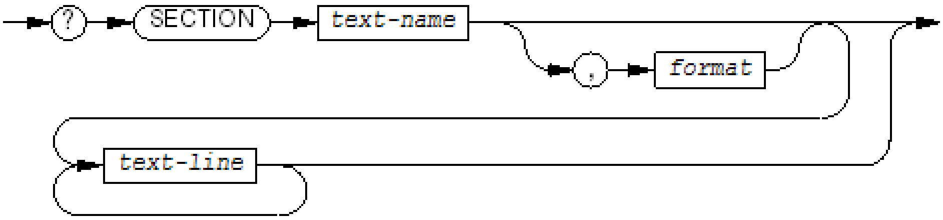
VST251.vsd

Format of a COPY Library



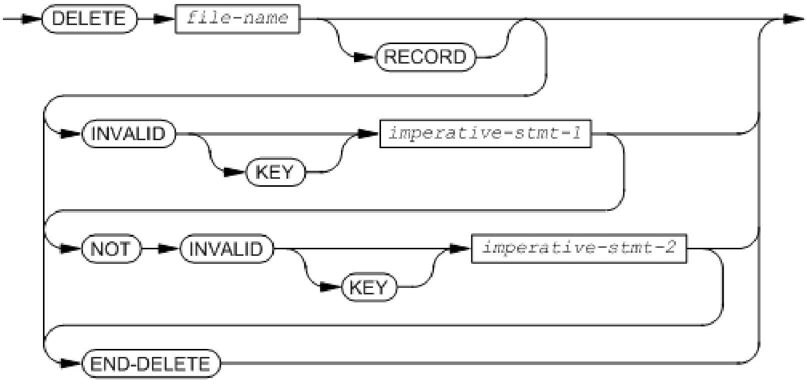
VST252.vsd

section-text



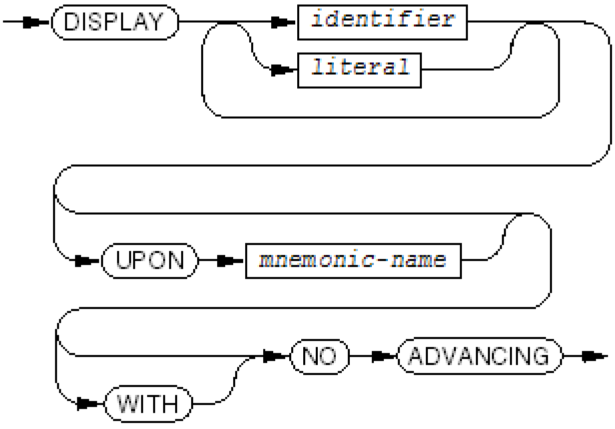
VST253.vsd

DELETE



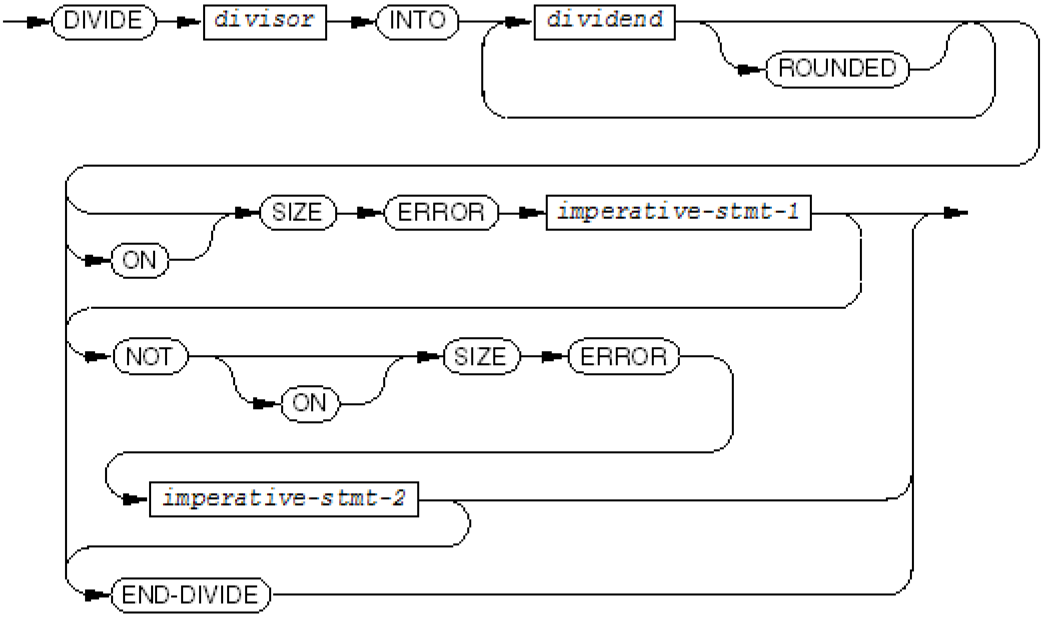
VST143.vsd

DISPLAY



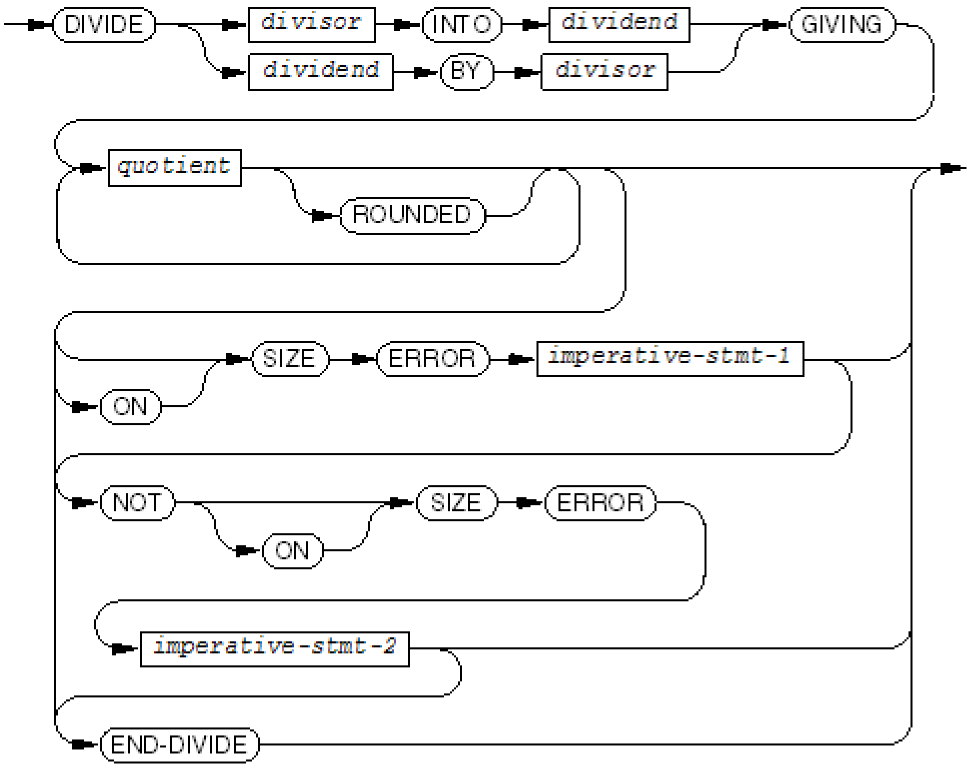
VST144.vsd

DIVIDE INTO



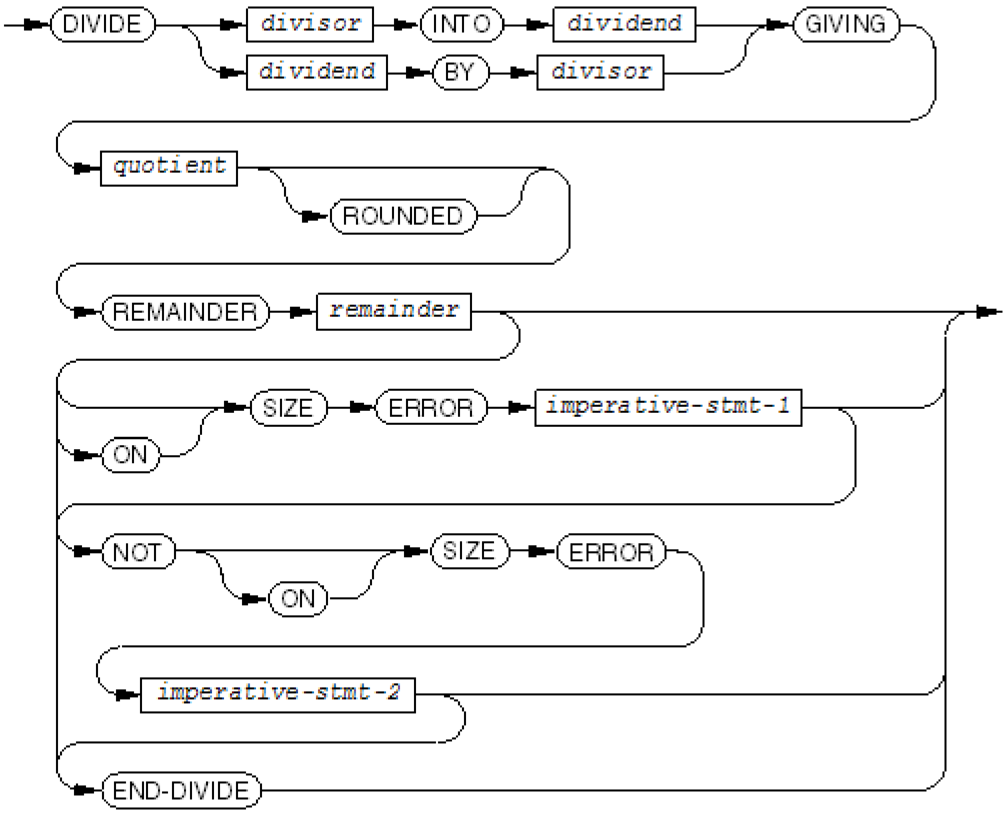
VST145.vsd

DIVIDE GIVING



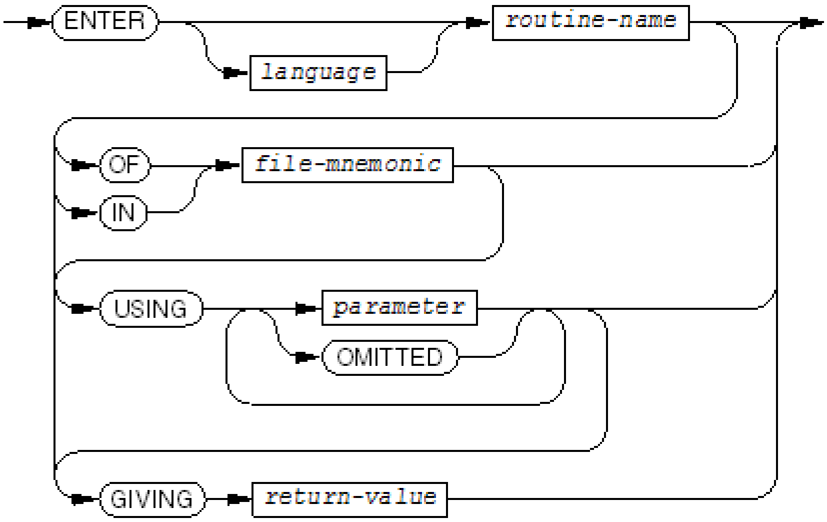
VST146.vsd

DIVIDE GIVING REMAINDER



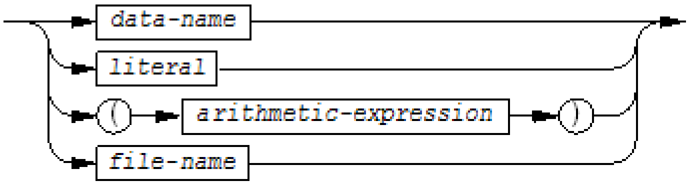
VST147.vsd

ENTER



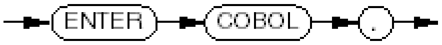
VST148.vsd

parameter



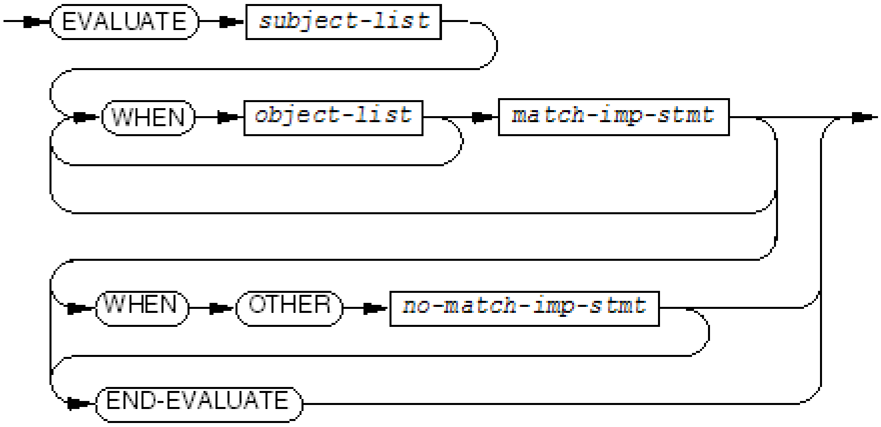
VST149.vsd

ENTER COBOL



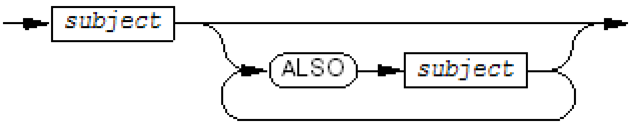
VST150.vsd

EVALUATE



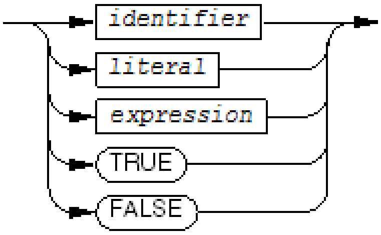
VST151.vsd

subject-list



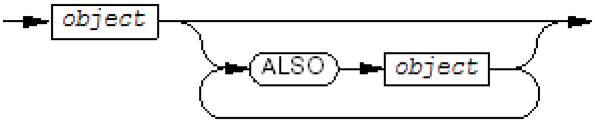
VST616.vsd

subject



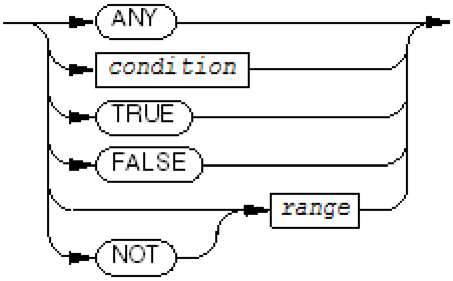
VST152.vsd

object-list



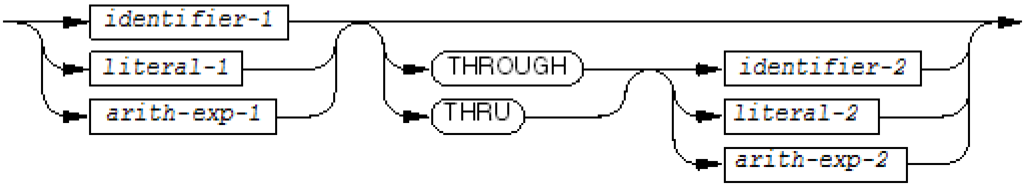
VST153.vsd

object



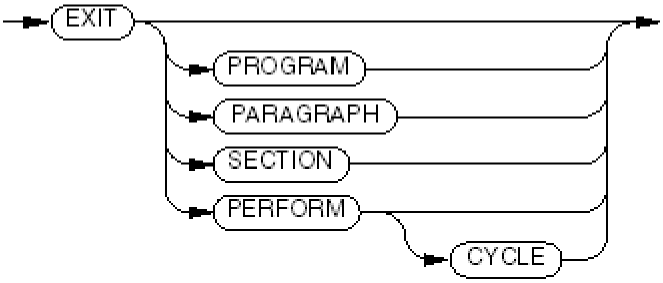
VST154.vsd

range



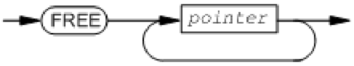
VST155.vsd

EXIT



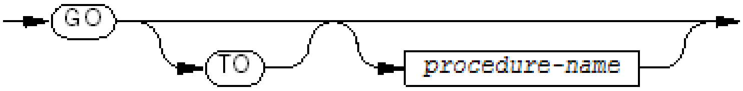
VST156.vsd

FREE



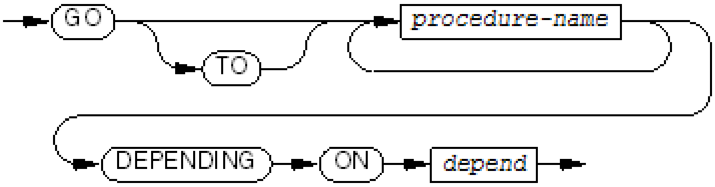
VST846.vsd

Unconditional GO TO



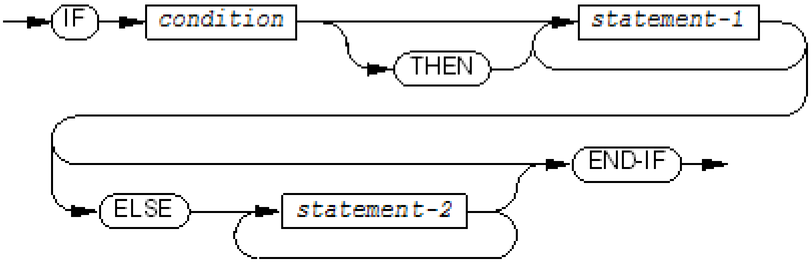
VST158.vsd

Conditional GO TO



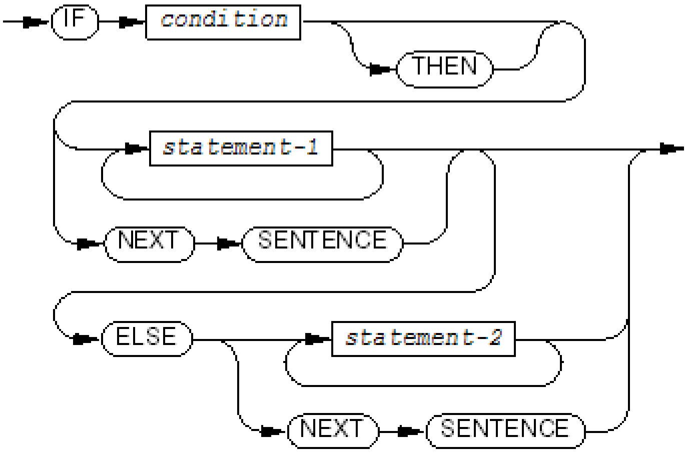
VST159.vsd

Delimited-Scope IF



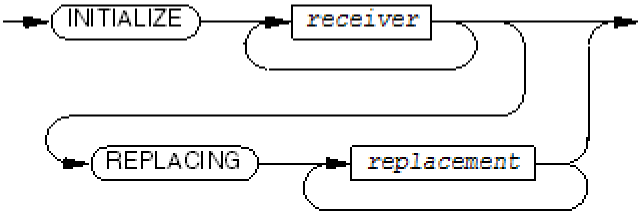
VST160.vsd

Conditional IF



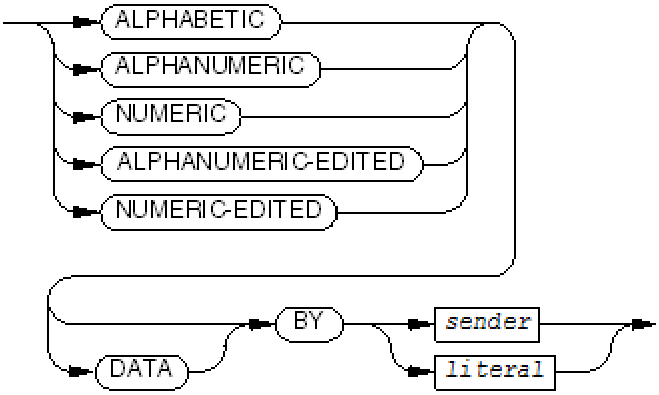
VST161.vsd

INITIALIZE



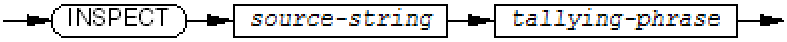
VST162.vsd

replacement



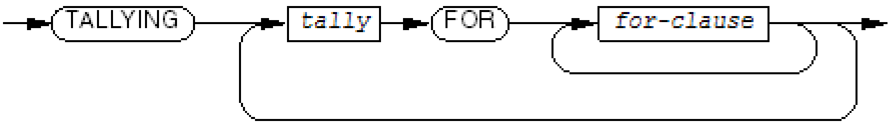
VST163.vsd

INSPECT TALLYING



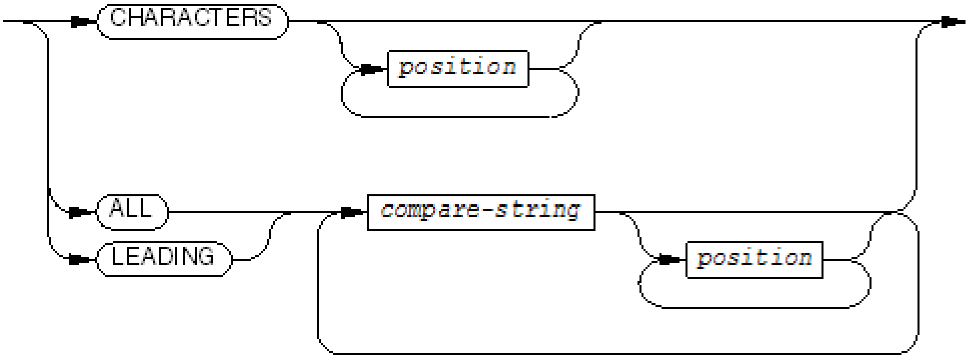
VST164.vsd

tallying-phrase



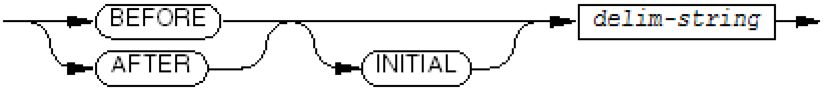
VST165.vsd

for-clause



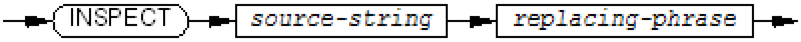
VST166.vsd

position



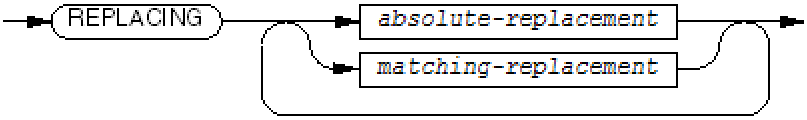
VST167.vsd

INSPECT REPLACING



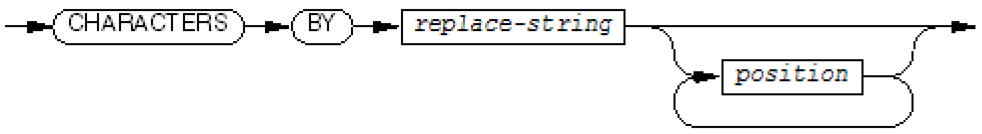
VST168.vsd

replacing-phrase



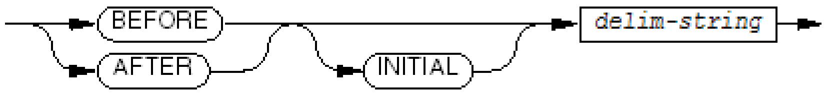
VST169.vsd

absolute-replacement



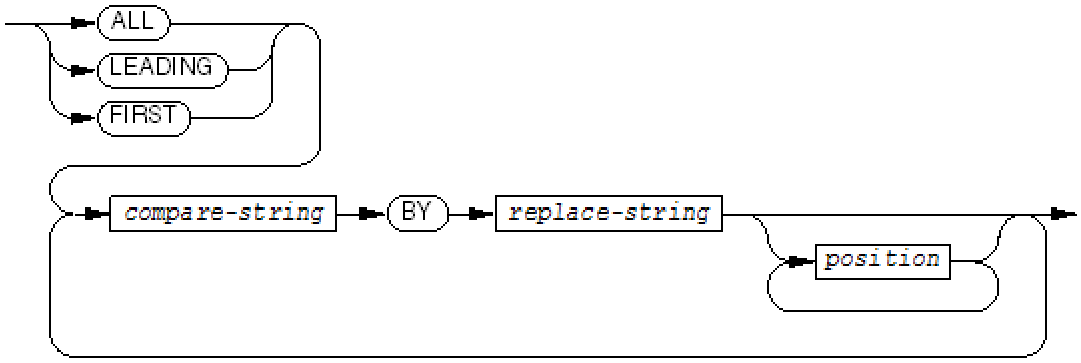
VST170.vsd

position



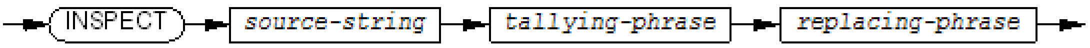
VST167.vsd

matching-replacement



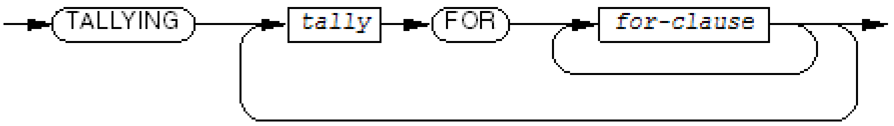
VST172.vsd

INSPECT TALLYING REPLACING



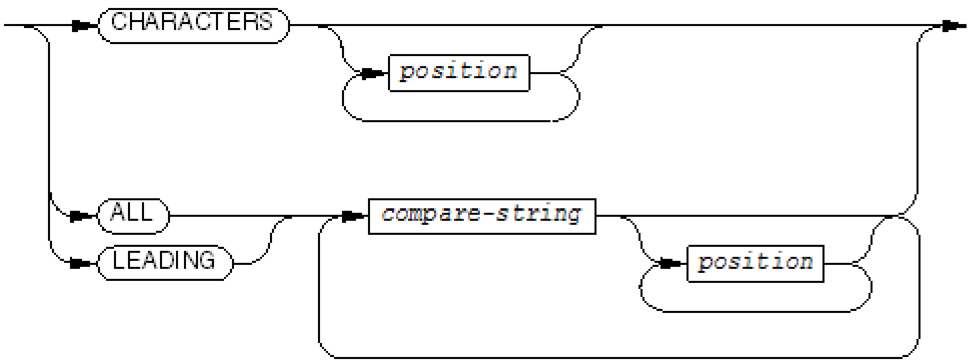
VST174.vsd

tallying-phrase



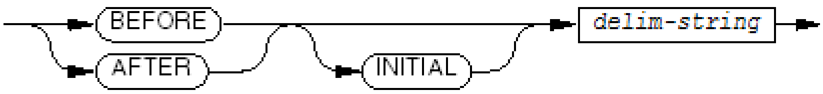
VST165.vsd

for-clause



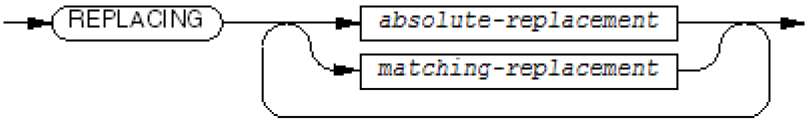
VST166.vsd

position



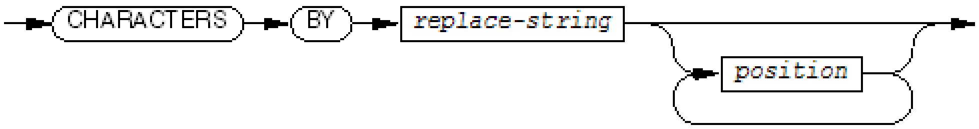
VST167.vsd

replacing-phrase



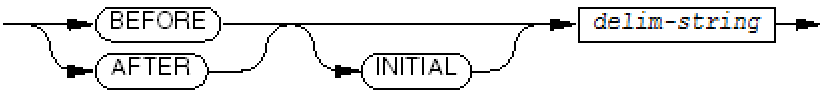
VST169.vsd

absolute-replacement



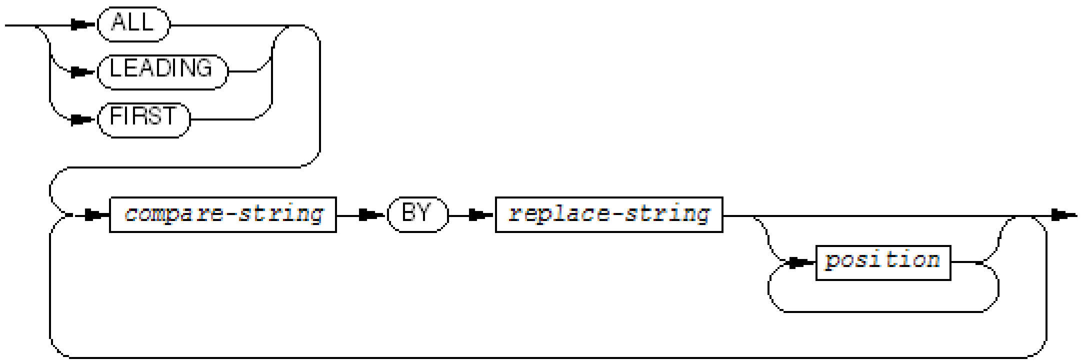
VST170.vsd

position



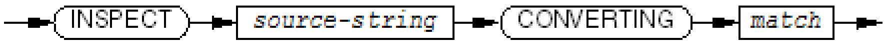
VST167.vsd

matching-replacement



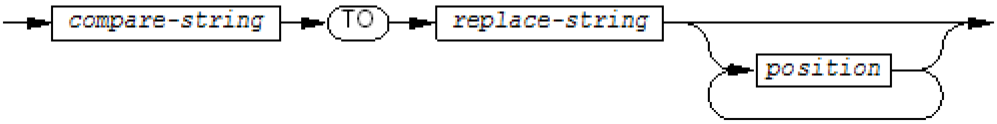
VST172.vsd

INSPECT CONVERTING



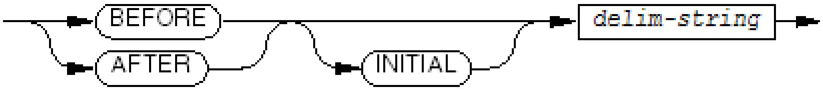
VST175.vsd

match



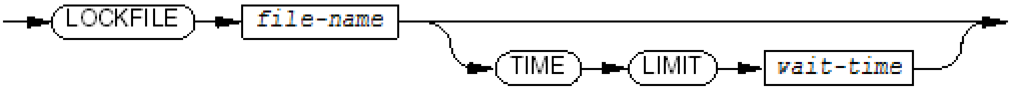
VST173.vsd

position



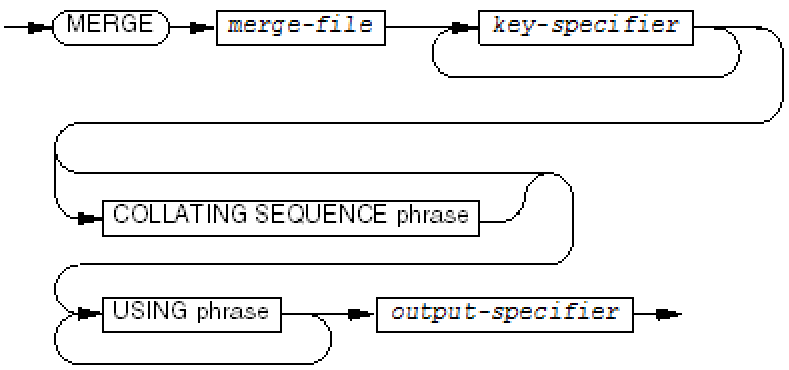
VST167.vsd

LOCKFILE



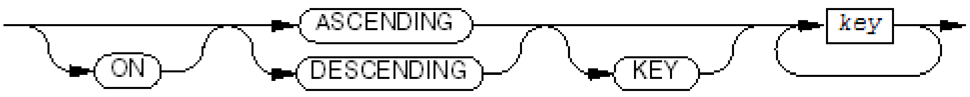
VST176.vsd

MERGE



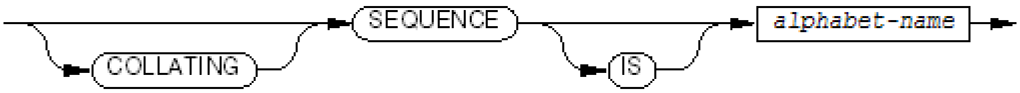
VST177.vsd

key-specifier



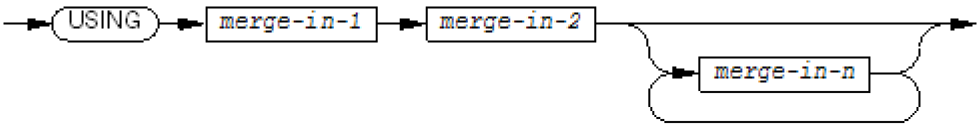
VST178.vsd

COLLATING SEQUENCE phrase



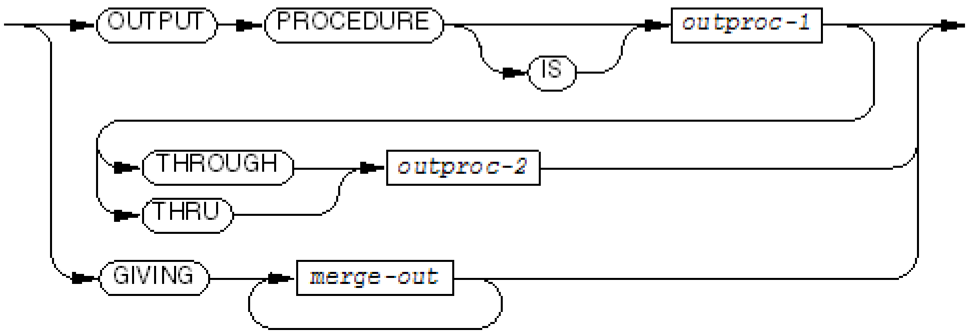
VST179.vsd

USING phrase



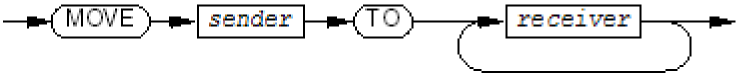
VST180.vsd

output-specifier



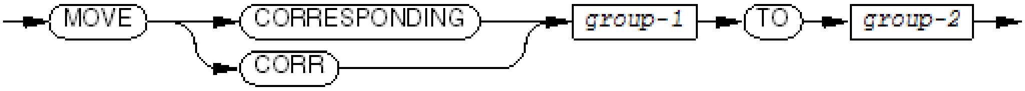
VST181.vsd

MOVE TO



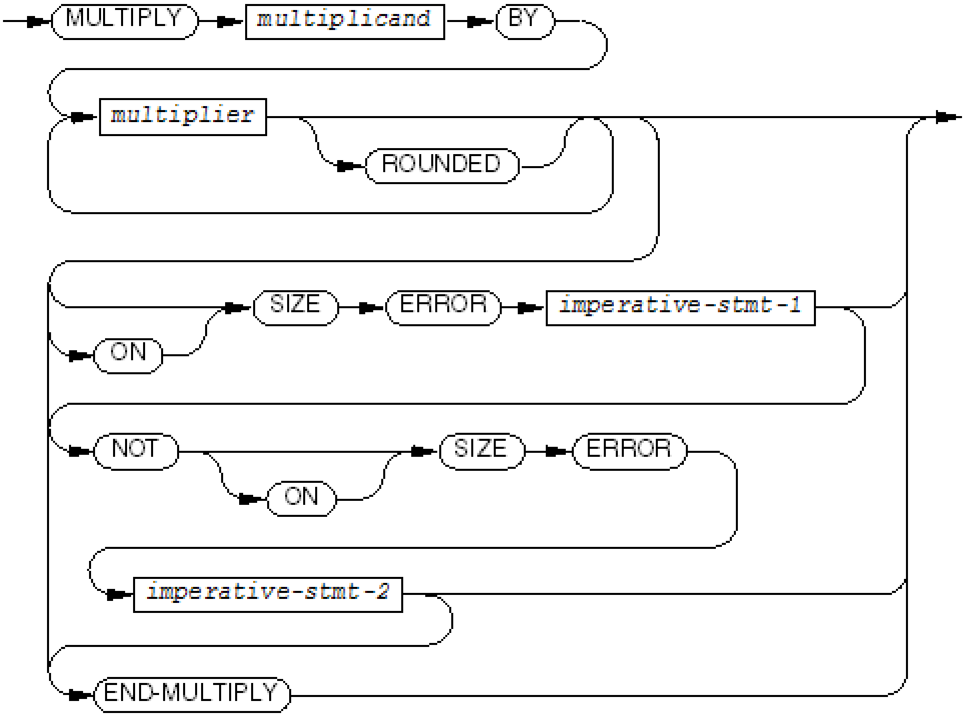
VST182.vsd

MOVE CORRESPONDING



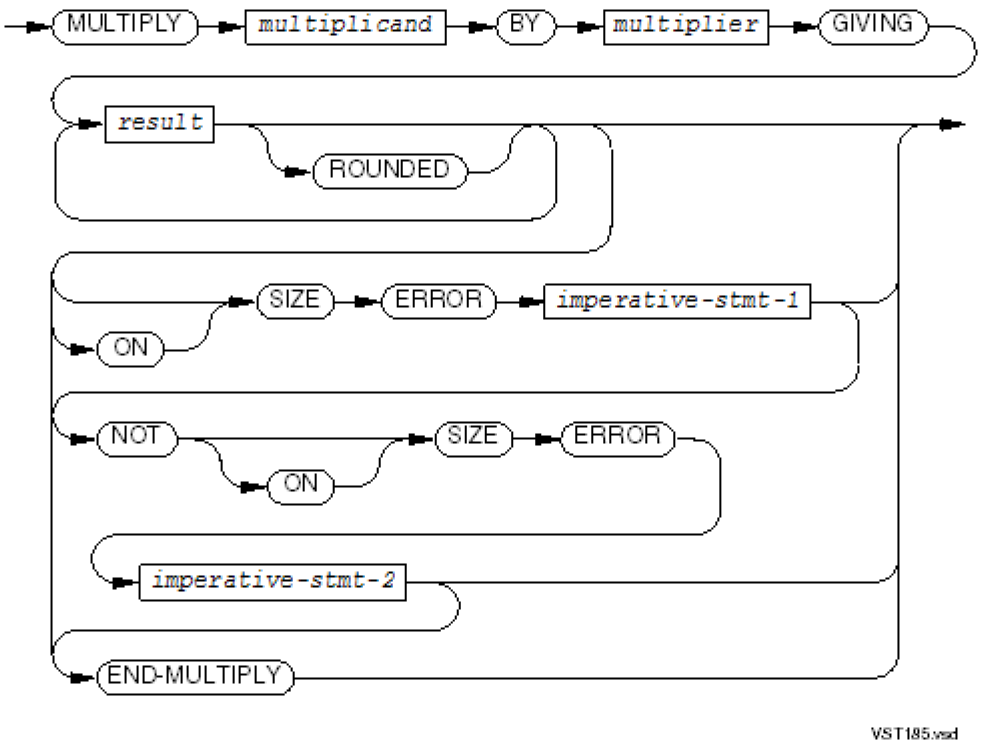
VST183.vsd

MULTIPLY BY

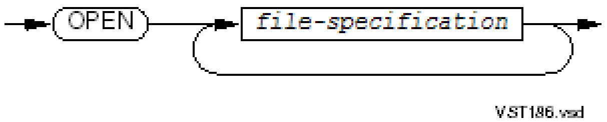


VST184.vsd

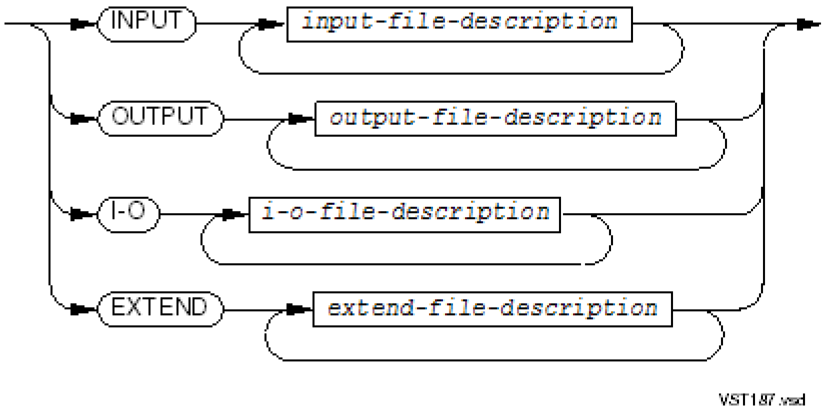
MULTIPLY GIVING



OPEN

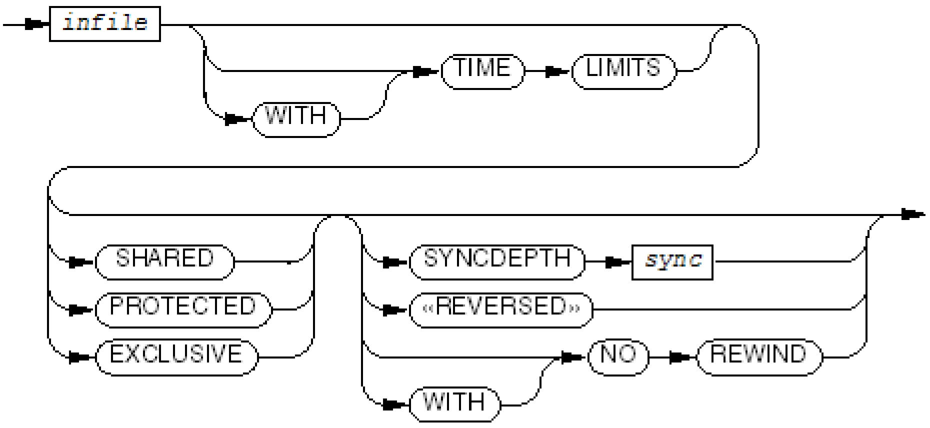


file-specification



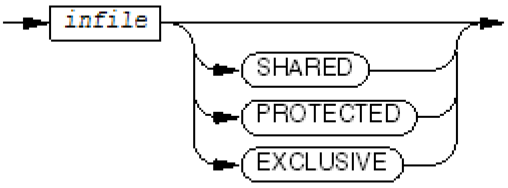
input-file-description

For a sequential, relative, indexed, or queue file:



VST188.vsd

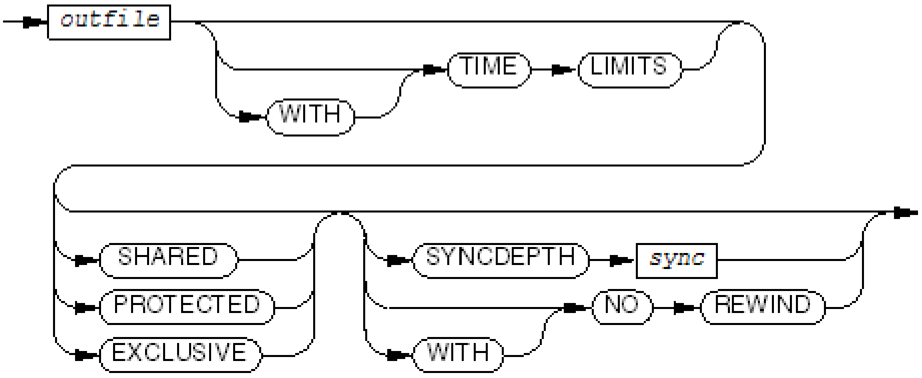
For a line sequential file:



VST627.vsd

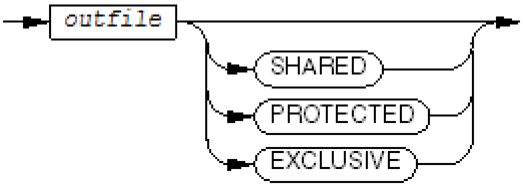
output-file-description

For a sequential, relative, indexed, or queue file:



VST189.vsd

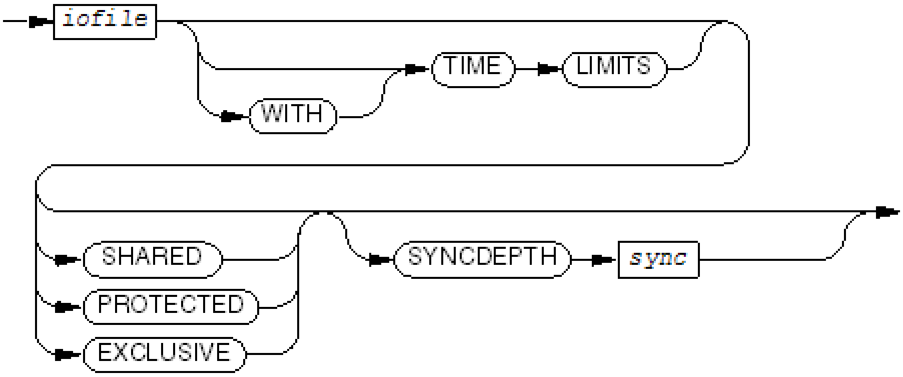
For a line sequential file:



VST628.vsd

i-o-file-description

For a sequential, relative, indexed, or queue file:

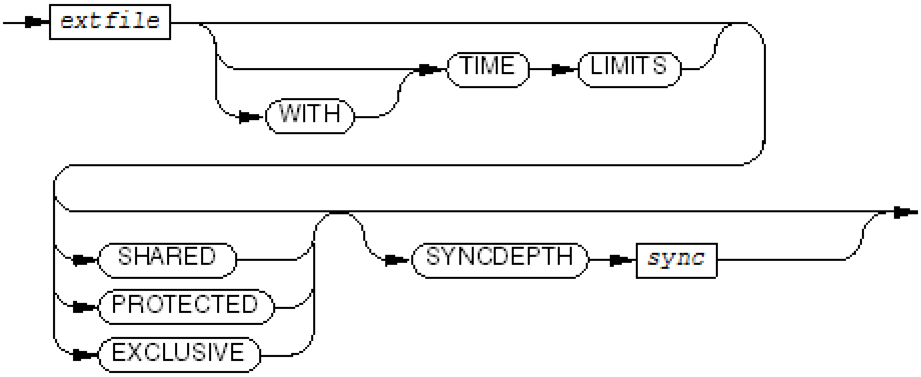


VST190.vsd

I-O mode is not supported for line sequential files.

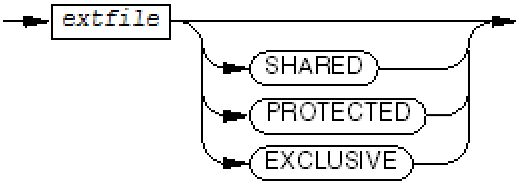
extend-file-description

For a sequential, relative, indexed, or queue file:



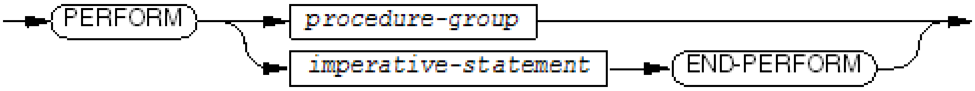
VST191.vsd

For a line sequential file:



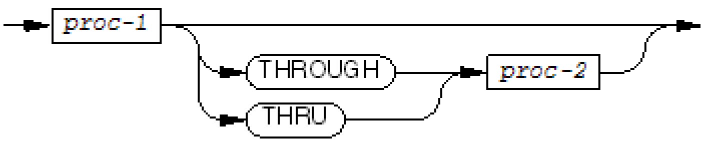
VST629.vsd

Unconditional PERFORM



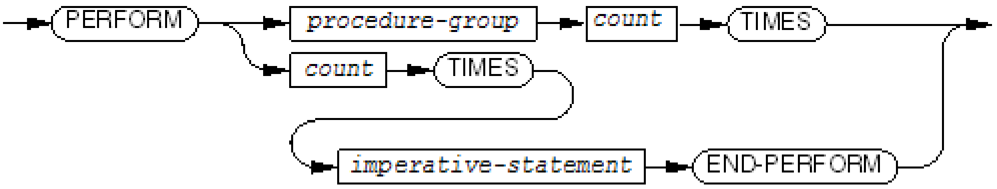
VST192.vsd

procedure-group



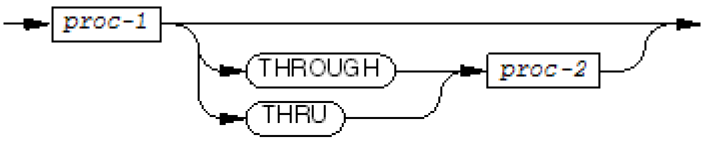
VST199.vsd

PERFORM TIMES



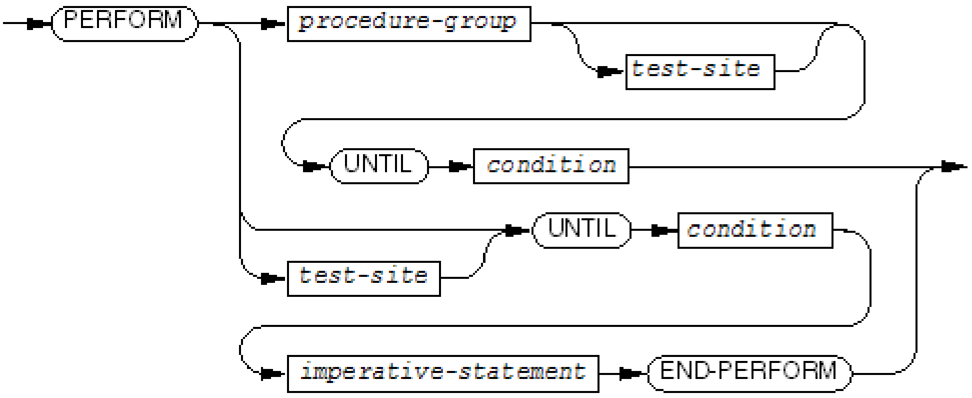
VST194.vsd

procedure-group



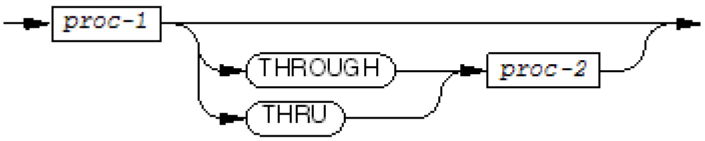
VST199.vsd

PERFORM UNTIL



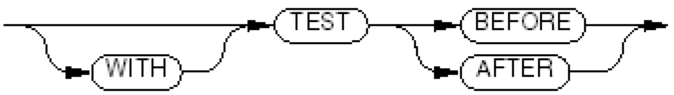
VST196.vsd

procedure-group



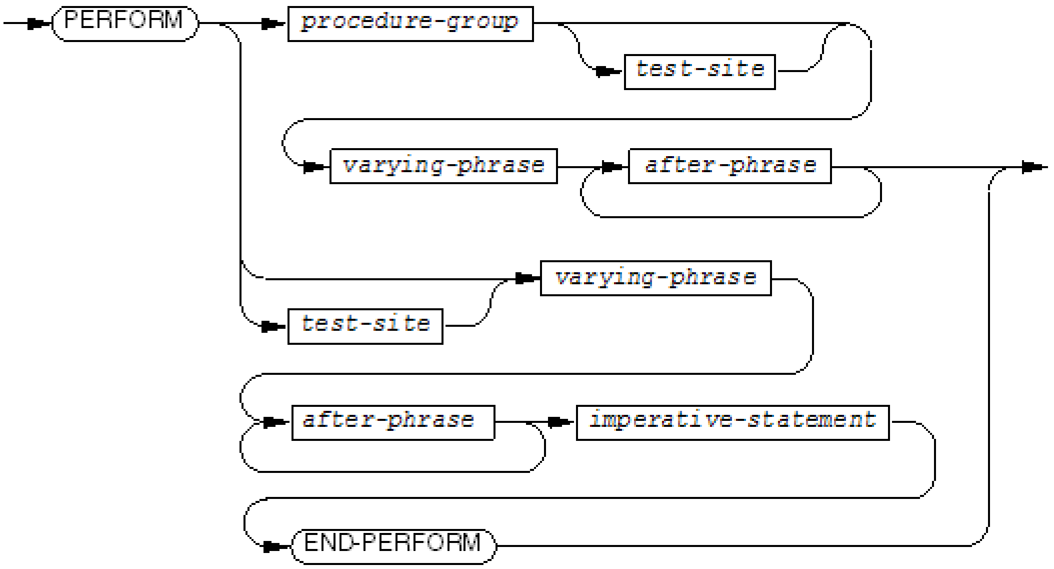
VST199.vsd

test-site



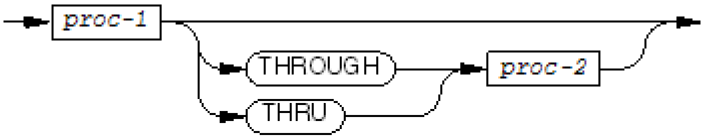
VST197.vsd

PERFORM VARYING



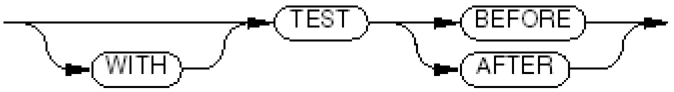
VST198.vsd

procedure-group



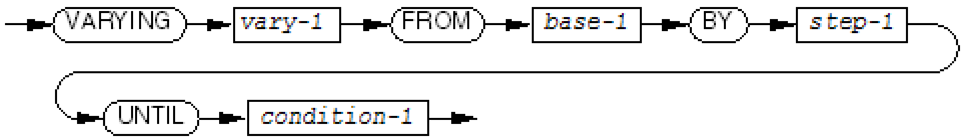
VST199.vsd

test-site



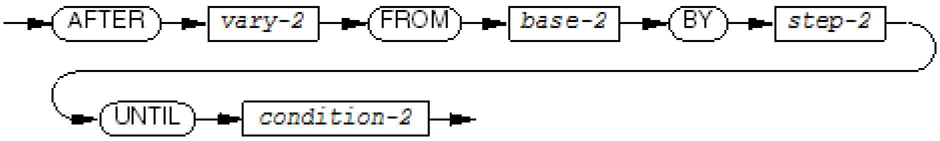
VST197.vsd

varying-phrase



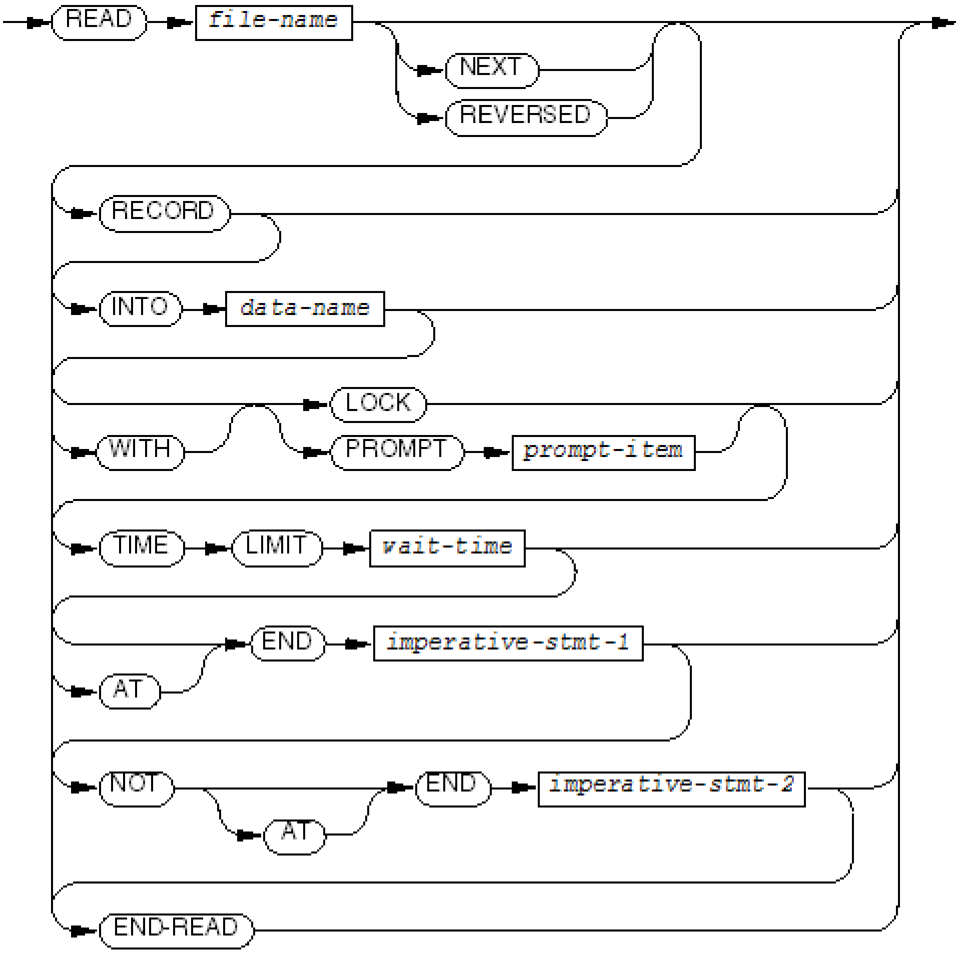
VST199.vsd

after-phrase



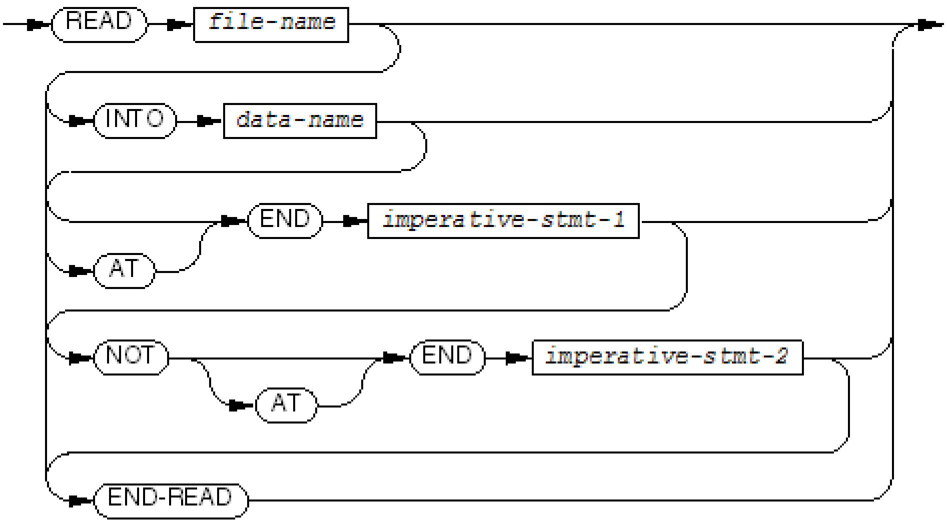
VST200.vsd

READ for Sequential or Dynamic Access



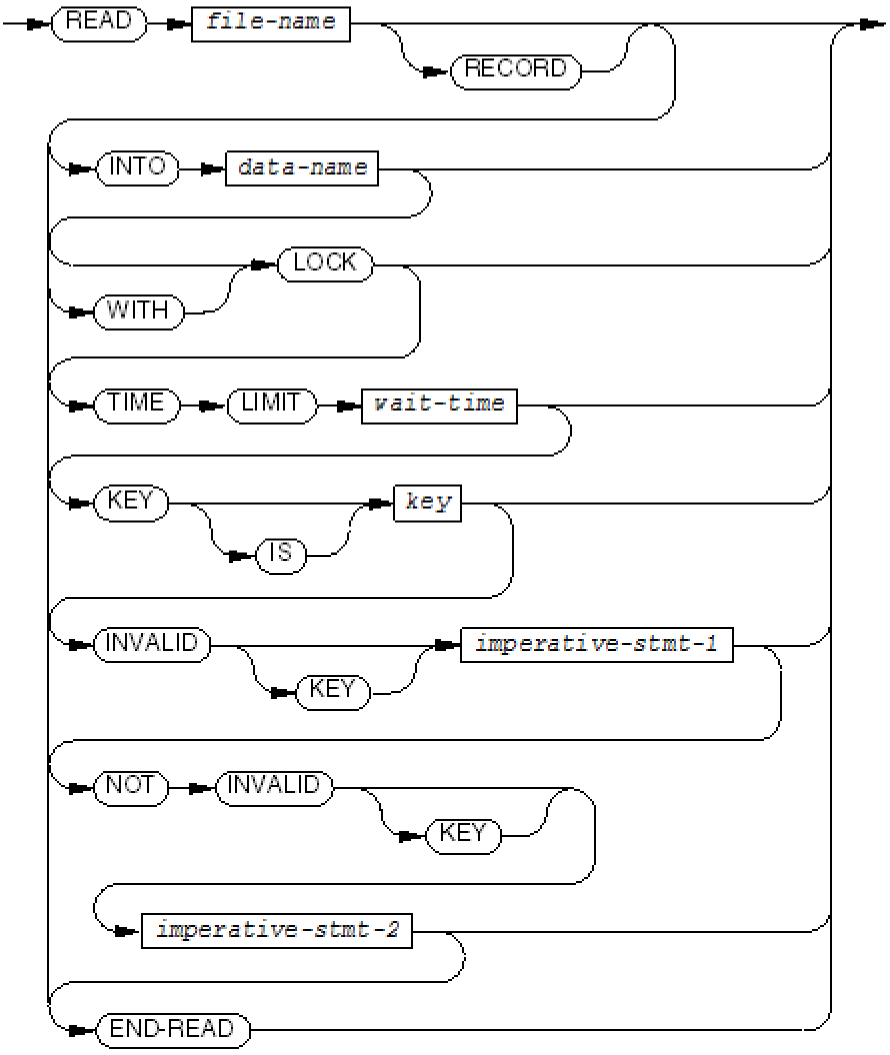
VST201.vsd

READ for Line Sequential Access



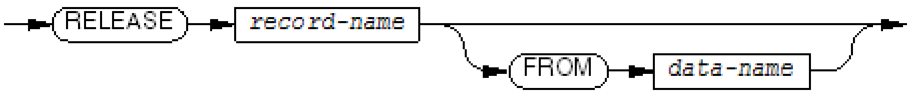
VST630.vsd

READ for Random or Dynamic Access



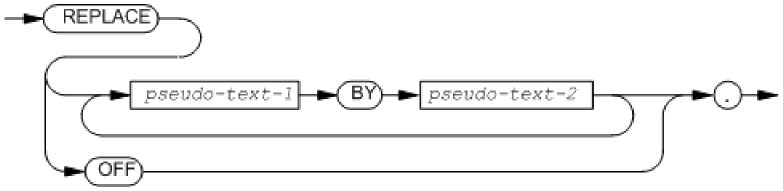
VST202.vsd

RELEASE

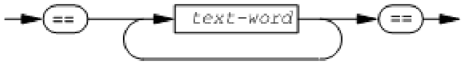


VST209.vsd

REPLACE

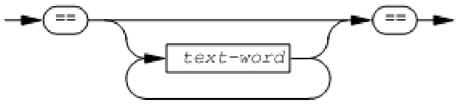


pseudo-text-1



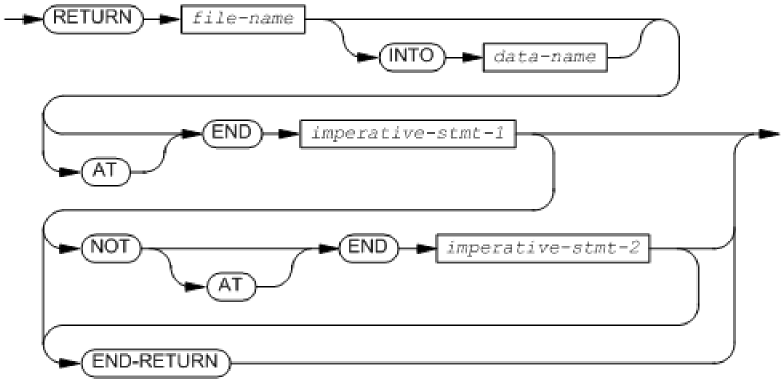
VST250.vsd

pseudo-text-2



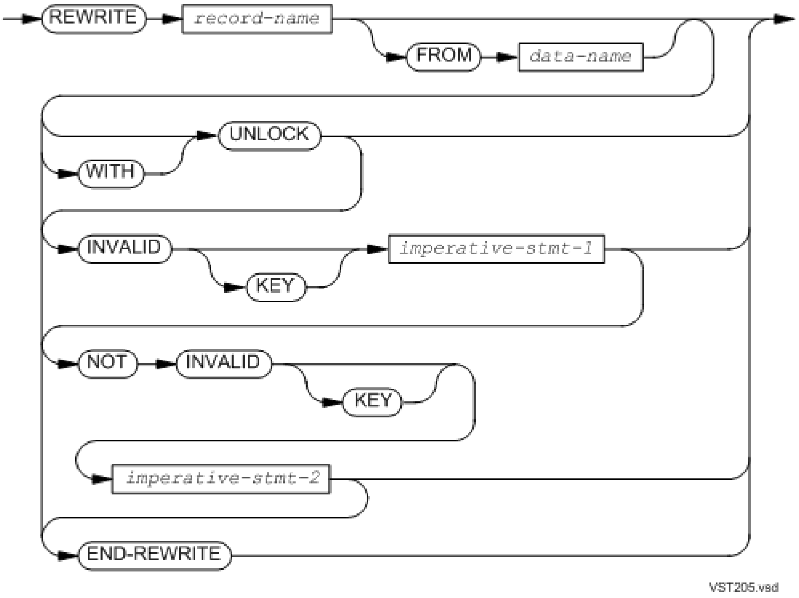
VST251.vsd

RETURN



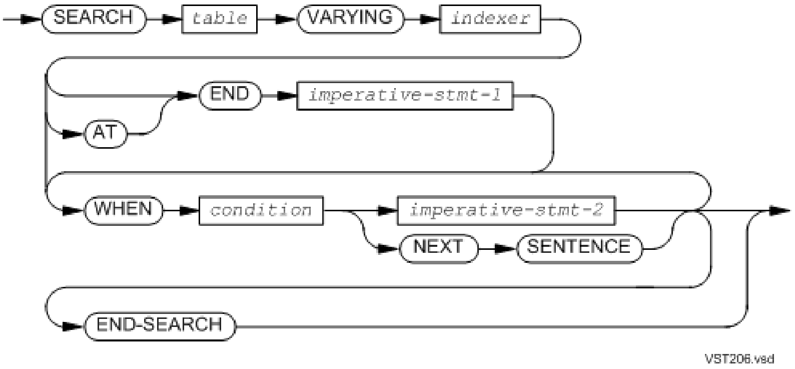
VST204.vsd

REWRITE for Sequential, Relative, Indexed, and Queue Files

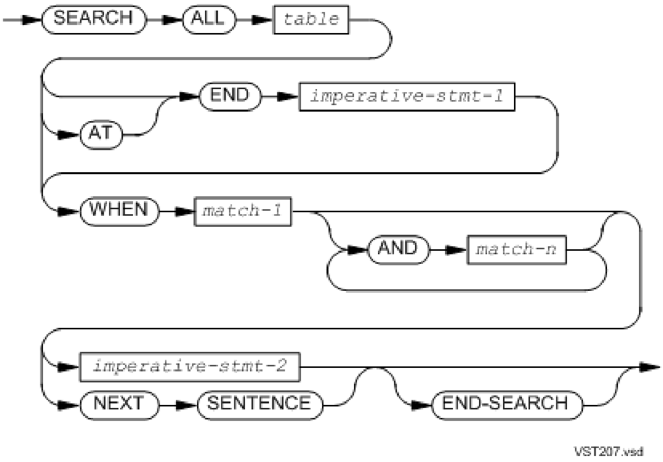


REWRITE is not supported for line sequential files.

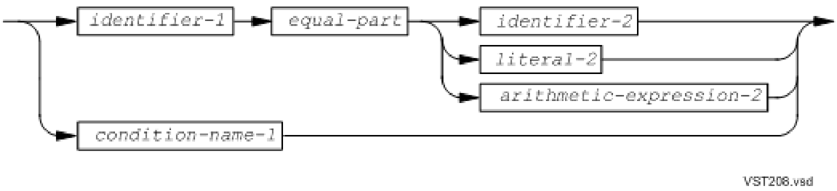
SEARCH VARYING



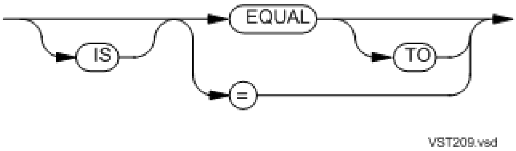
SEARCH ALL



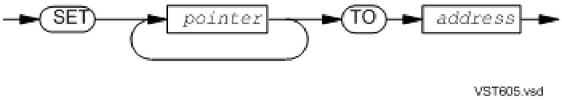
match-1, match-n



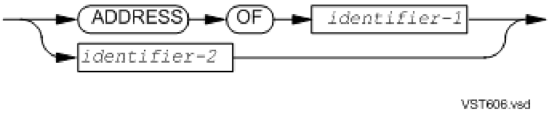
equal-part



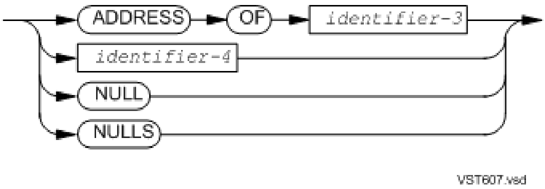
SET TO for Pointer Data Items



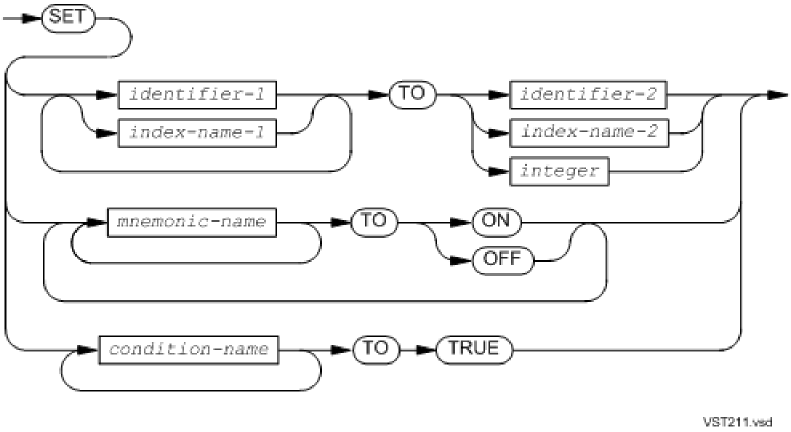
pointer



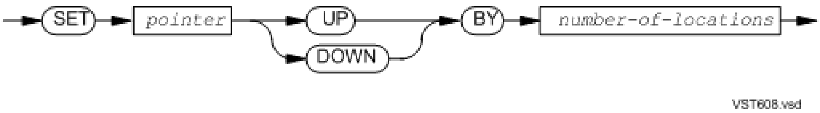
address



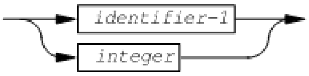
SET TO for Nonpointer Data Items



SET UP or SET DOWN for Pointer Data Items

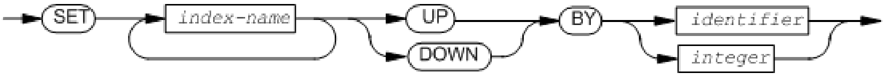


number-of-locations



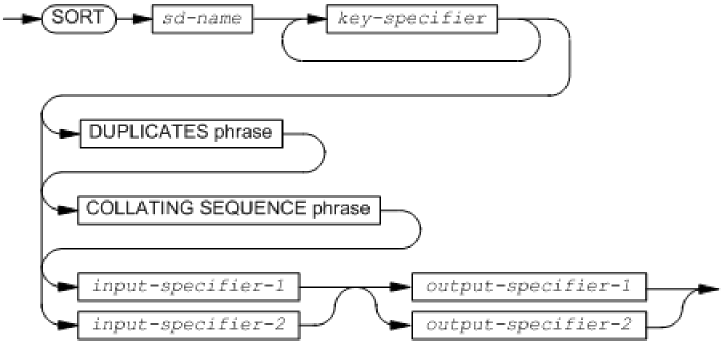
VST609.vsd

SET UP or SET DOWN for Nonpointer Data Items



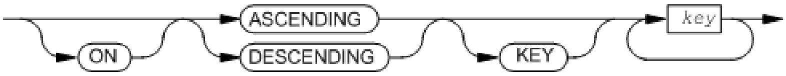
VST212.vsd

SORT



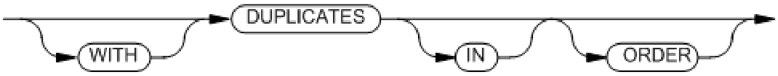
VST213.vsd

key-specifier



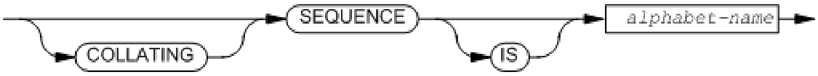
VST214.vsd

DUPLICATES phrase



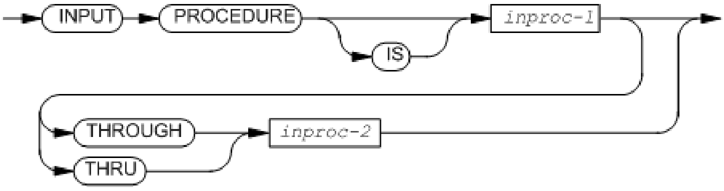
VST215.vsd

COLLATING SEQUENCE phrase



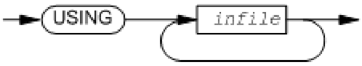
VST216.vsd

input-specifier-1



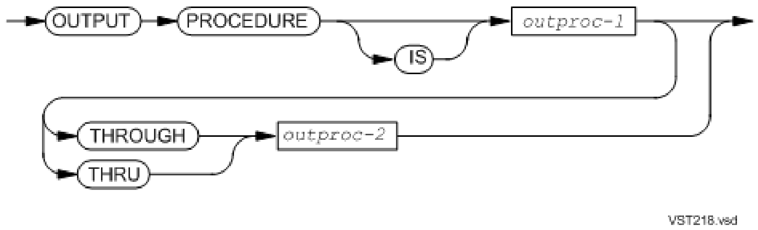
VST217.vsd

input-specifier-2

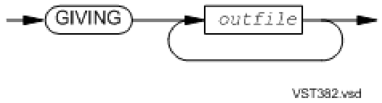


VST381.vsd

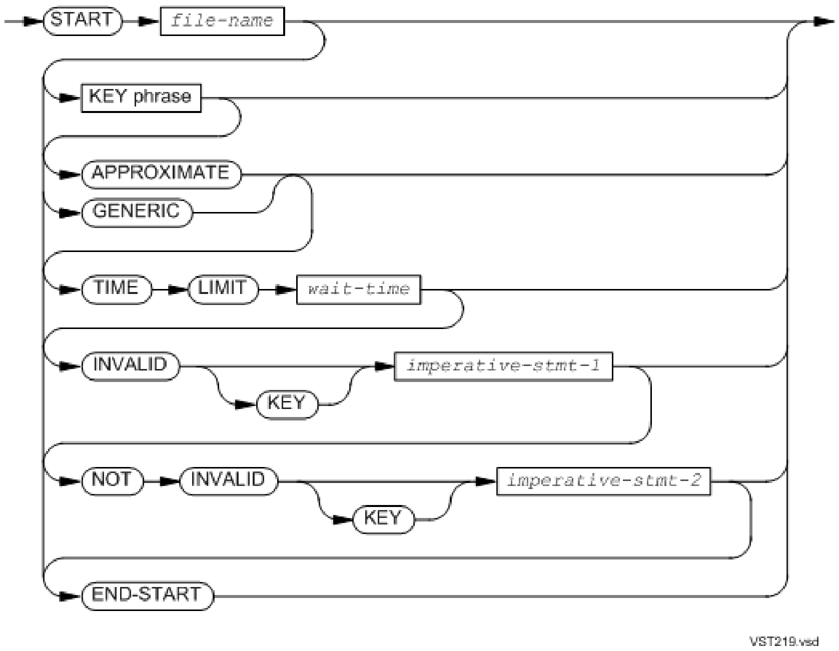
output-specifier-1



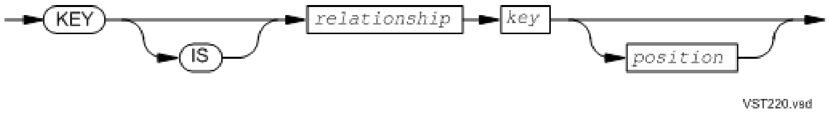
output-specifier-2



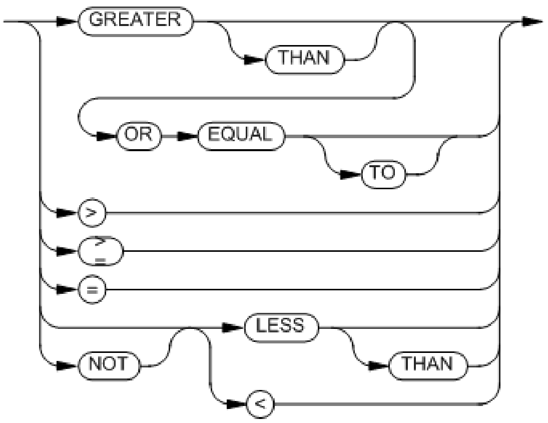
START



KEY phrase

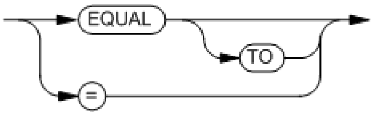


relationship



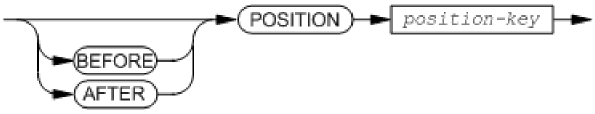
VST221.vsd

If *position* is present, *relationship* is limited to:



VST223.vsd

position

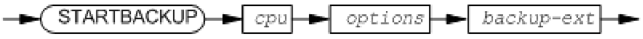


VST222.vsd

NOTE: BEFORE is available only for the ECOBOL compiler.

STARTBACKUP

NOTE: Do not use this directive in the OSS environment.

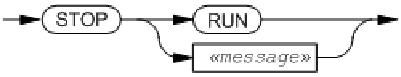


VST224.vsd

options

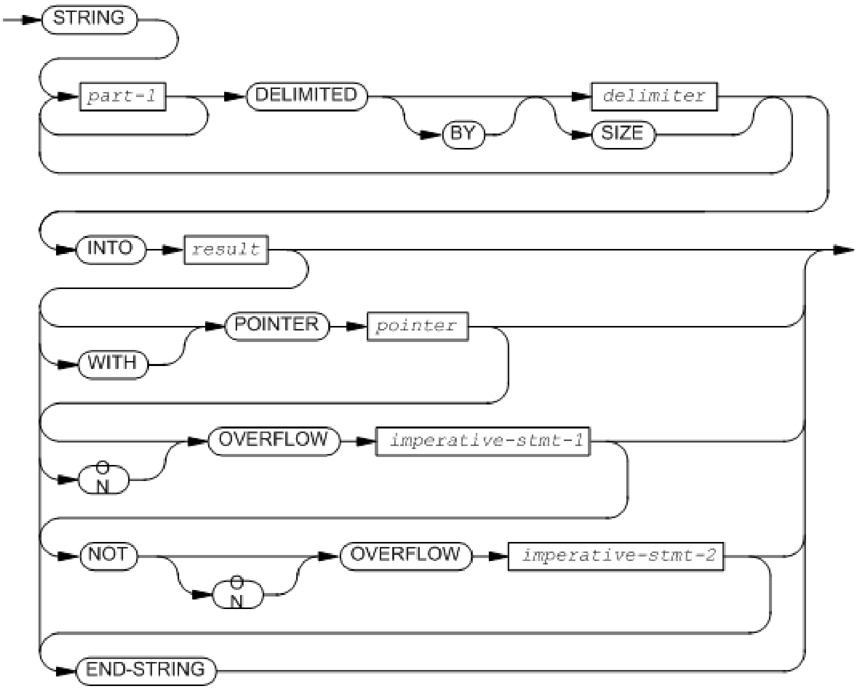
0	The fault-tolerant facility is to read and process system messages, and the primary process is to terminate abnormally if a trap condition occurs.
1	Same as option 0 except that if the primary process stops, the backup process takes over processing of the application.
2	Same as option 1 except that if the primary process encounters a trap condition, it enters the DEBUG procedure instead of being terminated abnormally.
3	The primary process, rather than the fault-tolerant facility, reads the $RECEIVE file and takes appropriate action for system messages. In addition, if the primary process encounters a trap condition, it enters the DEBUG procedure. When this option is in force and the backup process fails, the primary process must re-execute the STARTBACKUP statement to reestablish the backup.

STOP



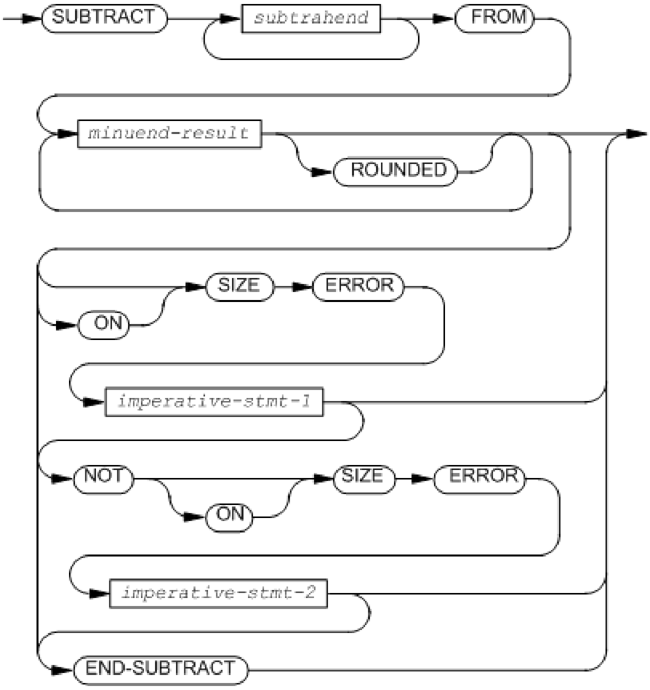
VST225.vsd

STRING



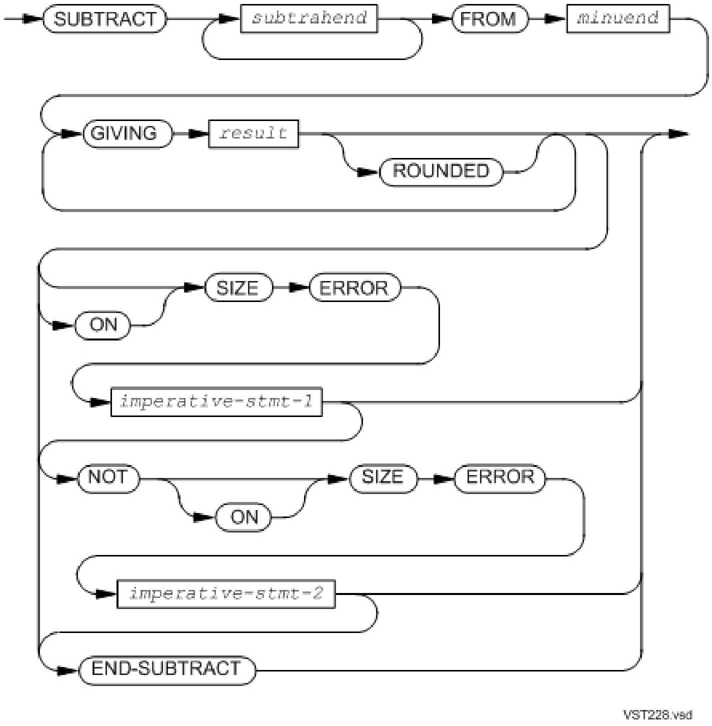
VST226.vsd

SUBTRACT FROM



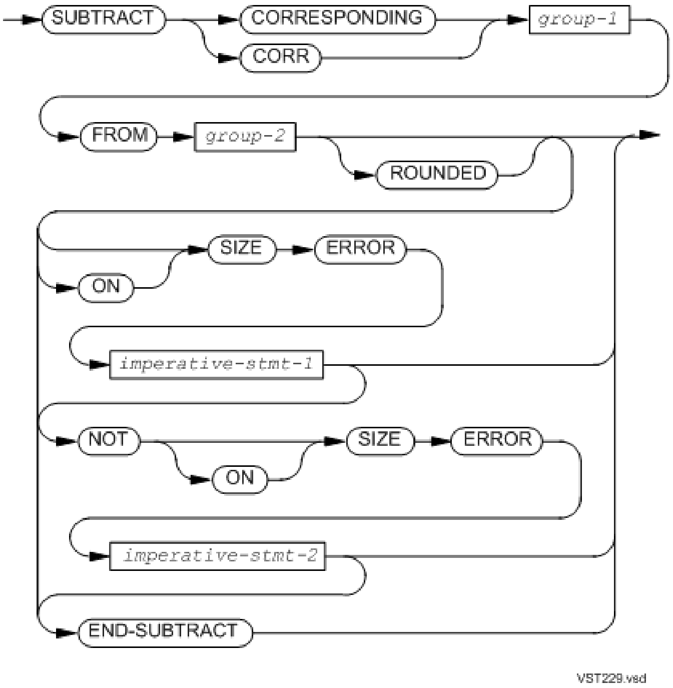
VST227.vsd

SUBTRACT GIVING

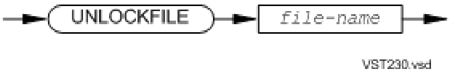


SUBTRACT CORRESPONDING

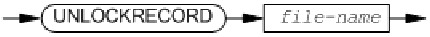
△ **CAUTION:** SUBTRACT CORRESPONDING is not recommended, because minor changes to one data structure can change the correspondence between its elements and those of the other data structure, and this is difficult to detect.



UNLOCKFILE

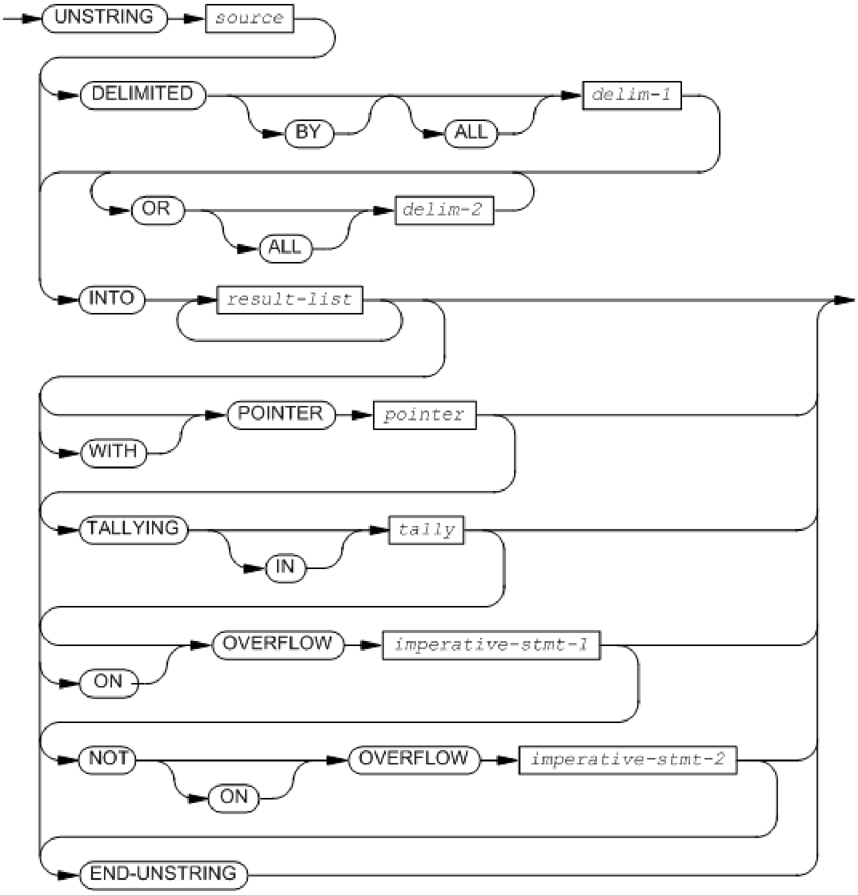


UNLOCKRECORD



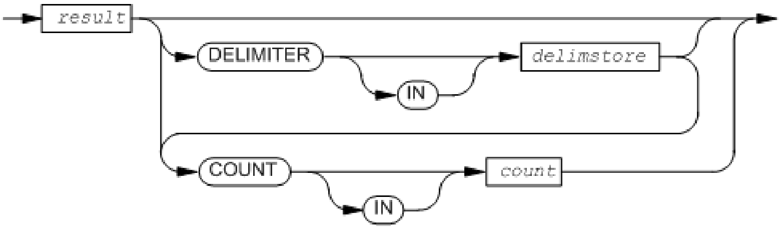
VST231.vsd

UNSTRING



VST232.vsd

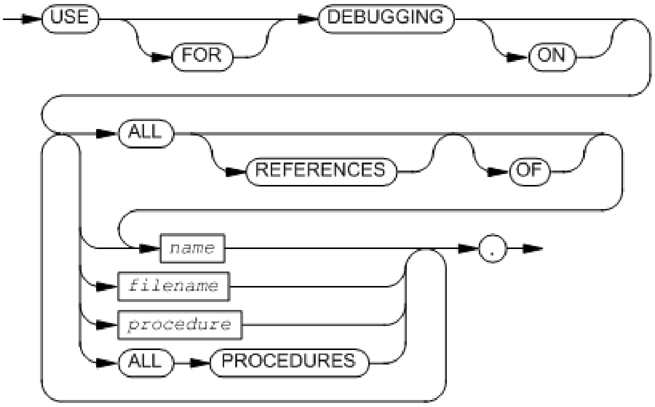
result-list



VST233.vsd

USE DEBUGGING

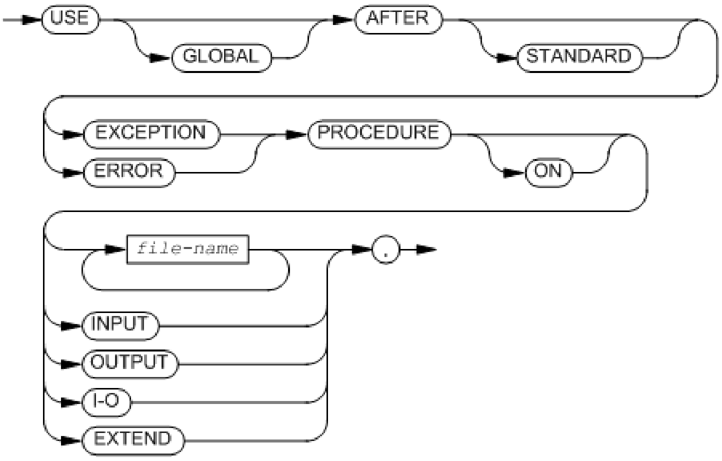
NOTE: The 1985 COBOL standard classifies USE DEBUGGING as **obsolete**. The compiler does not recognize it.



VST234.vsd

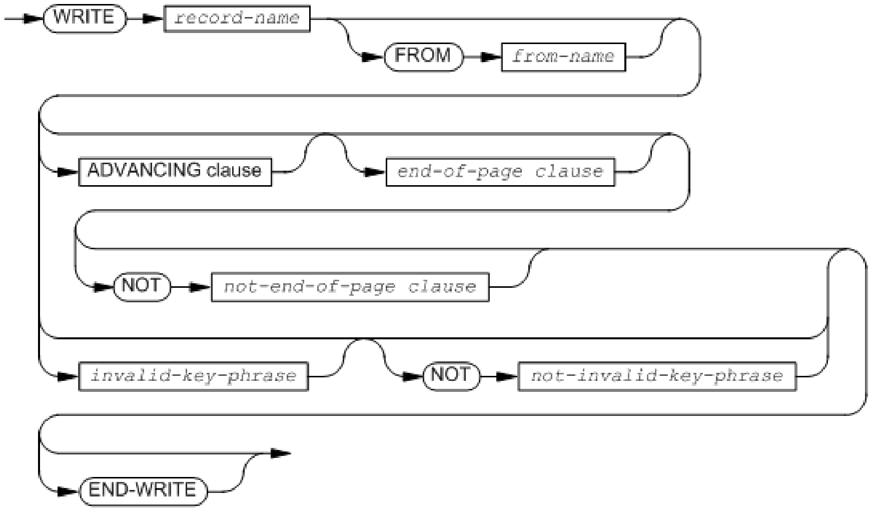
USE AFTER EXCEPTION

NOTE: Do not use GLOBAL in a USE AFTER EXCEPTION statement in the declaratives-portion.



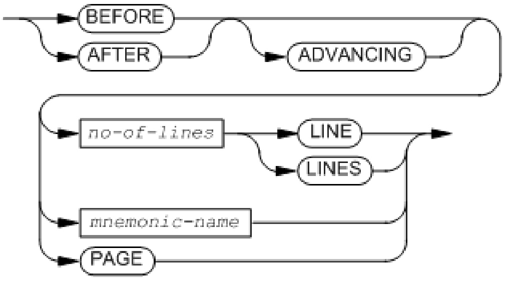
VST235.vsd

WRITE for Sequential Files



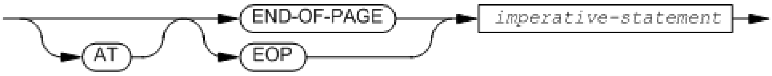
VST236.vsd

ADVANCING clause



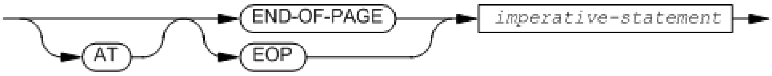
VST237.vsd

end-of-page clause



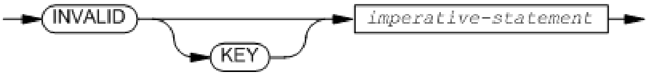
VST238.vsd

not-end-of-page clause



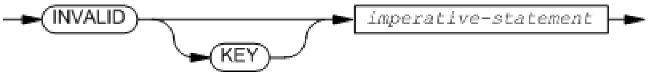
VST238.vsd

invalid-key-phrase



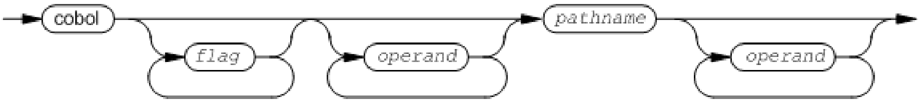
VST239.vsd

not-invalid-key-phrase



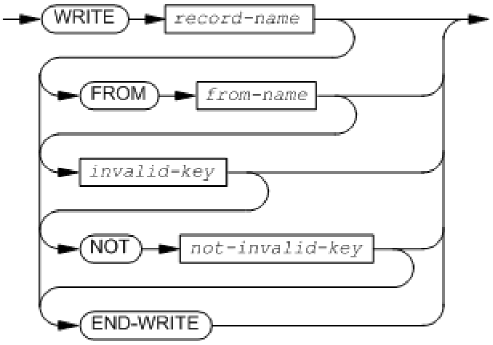
VST239.vsd

WRITE for Line Sequential Files



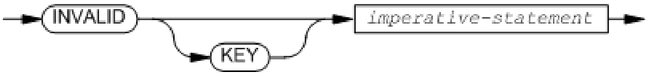
VST621.vsd

WRITE for Relative, Indexed, and Queue Files



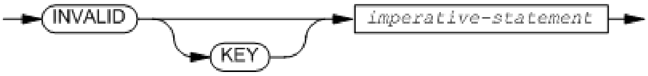
VST240.vsd

invalid-key



VST239.vsd

not-invalid-key

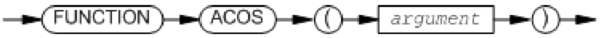


VST239.vsd

45 Intrinsic Function Calls

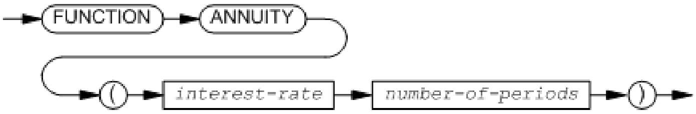
- ACOS
- ANNUITY
- ASIN
- ATAN
- CHAR
- COS
- CURRENT-DATE
- DATE-OF-INTEGER
- DAY-OF-INTEGER
- FACTORIAL
- INTEGER
- INTEGER-OF-DATE
- INTEGER-OF-DAY
- INTEGER-PART
- LENGTH
- LOG
- LOG10
- LOWER-CASE
- MAX
- MEAN
- MEDIAN
- MIDRANGE
- MIN
- MOD
- NUMVAL
- NUMVAL-C
- ORD
- ORD-MAX
- ORD-MIN
- PRESENT-VALUE
- RANDOM
- RANGE
- REM
- REVERSE
- SIN
- SQRT
- STANDARD-DEVIATION
- SUM
- TAN
- UPPER-CASE
- VARIANCE
- WHEN-COMPILED

ACOS



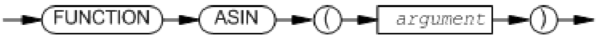
VST421.vsd

ANNUITY



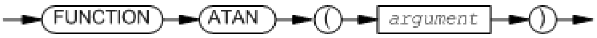
VST422.vsd

ASIN



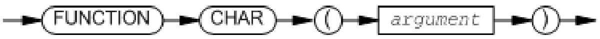
VST423.vsd

ATAN



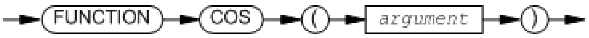
VST424.vsd

CHAR



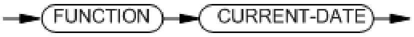
VST425.vsd

COS



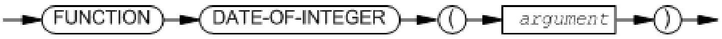
VST426.vsd

CURRENT-DATE



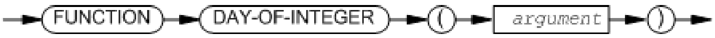
VST427.vsd

DATE-OF-INTEGER



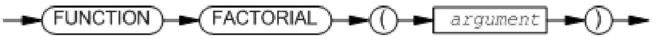
VST428.vsd

DAY-OF-INTEGER



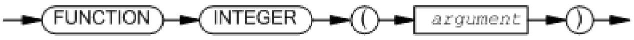
VST429.vsd

FACTORIAL



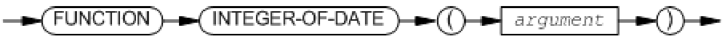
VST430.vsd

INTEGER



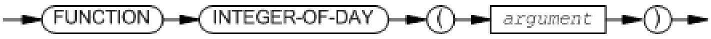
VST431.vsd

INTEGER-OF-DATE



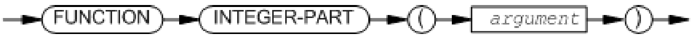
VST432.vsd

INTEGER-OF-DAY



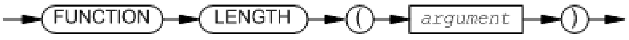
VST433.vsd

INTEGER-PART



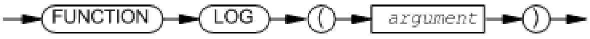
VST434.vsd

LENGTH



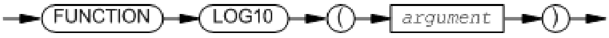
VST435.vsd

LOG



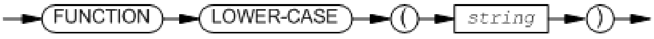
VST436.vsd

LOG10



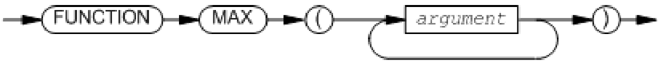
VST437.vsd

LOWER-CASE



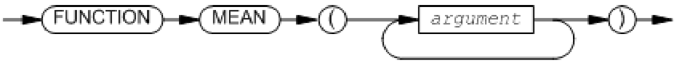
VST438.vsd

MAX



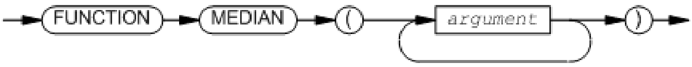
VST439.vsd

MEAN



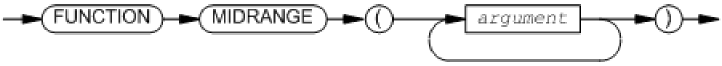
VST440.vsd

MEDIAN



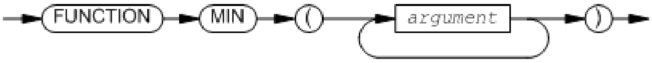
VST441.vsd

MIDRANGE



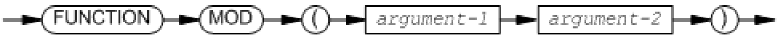
VST442.vsd

MIN



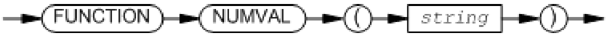
VST443.vsd

MOD



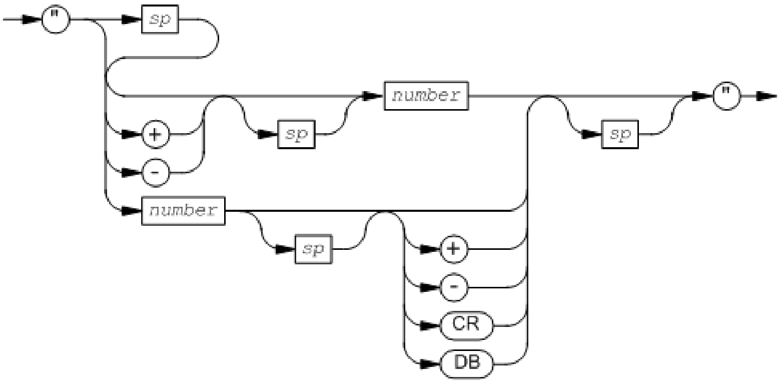
VST444.vsd

NUMVAL



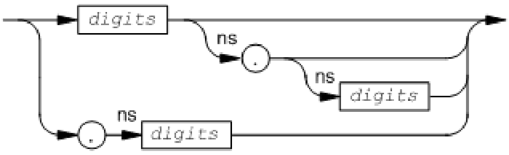
VST445.vsd

string



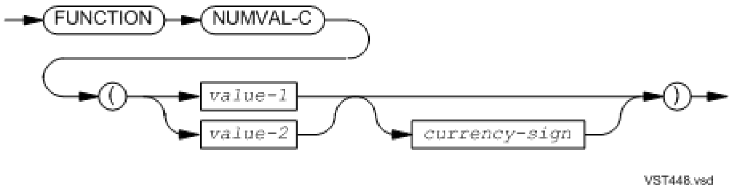
VST446.vsd

number

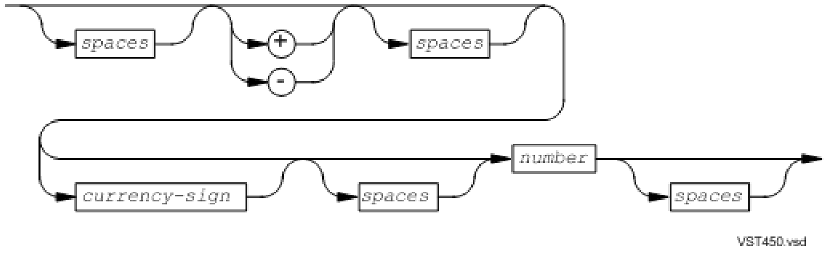


VST447.vsd

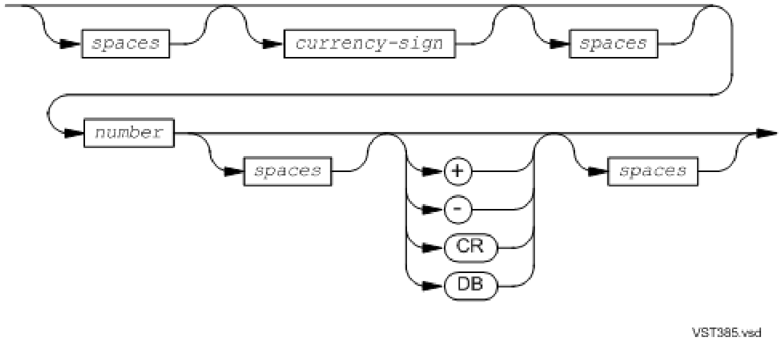
NUMVAL-C



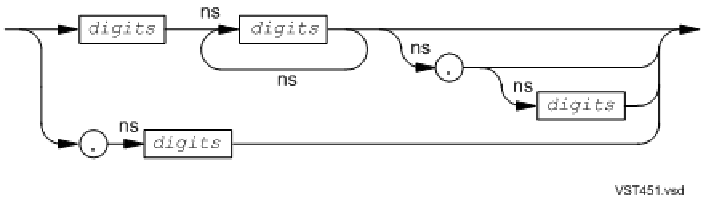
value-1



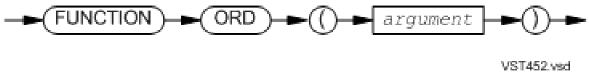
value-2



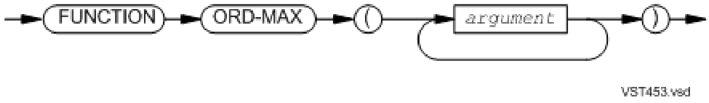
number



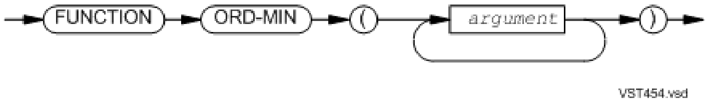
ORD



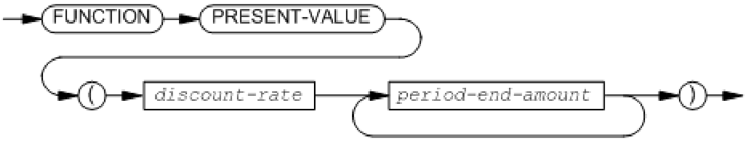
ORD-MAX



ORD-MIN

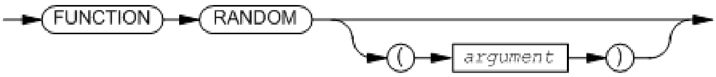


PRESENT-VALUE



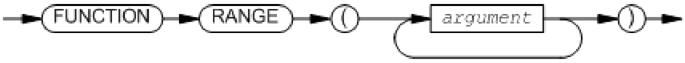
VST455.vsd

RANDOM



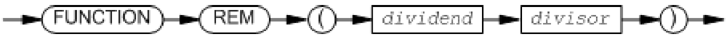
VST456.vsd

RANGE



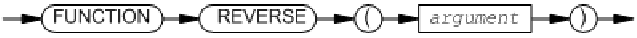
VST457.vsd

REM



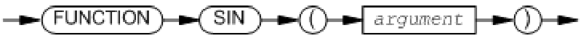
VST458.vsd

REVERSE



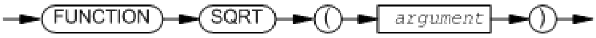
VST459.vsd

SIN



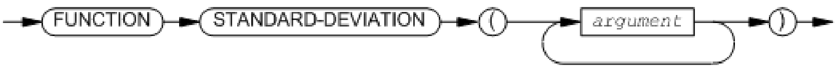
VST460.vsd

SQRT



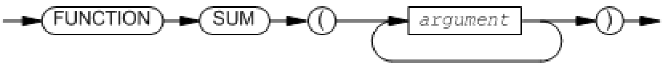
VST461.vsd

STANDARD-DEVIATION



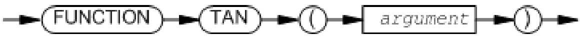
VST462.vsd

SUM



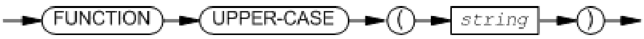
VST463.vsd

TAN



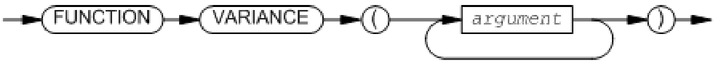
VST464.vsd

UPPER-CASE



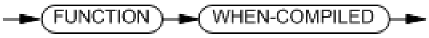
VST465.vsd

VARIANCE



VST466.vsd

WHEN-COMPILED



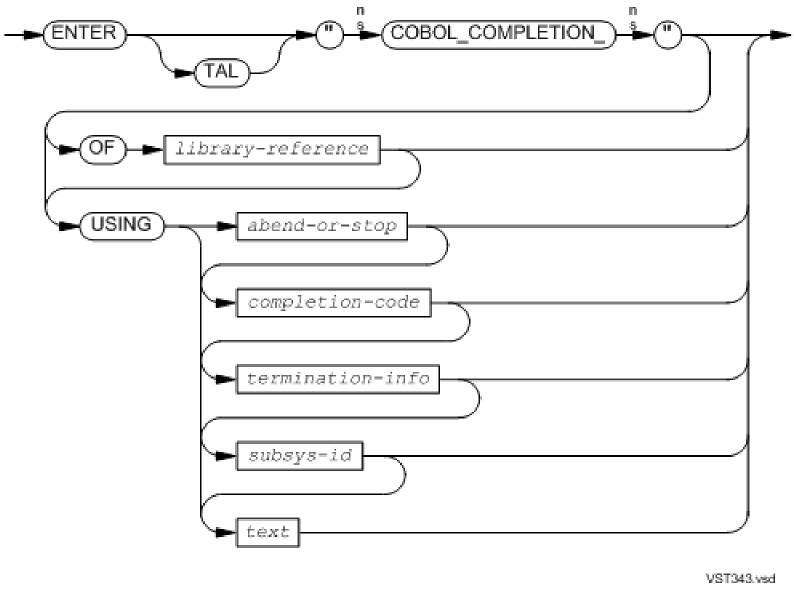
VST467.vsd

46 ZCOBDLL Routine Calls

If you omit an optional parameter when you call a ZCOBDLL routine, you must put the keyword OMITTED in its position if you specify subsequent parameters. Trailing OMITTEDs are not required. (The syntax diagrams do not reflect this, because it would make them much harder to read.)

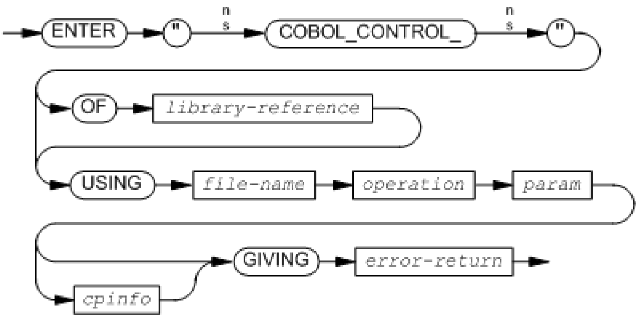
- COBOL_COMPLETION_
- COBOL_CONTROL_
- COBOL_GETENV_
- COBOL_PUTENV_
- COBOL_RETURN_SORT_ERRORS_
- COBOL_REWIND_SEQUENTIAL_
- COBOL_SET_SORT_PARAM_TEXT_
- COBOL_SET_SORT_PARAM_VALUE_
- COBOL_SET_MAX_RECORD_
- COBOL_SETMODE_
- COBOL_SPECIAL_OPEN_

COBOL_COMPLETION_



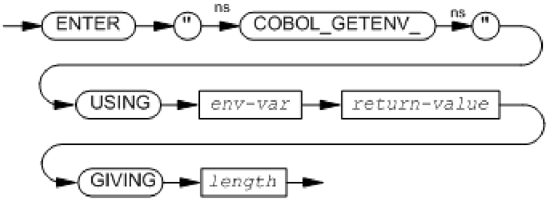
COBOL_CONTROL_

CAUTION: The HP COBOL run-time library does not attempt to validate calls to CONTROL. To avoid interfering with the operation of the HP COBOL run-time library, use COBOL_CONTROL_ only if absolutely necessary.



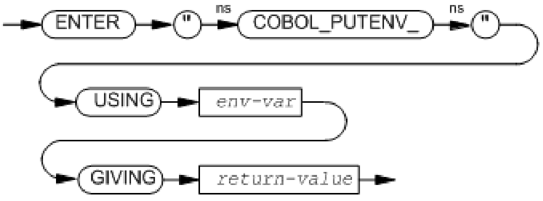
VST610.vsd

COBOL_GETENV_



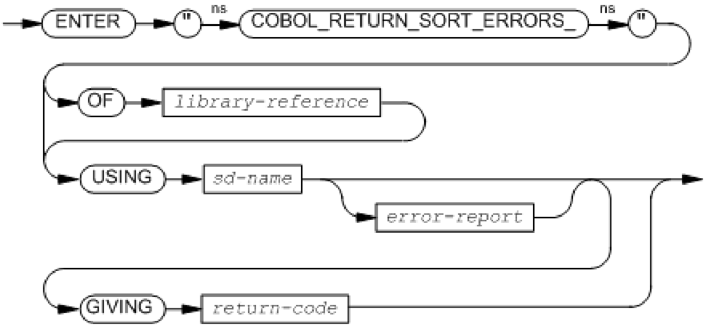
VST722.vsd

COBOL_PUTENV_



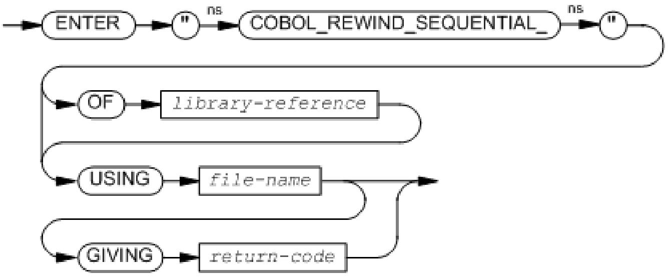
VST723.vsd

COBOL_RETURN_SORT_ERRORS_



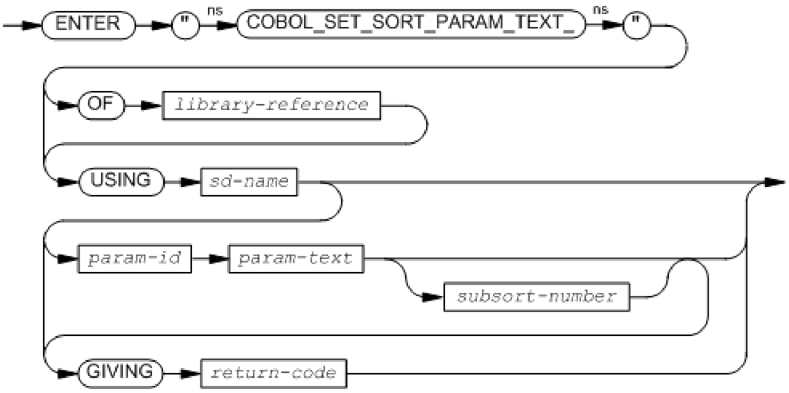
VST809.vsd

COBOL_REWIND_SEQUENTIAL_



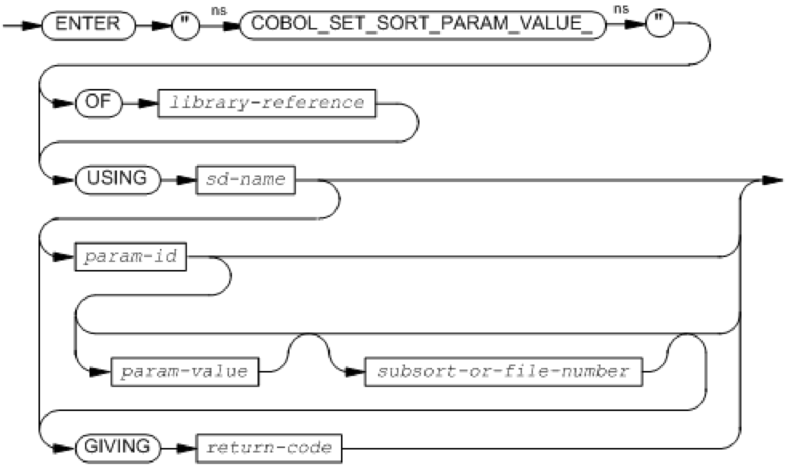
VST810.vsd

COBOL_SET_SORT_PARAM_TEXT_



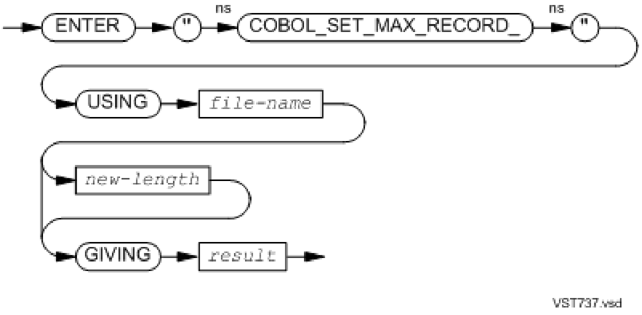
VST811.vsd

COBOL_SET_SORT_PARAM_VALUE_



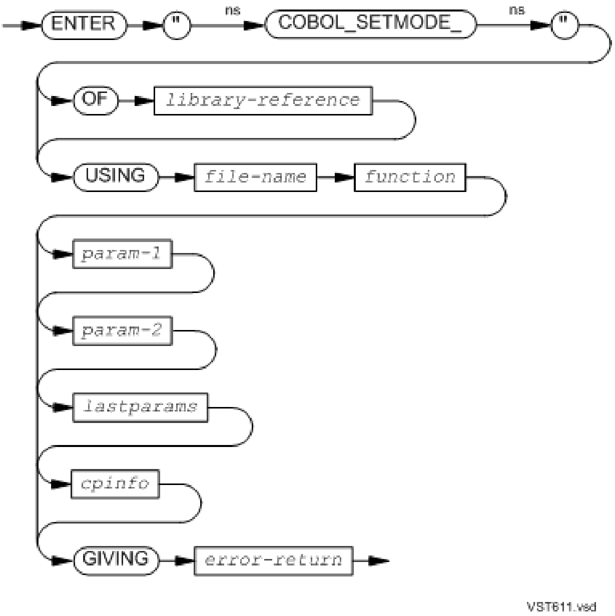
VST811.vsd

COBOL_SET_MAX_RECORD_



COBOL_SETMODE_

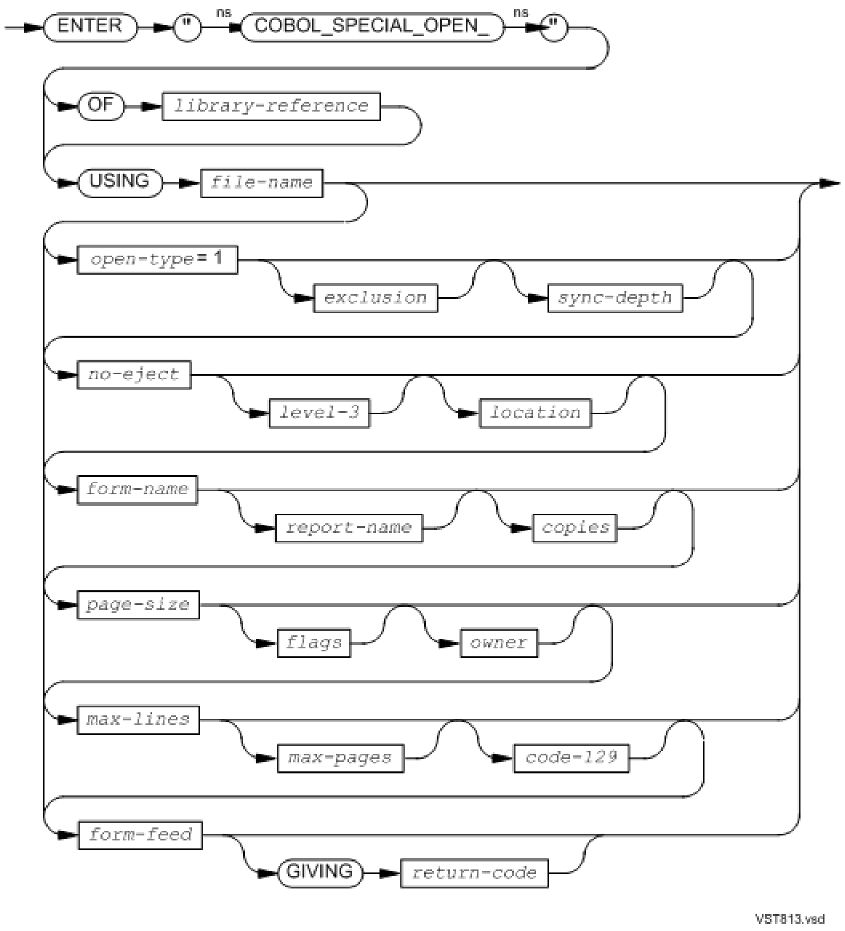
CAUTION: The HP COBOL run-time library does not attempt to validate calls to SETMODE. To avoid interfering with the operation of the HP COBOL run-time library, use COBOL_SETMODE_ only if absolutely necessary.



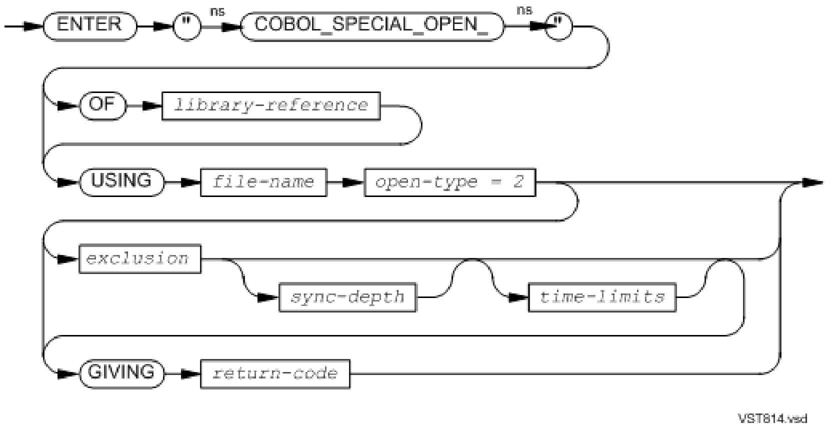
COBOL_SPECIAL_OPEN_

- For Spoolers and Printers
- For System Log Files
- For Partitioned Disk Files
- For Tape Files

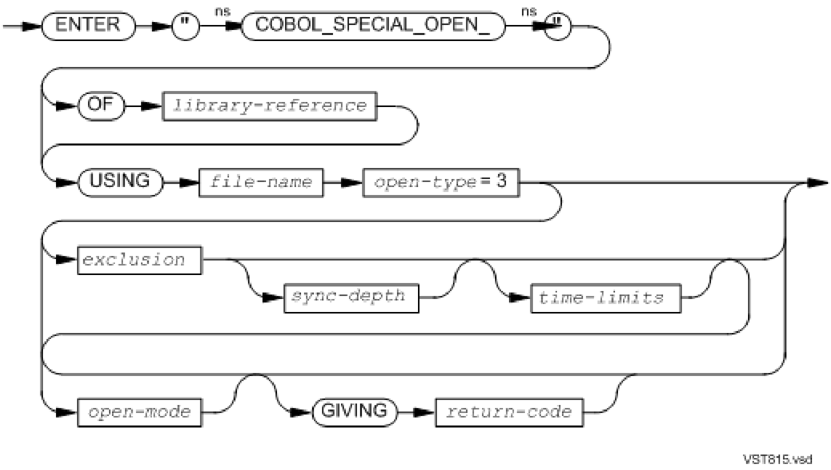
For Spoolers and Printers



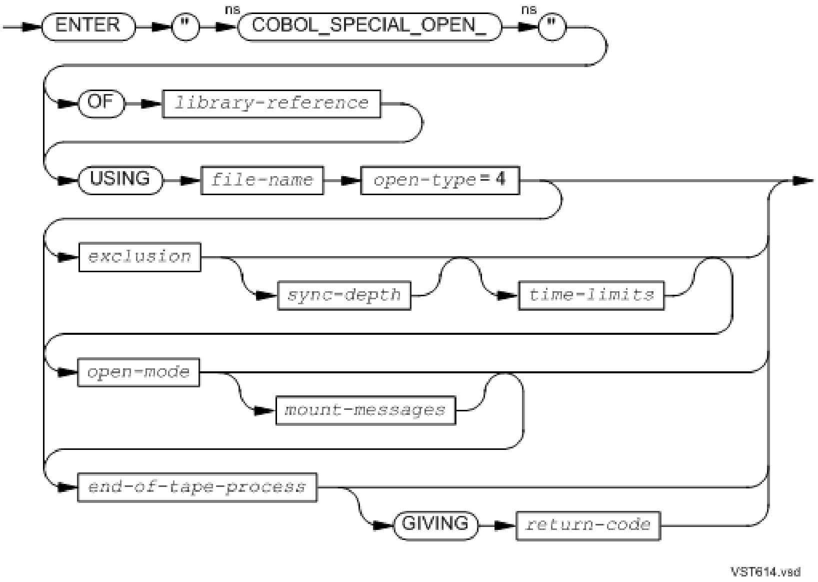
For System Log Files



For Partitioned Disk Files



For Tape Files



47 ZCREDLL Routine Calls

See the *CRE Programmer’s Guide*.

48 Compiler Diagnostic Messages

Table 48-1 Warning, Error, and Failure Characteristics

Message Type	Reports	Compilation Continues	Code Generation is Suppressed	Object File is Suppressed
Warning	Questionable condition	Yes	No	No
Error	Serious syntactic or semantic violation	Yes	For current separately compiled program	In some cases
Failure	Condition so severe that compilation cannot continue	No	Yes	Yes

Message Indicator Line

A message indicator line precedes each compiler diagnostic message. If possible, the message indicator line specifies the location of the problem or the unacceptable object with one or more of these:

- A caret (^) showing where in the source line the compiler recognized the problem
- The user-defined name of the object that the problem involves

The message indicator line cannot always tell you exactly where the problem is because the problem might not be recognizable (and reportable) until the compiler is past the source line that contains it.

Table 48-2 Problem Discovery Time, Message Indicator Line Contents, and Problem Location

Problem Discovery Time	Message Indicator Line Contents	Problem Location
During textual or syntactic analysis of a particular source file line	Caret (^) (usually), "Problem on line *nnnnn*" (maybe)	Language element marked by caret, source file line number *nnnnn*
During semantic analysis of an operand phrase, clause, or statement	"Problem on line *nnnnn*" (usually), caret (^) (maybe)	In vicinity of source file line number *nnnnn*, language element marked by caret
After analyzing many source lines, possibly an entire source program	User-defined name of object, "Problem on line *nnnnn*" (maybe)	With the named object, in vicinity of source file line number *nnnnn*

Example 48-1 Message Indicator Line

```
   13          MOVE "" TO A.
*** Error:               ^
--> Null literal [Error 30]
```

Warning Message Format

A warning message has this format:

```
*** Warning:
--> message-text [Warning message-number]
```

The brackets are part of the message, not indicators that the bracketed material is optional.

Error Message Format

An error message has this format:

```
*** Error:
--> message-text [Error message-number]
```

The brackets are part of the message, not indicators that the bracketed material is optional.

Failure Message Format

A failure message has this format:

```
*** Failure:
--> message-text [Failure message-number]
```

The brackets are part of the message, not indicators that the bracketed material is optional.

Message List

Messages are listed in numeric order by message number, with message type (warning, error, or failure) noted.

0 (Failure)

`Compiler logic error`

Cause The compiler's internal consistency checks found a logic error. Please report this failure to your service provider.

1 (Failure)

`CREATE failure on work file (`*`file-name`*`): `*`ddd`*

Cause The compiler cannot create one of its work files. *`file-name`* is the external form of the file name. *`ddd`* is the file management error code returned by the operating environment.

2 (Failure)

`OPEN failure on `*`xxxx`*` file (`*`file-name`*`): `*`ddd`*

Cause One of:

- The compiler cannot open the file named *`file-name`*.
- The file name parameter of the CONSULT, SEARCH, LIBRARY, or ERRORFILE directive is either absent or does not have the form of a disk file name.

The *`xxxx`* parameter is the type of compiler file (source, copy, list, or work). *`file-name`* is the external form of the file name. *`ddd`* is the file management error code that the operating environment returned.

3 (Failure)

`Unable to use `*`xxxx`*` file (`*`file-name`*`)`

Cause The compiler cannot use the file named *`file-name`*. *`xxxx`* is the type of compiler file (source, SEARCH, COPY, or list). *`file-name`* is the external form of the file name. Some causes of this failure are:

- The file *`file-name`* does not exist.
- The file *`file-name`* is not a disk file.
- The file *`file-name`* is specified as the source file, but its attributes are inappropriate for a source file (usually because the device type does support read operations).
- The file *`file-name`* is specified as a SEARCH or CONSULT file, but it is not a code 800 object file.
- The file *`file-name`* is specified as a SEARCH file, but it was not created by a compilation with symbols specified.
- The file *`file-name`* is specified as a COPY library, but its attributes are inappropriate for a COPY file (*`file-name`* is not an EDIT file or has been modified since the start of the current compilation).

- The default COPY library named `file-name` in the command that called the compiler has improper punctuation or its content does not have the form of a disk file name.
- The file `file-name` is specified as the compiler listing, but it's attributes ar inappropriate for a compiler listing (the device type does support write operations, the record length is too short, the file is an EDIT file or a keyed-access disk file, and so on).

4 (Failure)

`OPEN edit failure on xxxx file (file-name): ddd`

Cause The compiler cannot initialize the indicated EDIT file for reading. *xxxx* is the type of compiler file (source or copy). *file-name* is the external form of the file name. *ddd* is the error code describing the problem. A negative *ddd* value indicates a format error in the file. A nonnegative value indicates a Guardian file management error.

5 (Failure)

`READ edit failure on xxxx file (file-name): ddd`

Cause The compiler cannot read a record from the EDIT file indicated. *xxxx* is the type of compiler file (source or copy). *file-name* is the external form of the file name. *ddd* is the Guardian file management error code describing the problem.

6 (Failure)

`READ failure on xxxx file (file-name): ddd`

Cause The compiler cannot read a record from the file named *file-name*. *xxxx* is the type of compiler file (source or work). *file-name* is the external form of the file name. *ddd* is the file management error code that the operating environment returned.

7 (Failure)

`WRITE failure on xxxx file (file-name): ddd`

Cause The compiler cannot write a record to the file named *file-name*. *xxxx* is the type of compiler file (list or work). *file-name* is the external form of the file name. *ddd* is the file management error code that the operating environment returned.

8 (Failure)

`I/O failure on xxxx file (file-name): ddd`

Cause The compiler cannot perform a miscellaneous input-output operation on the file named *file-name*. *xxxx* is the type of compiler file (list or work). *file-name* is the external form of the file name. *ddd* is the file management error code that the operating environment returned.

9 (Failure)

`Spooler failure on list file (file-name): ddd`

Cause The compiler either cannot initiate or cannot terminate spooler processing for its list file. *file-name* is the external form of the file name. *ddd* is the error code describing the problem (explained in documentation of the SPOOLSTART and SPOOLEND procedures).

10 (Failure)

`Unable to allocate compiler data space: ddd`

Cause The compiler cannot allocate its extended data segment. *ddd* is the error code describing the problem (explained in the documentation of the ALLOCATESEGMENT procedure).

11 (Failure)

```
Source line exceeds 132 characters
```

Cause A source file or COPY library file contains more than 132 characters.

12 (Failure)

```
Improper context for source text directive
```

Cause One of:

- The reserved word COPY appears where it is not expected (for example, within COPY library text).
- The reserved word REPLACE appears where it is not expected (for example, within source text produced by the editing activities of a REPLACE statement).

13 (Failure)

```
Expected IDENTIFICATION
```

Cause The source program does not begin with an Identification Division header.

14 (Failure)

```
End of file reached during error recovery
```

Cause The compiler reached the end of the source file during an attempt to recover from a syntax error. This usually occurs in one of these situations:

- A syntax error is near the end of the source file, so the remaining text does not provide enough context for the compiler to decide how to recover.
- A syntax error is such that the compiler fails to discover an acceptable method of recovery and flushes the remaining source text.

15 (Failure)

```
Compiler internal resource failure
```

Cause The compiler cannot continue execution because it has exhausted some internal resource. The message can have one of these suffixes:

```
- statement nesting too deep
- editing space overflow
- too much pseudo text
- too many editable lines
- too many edited lines
```

If the message has the suffix

```
- statement nesting too deep
```

rewrite the statement, reducing the number of nested scopes. Use PARAM SYMBOL-BLOCKS (page 537) to allocate more space.

If PARAM SYMBOL-BLOCKS does not solve the problem, and the program uses COPY, REPLACE, or both COPY and REPLACE:

- Reduce the size of the pseudotext or literal in the REPLACE statement or REPLACING phrase.
- Use REPLACE OFF when replacement is no longer needed.
- Reduce the number of contiguous comment lines that can be read while COPY LIBRARY or REPLACE is active.
- Avoid having COPY and REPLACE active at the same time.

16 (Failure)

```
Too many errors
```

Cause The number of error messages exceeds the specified limit (100 unless specified otherwise in an ERRORS directive).

17 (Failure)

`Dictionary overflow`

Cause The compiler's symbol dictionary has insufficient space to describe all entities defined in the current separately compiled program. Divide the program into two or more separately compiled programs. Use PARAM SYMBOL-BLOCKS to allocate more space.

18 (Failure)

`Format switch during editing`

Cause A format switch directive (ANSI or TANDEM) appears in text edited under the control of a COPY or REPLACE statement. The compiler cannot process the source text correctly.

19 (Failure)

`Server failure`

Cause The compiler cannot continue execution because one of its server processes reported a failure or terminated abnormally.

20 (Failure)

`Assigned CPU is not licensed for this compiler`

Cause The processor in which the compiler is to run is not licensed for this compiler's execution.

21 (Failure)

`SOURCE nesting too deep`

Cause The compiler ran out of storage space because the nesting of SOURCE directives summoning text that includes other SOURCE directives is too deep.

25 (Warning)

`Blank continuation line`

Cause A source line marked as a continuation line (having a hyphen in the indicator area) contains only spaces as its text.

26 (Warning)

`Improper indicator character`

Cause One of:

- The character in the indicator area of a source line is not minus (-), asterisk (*), slash (/), question mark (?), *d*, *D*, or space.
- A continuation line is part of a comment-entry in a paragraph of the Identification Division.

27 (Error)

`Improper character`

Cause The character indicated is not permitted in this context. Because the character might not be printable, its internal (octal) value is appended to the message.

28 (Error)

`Text not permitted here`

Cause One of:

- A division header is followed by other text on the same source line.
- A section header is followed by other text on the same source line (other than a USE statement).
- The DECLARATIVES or END DECLARATIVES header is followed by other text on the same source line.
- A compiler directive that must be the last one on its line is followed by other text on the same directive line.

29 (Error)

```
Missing quote character
```

Cause The terminating quotation-mark character (") is missing from a nonnumeric literal.

30 (Error)

```
Null literal
```

Cause A nonnumeric literal contains no characters (has no textual value).

31 (Error)

```
Literal exceeds 160 characters
```

A nonnumeric literal contains more than 160 characters.

32 (Error)

```
Token exceeds 160 characters
```

Cause A character-string appears (to the compiler) to contain more than 160 characters. If the indicated text is intended to represent several consecutive language elements, correct the problem by inserting space separators between them.

33 (Error)

```
Numeric literal exceeds 18 digits
```

Cause A numeric literal contains more than 18 digits.

34 (Error)

```
Word exceeds 30 characters
```

Cause A COBOL word contains more than 30 characters.

35 (Error)

```
Word ends with '-'
```

Cause The last character of the indicated COBOL word is a hyphen.

36 (Warning or Error)

```
Improper use of reserved word
```

Cause **Warning:** A COBOL reserved word is the text-name or library-name in a COPY statement.

Error: A COBOL reserved word is in an improper context. Usually this message reports the use of a noncritical reserved word (one defined in an unsupported language module) as a user-defined word.

37 (Error)

```
Do not quote PICTURE string
```

Cause A PICTURE character-string is specified as a nonnumeric literal.

38 (Warning)

`Missing separator`

Cause One of:

- A comma, semicolon, or period character is not followed by the space needed to make it a separator.
- A character-string is not followed by an appropriate separator.

39 (Warning, Error or Failure)

`Improper actual file name`

Cause **Warning:** The run unit file name (object file name) field in the command that called the compiler has improper punctuation or its content does not have the form of a disk file name. The compiler supplies RUNUNIT (with appropriate default volume and subvolume components) as the run unit file name.

Error: A system-name is in a context where it must identify an actual file (for example, in a File-Mnemonic clause), but it does not have the form of an actual file name.

Failure: A COPY or SOURCE file does not have the form of a disk file name.

40 (Error)

`Improper syntax`

Cause One of:

- The analysis of a COPY or REPLACE statement failed when it reached the indicated element because of a syntactic problem (for example, a required component is missing).
- The compiler found a syntactic problem in a Procedure Division statement after the general syntax analysis completed with apparent success.

41 (Error)

`Syntax error - replacing unexpected token with` *`xxxx`*

Cause The compiler found a syntax error at the indicated point and recovered by replacing that token (a character-string or a separator) with the token *xxxx* (a different character-string or separator). This replacement is probably just one of several possible corrections. You must determine the proper correction through careful examination of the text.

42 (Error)

`Syntax error - inserting missing token` *`xxxx`*

Cause The compiler found a syntax error at the indicated point and recovered by inserting the token *xxxx* (a character-string or a separator) just prior to that point. Inserting that token is probably just one of several possible corrections. You must determine the proper correction through careful examination of the text.

43 (Error)

`Syntax error - deleting unexpected token` *`xxxx`*

Cause The compiler found a syntax error at the token *xxxx* (a character-string or a separator) and recovered by deleting that token. The problem could have been a misspelled optional reserved word or an extraneous separator. Deleting the token is probably just one of several possible corrections. You must determine the proper correction through careful examination of the text.

44 (Error)

`Syntax error detected at token` *`xxxx`*

Cause The compiler's syntax analyzer cannot accept the token *xxxx* (a character-string or a separator) and the recovery mechanism cannot find a simple correction. The compiler tried to recover by discarding text following the token, along with as little text preceding the token as is necessary, until the remaining text was syntactically acceptable.

45 (Warning)

`Parsing resumed at token xxxx`

Cause The compiler discarded all tokens from the one indicated in the message for error 44 up to, but not including, the token *xxxx*. Also, the compiler might have discarded some text prior to the point indicated in the preceding message. This error correction usually is not the preferred one. You must determine the proper correction through careful examination of the text at the point indicated by the preceding syntax error message.

46 (Error)

`Expected directive`

Cause The compiler expected to find a directive and did not. Directives are expected in these places:

- At the beginning of a directive line
- After the punctuation following a completed directive

47 (Error)

`Unknown directive`

Cause The indicated word must be a directive keyword, but it is not.

48 (Warning or Error)

`Improper context for this directive`

Cause The compiler encountered an improperly-placed directive.

Warning: One of:

- The SYMBOLS or NOSYMBOLS directive is within a separately compiled program. It is ignored in this context.
- The STANDARD 2002 directive is in effect when the FIPS or SUBSET directive appears, or a FIPS or SUBSET directive is in effect when the STANDARD 2002 directive appears. The directive that appeared first remains in effect.

Error: One of:

- The indicated directive (MAIN or NONSTOP) cannot appear at this point in the source text. It must precede the first source program's Identification Division header.
- The indicated directive (SYNTAX or COMPILE) cannot appear at this point in the source text. It must appear before the first separately compiled program or between separately compiled programs.
- The SECTION directive cannot appear at this point in the directive line. It must be the first one on its directive line.
- The program contains the SQLMEM directive but not the SQL directive.

49 (Error)

`Expected directive parameter`

Cause The compiler did not find the directive parameter keyword that it expected in one of these places:

- Following a directive keyword
- After the punctuation following a completed directive parameter

50 (Error)

```
Unknown directive parameter
```

Cause The indicated word must be a parameter keyword of the current directive, but is not.

51 (Error)

```
Expected quoted string
```

Cause The HEADING directive has a parameter that is not a string surrounded by quotation marks.

52 (Error)

```
Expected program-name
```

Cause The MAIN directive has either no parameter or a parameter that is not a COBOL word.

53 (Error)

```
Expected comma or other delimiter valid for context
```

Cause One of:

- Multiple compiler directives appear on the same line but are not separated by semicolons.
- Multiple parameters of a compiler directive are not separated by commas.
- A compiler directive parameter list begun with a left parenthesis is not terminated by a right parenthesis.

54 (Warning)

```
No symbol table generated
```

Cause One of:

- The compiler finds a SYMBOLS directive when a SYNTAX directive is active. The compiler ignores the SYMBOLS directive.
- The compiler finds a SYNTAX directive when the SYMBOLS directive is active. The compiler proceeds as if a NOSYMBOLS directive preceded the SYNTAX directive.

55 (Error)

```
Parameters ignored
```

Cause A NOCROSSREF directive is followed by directive parameters, which it cannot have.

56 (Warning)

```
This directive permitted only on the command line
```

Cause The SQL directive must be specified on the command line. An SQL directive in the source program will be ignored.

58 (Error)

```
Unable to convert continuation to debugging line
```

Cause The compiler could not convert a source line processed under the control of a COPY or REPLACE statement to a debugging line (by putting *`D`* or *`d`* in the indicator area) because the line is also a continuation line.

59 (Warning)

```
Logic may differ from COBOL 74 - tttt
```

Cause The compiler's interpretation of the source program might differ from the HP COBOL 74 compiler's interpretation. *tttt* indicates which aspect of the program is being reported. See DIAGNOSE-74 and NODIAGNOSE-74 (page 554).

You must determine what effects the compiler's interpretation has on the program's execution logic, decide if those effects are desired or not, and then revise the source program as needed. The compiler reports potential logic differences only when the DIAGNOSE-74 directive is specified.

60 (Warning or Error)

```
Not supported
```

Cause The compiler found an optional element of the COBOL language that HP COBOL does not support.

Warning: Examples of language elements that cause this warning are:

- More than one system-name in an ASSIGN clause (the compiler ignores all system-names after the first one)
- The VALUE clause of a file description entry (the compiler ignores the VALUE clause)

Error: Use of an unsupported language element, for example:

- The OPEN … REVERSED option
- Elements defined in the Communications module
- Elements defined in the Report Writer module
- An I/O statement in a nested program that implicitly references a GLOBAL declarative that is defined in a containing program

61 (Warning)

```
Logic may differ from COBOL85 - move alphanumeric to numeric with invalid
data
```

Cause The ECOBOL compiler's interpretation of the source program might differ from the COBOL85 compiler's interpretation. See DIAGNOSE-85 and NODIAGNOSE-85 (page 556).

You must determine what effects the ECOBOL interpretation has on the program's execution logic, decide if those effects are desired, and then revise the source program if needed. The ECOBOL compiler reports potential logic differences only when the DIAGNOSE-85 directive is specified.

If you suspect that the data item that caused this warning contains trailing spaces, use the NUMVAL function; for example, change:

```
MOVE a TO b
```

To:

```
MOVE FUNCTION NUMVAL (a) TO b
```

62 (Error)

```
Name conflict
```

Cause One of:

- The definition of a user-defined word conflicts with its prior definition as the name of an object in another class.
- The definition of a user-defined word conflicts with its prior definition as the name of an object in the same class, because all objects in that class must have unique names.
- The name of a special register is specified as the name of a user-defined object.

63 (Error)

```
Ambiguous reference
```

Cause A reference has insufficient qualification to identify a unique entity.

64 (Error)

```
Zero not permitted in this context
```

Cause The indicated integer numeric literal has the value 0, which is not permitted in this context.

65 (Error)

```
Integer not within expected range
```

Cause One of:

- The value of an integer numeric literal is either too small or too large for its context. The expected range can be absolute (for example, many literals cannot exceed 32,767) or relative (for example, the FOOTING value in a LINAGE clause cannot exceed the number of lines in the page body).
- The value of an integer numeric literal used as a subscript is less than 1 or greater than the maximum occurrence number defined by the appropriate OCCURS clause.

66 (Error)

```
Expected unsigned integer
```

Cause Only an unsigned integer numeric literal is permitted in this context, and the compiler found something else.

67 (Error)

```
Expected single character
```

Cause The indicated nonnumeric literal contains more than one character, which is not allowed in this context.

69 (Error)

```
Expected alphabet-name
```

Cause One of:

- The alphabet-name specified in the PROGRAM COLLATING SEQUENCE clause is not defined.
- A reference that must identify an alphabet-name does not.

70 (Error)

```
Expected file name
```

Cause A reference that must identify a file name does not.

71 (Error)

```
Expected symbolic character
```

Cause The indicated COBOL word appears in a context where a symbolic-character is expected, but it does not identify one.

72 (Error)

```
Expected section-name
```

Cause One of:

- The section-name parameter is missing from a SECTION directive.
- The text-name specified in a COPY statement is not a section-name in the COPY library.
- A section-name specified in a SOURCE directive is not in the source library.

73 (Error)

```
Improper range
```

Cause One of:

- The first value in a numeric range exceeds the last value (or is equal to the last value in a context in which it must be less).
- The first value in a nonnumeric range is greater than the last value.

74 (Error)

```
Too many actual parameters
```

Cause One of:

- A CALL statement specifies more parameters in its USING phrase than HP COBOL supports.
- An ENTER statement specifies more parameters in its USING phrase than the routine expects.

75 (Error)

```
Out of order
```

Cause The indicated language element does not appear in the proper position within the source program.

76 (Error)

```
Duplicate phrase
```

Cause The indicated phrase duplicates the function of a preceding one.

77 (Error)

```
Duplicate clause
```

Cause The indicated clause duplicates the function of a preceding one.

78 (Error)

```
Duplicate paragraph
```

Cause The indicated paragraph header duplicates a preceding one.

79 (Error)

```
Duplicate section
```

Cause The indicated section header duplicates a preceding one.

80 (Error)

```
Clause not permitted in this context
```

Cause The indicated clause cannot appear in the current entry.

81 (Error)

```
Not permitted within contained program
```

Cause One of:

- The Configuration Section is within a contained program.
- The RECEIVE-CONTROL paragraph is within a contained program.
- A program is not terminated by an END PROGRAM statement or an ENDUNIT directive.

82 (Error)

```
Too many keys
```

Cause One of:

- A file has more than 31 alternate record keys.
- An OCCURS clause has more than 31 key references.
- A SORT or MERGE statement has more than 31 keys.

83 (Error)

```
Too many receiver items
```

Cause The number of receiver items in the statement exceeds the maximum number allowed. For the maximum number allowed, see Chapter 20: Using HP COBOL in the OSS Environment (page 721).

84 (Error)

```
Too few actual parameters
```

Cause An ENTER statement specifies fewer parameters in its USING phrase than the routine expects.

85 (Warning)

```
Arithmetic expression too complex - floating-point used
```

Cause An arithmetic expression was too complex to produce a correct answer, probably because the intermediate result exceeded the capacity of the intermediate data item (36 digits) or because there was more than one division operation.

The expression uses a floating-point, intermediate, data item, which might cause the result to be slightly off in the rightmost positions. If you want exact results, break the expression into more than one statement, using temporary data items that have the precision you need.

86 (Error or Failure)

```
Program nesting too deep
```

Cause **Error:** Programs are nested deeper than the limit of 7. **Failure:** Program nesting depth is too great.

87 (Error)

```
Missing PROGRAM-ID paragraph
```

Cause The Identification Division has no PROGRAM-ID paragraph (the compiler generates a name for the source program).

88 (Error or Failure)

```
Duplicate program-name
```

Cause Within a separately compiled program, the same name identifies more than one source program.

89 (Error)

```
Program-name conflicts with routine-name
```

Cause The same name identifies both a separately compiled COBOL program and a routine in the same run unit. The conflict arises because the name is the subject of CALL or CANCEL statements in the current separately compiled program, and these references must be resolved to another separately compiled COBOL program not yet defined in the source text; however, the compiler has already found references to this name as the identifier of a routine.

90 (Error)

```
Permitted only within contained program
```

Cause The COMMON phrase appears somewhere other than within a contained program.

91 (Error)

```
System-name not defined in this context
```

Cause One of:

- The reference in the CHARACTER-SET clause is not one of the system-names defined for that clause.
- The system-name in the ALPHABET clause is not EBCDIC, the only value that HP COBOL supports.
- The standard system-name in the RECORD DELIMITER clause is not STANDARD-1, the only value that HP COBOL supports.
- A reference in the REPORT clause of the RECEIVE-CONTROL paragraph is not one of the system-names defined for that clause.

92 (Error)

```
STATUS phrase permitted only for switch
```

Cause A STATUS phrase appears somewhere other than in a system-name clause whose subject is one of the supported external switch names.

93 (Warning)

```
Missing ALPHABET keyword
```

Cause A clause defining an alphabet-name does not begin with the keyword ALPHABET.

94 (Error)

```
Symbolic character not permitted
```

Cause A symbolic-character appears within the definition of an alphabet-name or class-name.

95 (Error)

```
Duplicates not permitted
```

Cause One of:

- A character has more than one collating position in a character set.
- A character appears more than once in the first operand of the CONVERTING phrase in an INSPECT statement.
- A section-name appears more than once in a SOURCE directive.

96 (Error)

```
Too many or too few values
```

Cause The number of values in the SYMBOLIC CHARACTER clause differs from the number of symbolic-characters being defined.

97 (Error)

```
Improper currency symbol
```

Cause Either the alternative currency symbol specified is not a single character or the given character is not permitted in this context.

98 (Error)

```
Missing ASSIGN clause
```

Cause A file-control entry does not have an ASSIGN clause.

99 (Error)

```
Missing relative key
```

Cause The description of a file with relative organization and either random or dynamic access does not have a RELATIVE KEY clause.

100 (Error)

```
Missing record key
```

Cause The description of a file with indexed organization does not have a RECORD KEY clause.

101 (Error)

```
Relative key permitted only for relative organization
```

Cause A file that does not have relative organization has a RELATIVE KEY clause in its file-control entry.

102 (Error)

```
Record key permitted only for indexed organization
```

Cause A file that does not have indexed organization has a RECORD KEY clause in its file-control entry.

103 (Error)

```
Sequential organization requires sequential access
```

Cause The program tried to access a file that was described with sequential organization randomly or dynamically.

104 (Error)

```
PADDING clause permitted only for sequential organization
```

Cause A file that does not have sequential organization has a PADDING clause in its file-control entry.

105 (Error)

```
RECORD DELIMITER clause permitted only for sequential organization
```

Cause A file that does not have sequential organization has a RECORD DELIMITER clause in its file-control entry.

106 (Error)

```
Expected local file name
```

Cause The indicated name is not defined in a file-control entry of the current source program.

107 (Warning)

```
Duplicate file-name in SAME clause
```

Cause A file name appears more than once in a single SAME AREA, SAME RECORD AREA, or SAME SORT AREA clause.

108 (Error)

```
File name permitted in only one SAME clause
```

Cause One of:

- A particular file name appears in more than one SAME AREA clause.
- A particular file name appears in more than one SAME RECORD AREA clause.
- A particular sort-merge file name appears in more than one SAME SORT AREA clause.

109 (Warning or Error)

```
Expected two or more file names
```

Cause **Warning:** The SAME AREA, SAME RECORD AREA, or SAME SORT AREA clause contains fewer than two file names.

Error: The USING phrase in a MERGE statement contains fewer than two file names.

110 (Error)

```
Inconsistent with other SAME clause
```

Cause One of these rules was violated:

- When a file name that appears in a SAME AREA clause also appears in a SAME RECORD AREA clause, every other file name in that SAME AREA clause must also appear in the SAME RECORD AREA clause.
- When a file name that appears in a SAME AREA clause also appears in a SAME SORT AREA clause, every other file name in that SAME AREA clause must also appear in the SAME SORT AREA clause.

111 (Error)

```
Too many MULTIPLE FILE TAPE clauses
```

Cause A source program has more than 8 MULTIPLE FILE TAPE clauses.

112 (Error)

```
Too many file names
```

Cause A source program has too many CONSULT, LIBRARY, and SEARCH files for the available memory.

113 (Error)

```
file name permitted only once in MULTIPLE FILE TAPE clauses
```

Cause The same file name appears more than once in a MULTIPLE FILE TAPE clause or in more than one MULTIPLE FILE TAPE clause.

114 (Error)

```
File position required
```

Cause A file name in a MULTIPLE FILE TAPE clause follows one or more other file names that have explicit position numbers, but it does not have an explicit position number.

115 (Error)

```
Duplicate file position
```

Cause Two or more file names in a MULTIPLE FILE TAPE clause have the same position number.

116 (Error)

```
MULTIPLE FILE TAPE clause requires sequential organization
```

Cause A MULTIPLE FILE TAPE clause has a file name that is not described with sequential organization.

117 (Error)

```
Receive control table too large
```

Cause The size of the receive control table as specified in the RECEIVE-CONTROL paragraph exceeds 62 KB (65,400 characters). Reduce *`table-length`* in the TABLE OCCURS phrase or reduce another factor in these formulas, which determine the size of the receive control table:

```
((max_requesters + 1) * 26) +
((((max_reply + 1) / 2) + 4) * (max_requesters * sync)) +4
```

121 (Error)

```
Improper level-number
```

Cause A level-number is not 66, 77, 88, or in the range 01 through 49. The compiler converts the improper level-number to 50.

122 (Error)

```
Missing 01 level entry
```

Cause A data description entry with a level-number from 02 through 49 inclusive is not subordinate to a data description entry with level-number 01.

123 (Error)

```
Not preceded by record
```

Cause A data description entry with level-number 66 is not preceded by a record description. (Any intervening data description entries must also have level-number 66.)

124 (Error)

```
Not permitted within this section
```

Cause One of:

- A level-77 data item is defined in the File Section.
- A level-01 data item defined in the File Section is described with the REDEFINES clause.
- The EXTERNAL clause for a data description entry is not within the Working-Storage Section or Extended-Storage Section.
- The GLOBAL clause appears in descriptions that are not within the File Section, Working-Storage Section, or Extended-Storage Section.
- The description of a data item defined within the File Section or Linkage Section includes a VALUE clause.

125 (Error)

```
Not preceded by conditional variable
```

Cause The definition of a condition-name (a name whose data description entry has level-number 88) is not preceded by the entry for the data item whose value it tests. (Any intervening data description entries must also have level-number 88.)

126 (Error)

```
Inconsistent level-number
```

Cause A level-number is neither greater than the level-number of the preceding data description entry nor equal to that of some preceding data description entry in the same record description.

127 (Error)

```
Not permitted after variable occurrence table
```

Cause Within a record description, a data description entry that includes an OCCURS clause with the DEPENDING phrase is followed by a data description entry with a lesser level-number.

128 (Error)

`FILLER not permitted for 01 level external data item`

Cause A record data item that has no name, or is a FILLER data name, is either described with the EXTERNAL clause or is subordinate to a file description entry that includes the EXTERNAL clause.

129 (Error)

`FILLER not permitted for 01 level global data-name`

Cause A record data item that has no name, or is a FILLER data name, is either described with the GLOBAL clause or is subordinate to a file description entry that includes the GLOBAL clause.

130 (Error)

`FILLER not permitted for this level-number`

Cause One of:

- A data description entry with level-number 66 or 77 does not include the data-name.
- A data description entry with level-number 88 does not include the condition-name.

131 (Error)

`Redefined data item not found`

Cause The reference in a REDEFINES clause does not identify a data item. (When a REDEFINES clause appears in a record description, only that record is searched for the redefined item.)

132 (Error)

`Redefined data item has conflicting level-number`

Cause The data item to be redefined does not have the same level-number as the redefinition data item.

133 (Error)

`Redefined and redefinition data items not subordinate to same levels`

Cause A redefined data item is a subordinate of a file description entry and/or one or more data structure description entries, but the redefinition item is not a subordinate of these entries.

134 (Error)

`Redefined data item is a redefinition`

Cause The program tried to redefine a data item that was described with a REDEFINES clause. (A subordinate of a redefinition can be redefined unless it is also described with a REDEFINES clause.)

135 (Error)

`Redefined data item not preceding item at this level`

Cause The data description entry of a redefinition was separated from the data description entry of the redefined item by another data description entry with the same level-number, and the intervening entry did not redefine the same data item.

136 (Error)

```
Redefined data item is table or has variable size
```

Cause The program tried to redefine a table or a data structure with a variable size.

137 (Error)

```
Name conflict with other 01 level external data item
```

Cause Two different records in the same source program have the same record-name and both have the EXTERNAL attribute. Only one can have the EXTERNAL attribute.

138 (Error)

```
Redefinition not permitted for 01 level external data item
```

Cause A record item described with a REDEFINES clause has the EXTERNAL clause.

139 (Error)

```
Picture string exceeds 30 characters
```

Cause A PICTURE character-string has more than 30 characters.

140 (Error)

```
Improper picture string
```

Cause The PICTURE character-string does not conform to the rules of the COBOL language, possibly for one of these reasons:

- The characters are undefined in this context.
- The characters are combined improperly.
- The character-string has unmatched parentheses.
- The character-string has no positions for data characters.

141 (Error)

```
Too many digit positions
```

Cause The PICTURE character-string for a numeric or numeric-edited data item has more than 18 digit positions.

142 (Error)

```
Too many character positions
```

Cause The PICTURE character-string has more than 134,217,726 character positions.

143 (Error)

```
PICTURE clause not permitted for specified usage
```

Cause The PICTURE clause describes a data item with USAGE INDEX, NATIVE-2, NATIVE-4, or NATIVE-8.

144 (Error)

```
Subordinate usage conflicts with group usage
```

Cause A data item that is subordinate to a data structure described with a USAGE clause is described with a USAGE clause specifying a different usage.

145 (Error)

```
Specified usage permitted only for numeric data item
```

Cause Only a numeric data item can be described with the specified usage.

146 (Error)

```
Display usage required in group with value or condition-names
```

Cause A data item that is either subordinate to a data structure described with a VALUE clause or associated with condition-names does not have USAGE DISPLAY, as it must.

147 (Error)

```
Display usage required when SIGN clause applies
```

Cause One of these items does not have USAGE DISPLAY, as it must:

- A data item described with a SIGN clause
- A signed numeric data item that is subordinate to a data structure described with a SIGN clause

148 (Error)

```
Only signed numeric picture permitted when SIGN clause specified
```

Cause A data item is described with a SIGN clause, but either its category is not numeric or its PICTURE character-string does not contain *S*.

149 (Error)

```
SYNCHRONIZED clause not permitted in group with value or condition-names
```

Cause A data item is described with a SYNCHRONIZED clause, but either the data item is subordinate to a data structure described with a VALUE clause or the data item is associated with condition-names.

150 (Error)

```
JUSTIFIED clause requires display usage
```

Cause A data item is described with a JUSTIFIED clause, but the data item does not have USAGE DISPLAY, as it must.

151 (Error)

```
JUSTIFIED clause not permitted for numeric or edited data item
```

Cause A numeric, numeric-edited, or alphanumeric-edited data item is described with a JUSTIFIED clause.

152 (Error)

```
JUSTIFIED clause not permitted in group with value or condition-names
```

Cause A data item is described with a JUSTIFIED clause, but either the data item is subordinate to a data structure described with a VALUE clause or the data item is associated with condition-names.

153 (Error)

```
BLANK WHEN ZERO clause requires display usage
```

Cause A data item is described with a BLANK WHEN ZERO clause, but the data item does not have USAGE DISPLAY, as it must.

154 (Error)

```
BLANK WHEN ZERO clause requires numeric or numeric-edited picture
```

Cause A data item is described with a BLANK WHEN ZERO clause, but the data item is not numeric or numeric-edited, as it must be.

155 (Error)

```
BLANK WHEN ZERO clause not permitted for picture with '*'
```

Cause A data item is described with both the BLANK WHEN ZERO clause and a PICTURE character-string containing the asterisk symbol (*).

156 (Error)

```
Access mode conflict for redefinition or subordinate
```

Cause One of:

- A data item that is part or all of a redefinition is described with an ACCESS MODE clause specifying a different mode than the data item it redefines.
- A data item that is subordinate to a data structure described with an ACCESS MODE clause is described with an ACCESS MODE clause specifying a different mode.

157 (Error)

```
VALUE clause not permitted for index data item
```

Cause A data item with USAGE INDEX is described with a VALUE clause.

158 (Error)

```
VALUE clause not permitted for redefinition
```

Cause A data item that is part or all of a redefinition is described with a VALUE clause.

159 (Error)

```
VALUE clause not permitted for external data item
```

Cause An external data item is described with a VALUE clause.

160 (Error)

```
VALUE clause not permitted in group with initial value
```

Cause A data item that is subordinate to a data structure described with a VALUE clause is described with its own VALUE clause.

161 (Error)

```
Numeric literal not compatible with nonnumeric literal
```

Cause The literals of a range in a VALUE clause are not both numeric or both nonnumeric, as they must be.

162 (Error)

```
Value range not permitted for initial value
```

Cause The VALUE clause describing a data item contains a range of literals.

163 (Error)

```
Only one initial value permitted
```

Cause The VALUE clause describing a data item contains more than one value.

164 (Error)

```
Table nesting too deep
```

Cause A table has more than 7 OCCURS clauses.

165 (Error)

```
Variable occurrences not permitted for subordinate table
```

Cause A data item subordinate to a table item is described with an OCCURS clause that includes the DEPENDING phrase.

166 (Error)

```
Variable occurrences not permitted in redefinition
```

Cause A data item that is part or all of a redefinition is described with an OCCURS clause that includes the DEPENDING phrase.

167 (Error)

```
Renamed object not data item
```

Cause The RENAMES clause does not refer to a data item.

168 (Error)

```
Renamed data item not subordinate of preceding record
```

Cause The RENAMES clause references a data item that was not defined within the preceding record description.

169 (Error)

```
Renaming not permitted for 66 level data item
```

Cause The RENAMES clause references a 66-level item.

170 (Error)

```
Renamed data item in table or has variable size
```

Cause The RENAMES clause references a table item, a subordinate of a table item, or a data structure that has a variable size.

171 (Error)

```
Improper range for renamed data items
```

Cause One of:

- The second data item in a RENAMES clause contains no character positions that are not contained in the first data item.
- The initial character position of the second data item in a RENAMES clause precedes the initial character position of the first data item within their record item.

172 (Error)

```
Missing PICTURE clause
```

Cause An elementary data item that is described with neither USAGE INDEX nor a PICTURE clause. It must be described with at least one of these.

173 (Error)

```
Missing RENAMES clause
```

Cause A data description entry with level-number 66 does not have a RENAMES clause, as it must.

174 (Error)

```
Missing VALUE clause
```

Cause A data description entry with level-number 88 does not have a VALUE clause, as it must.

175 (Error)

```
Elementary data item clause specified for group data item
```

Cause The description of a data structure includes a BLANK WHEN ZERO, JUSTIFIED, SYNCHRONIZED, or PICTURE clause.

176 (Warning)

```
Group with SIGN clause has no signed numeric subordinate item
```

Cause A data structure described with a SIGN clause has no signed numeric subordinate data items.

177 (Error)

```
Data item attributes not compatible with CODE-SET clause
```

Cause A file description that includes the CODE-SET clause violates one of these rules:

- All data items defined in the record descriptions must have USAGE DISPLAY.
- All signed numeric data items must be described as having the sign in a separate character position.

178 (Error)

```
Redefinition not properly aligned
```

Cause The redefinition is aligned to the first character position of the area it redefines. HP COBOL does not permit a redefinition that requires allocation of an implicit FILLER item to properly align the first elementary item.

179 (Error)

```
Redefinition too large
```

Cause The number of character positions occupied by a redefinition exceeds the number of character positions occupied by the redefined item, and the redefined item is not an internal record item.

181 (Error)

```
Key object not data item subordinate to its table
```

Cause A reference in the KEY phrase of an OCCURS clause identifies neither the table item nor a data item defined in the table item that the clause describes.

182 (Error)

```
Key data item within subordinate table
```

Cause A table key data item is defined in, or as, a subordinate table item.

183 (Error)

```
Table item permitted only as first key data item
```

Cause A reference in the KEY phrase of an OCCURS clause identifies the table item itself, and that reference is not the first reference in the KEY phrase.

184 (Error)

```
Nonnumeric literal not permitted for numeric data item
```

Cause A literal in the VALUE clause describing a numeric data item or a condition-name associated with a numeric data item is neither ZERO (or one of its equivalents) nor a numeric literal, as it must be.

185 (Error)

```
Nonnumeric literal exceeds item size
```

Cause A literal in the VALUE clause describing a nonnumeric data item or data structure, or a condition-name associated with either type of item, is larger than the data item itself.

186 (Error)

```
Numeric literal not permitted for nonnumeric or group data item
```

Cause The VALUE clause describing a nonnumeric data item or data structure, or a condition-name associated with either type of item, contains a numeric literal.

187 (Error)

```
Signed literal not permitted for unsigned numeric data item
```

Cause The VALUE clause describing an unsigned numeric data item or a condition-name associated with an unsigned numeric data item contains a signed numeric literal.

188 (Error)

```
Numeric literal value inconsistent with numeric data item
```

Cause The VALUE clause describing a numeric data item, or a condition-name associated with a numeric data item, contains a literal that cannot be assigned to the data item without truncating nonzero digits.

189 (Error)

```
01 or 77 level data item too large for section
```

Cause The size of a data item exceeds the maximum size permitted for the Data Division section in which it is defined.

190 (Error)

```
Described with clauses not permitted for sort-merge file
```

Cause The file-control entry for a sort-merge file includes clauses other than the SELECT and ASSIGN clauses.

191 (Error)

```
EXTERNAL clause not compatible with SAME clause
```

Cause The file description entry of a file name that appears in a SAME AREA, SAME RECORD AREA, or SAME SORT clause includes the EXTERNAL clause.

192 (Error)

```
GLOBAL clause not compatible with SAME RECORD AREA clause
```

Cause The file description entry or a subordinate record description entry of a file name that appears in a SAME RECORD AREA clause includes the GLOBAL clause.

193 (Error)

```
LINAGE permitted only for sequential organization with no record keys
```

Cause One of:

- A file description entry includes the LINAGE clause, but the file organization is not sequential.
- A file description entry includes both the LINAGE clause and ALTERNATE RECORD KEY clauses.

194 (Error)

```
Logical page too large
```

Cause The sum of the top margin, page body, and bottom margin of a logical page exceeds 9999 lines.

195 (Error)

```
CODE-SET permitted only for sequential organization
```

Cause A file is described with the CODE-SET clause, but it does not have sequential organization, as it must.

196 (Error)

```
Referenced alphabet-name not permitted in this context
```

Cause One of:

- The alphabet-name specified in the CODE-SET clause is defined by a sequence of literal phrases.
- The alphabet-name specified in the CODE-SET clause is defined by NATIVE, STANDARD-1, or STANDARD-2 when the file's description includes ALTERNATE RECORD KEY clauses.

197 (Error)

```
File description has no record descriptions
```

Cause A file description entry is not followed by any record description entries.

198 (Warning)

```
Record-names inconsistent with file description entry
```

Cause At least one reference in the DATA RECORDS clause does not identify any record-name defined by the record description entries.

199 (Error)

```
Record sizes inconsistent with file description entry
```

Cause A record description entry describes a record longer or shorter than the maximum or minimum specified in the RECORD clause.

200 (Error)

```
Record size exceeds block size
```

Cause The maximum record size for the file exceeds the block size specified by the BLOCK CONTAINS clause.

201 (Error)

```
Fixed-size records incompatible with RECORD DELIMITER clause
```

Cause A file described with the RECORD DELIMITER clause does not have variable-size records, as it must.

202 (Error)

```
Block size too large
```

Cause A block exceeds 32,767 characters.

203 (Error)

`Record key not found within file record`

Cause The reference in a RECORD KEY or ALTERNATE RECORD KEY clause does not identify a data item defined in the file's record description entries.

204 (Error)

`Record key not simple fixed-size alphanumeric item`

Cause The object referenced in a FILE STATUS clause either is not a fixed-size alphanumeric data item or is a table.

205 (Error)

`Record key aligned with another record key`

Cause The first character position of one record key coincides with the first character position of another record key.

206 (Error)

`Missing file description entry for file name`

Cause A file name introduced in a file-control entry is not defined in a file description entry.

207 (Error)

`Padding item not found`

Cause The PADDING clause references an undefined name.

208 (Error)

`Padding item not simple one character alphanumeric`

Cause The PADDING clause references either an object that is not a fixed-size one-character alphanumeric data item or an object that is a table or special register.

209 (Error)

`Padding item not external`

Cause The PADDING clause for an external file references a data item that is not external.

210 (Error)

`Padding item in improper section`

Cause The padding data item for a file is not defined in the Working-Storage, Extended-Storage, or Linkage Section.

211 (Error)

`Relative key item not found`

Cause The RELATIVE KEY clause references an undefined name.

212 (Error)

`Relative key item not simple unscaled integer`

Cause The RELATIVE KEY clause references an object about which one of these is true:

- It is not an unsigned integer data item.
- It is described with `P`s in its PICTURE clause.
- It is a table or special register.

213 (Error)

`Relative key item not external`

Cause The RELATIVE KEY clause for an external file references a data item that is not external.

214 (Error)

`Relative key item within its file record`

Cause The relative key data item for a file is defined in the file's record description entries.

215 (Error)

`File status item not found`

Cause The FILE STATUS clause references an undefined name.

216 (Error)

`File status item not simple two character alphanumeric`

Cause The FILE STATUS clause references either an object that is not a fixed-size two-character alphanumeric data item or an object that is a table or special register.

218 (Error)

`File status in improper section`

Cause The FILE STATUS clause references a data item that is defined in the File Section.

219 (Error)

`Linage control item not found`

Cause The LINAGE clause references an undefined name.

220 (Error)

`Linage control item not simple unsigned integer`

Cause The LINAGE clause references either an object that is not an unsigned integer data item or an object that is a table or special register.

221 (Error)

`Linage control item not external`

Cause The LINAGE clause for an external file references a data item that is not external.

222 (Error)

`Depending item not found`

Cause A name referenced in the DEPENDING phrase of an OCCURS or RECORD clause is not defined.

223 (Error)

`Depending item not simple unsigned integer`

Cause The DEPENDING phrase of a RECORD clause references either an object that is not an unsigned integer data item or an object that is a table or special register.

224 (Error)

`Depending item not simple integer`

Cause The DEPENDING phrase of a OCCURS clause references either an object that is not an integer data item or an object that is a table or special register.

225 (Error)

```
Depending item in improper section
```

Cause The data item referenced in the DEPENDING phrase of a RECORD clause is not defined in the Working-Storage, Extended-Storage, or Linkage Section.

226 (Error)

```
Depending item not external
```

Cause The DEPENDING phrase of an OCCURS or RECORD clause for an external data item or an external file references a data item that is not external.

227 (Error)

```
Depending item not global
```

Cause The data item in the DEPENDING phrase of an OCCURS clause controls the number of occurrences of a table that has a global data-name, but the data item itself does not have a global data-name, as it must.

228 (Error)

```
Depending item within its table
```

Cause The data item in the DEPENDING phrase of an OCCURS clause is defined within the space of the table whose number of occurrences it controls.

229 (Error)

```
Message source item not found
```

Cause The MESSAGE SOURCE references an undefined name.

230 (Error)

```
Message source item not simple alphanumeric, has improper size, or not
word-aligned
```

Cause The MESSAGE SOURCE clause references an object about which one of these is true:

- It is not a fixed-size alphanumeric data item.
- It is a table item.
- It has fewer than 32 character positions.
- It begins on an odd-character boundary within a record.

231 (Error)

```
Error code item not found
```

Cause The ERROR CODE references an undefined name.

232 (Error)

```
Error code item not simple unsigned integer
```

Cause The ERROR CODE clause references either an object that is not an unsigned integer data item or an object that is a table or special register.

233 (Warning or Error)

```
Permitted value exceeds data item size
```

Cause **Warning:** One of:

- The maximum value assignable to the data item referenced in the DEPENDING phrase of an OCCURS clause is less than the maximum occurrence number of the table.
- The maximum value assignable to the data item referenced in the DEPENDING phrase of a RECORD clause is less than the maximum record size.

Error: One of:

- The maximum value assignable to the POINTER operand in a STRING statement is less than or equal to the size of the receiving operand.
- The maximum value assignable to the POINTER operand in an UNSTRING statement is less than or equal to the size of the sending operand.

234 (Error)

```
No sort-merge file name in SAME SORT clause
```

Cause None of the file names in the SAME SORT AREA clause identifies a sort-merge file.

235 (Error)

```
Too many external objects
```

Cause The program defines more external objects (files and records) than the compiler can allocate.

236 (Error)

```
Too many internal files
```

Cause The program defines more internal files than the compiler can allocate.

237 (Error)

```
Improper target for ADDRESS OF clause
```

Cause One of these is true about the `identifier-1` parameter in an ADDRESS OF clause:

- It is not a level-01 or level-77 data item in the Linkage Section.
- It is not a level-01 or level-77 data item declared with the BASED clause.

240 (Error)

```
USING operand not linkage section data-name
```

Cause The USING phrase specifies an operand that is not defined as a data-name in the Linkage Section of the current program.

241 (Error)

```
USING operand subordinate or redefinition data-name
```

Cause The USING phrase specifies an operand about which one of these is true:

- It is not described as a level 01 or level 77 data-name.
- It is described as a redefinition of another data-name.

242 (Error)

```
Data-name permitted only once as USING operand
```

Cause An operand appears more than once in the USING phrase.

243 (Error)

```
Too many USING operands
```

Cause The USING phrase has more than 126 operands.

244 (Warning or Error)

```
Linkage section data item not found as USING operand
```

Cause **Warning:** A data item defined in the Linkage Section is not referenced as a USING operand. Any references to it, any subordinates, or any redefinitions will produce unpredictable effects.

Error: The compiler found a data item in the Linkage Section but not in the USING phrase of any CALL statement in the Procedure Division.

245 (Error)

```
Improper context for DECLARATIVES
```

Cause The DECLARATIVES header does not precede all procedures in the Procedure Division.

246 (Error)

```
Declarative paragraph not within section
```

Cause A paragraph-name in the Declaratives Portion of the Procedure Division is not subordinate to any section-name, as it must be.

247 (Error)

```
Improper context for USE sentence
```

Cause The USE sentence does not immediately follow a section header in the Declaratives Portion of the Procedure Division, as it must.

248 (Error)

```
Debugging procedures must be first declaratives
```

Cause Not all debugging procedures precede all nondebugging procedures in the Declaratives Portion of the Procedure Division.

249 (Error)

```
Undefined debugging reference
```

Cause The indicated USE DEBUGGING statement cannot be resolved to any defined entity.

250 (Error)

```
Debugging not supported for this object
```

Cause HP COBOL does not support debugging for this type of object.

251 (Error)

```
Debugging not permitted for this object
```

Cause The COBOL language does not define debugging for this type of object.

252 (Error)

```
Debugging not permitted for debugging procedures
```

Cause A USE DEBUGGING statement references a paragraph- name or section-name that belongs to a debugging procedure.

253 (Error)

```
Conflicting debugging declarative assignments
```

Cause One of:

- A USE DEBUGGING statement specifies the same object more than once.
- More than one USE DEBUGGING statement in a source program specifies the same object.
- The program has more than one USE DEBUGGING ALL PROCEDURES statement.
- The program has a USE DEBUGGING ALL PROCEDURES statement and other USE DEBUGGING statements that reference paragraph-names or section-names.

254 (Error)

```
Conflicting exception declarative assignments
```

Cause One of:

- A USE AFTER EXCEPTION PROCEDURE statement specifies the same file name more than once.
- More than one USE AFTER EXCEPTION PROCEDURE statement in a source program specifies the same file name.
- More than one USE AFTER EXCEPTION PROCEDURE statement in a source program specifies the same open mode (INPUT, OUTPUT, I-O, or EXTEND).

255 (Error)

```
Missing END DECLARATIVES
```

Cause The Procedure Division does not contain the END DECLARATIVES header needed to terminate its Declaratives Portion.

256 (Error)

```
Improper context for END DECLARATIVES
```

Cause The Procedure Division contains an extraneous END DECLARATIVES header.

257 (Error)

```
No declarative procedures
```

Cause The Declaratives Portion exists, but contains no procedures.

258 (Error)

```
No nondeclarative procedures
```

Cause The current source program has an explicit Procedure Division, but its nondeclarative portion is empty.

259 (Error)

```
Exception phrase not compatible with file description
```

Cause An exception phrase corresponds to a statement with which it cannot be associated. Possible reasons are:

- The statement references a file whose description is incompatible with the phrase.
- A READ statement specifies the NEXT phrase and the exception phrase is INVALID KEY or NOT INVALID KEY.
- A READ statement does not specify the NEXT phrase and the exception phrase is AT END or NOT AT END.

260 (Error)

```
Improper context for exception phrase
```

Cause There is no preceding statement with which the exception phrase can be associated.

261 (Error)

```
Improper context for ELSE phrase
```

Cause There is no preceding statement with which the ELSE phrase can be associated.

262 (Error)

```
Improper context for WHEN phrase
```

Cause There is no preceding statement with which the WHEN phrase can be associated.

263 (Error)

```
Improper context for scope delimiter
```

Cause There is no preceding statement with which the explicit scope delimiter can be associated.

264 (Error)

```
Missing AT END phrase
```

Cause A RETURN statement does not have an associated AT END phrase.

265 (Error)

```
Missing WHEN phrase
```

Cause A preceding EVALUATE or SEARCH statement does not include a WHEN phrase.

266 (Error)

```
Missing scope delimiter
```

Cause A statement appears in a context that requires an explicit scope delimiter, but the appropriate scope delimiter is not specified.

267 (Error)

```
Improper context for EXIT or empty GO TO statement
```

Cause An EXIT or GO TO statement that does not specify a procedure-name is not the only statement in its sentence, as it must be.

268 (Error)

```
Improper context for NEXT SENTENCE
```

Cause NEXT SENTENCE is neither in an IF or SEARCH statement nor a replacement for an imperative-statement.

269 (Error)

```
Prior statement must be last in its sequence
```

Cause The prior statement appears in a sequence of statements, but is not the last one in the sequence, as it must be.

270 (Error)

```
Expected integer numeric literal
```

Cause One of:

- Only an integer numeric literal can appear in this context.
- When a literal appears in this context, it must be an integer numeric literal.

271 (Error)

```
Undefined object reference
```

Cause The reference cannot be resolved to any object that the current source program can access. (Check the spelling of the object name and verify that all qualifiers are correct and in the proper order.)

272 (Error)

```
Improper context for special register
```

Cause Usually, a special register has been used as a receiving operand, which is not allowed.

273 (Error)

```
Improper context for debugging special register
```

Cause References to the special register DEBUG-ITEM or any of its subordinates are not in a debugging procedure in the Declaratives Portion of the Procedure Division, as they must be.

274 (Error)

```
Expected data item
```

Cause The specified operand cannot appear in this context. (A data item with appropriate attributes would be acceptable here. Other types of operands, such as literals, might also be acceptable.)

275 (Error)

```
Expected display data item
```

Cause The data item does not have USAGE DISPLAY, as it must.

276 (Error)

```
Expected display data item, no Ps if numeric
```

Cause Either the data item does not have USAGE DISPLAY (as it must), or it is a numeric item with *P*s in its PICTURE character-string (which it must not have).

277 (Error)

```
Expected display data item, not edited nor justified
```

Cause One of is true about the data item:

- It does not have USAGE DISPLAY.
- It is an alphanumeric-edited or numeric-edited item.
- It is described with a JUSTIFIED clause.

278 (Error)

```
Expected alphanumeric data item
```

Cause The data item is not alphanumeric, as it must be.

279 (Error)

```
Expected numeric or numeric-edited data item
```

Cause The data item is neither numeric nor numeric-edited, as it must be.

280 (Error)

```
Expected numeric data item
```

Cause The data item is not numeric, as it must be.

281 (Error)

`Expected integer numeric data item`

Cause The data item is not numeric without fraction digits, as it must be.

282 (Error)

`Expected unscaled numeric data item`

Cause Either the data item is not numeric without fraction digits, or it has Ps in its PICTURE character-string.

283 (Error)

`Expected index data item or integer numeric data item`

Cause The data item does not have USAGE INDEX and is not a numeric item without fraction digits.

284 (Error)

`Expected group data item`

Cause The data item is not a data structure, as it must be.

285 (Error)

`Expected data file record data item`

Cause The data item is not a level-01 item subordinate to a data file description entry (level indicator FD), as it must be.

286 (Error)

`Expected sort-merge file record data item`

Cause The data item is not a level-01 item subordinate to a sort-merge file description entry (level indicator SD), as it must be.

287 (Error)

`Expected level 01 or level 77 or elementary data item`

Cause One of:

- The FIPS directive specified NONSTANDARDEXT, but the data item is not a level-01 item, a level-77 item, or an elementary item of a record.
- No FIPS directive specified NONSTANDARDEXT, but the data item is not elementary, level-01, level-77, or at a level other than 01 or 77 and aligned on a 2-byte boundary.

If the data item is subscripted, its first occurrence must be on a 2-byte boundary and the number of occurrences must be even.

288 (Error)

`Extended storage data item not permitted`

Cause The data item is either defined in the Extended-Storage Section or described with the ACCESS MODE EXTENDED-STORAGE clause.

289 (Error)

`Index data item not permitted`

Cause A reference to an index data item cannot appear in this context.

290 (Error)

`Alphabetic data item not permitted`

Cause The receiving data item is described as alphabetic when the data source for an ACCEPT statement is DATE, DAY, DAY-OF-WEEK, or TIME.

291 (Error)

`Numeric-edited data item not permitted`

Cause A numeric-edited data item is specified as an arithmetic operand.

292 (Error)

`Improper context for subscripts`

Cause A subscript is in the specification of either the table operand in a SEARCH statement or the key operand in a START statement.

293 (Error)

`Too many or too few subscripts`

Cause The reference to a table item or condition-name associated with a table item does not have the correct number of subscripts.

294 (Error)

`Table item not permitted in subscript`

Cause A reference to a table item appears as a subscript.

295 (Error)

`Index-name not associated with table level`

Cause The index-name appears in a reference whose subject is not described by the OCCURS clause that defines the index-name.

296 (Error)

`Reference modifier permitted only for display data item`

Cause A reference modifier is specified, but the subject of the reference is not a data-name described with USAGE DISPLAY.

297 (Error)

`Improper context for reference modifier`

Cause A reference modifier cannot be specified for the data item indicated.

298 (Error)

`Expected arithmetic operand`

Cause The operand of an arithmetic operator is not a numeric literal, a numeric data item, or an arithmetic expression, as it must be.

299 (Warning)

`Division by zero`

Cause The value of a divisor is 0.

300 (Error)

`Expected relation operand`

Cause The operand of a relational operator is not a literal, index-name, data item, or arithmetic expression, as it must be.

301 (Error)

`Have no implicit operand for relation`

Cause There is no implicit left-hand operand to complete the relation.

302 (Error)

`Relation operands not compatible`

Cause One of:

- An arithmetic expression was compared with something other than another arithmetic expression, a numeric data item, or a numeric literal.
- An index-name was compared with something other than another index-name, an index data item, a numeric data item, or a numeric literal.
- An index data item was compared with something other than another index data item or an index-name.
- A pointer (a data item with USAGE POINTER, or ADDRESS OF *`data-item`*, or NULL) can be compared only with another pointer.

303 (Error)

`No variable among relation operands`

Cause A relation includes no references to data items; it is constructed solely from literal operands.

304 (Error)

`Expected class-name`

Cause A name referenced in a context where it must identify a class-name does not.

305 (Error)

`Improper operand for class test`

Cause The subject of a class test is not an identifier, as it must be.

306 (Error)

`Improper item for alphabetic or class-name test`

Cause The subject of a class-name such as ALPHABETIC or NOT ALPHABETIC is numeric.

307 (Error)

`Improper item for numeric test`

Cause The subject of a NUMERIC or NOT NUMERIC test is either alphabetic or is a data structure that has a subordinate described with an embedded operational sign.

308 (Error)

`Expected condition as operand`

Cause The operand indicated appears in a context that requires a simple or complex condition, such as the operand of an AND, OR, or NOT operator or the operand of a WHEN phrase.

309 (Error)

`Expected data file`

Cause Only a file name defined in a data file description entry (level indicator FD) is permitted in this context.

310 (Error)

```
Expected sort-merge file
```

Cause Only a file name defined in a sort-merge file description entry (level indicator SD) is permitted in this context.

311 (Warning)

```
No corresponding pairs
```

Cause The two data structures specified in a CORRESPONDING operation contain no corresponding subordinates.

312 (Error)

```
Phrase permitted only for sequential organization
```

Cause One of these phrases is used on a file that does not have sequential organization:

- ADVANCING
- NO REWIND
- PROMPT
- REEL
- UNIT

313 (Error)

```
FROM or INTO data item overlaps file record
```

Cause The data item specified in the INTO or FROM phrase overlaps part of the file record area.

314 (Error)

```
Expected mnemonic-name associated with a file name
```

Cause The referenced operand is not a mnemonic-name associated with an external file name in the SPECIAL-NAMES paragraph.

315 (Error)

```
Permitted only when NONSTOP directive specified
```

Cause The program has a CHECKPOINT or STARTBACKUP statement, but was not compiled with the NONSTOP directive.

316 (Error)

```
Improper context for ROUNDED
```

Cause ROUNDED appears in the REMAINDER phrase of a DIVIDE statement.

317 (Error)

```
DELETE not permitted for sequential organization
```

Cause The file name specified in a DELETE statement identifies a file with sequential organization.

318 (Error)

```
Improper context for ALSO phrase
```

Cause The ALSO phrase appears in a SEARCH statement.

319 (Error)

```
Improper context for condition
```

Cause A condition appears in an object that includes the THROUGH phrase.

320 (Error)

```
Range operands not compatible
```

Cause Operands in an object that includes the THROUGH phrase are not of the same class (alphabetic, alphanumeric, numeric).

321 (Error)

```
Too many subject operands
```

Cause An EVALUATE statement has more than 127 subjects.

322 (Error)

```
Too many object operands
```

Cause The number of objects in the WHEN phrase is greater than the number of subjects in the EVALUATE statement.

323 (Error)

```
Too few object operands
```

Cause The number of objects in the WHEN phrase is less than the number of subjects in the EVALUATE statement.

324 (Error)

```
Corresponding operands not compatible
```

Cause The object in the WHEN phrase is incompatible with the corresponding subject in the EVALUATE statement.

325 (Error)

```
No variable among corresponding operands
```

Cause Both the object in the WHEN phrase and the corresponding subject in the EVALUATE statement contain only literal operands.

326 (Error)

```
Variable occurrences or 66 level data item not permitted
```

Cause One of:

- An OCCURS clause that includes the DEPENDING phrase appears in the description of a receiving operand in the INITIALIZE statement or in the description of any subordinate data item.
- A receiving operand of the INITIALIZE statement has level-number 66.

327 (Error)

```
Receiving category not compatible with sending category
```

Cause The receiving operand cannot be assigned a value derived from the sending operand; for example:

- SPACE (or one of its equivalents), an alphabetic data item, or an alphanumeric-edited data item cannot be assigned to a numeric or numeric-edited data item.
- ZERO (or one of its equivalents), a numeric literal, a numeric data item, or a numeric-edited data item cannot be assigned to an alphabetic data item.

This message usually results from a MOVE statement. It can also result from statements whose execution includes assignment operations that follow MOVE statement rules.

328 (Error)

```
Receiving operand not compatible with sending operand
```

Cause The attributes of the receiving operand do not permit assigning it a value derived from the sending operand; for example:

- A numeric data item is being set to the value of a nonnumeric literal, nonnumeric data item, or index data item.
- An index data item is being set to the value of a nonnumeric literal or nonnumeric data item.
- A data item other than an index-name is being set UP or DOWN.
- A data item other than a mnemonic-name associated with an external switch is being set ON or OFF.
- A data item other than a condition-name associated with a data item is being set to TRUE or FALSE.

329 (Error)

```
INSPECT operand sizes not compatible
```

Cause One of:

- The replaced and replacing operands have incompatible sizes. When replacing CHARACTERS or a figurative constant, the size of the replacing operand must be one character. When replacing a numeric or nonnumeric literal or a data item, the size of the replacing operand must be the same as that of the replaced operand.
- The CONVERTING phrase operands are not the same size.

330 (Error)

```
NO REWIND phrase not permitted for I-O or EXTEND mode
```

Cause The NO REWIND phrase appears when the open mode is I-O or EXTEND.

331 (Error)

```
EXTEND mode permitted only for sequential access
```

Cause EXTEND mode is specified for a file access whose access is not sequential.

332 (Error)

```
EXTEND mode not permitted for MULTIPLE FILE TAPE file
```

Cause EXTEND mode is specified for a file name that is specified in a MULTIPLE FILE TAPE clause.

333 (Error)

```
EXTEND mode not permitted for LINAGE file
```

Cause EXTEND mode is specified for a file name that is described with a LINAGE clause.

334 (Error)

```
Too many AFTER phrases
```

Cause A PERFORM statement has more than 6 AFTER phrases.

335 (Error)

```
NEXT/REVERSED not compatible with random access
```

Cause The NEXT phrase is used with a READ statement that specifies a file name described with random access.

336 (Error)

```
PROMPT data item overlaps file record
```

Cause The data item specified in the PROMPT phrase overlaps the file record area, but the data item and file record area do not begin at the same character position.

337 (Warning)

```
PROMPT data item exceeds record size
```

Cause The size of the PROMPT phrase operand exceeds the maximum record size for the file. (The excess characters will not appear in the prompt.)

338 (Error)

```
Operands not compatible for INTO assignment
```

Cause One of:

- The record descriptions of the file referenced in this statement require that the INTO phrase operand be alphanumeric, and it is not.
- The record descriptions of the file referenced in this statement prohibit specification of the INTO phrase.

339 (Warning)

```
No editing occurs for INTO assignment
```

Cause The value of the INTO phrase operand is assigned according to the rules for a group move; thus, no editing operations occur.

340 (Error)

```
Expected table item with index-names
```

Cause The description of the table operand in a SEARCH statement has an OCCURS clause that does not include the INDEXED phrase.

341 (Error)

```
Expected table item with keys
```

Cause The description of the table operand in a SEARCH ALL statement has an OCCURS clause with no KEY phrases.

342 (Error)

```
VARYING phrase not permitted with ALL
```

Cause The VARYING phrase is appears in a SEARCH ALL statement.

343 (Error)

```
Not permitted in declaratives
```

Cause A SORT or a MERGE statement appears in the Declaratives Portion of the Procedure Division.

344 (Error)

```
Key within table or has variable size
```

Cause A reference in an ASCENDING KEY or DESCENDING key phrase identifies a data item that has a variable size or is referenced by an OCCURS clause.

345 (Error)

```
Key not found within sort-merge file record
```

Cause A reference in an ASCENDING KEY or DESCENDING key phrase identifies a data item defined in the record descriptions of the sort-merge file.

346 (Error)

```
Key not contained within minimum sort-merge file record
```

Cause The key data item is not completely contained within the shortest possible record of the sort-merge file.

347 (Error)

```
Prime record key inconsistent with major sort-merge key
```

Cause A GIVING phrase specifies an indexed file, but the major sort-merge key does not have the ASCENDING attribute or does not occupy the same character positions within the sort-merge file records as the prime key item occupies within the indexed file records.

348 (Error)

```
Record size incompatible with sort-merge file description
```

Cause One of:

- The longest possible record of a USING file exceeds the record capacity of the sort-merge file.
- The longest possible record of the sort-merge file exceeds the record capacity of a GIVING file.

349 (Error)

```
Random access not permitted
```

Cause A MERGE, SORT, or START statement specifies a file name described with random access.

350 (Error)

`File set conflict from` *`xxxx`*

Cause A SORT or MERGE statement specifies conflicting files. *xxxx* is one of these phrases:

Phrase	Meaning
duplicate file names	The same file name appears more than once in a MERGE statement.
SAME AREA clause	Two or more file names specified in one SAME AREA clause appear in a MERGE statement or in the GIVING phrase of a SORT statement.
SAME RECORD clause	Two or more file names specified in one SAME RECORD clause appear in the USING phrase of a MERGE statement.
MULTIPLE FILE TAPE clause	Two or more file names specified in one MULTIPLE FILE TAPE clause appear in a MERGE statement or in the GIVING phrase of a SORT statement.

351 (Error)

```
KEY phrase not permitted for sequential access
```

Cause The KEY phrase applies to a file whose access is not sequential.

352 (Error)

```
KEY phrase not compatible with NEXT phrase
```

Cause The KEY phrase appears with a NEXT phrase.

353 (Error)

```
KEY item not found within file record
```

Cause The reference in the KEY phrase does not identify a key of the specified file.

354 (Error)

```
Length of combined keys exceeds 253 characters
```

Cause In a file-control entry with multiple keys, the sum of the lengths of the keys exceeds 253 characters.

355 (Error)

```
File has no default prime key
```

Cause The KEY phrase appears in a statement other than a START statement, and the file is neither indexed nor described with a RELATIVE KEY phrase.

356 (Error)

```
Improper relation for this context
```

Cause One of:

- The operator in the KEY phrase is not equal sign (=), greater-than sign (>), greater-than-or-equal sign (>=), or one of their equivalents.
- The POSITION or GENERIC phrase is present, and the operator in the KEY phrase is not the equal sign (=) or EQUAL.

357 (Error)

```
Expected file key or equivalent
```

Cause The operand in the KEY phrase is neither a data item described as a file key, nor a data item whose leftmost character position coincides with the leftmost character position of a file key and whose length is less than or equal to the length of that file key.

358 (Error)

```
Position data item must be prime key
```

Cause The operand in the POSITION phrase is not the file's prime key. (The prime key of a relative file is defined by the RELATIVE KEY clause; the prime key of an indexed file is defined by the RECORD KEY clause.)

359 (Error)

```
ADVANCING mnemonic-name not compatible with LINAGE
```

Cause The ADVANCING mnemonic-name phrase is used for a file described with the LINAGE clause.

360 (Error)

```
Operand too large
```

Cause The size of the operand exceeds the maximum supported for its context.

361 (Error)

```
Improper object file name
```

Cause A reference to a routine or COBOL program specifies the object file from which it must be selected, but the external file name associated with the specified mnemonic-name does not identify a disk-resident file.

362 (Error)

`Inconsistent object file reference`

Cause One of:

- Not all of the CALL and CANCEL statements specifying a particular COBOL agree about the object file from which it must be selected.
- Not all of the ENTER statements specifying a particular routine agree about the object file from which it must be selected.

363 (Error)

`Improper language name`

Cause The *`language`* in an ENTER statement must be C or TAL.

364 (Error)

`Improper unit-name for language`

Cause One of:

- The program-name does not conform to the spelling rules of COBOL.
- The routine-name does not conform to the spelling rules of its language.

365 (Error)

`Unit of proper language not found`

Cause One of:

- Either the requested COBOL program does not exist in the specified object file, or an entity identified by the program-name exists but is not a COBOL program.
- Either the requested routine does not exist in the specified object file, or an entity identified by the routine-name exists but is not a routine of the proper language.

If the called program is a system routine, the called program might not be in the COBOLEXT or NMCOBEXT file, and you must compile the program with the one of these directives:

- `CONSULT $SYSTEM.SYSTEM.COBOLEX0`
- `CONSULT $SYSTEM.SYSTEM.NMCOBEX0`

366 (Error)

`Inconsistent language reference for unit-name`

Cause Not all of the ENTER statements specifying a particular routine agree about its language.

367 (Error)

`Unit has invalid attribute`

Cause An called routine has an invalid parameter.

369 (Warning)

`NONSTOP attribute conflict`

Cause One of:

- The source text from which the indicated COBOL program was compiled included the NONSTOP directive, but the current source text does not.
- The current source text includes a NONSTOP directive, but the source text from which the indicated COBOL program was compiled did not.
- The program searched an import library that includes a COBOL program that was compiled with the directive NONSTOP (page 566).

  The compiler assumes that programs in an import library were compiled without the NONSTOP directive. If this is not true, the compiler issues warning 369, which you can ignore.

Run-time problems might occur if proper operation of the program depends on the run unit's executing as a process pair.

370 (Error)

```
Inconsistent parameter attributes
```

Cause One of:

- Different CALL statements referencing the same COBOL program specify a different number of parameters in their USING phrases.
- The number of parameters defined in the Procedure Division of a COBOL program differs from the number of parameters expected by the callers.
- For one or more parameters of a COBOL program, the access mode (STANDARD or EXTENDED-STORAGE) defined in the program differs from the access mode expected by the callers.

371 (Error)

```
Formal parameter not a variable
```

Cause The formal parameter of the called routine is neither a data variable nor the word OMITTED.

372 (Error)

```
Actual parameter is a file, so formal parameter must be a struct
```

Cause The actual parameter of an ENTER statement is a file name (FD or SD), but it does not correspond to a formal parameter that expects a struct.

373 (Warning)

```
Potential odd byte address converted to even byte address
```

Cause The compiler cannot determine if a problem exists. Verify that the correct address is passed to the routine.

Most COBOL data items are referenced internally by byte addresses. When one of these appears as the actual parameter corresponding to the formal parameter of a routine that expects a 2-byte address, the compiler must convert the item's byte address to a 2-byte address. If the data item begins on an odd-byte boundary, that information is lost by the conversion operation (thus the value space of the item appears, to the called routine, to begin one byte before it actually does). If the data item begins on an even-byte boundary, there is no problem.

374 (Error)

```
OMITTED permitted only for extensible or variable routine
```

Cause OMITTED appears as an actual parameter, but the called routine does not have the EXTENSIBLE or VARIABLE attribute.

375 (Error)

```
GIVING phrase permitted only for function routine
```

Cause An ENTER statement includes the GIVING phrase, but the called routine is not a function (that is, it does not return a value).

376 (Error)

```
Routine type not supported
```

Cause The routine returns a value of a type that HP COBOL does not support; therefore, it cannot have a GIVING phrase.

377 (Warning)

```
Reference with mnemonic-name resolved to contained program
```

Cause References to a COBOL program specify an object file from which it should be selected, but because the program-name identifies an accessible contained program, the compiler resolves the references to that program. Examine the program logic to verify that this result is acceptable.

378 (Warning)

```
Expected digit string as sending operand
```

Cause A nonnumeric operand moved to a numeric operand has a value not composed entirely of digits. The result of the assignment is not defined.

379 (Error)

```
Formal parameter type not supported
```

Cause One of:

- A formal parameter has a type that is not supported.
- A value parameter was passed, but the corresponding formal parameter's type is not numeric or character.
- An actual parameter was omitted, but the corresponding formal parameter is not a type that can be omitted.

380 (Error)

```
Too many procedure names
```

Cause The number of procedure names in the statement exceeds the maximum number allowed in the statement.

381 (Error)

```
PROMPT/LOCK phrase cannot be specified with REVERSED
```

Cause A READ statement with a REVERSED phrase also has a PROMPT or LOCKED phrase.

382 (Error)

```
Undefined procedure reference
```

Cause One of:

- The name indicated appears as a procedure reference, but does not identify a paragraph-name or section-name defined within the Procedure Division of the current source program.
- The name indicated appears in a procedure reference with a qualifier, but does not identify a paragraph-name defined both as a subordinate of the section identified by the qualifier and within the Procedure Division of the current source program.

383 (Error)

```
Ambiguous procedure reference
```

Cause The name indicated appears in a procedure reference but does not identify a unique object within the current source program.

384 (Error)

```
Improper ALTER subject
```

Cause A procedure reference that appears as an ALTER statement subject does not identify a paragraph or section containing only a single unconditional GO TO statement.

385 (Warning or Error)

```
Improper ALTER context
```

Cause **Warning:** An ALTER statement modifies the GO TO statement of a paragraph or section in an independent segment. HP COBOL does not support the semantics for independent segments described in the ISO/ANSI COBOL standard (it does not restore the original specification of the GO TO statement if control leaves the independent segment and then returns to it later).

Error: An ALTER statement appearing in one independent segment modifies a GO TO statement of a paragraph or section in another independent segment.

386 (Warning or Error)

```
Improper ALTER scope
```

Cause **Warning:** One of:

- An ALTER statement in the nondeclarative portion of the Procedure Division references a paragraph-name or section-name defined in a declarative procedure.
- An ALTER statement in a declarative procedure references a paragraph-name or section-name defined in another declarative procedure.
- An ALTER statement modifies a GO TO statement so that its execution will transfer control from one declarative procedure to another declarative procedure.

Error: One of:

- An ALTER statement in a declarative procedure references a paragraph-name or section-name defined in the nondeclarative portion of the Procedure Division.
- An ALTER statement in a nondebugging declarative procedure references a paragraph-name or section-name defined in a debugging declarative procedure.
- An ALTER statement modifies a GO TO statement so that its execution will transfer control into or out of the Declaratives Portion the Procedure Division.
- An ALTER statement modifies a GO TO statement so that its execution will transfer control between a debugging declarative procedure and a nondebugging declarative procedure.

387 (Warning or Error)

```
Improper GO TO scope
```

Cause **Warning:** A GO TO statement transfers control from one declarative procedure to another declarative procedure.

Error: One of:

- A GO TO statement transfers control between the declarative and nondeclarative portions of the Procedure Division.
- A GO TO statement transfers control between a debugging declarative procedure and a nondebugging declarative procedure.

388 (Warning or Error)

```
Improper PERFORM scope
```

Cause **Warning:** A PERFORM statement specifies a range that includes two references. One identifies a paragraph-name or section-name defined in a declarative procedure. The other identifies a paragraph-name or section-name defined in another declarative procedure. It is recommended that both references identify items in the same declarative procedure.

Error: One of:

- A PERFORM statement in an independent segment references a paragraph-name or section-name defined in another independent segment.
- A PERFORM statement in a declarative procedure references a paragraph-name or section-name defined in the nondeclarative portion of the Procedure Division.
- A PERFORM statement references a paragraph-name or section-name defined in a debugging declarative procedure, but the statement itself is not in a debugging declarative procedure.
- When the range specified in a PERFORM statement includes two references, they must identify compatible paragraph-names or section-names. If one is defined in the Declaratives Portion of the Procedure Division, the other must also be defined in the Declaratives Portion. If one is defined in an independent segment, the other must be defined in the same independent segment.

389 (Warning or Error)

```
Improper SORT/MERGE scope
```

Cause **Warning:** An INPUT PROCEDURE or OUTPUT PROCEDURE phrase specifies a range that includes two references. One identifies a paragraph-name or section-name defined in a declarative procedure. The other identifies a paragraph-name or section-name defined in a different declarative procedure.

Error: One of:

- An INPUT PROCEDURE or OUTPUT PROCEDURE phrase of a SORT or MERGE statement in an independent segment references a paragraph-name or section-name defined in a different independent segment.
- An INPUT PROCEDURE or OUTPUT PROCEDURE phrase of a SORT or MERGE statement specifies a range that includes two references, and they identify incompatible paragraph-names or section-names. Either one reference is defined in the Declaratives Portion of the Procedure Division and the other is not, or one is defined in one independent segment and the other is defined in another independent segment.

390 (Warning)

```
Referenced program not found by compiler
```

Cause A CALL or CANCEL statement referenced a program that the compiler could not find in the source text or in any file on any search list. These programs must be bound into the run unit before it is executed.

The compiler issues this warning at the point of reference to the program unit (that is, at a CALL or CANCEL statement); however, for any given missing program, the compiler issues the warning only at the first point of reference to that program.

391 (Error)

```
Improper program-name specified
```

Cause One of:

- A CALL or CANCEL statement specifies the program-name of the separately compiled program within which it appears.
- The END PROGRAM statement identifies a program that has already been terminated.

392 (Error)

```
Expected current program-name
```

Cause The END PROGRAM statement specifies a name that does not match any program-name within the separately compiled program.

393 (Error)

Missing END PROGRAM: *xxxx*

Cause No END PROGRAM statement appeared for the contained program whose program-name is *xxxx*.

395 (Error)

```
More than one implicit MAIN program
```

Cause Both the current COBOL program and a preceding one qualify for the MAIN attribute. Because the compiler cannot determine which one should be the main program, it does not produce an object file.

396 (Warning)

```
Condition-name has too many values to be referenced by INSPECT
```

Cause A condition-name has more values than the compiler can send to the Inspect debugger. The compiler sends only the first value. To make all of the values available to the Inspect debugger, replace the condition-name with several condition names, each with a less demanding VALUE clause, and reference all of them wherever the original condition-name is currently referenced.

397 (Error)

```
Program data space overflow
```

Cause The data items defined in the File and Working-Storage Sections of a source program need more program data space than is available for the entire run unit.

398 (Error)

```
Program extended data space overflow
```

Cause The data items defined in the Extended-Storage Section of a source program need more program extended data space than is available for the entire run unit.

399 (Error)

```
Program control space overflow
```

Cause The objects defined in a source program need more run-time program control space than is available for the entire run unit.

400 (Error)

```
Program code exceeds 65500 words
```

Cause The object code generated for a separately compiled program exceeds 131 KB, which is the maximum size supported.

Recovery Recompile the program without the CHECK directive and with the NOBLANK and LESS-CODE 1 directives. If error 400 still occurs, change the source program:

- Remove unused code.
- Break the program into two or more compilation units (see Compilation Units (page 523)).
- Find sequences of code that occur several times and make each of them a procedure (see Procedures (page 247) and Procedure Execution (page 252)). To transfer control to a paragraph or procedure, use the statement PERFORM (page 399).
- Change level-77 data items and elementary level-01 data items that are numeric from USAGE DISPLAY to USAGE COMP (see USAGE Clause (page 214)).

401 (Error)

```
Embedded program code exceeds 32767 words
```

Cause The object code generated for a contained program exceeds 65,534 bytes, which is the maximum size supported.

402 (Error)

```
Reference modifier first character out of range
```

Cause In a reference modifier, the value of the first character position exceeds the size of the subject data item.

403 (Error)

```
Reference modifier length out of range
```

Cause In a reference modifier, the value of the last character position *first-character-position* + *length* - 1 exceeds the size of the subject data item.

404 (Warning)

```
Size error on literal expression
```

Cause The compiler has determined that this expression, composed solely of literal operands, will always generate the size error condition during execution. If the associated statement includes the SIZE ERROR clause, the program takes the size error branch; otherwise, an arithmetic overflow can result, or (if the receiving item is of USAGE COMPUTATIONAL) the program can store a number larger than the maximum value allowed for the item, leading to an arithmetic overflow later in the execution.

405 (Warning)

```
Relation truth value is constant
```

Cause The compiler has determined that the value of this relation condition will be always TRUE or always FALSE. (Example: a literal is compared to a shorter data item.) This message might point only to the statement in error, not to the erroneous line of the statement.

406 (Warning)

```
SET statement literal not in subscript range
```

Cause A value of a numeric literal specified as the sending operand in a SET statement is greater than the maximum occurrence number defined for assignment to an index-name specified as a receiving operand. Execution of the statement will assign an improper value to the index-name.

407 (Warning)

```
Improper SEARCH ALL condition - serial search used
```

Cause A SEARCH ALL statement specifies a condition that does not conform to the rules of the COBOL language. HP COBOL employs a serial rather than a binary search technique in this situation.

408 (Error)

```
Program buffer space exceeds 31000 words
```

Cause The sum of the buffer space needed for files exceeds 62 KB. (Reduce the size of blocks specified in the BLOCK CONTAINS clause in one or more File Description entries or reduce the number of files.)

425 (Error)

```
Usage clause not permitted with specified picture
```

Cause A national data item contains a USAGE clause.

426 (Error)

```
Synchronized clause not permitted with specified picture
```

Cause A national data item contains a SYNCHRONIZED clause.

427 (Error)

```
Multi-byte not permitted in this context
```

Cause A national data item or national literal appears where it is not permitted.

428 (Error)

```
Literal and data item must both be multi-byte
```

Cause A literal or data item that is not in the national class appears where all the literals and data items must be in the national class.

429 (Error)

```
Can only compare multi-byte type to multi-byte type
```

Cause In a conditional statement, a national data item or national literal is compared to something other than another national data item or national literal.

430 (Error)

```
Expecting multi-byte data item or literal
```

Cause A literal or data item that is not in the national class appears where all the literals and data items must be in the national class.

440 (Error)

```
External file not allowed in library object
```

Cause A program in a user library references an external file.

441 (Error)

```
External data item not allowed in library object
```

Cause A program in a user library references an external data item.

442 (Error)

```
Library object cannot contain a main program
```

Cause A main program is in a user library.

443 (Error)

```
Library object can only contain initial programs
```

Effect A program that is not an initial program is in a user library.

444 (Error)

`Embedded programs are not allowed in library object`

Cause An embedded program is in a user library.

445 (Error)

`Global data item not allowed in library object`

Cause A data item with the GLOBAL attribute is in a user library.

446 (Error)

`COLLATING SEQUENCE clause not allowed in library`

Cause A program in a COPY library or user library has a PROGRAM COLLATING SEQUENCE clause in its Environment Division.

447 (Error)

`Global file not allowed in library object`

Cause A file whose COBOL file name has the GLOBAL attribute is in a user library.

448 (Error)

`EXTENDED-STORAGE not allowed in library object`

Cause A program with an EXTENDED-STORAGE section is in a user library.

450 (Error)

`COBOLSPOOLOPEN allowed only in old environment`

Cause An object file contains a call to the COBOLSPOOLOPEN routine.

461 (Error)

`Internal SQL error`

Cause The compiler's internal consistency checker discovered a logic error. Please report this failure to your service provider.

462 (Error)

`Improper usage clause for host variable`

Cause An SQL/MP or SQL/MX host variable has an improper USAGE clause.

463 (Error)

`Host variable picture string cannot contain 'P'`

Cause An SQL/MP or SQL/MX host variable's PICTURE string contains *P*.

464 (Error)

`Improper category for host variable`

Cause An SQL/MP or SQL/MX host variable is in the wrong category.

465 (Error)

`Host variable's picture string cannot contain '*'`

Cause An SQL/MP or SQL/MX host variable's PICTURE string contains an asterisk (*).

466 (Error)

`JUSTIFIED clause not allowed for host variables`

Cause An SQL/MP or SQL/MX host variable contains a JUSTIFIED clause.

467 (Error)

```
OCCURS clause not allowed for host variables
```

Cause An SQL/MP or SQL/MX host variable contains a OCCURS clause.

468 (Error)

```
The RENAMES is ignored
```

Cause An SQL/MP or SQL/MX host variable contains a RENAMES clause.

470 (Error)

```
Missing END-EXEC
```

Cause The compiler processed the maximum number of SQL/MP or SQL/MX statement lines without encountering the SQL/MP or SQL/MX statement terminator END-EXEC.

471 (Error)

```
Ambiguous reference to SQLCA
```

Cause More than one SQLCA is declared in the program, and the SQLCA in this statement is ambiguous in this context.

472 (Error)

```
SQLCA is declared incorrectly
```

Cause SQLCA is declared incorrectly (it must be a data structure).

473 (Error)

```
Ambiguous reference to SQLCODEX
```

Cause More than one SQLCODEX is declared in the program, and the SQLCODEX in this statement is ambiguous in this context.

474 (Error)

```
SQLCODEX is declared incorrectly
```

Cause SQLCODEX is declared incorrectly (it must be a COMPUTATIONAL data item with no editing characters).

475 (Error)

```
Ambiguous reference to SQLCODE
```

Cause More than one SQLCODE is declared in the program, and the SQLCODE in this statement is ambiguous in this context.

476 (Error)

```
SQLCODE is declared incorrectly
```

Cause SQLCODE is declared incorrectly (it must be a COMPUTATIONAL data item with no editing characters).

477 (Error)

```
Program is missing SQLCODE
```

Cause SQLCODE is required in the program, but is missing.

478 (Error)

```
Ambiguous reference to SQLSA
```

Cause More than one SQLSA is declared in the program, and the SQLSA in this statement is ambiguous in this context.

479 (Error)

```
SQLSA is declared incorrectly
```

Cause SQLSA is declared incorrectly (it must be a COMPUTATIONAL data item with no editing characters).

480 (Error)

```
Improper context for INCLUDE
```

Cause An SQL/MP or SQL/MX INCLUDE statement is outside of the Data Division. (It must be inside.)

481 (Error)

```
Improper context for CHARACTER SET clause
```

Cause A CHARACTER-SET clause appears somewhere other than the final clause in an OBJECT-COMPUTER paragraph.

482 (Error)

```
Expecting elementary data item
```

Cause A data structure appears where an elementary data item is expected.

490 (Error)

```
Improper context for ALL subscript
```

Cause The ALL subscript applied to an argument to an intrinsic function is either not allowed in this context or not allowed on any argument to this function.

491 (Error)

```
Function nesting too deep
```

Cause Functions are nested so deeply within a single statement that the compiler cannot process them.

492 (Error)

```
Subscripting not allowed on a function
```

Cause A subscript appears on a function. (A subscript can only appear on a table.)

493 (Error)

```
Improper context for function
```

Cause An intrinsic function appears where it is not allowed (as a receiving item in a statement, for example).

494 (Error)

```
Unknown function
```

Cause The compiler does not recognize the function name.

495 (Error)

```
Expression is too complex
```

Cause An expression is so complex that the compiler cannot process it.

496 (Error)

```
Improper argument
```

Cause The argument has at least one wrong characteristic (such as type, class, category, or length).

497 (Error)

```
Too many arguments
```

Cause More arguments have been supplied to the function than the function takes.

498 (Error)

```
Argument not within expected range
```

Cause The literal being used as an argument to the function is outside the range allowed for that argument.

499 (Error)

```
Too few arguments
```

Cause The function requires more arguments than have been supplied.

500 (Error)

```
Expecting nonnumeric item
```

Cause A nonnumeric argument was supplied where a numeric argument is required.

502 (Error)

```
Expected alphabetic or alphanumeric data item
```

Cause A nonalphabetic, nonalphanumeric argument was supplied where an alphabetic or alphanumeric argument is required.

503 (Error)

```
Expected alphabetic data item
```

Cause A nonalphabetic argument was supplied where an alphabetic argument is required.

504 (Error)

```
Reference modifier permitted only for alphanumeric functions
```

Cause A reference modifier was applied to a function whose return type is not alphanumeric.

510 (Error)

```
EXIT PERFORM [CYCLE] must be within an in-line perform
```

Cause An EXIT PERFORM or EXIT PERFORM CYCLE statement appears in a context other than within an inline PERFORM statement.

511 (Error)

```
Expected key-word CYCLE or other verb
```

Cause The keyword PERFORM is followed by something other than the keyword CYCLE or another verb.

512 (Error)

```
Expected key-word PARAGRAPH
```

Cause The keyword EXIT is followed by something other than the keyword PARAGRAPH.

513 (Error)

```
An EXIT PARAGRAPH must be contained in a paragraph
```

Cause An EXIT PARAGRAPH statement is not inside a paragraph. (You must be within a paragraph to exit from one.)

514 (Error)

```
An EXIT SECTION must be contained in a section
```

Cause An EXIT SECTION statement is not inside a section. (You must be within a section to exit from one.)

515 (Error)

```
Expected YYYYMMDD after DATE or YYYYDDD after DAY
```

Cause In the ACCEPT statement, either DATE appears followed by something other than YYYYMMDD or DAY appears followed by something other than YYYYDDD.

520 (Error)

```
Possible odd byte address for parameter
```

Cause A data structure that could start on an odd byte was passed as a parameter to a CALL statement. If the data structure actually does start on an odd byte, the wrong address will be passed to the CALL statement. Certain subscript combinations cause this problem.

521 (Error)

```
Parameter must be passed BY REFERENCE to a FORTRAN routine
```

Cause An ENTER statement tried to pass a parameter by value to a FORTRAN routine. This parameter must be passed by reference.

522 (Error)

```
Parameter must be passed BY REFERENCE to a C function
```

Cause An ENTER statement tried to pass a parameter by value to an HP C function. This parameter must be passed by reference.

525 (Error)

```
This directive is not permitted in the OSS environment
```

Cause The program was compiled in the OSS environment with one of these compiler directives:

- ENV
- NONSTOP
- SAVE
- SUBTYPE

526 (Error)

```
This phrase is not permitted for LINE SEQUENTIAL files
```

Cause The source program applies one of these phrases to a LINE SEQUENTIAL file:

- In a SELECT clause:
  - ACCESS MODE IS DYNAMIC
  - ACCESS MODE IS RANDOM
  - ALTERNATE RECORD KEY

  - PADDING CHARACTER
  - RECORD DELIMITER
- In the I-O-CONTROL paragraph:
  - MULTIPLE FILE
- In a file description entry:
  - BLOCK CONTAINS
  - CODE-SET
  - LINAGE
- In an OPEN statement:
  - I-O
  - REVERSED
  - SYNCDEPTH
  - TIME LIMITS
- In a READ statement:
  - PROMPT
  - REVERSED
  - TIME LIMITS
- ADVANCING in a WRITE statement

527 (Error)

`LINE SEQUENTIAL files may not be opened in I-O mode`

Cause The program contains a statement of the form

`OPEN I-O` *`i-o-file-description`*

where *`i-o-file-description`* is a LINE SEQUENTIAL file.

600 (Warning)

`Defunct directive ignored`

Cause The compiler ignores this directive.

601 (Warning)

`Feature not yet available -- directive ignored`

Cause The compiler ignores this directive.

602 (Error)

`No symbols for external name`

Cause A CALL or ENTER statement calls a program for which the object file contains a name but not a symbol. (If a native HP COBOL program references an object in a CALL or ENTER statement, the object must have been compiled with symbols.)

603 (Error)

`External name not found`

Cause The program called by a CALL or ENTER statement is not in the specified object file (if an object file is specified) or is not in any object file on the search list(s).

604 (Error)

`Error reading object file`

Cause One of:

- A CALL or ENTER statement calls a program, specifying the object file that contains the program, but that object file cannot be read.
- A CALL or ENTER statement calls a program without specifying the object file that contains it, and one of the object files on the search list(s) cannot be read.

605 (Error)

`The NO prefix is not allowed on this directive`

Cause The compiler does not accept this directive, although it does accept its opposite (for example, the compiler accepts CONSULT, SEARCH, and SQL, but not NOCONSULT, NOSEARCH, or NOSQL).

606 (Warning)

`ACCESS MODE STANDARD is ignored for a non-parameter data item in the linkage section`

Cause ACCESS MODE STANDARD is specified for a data item described in the Linkage Section, and the data item is not referenced in the USING phrase of the PROCEDURE DIVISION header. Delete the ACCESS MODE phrase.

607 (Error)

`Numeric hexadecimal literal exceeds 16 hex digits`

Cause A hexadecimal literal has more than 64 bits (16 hexadecimal digits).

608 (Error)

`More than 255 corresponding pairs`

Cause A statement with a CORRESPONDING phrase has more than 255 matching pairs. Simplify the data structures.

610 (Error)

`Pointer data item not permitted`

Cause A data item described as USAGE POINTER is specified where it is not allowed.

611 (Error)

`Expected index or pointer data item`

Cause A statement (for example, SET) expects either an index data item or a pointer data item as a sending or receiving operand.

612 (Error)

`Expected pointer data item`

Cause A statement expects a pointer data item as a sending or receiving operand.

613 (Warning)

`Directives: `*`directive-1`*`;`*`directive-2`*` Duplicate setting, previous setting ignored`

Cause The compilation unit contains incompatible directives; for example, CALL-SHARED and NON-SHARED.

614 (Warning)

`Cannot use file specified in CONSULT or SEARCH directive `*`filename`*

Cause A CONSULT or SEARCH directive references a file that is either nonexistent, corrupt, or not a TNS/E object file compiled with debugging symbols.

700 (Warning)

```
Reserved word in next standard
```

Cause The MIGRATION-CHECK and STANDARD 1985 directives are in effect, and the indicated word is a reserved word when the STANDARD 2002 directive is in effect.

701 (Error)

```
BASED clause not permitted on a redefinition
```

Cause The data item has both the BASED and REDEFINES clause specified.

702 (Error)

```
BASED clause not permitted for external data item
```

Cause The data item has both the BASED and EXTERNAL clauses specified.

703 (Error)

```
Operand must not be BASED
```

Cause One of:

- The USING phrase of the Procedure Division header specifies a BASED data item.
- A CHECKPOINT statement specifies a BASED data item.

704 (Error)

```
Expected level 01 or level 77 BASED data item
```

Cause The data item must be a level-01 or level-77 item with the BASED clause.

705 (Error)

```
Expected key-word INITIALIZED, RETURNING, or a verb
```

Cause The syntax of the ALLOCATE statement is incorrect.

Other Products' Error Messages

Products that are merged with the compiler sometimes cause compile-time error messages when you run the compiled program. Examples of such products are:

- ECOBOL Compiler Back End
- SCI, the SQL/MP compiler interface

ECOBOL Compiler Back End

Example 48-2 ECOBOL Compiler Back End Run-Time Error Message

```
*** Failure:
--> Compiler logic error 0_46204 [Failure 0]
```

Cause The compiler's internal consistency checker discovered a logic error. Please report this failure to your service provider.

SCI

Example 48-3 SCI Run-Time Error Message

```
*** Embedded SQL Fatal Error ***
*** ERROR 11998 - Embedded SQL memory exceeded during parsing --
                  report to HP.
Problem on line nnn
** Failure  19 **  Server failure
```

Cause Too few pages of memory were allocated to the SQL compiler interface (SCI) for processing SQL/MP statements.

Recovery Compile your program with the directive SQL PAGES *n*, where *n* is between 385 and 1000. If this error still occurs, recompile the program with a larger value of *n*.

49 Run-Time Diagnostic Messages

This section explains the run-time diagnostic messages that can be reported if an HP COBOL process encounters an error condition.

Where Messages Are Reported

If an EXECUTION-LOG parameter is currently defined, run-time diagnostic messages are reported to the destination that the EXECUTION-LOG parameter specifies (see PARAM Command (page 594)); otherwise, they are delivered to the home terminal of the process.

Standard Message Format

The standard format of a run-time diagnostic message is:

```
  pn - *** Run-time Error nnn  ***
  pn - message
[ pn - File COBOL-file-name = assigned-file-name, open-status ]
  pn - From prog-id + %pppppp, UC.00
[ pn -      prog-id + %pppppp ] ...
[ pn - additional-message ] ...
```

[ ]	surrounds optional material.
pn	is the process name or process ID of the process that caused the error report.
nnn	is the message number from the CRE.
prog-id + %*pppppp*	is the specification of the code location of the error.
prog-id	is the name of the program (from the PROGRAM-ID paragraph) in which the error occurred.
%*pppppp*	is the octal offset within the code block.If a series of similar lines (without the word *From*) follows, it is a traceback of one or more lines to show the CALL history, starting with the program in which the error occurred, then its caller, and so on back to the main program.
message	is the error message text.
additional-message	is any additional information about the problem, such as the name of the file that caused the error.
COBOL-file-name	applies only to I-O errors and is the name of the file in error (the *file-name* in the SELECT clause).
assigned-file-name	is the file system name of the file assigned to *COBOL-file-name*.
open-status	is an open mode (INPUT, OUTPUT, I-O, EXTEND or CLOSED).

Input-Output Error Messages

An input-output error message refers to a COBOL file or to a device specified in an ACCEPT or DISPLAY statement. An input-output error message has additional messages that report:

- A Guardian error number
- A line identifying the file that caused the error

Example 49-1 Input-Output Error Message (CRE)

```
\DRP12.$:0:622:163577460 - *** Run-time Error 181 ***
\DRP12.$:0:622:163577460 - OPEN operation failed with error 48
\DRP12.$:0:622:163577460 - File BWORK = \DRP12.$DATA4.PTR.GRIP, closed
\DRP12.$:0:622:163577460 - From COBLIB_IO_ERROR_ + 0x660 (DLL zcobdll)
\DRP12.$:0:622:163577460 -      COBLIB_OPEN_ + 0x5120 (DLL zcobdll)
\DRP12.$:0:622:163577460 -      .BBB + 0x1C0 (UCr)
\DRP12.$:0:622:163577460 -      .AAA + 0x90 (UCr)
\DRP12.$:0:622:163577460 -      MANE + 0x130 (UCr)
```

SORT Error Messages

A SORT error message arises from a SORT, RELEASE, or RETURN statement. A SORT error message is followed by a SORT diagnostic line. For explanation of the SORT diagnostic line, see the *FastSort Manual*.

Guardian Abnormal Termination Messages

A Guardian abnormal termination message results from circumstances that cause the Guardian environment to terminate the execution of a run unit. The operating environment delivers its own message—the Guardian abnormal termination message—after (or instead of) delivering a COBOL run-time diagnostic message. A Guardian abnormal termination message is more cryptic than a COBOL run-time diagnostic message.

Here is a typical Guardian abnormal termination message:

```
ABENDED: 10,355
CPU time: 0:00:00.005
3: Premature process termination with fatal errors or diagnostics
```

The run unit was operating in an environment with traps armed, and an arithmetic overflow occurred. The Guardian environment placed the job into the selected debugger. You can use the "#X + 77I" information to find out where in the source program the problem arose.

The three most common causes of Guardian abnormal termination messages are:

- Abend (abnormal termination)
- Arithmetic Overflow
- Stack Overflow

These topics explain abnormal termination messages and describe what to do if they occur.

Abend

Abend is short for "abnormal end." It follows any fatal error message that the run-time routines generate. If it appears without a run-time diagnostic message, you might want to execute the run unit again with the SAVEABEND attribute set so that the selected debugger makes a save file. You might also want to recompile with SYMBOLS and INSPECT directives, and then execute with SAVEABEND so that you can make easier use of the save file.

Arithmetic Overflow

Arithmetic overflow means that an arithmetic operation resulted in an arithmetic overflow. Typical causes include:

- Invalid data is associated with an item, such as an uninitialized USAGE COMPUTATIONAL or index item.
- A receiving item in an arithmetic statement is too small.
- An intermediate result in a COMPUTE statement exceeds the 36-digit maximum.

- A USAGE COMPUTATIONAL operand in an arithmetic statement contains a number larger than the number of digits specified in the PICTURE clause for that item.
- A subscript or a reference modifier is too big.

You can determine the cause of arithmetic overflow in any of these ways:

- Without the Debugger

  This method is recommended for production runs.

  If the PARAM INSPECT OFF command is active, the process stops executing with a trace-back to the highest-level program. The first item in the trace-back is the offset within the program of the statement that caused the problem. To find the statement itself, compile the program with the INNERLIST directive. Except for adding the INNERLIST directive, compile the program with exactly the same directives as you did before the problem occurred. (Some directives, such as SYMBOLS, can slightly change the generated code.)
- With the Debugger

  If the PARAM INSPECT ON command is active, the process stops and enters the selected debugger . You can use the debugger to find the problem. If the selected debugger is symbolic, using it is easier if you compiled the program with the SYMBOLS directive. If you did not, you might have to recompile the program with SYMBOLS to find the problem with the debugger.
- With a Saveabend File

  If the program is compiled with the SAVEABEND directive, the process generates a saveabend file if it terminates abnormally. You can determine the cause of the problem by examining the saveabend file.

Stack Overflow

Stack overflow means that the control and data stack has overflowed.

The most likely cause of stack overflow is calling one or more initial programs that have large amounts of data declared in the Working-Storage Section or Extended-Storage Section. Data in initial programs is allocated on the stack, and the area of memory reserved for the stack is much smaller than the area reserved for static data. There are several ways to fix this problem:

- Move one or more large data items (such as large tables) from the initial program to a noninitial program that calls it, and pass the data items as parameters.
- Reduce arrays to fewer elements.
- Eliminate unused data items.
- Remove the INITIAL phrase from the program header and explicitly cancel the program after every call to it (see CANCEL (page 312)).

How to Use the Message List

This topic lists and explains all the run-time diagnostic messages that can be reported for an HP COBOL program that either runs in the non-CRE environment or runs in the CRE but does not call any non-COBOL routines.

If your HP COBOL program calls non-COBOL routines, errors in those routines can cause additional messages to be reported. For a complete list of CRE run-time diagnostic messages, see the *CRE Programmer’s Guide*.

The messages in Message List are in numeric order by message number.

Message List

The run-time diagnostic messages associated with COBOL85, in numeric order by CRE message number, are:

1

`Unknown trap`

Cause The CRE trap processing function was called with an unknown trap number.

Effect The run unit terminates abnormally.

Recovery Check the program’s logic. Use the selected debugger to help isolate the problem or consult your system administrator.

2

`Illegal address reference`

Cause An address was specified that was not within either the virtual code area or the virtual data area allocated to the process. In most cases, a subscript or reference modifier was out of bounds.

Effect The run unit terminates abnormally.

Recovery Try recompiling with the CHECK 3 directive.

3

`Instruction failure`

Cause An attempt was made to:

- Execute a code word that is not an instruction
- Execute a privileged instruction by a nonprivileged process
- Reference an illegal extended address

In most cases, a compiler error or a subscript or reference modifier was out of bounds.

Effect The run unit terminates abnormally.

Recovery Try recompiling with the CHECK 3 directive.

4

`Arithmetic fault`

Cause An arithmetic overflow occurred for one of these reasons:

- The result of a signed arithmetic operation could not be represented with the number of bits available for the particular data type.
- A division operation was attempted with a zero divisor.

In most cases, a receiving item in an arithmetic statement is too small.

Effect The run unit terminates abnormally.

Recovery Check the statement in your program at the address given by *`offset`* and correct it. Also, see Arithmetic Overflow for alternative ways to handle this problem.

5

`Stack overflow`

Cause A stack overflow fault occurs if:

- An attempt was made to execute a program whose dynamically allocated data did not fit within the remaining area available for the stack.
- There was not enough remaining virtual data space for an operating environment procedure to execute.

Operating environment procedures require approximately 700 bytes of user-data stack space to execute.

Effect The run unit terminates abnormally.

Recovery See Stack Overflow for an explanation of this condition and recovery information.

6

`Process loop-timer timeout`

Cause The new time limit specified in the latest call to SETLOOPTIMER has expired.

Effect The run unit terminates abnormally.

Recovery Report this error to your service provider.

7

`Memory manager read error`

Cause An unrecoverable, read error occurred while the program was trying to bring in a page from virtual memory.

Effect The run unit terminates abnormally.

Recovery Report this error to your service provider.

8

`Not enough physical memory`

Cause This fault occurs for one of these reasons:

- A page fault occurred, but there were no physical memory pages available for overlay.
- Disk space could not be allocated while the program is using extensible segments.

Effect The run unit terminates abnormally.

Recovery Report this error to your service provider.

9

`Uncorrectable memory error`

Cause An uncorrectable memory error occurred.

Effect The run unit terminates abnormally.

Recovery Report this error to your service provider.

11

`Corrupted environment`

Cause CRE run-time library data is invalid.

Effect The CRE calls PROCESS_STOP_, specifying the ABEND variant and the text “Corrupted environment.”

Recovery To correct the problem, initialize uninitialized pointers.

12

`Logic error`

Cause The CRE or run-time library detected a logic error within its own domain. For example, although each data item it is using is valid, the values of the data items are mutually inconsistent.

Effect The CRE calls PROCESS_STOP_, specifying the ABEND variant and the text “Logic error.”

Recovery To correct the problem, initialize uninitialized pointers.

The program might have written data in the upper 64 KB of the user data segment. The upper 64 KB are reserved for CRE and run-time library data. Check the program’s logic. Use the selected debugger to help isolate the problem or consult your system administrator.

13

`MCB pointer corrupt`

Cause The pointer at symbolic location _MCB to its primary data structure—the Master Control Block (MCB)—does not point to the MCB. Both the CRE and run-time libraries can report this error.

Effect The CRE attempts to restore the MCB pointer and to write a message to the standard log file; however, because its environment might be corrupted, the CRE might not be able to log a message. In that case, it calls PROCESS_STOP_, specifying the ABEND variant and the text "Corrupted Environment."

Recovery In a native object, the MCB is not at a fixed location. To find it, use the `enoft` command `EXTSYMTBL`. Then use the selected debugger to help isolate the problem. The `enoft` output can be very lengthy. Example 49-2 shows the location of the MCB for the native object file DOMS1400 is 0x0800e3e0.

Example 49-2 Using enoft to Find the MCB Pointer in a Native Object File

```
enoft> FILE DOMS1400
enoft> EXTSYMTBL
....
4 0 _MCB 0x0800e3e0 Glob
....
enoft>
```

14

`Premature takeover`

Cause The backup process received a Guardian message that it had become the primary process, but it had not yet received all of its initial checkpoint information from its predecessor primary process.

Effect The CRE calls PROCESS_STOP_, specifying the ABEND variant and the text "Premature takeover."

Recovery If the takeover occurred because of faulty program logic, correct the program's logic. If the takeover occurred for other reasons, such as a hardware failure, you might want to rerun the program, provided that doing so will not duplicate operations already performed, such as updating a database a second time.

15

`Checkpoint list inconsistent`

Cause A list of checkpoint item descriptors that a run-time library or CRE maintains for fault-tolerant processes was invalid.

Effect The run unit terminates abnormally.

Recovery The list of items to checkpoint is maintained in the program's address space. Check the program's logic. The program might have overwritten the checkpoint list. Use the selected debugger to help isolate the problem.

16

`Checkpoint list exhausted`

Cause The CRE did not have enough room to store all of the checkpoint information required by the program.

Effect The run unit terminates abnormally unless the error occurred in an SMU routine.

Recovery If the error occurred in an SMU routine, increase the size of the checkpoint list.

17

`Cannot obtain control space`

Cause The CRE or a run-time library could not obtain space for all of its data. If the offending statement is an ACCEPT or DISPLAY statement, its buffer could not be allocated due to a lack of allocatable space.

Effect The run unit terminates abnormally.

Recovery Increase the available space. The program should close files as soon as it is done using them.

20

`Cannot utilize file name`

Cause A string, expected to be a valid file name, could not be interpreted as a Guardian external file name.

Effect The run unit terminates abnormally.

Recovery Check that the file names in the program are valid Guardian file names.

21

`Cannot read initialization messages (` *`error`* `)`

Cause During program initialization, the CRE could not read all of the messages (start-up message, PARAM message, ASSIGN messages, and so forth) it expected from the file system. *`error`* is the file system error number the CRE received when it could not read an initialization message.

Effect The run unit terminates abnormally.

Recovery Consult your system administrator.

22

`Cannot obtain executable file name`

Cause The CRE could not obtain the name of the loadfile from the Guardian environment.

Effect The run unit terminates abnormally.

Recovery Consult your system administrator.

23

```
Cannot determine file name ( error )
program_name.logical_name
```

Cause The CRE could not determine the physical file name associated with *`program_name.logical_name`*.

Effect The run unit terminates abnormally.

Recovery Correct the *`program_name.logical_name`* and rerun your program. For information on ASSIGN commands, see ASSIGN Command (page 590) and the *TACL Reference Manual*.

24

```
Conflict in application of ASSIGN
program_name.logical_name
```

Cause ASSIGN values in your TACL environment conflict with each other. For example:

```
ASSIGN   A, $B1.C.D
ASSIGN *.A, $B2.C.D
```

The first ASSIGN specifies that the logical name A can appear in no more than one loadfile. The second assign specifies that the name A can appear in an arbitrary number of loadfiles. The CRE cannot determine whether to use the file C.D on volume $B1 or on volume $B2.

Effect The run unit terminates abnormally.

Recovery Correct the ASSIGNs in your TACL environment. For information on ASSIGN commands, see ASSIGN Command (page 590) and the *TACL Reference Manual*.

25

```
Ambiguity in application of ASSIGN
logical_name
```

Cause An unqualified file name (one without the program-name or the asterisk (*) prefix) specified by an active ASSIGN command corresponds to more than one COBOL file in the run unit.

Effect The run unit terminates abnormally.

Recovery Either qualify the file name or change the program to prevent duplicated names.

26

```
Invalid PARAM value text ( error )
PARAM name 'value')
```

Cause A PARAM specifies a value that is not defined by the CRE. For example, the value for a DEBUG PARAM must be either ON or OFF. The CRE reports this error if a DEBUG PARAM has a value other than ON or OFF. *error*, if present, is a Guardian file system error.

Effect The run unit terminates abnormally.

Recovery Modify the PARAM text and rerun your program. For more information on using PARAMs, see PARAM Command (page 594) and the *TACL Reference Manual*.

27

```
Ambiguity in application of PARAM
PARAM name 'value')
```

Cause A PARAM specifies a value that is ambiguous in the current context. For example, the PARAM specification:

```
PARAM PRINTER-CONTROL A
```

is ambiguous if the program contains more than one logical file named A.

Effect The run unit terminates abnormally.

Recovery Correct the PARAM in your TACL environment. See the *TACL Reference Manual* for more information on using PARAMs.

28

```
Missing language run-time library -- language
```

Cause The run-time library for a module that is written in *language* is not available to the program.

Effect The run unit terminates abnormally.

Recovery Consult your system administrator.

29

```
Program incompatible with run-time library -- COBOL85
```

Cause The run unit was compiled by a version of HP COBOL that is not compatible with this version of the COBOL85 run-time library.

Effect The run unit terminates abnormally.

Recovery Recompile the main program.

34

`Released space not allocated`

Cause The FREE statement attempted to release memory not obtained by ALLOCATE (or by the C run-time library function `malloc()`).

Effect The run unit terminates abnormally.

Recovery Check the program's logic. Use the selected debugger to help isolate the problem or consult your system administrator.

40

`Invalid function parameter`

Cause A function detected a problem with its parameters.

Effect Program behavior depends on the function that was called.

Recovery Correct the parameter you are passing.

41

`Range fault`

Cause An arithmetic overflow or underflow occurred while evaluating an arithmetic function.

Effect Program behavior is language and application dependent. An HP COBOL program cannot cause this error, so the problem must be in a non-COBOL program in the same run unit.

Recovery Modify the program to pass values to the arithmetic functions that do not cause overflow.

42

`Arccos domain fault`

Cause The parameter passed to the ACOS function was not in the range -1 to +1.

Effect Program behavior is language and application dependent.

Recovery Modify the program to pass a valid value to the ACOS function.

43

`Arcsin domain fault`

Cause The parameter passed to the ASIN function was not in the range -1 to +1.

Effect Program behavior is language and application dependent.

Recovery Modify the program to pass a valid value to the ASIN function.

44

`Arctan domain fault`

Cause Both of the parameters to an ATAN function were zero. At least one of the parameters must be nonzero.

Effect Program behavior is language and application dependent.

Recovery Modify the program to pass the correct value to the ATAN function.

46

```
Logarithm function domain fault
```

Cause The parameter passed to a logarithm function (LOG or LOG10) was less than or equal to zero. The parameter to a logarithm function must be greater than zero.

Effect Program behavior is language and application dependent.

Recovery Modify the program to pass a valid value to the logarithm function.

47

```
Modulo function domain fault
```

Cause The value of the second parameter to the MOD function was zero. The second parameter to the MOD function must be nonzero.

Effect Program behavior is language and application dependent.

Recovery Modify the program to pass a nonzero value to the MOD function.

48

```
Exponentiation domain fault
```

Cause Parameters to the exponentiation operator were not acceptable. Given the expression

```
x ** y
```

these parameter combinations produce this message:

```
x = 0 and y is not 0
x < 0 and y is not an integral value
```

Effect Program behavior is language and application dependent.

Recovery Modify the program to pass values that do not violate the above combinations.

49

```
Square root domain fault
```

Cause The parameter to the SQRT function was a negative number. The parameter must be greater than or equal to zero.

Effect Program behavior is language and application dependent.

Recovery Modify the program to pass a nonnegative value to the SQRT function.

55

```
Missing or invalid parameter
```

Cause A required parameter is missing or too many parameters were passed.

Effect Program behavior depends on the function that was called.

Recovery Correct the program to pass a valid parameter.

56

```
Invalid parameter value
```

Cause The value passed as a procedure parameter was invalid.

Effect Program behavior depends on the function that was called.

Recovery Correct the program to pass a valid parameter value.

57

```
Parameter value not accepted
```

Cause The value passed as a procedure parameter is not acceptable in the context in which it is passed. For example, the number of bytes in a write request is greater than the number of bytes per record in the file.

Effect Program behavior depends on the function that was called.

Recovery Correct the program to pass a valid parameter.

59

`Standard input file error ( *error* )`

Cause The Guardian file system reported an error when a routine tried to access the standard input file. *error* is a Guardian file system error code.

Effect The CRE can report this error when it closes your input file. All other instances are language and application dependent. An HP COBOL program terminates abnormally.

Recovery If the error was caused by a read request from your program, correct your program. You might need to verify that your program handles conditions that are beyond your control, such as losing a path to the device. Also refer to error handling in this manual and in the language manual for the routine in your program that detected the error.

If the error was caused by a read request from the CRE, consult your system administrator.

60

`Standard output file error ( *error* )`

Cause The Guardian file system reported an error when the CRE called a file system procedure to access standard output. *error* is the Guardian file system error.

Effect The CRE can report this error when it closes your output file. All other instances are language and application dependent. An HP COBOL program terminates abnormally.

Recovery If the error was caused by a write request from your program, correct your program. You might need to verify that your program handles conditions that are beyond your control such as losing a path to the device. Also refer to error handling in this manual and in the language manual for the routine in your program that detected the error.

If the error was caused by a write request from the CRE, consult your system administrator.

61

`Standard log file error ( *error* )`

Cause The Guardian file system reported an error when the CRE called a file system procedure to access the standard log file. *error* is the Guardian file system error.

Effect The CRE terminates your program.

Recovery Consult your system administrator.

62

`Invalid GUARDIAN file number`

Cause A value that is expected to be a Guardian file number is not the number of an open file.

Effect Program behavior is language and application dependent. An HP COBOL program terminates abnormally.

Recovery Consult your system administrator.

63

`Undefined shared file`

Cause A parameter was not the number of a shared file where one was expected.

Effect Program behavior is language and application dependent. An HP COBOL program terminates abnormally.

Recovery Consult your system administrator.

64

`File not open`

Cause A request to open a file failed because the file device is not supported.

Effect Program behavior is language and application dependent. An HP COBOL program terminates abnormally.

Recovery Consult your system administrator.

65

`Invalid attribute value`

Cause A parameter to an open operation was not a meaningful value. For example, the CRE_File_Open *sync_receive_depth* parameter must be a nonnegative number. This message might be reported if the *sync_receive_depth* parameter is negative.

Effect Program behavior is language and application dependent. An HP COBOL program terminates abnormally.

Recovery Consult your system administrator.

66

`Unsupported file device`

Cause The CRE received a request to access a device that it does not support.

Effect Program behavior is language and application dependent. An HP COBOL program terminates abnormally.

Recovery Consult your system administrator.

67

`Access mode not accepted`

Cause The value of *access* to an open operation was not valid in the context in which it was used. For example, it is invalid to open a spool file for input.

Effect Program behavior is language and application dependent. An HP COBOL program terminates abnormally.

Recovery Consult your system administrator.

68

`Nowait value not accepted`

Cause The value of *no_wait* to an open operation was not valid in the context in which it was used. For example, it is invalid to specify a nonzero value for *no_wait* for a device that does not support nowait operations.

Effect Program behavior is language and application dependent. An HP COBOL program terminates abnormally.

Recovery Consult your system administrator.

69

`Syncdepth not accepted`

Cause The value of the *`sync_receive_depth`* parameter to an open operation was not valid in the context in which it was used. For example, it is not valid to specify a *`sync_receive_depth`* greater than one for a shared file.

Effect Program behavior is language and application dependent. An HP COBOL5 program terminates abnormally.

Recovery Consult your system administrator.

70

`Options not accepted`

Cause The value of an open operation *`options`* parameter was not valid in the context in which it was used.

Effect Program behavior is language and application dependent. An HP COBOL program terminates abnormally.

Recovery Consult your system administrator.

71

`Inconsistent attribute value`

Cause A routine requested a connection to a shared file that was already open, and the attributes of the new open request conflict with the attributes specified when the file was first opened.

Effect Program behavior is language and application dependent. An HP COBOL program terminates abnormally.

Recovery If your program supplied the attribute values, correct and rerun your program; otherwise, consult your system administrator.

75

`Cannot obtain buffer space`

Cause An attempt to allocate buffer space failed. The possible reasons are:

- COBOL_SPECIAL_OPEN_ attempted to get space for level 3 spooling or for at least one record.
- An OPEN of a file assigned to $RECEIVE attempted to get space for various tables.
- An OPEN of an EDIT file attempted to get space to process the file.
- An OPEN of other files attempted to get space for ADVANCING processing.

Effect The OPEN statement is unsuccessful with I-O status code “91.” The COBOL_SPECIAL_OPEN_ call is unsuccessful with I-O status code “30.”

Recovery The program should close files as soon as it is done using them.

76

`Invalid external file name (` *`error`* `)`

Cause A value that was expected to be a Guardian external file name is not in the correct format.

Effect Program behavior is language and application dependent. An HP COBOL program terminates abnormally.

Recovery If you supplied an invalid file name, correct the file name and rerun your program. Otherwise, consult your system administrator.

77

`EDITREADINIT failed (` *`error`* `)`

Cause A call to EDITREADINIT failed. In CRE message 77, *error*, if present, gives the reason for the failure. Possible values of *error* are:

Error Code	Error Name
-1	End-of-file marker encountered
-2	I-O error
-3	Text file format error
-6	Invalid buffer address

Effect The OPEN statement is unsuccessful with I-O status code “91.”

Recovery If the code is -2, there was an input-output error; try the statement again. If the code is -3, the file is not an EDIT file. Assign the correct file. If the code is -4 there is a sequence number error. Use EDIT, TEDIT, or Codewright to correct the file.

78

`EDITREAD failed ( error )`

Cause A call to EDITREAD failed. In CRE message 78, *error*, if present, gives the reason for the failure. Possible values of *error* are:

Error Code	Error Name
-1	End-of-file marker encountered
-2	I-O error
-3	Text file format error
-4	Sequence number error
-5	Checksum error
-6	Invalid buffer address

Effect The OPEN statement is unsuccessful with I-O status code “91.”

Recovery If the code is -2, there was an input-output error; try the statement again. If the code is -3, there is a text file format error. Re-create the file. If the code is -4, there is a sequence number error. Use EDIT, TEDIT, or Codewright to correct the file. If the code is -5, there is a checksum error. Re-create the file.

79

`OpenEdit failed ( error )`

Cause The call to OpenEdit made during the OPEN processing for an EDIT file failed with Guardian *error* or *nnn*. A negative number is a format error. A positive number is a Guardian file system error.

Effect The OPEN statement is unsuccessful with I-O status code “30”.

Recovery Use the command interpreter ERROR command to discover the meaning of the Guardian error number. The message might suggest a corrective action.

80

`Spooler initialization failed ( error )`

Cause A Guardian *error* or *nnn* was returned during the initialization of a spooler file during a special OPEN operation.

Effect The ENTER "COBOL85_SPECIAL_OPEN_" is unsuccessful with I-O status code “30.”

Recovery You can use the command interpreter ERROR command to discover the meaning of the Guardian error number. The message might suggest a corrective action.

81

```
End of file
```

Cause A routine detected an end-of-file condition.

Effect Program behavior is language and application dependent. An HP COBOL program terminates abnormally.

Recovery Correct your program to allow for an end-of-file condition or verify that your program can determine when all of the data has been read.

82

```
Guardian I/O error nnn
```

Cause A Guardian operating system routine returned error *nnn*. This error is usually reported as a result of an event that is beyond program control (when a path or system is not available, for example).

Effect Program behavior is language and application dependent. An HP COBOL program terminates abnormally.

Recovery Consult your system administrator.

83

```
Operation incompatible with file type or status (GE)
```

Cause One of:

1. COBOL_CONTROL_ or COBOL_SETMODE_ was called, and one of these was true:
   a. The associated file was assigned to $RECEIVE.
   b. HP COBOL Fast I-O was selected for the associated file.
   c. The CONTROL or SETMODE call was rejected by the file system (“(GE)” shows the error code).
2. An OSS file (probably described with ORGANIZATION IS LINE SEQUENTIAL) was opened in a Guardian process.

Effect The run unit terminates abnormally.

Recovery One of these, depending on the cause number:

1. Either delete the call to COBOL_CONTROL_ or COBOL_SETMODE_ or correct it (depending on the cause):
   a. Change the file to a file other than $RECEIVE.
   b. Change the RESERVE clause so that it does not use HP COBOL Fast I-O.
   c. Take the action indicated by the Guardian error code “(GE).”
2. Correct the ASSIGN in the select clause or change LINE SEQUENTIAL to SEQUENTIAL.

124

```
System not licensed for COBOL programs
```

Cause An ECOBOL object is being run on a system that does not contain the run-time library. Instead, a “stub” is installed and it produces the diagnostic.

Effect The run unit terminates abnormally.

Recovery Ensure that either T0356 or T0357 is licensed to the system and installed correctly.

125

```
OCCURS DEPENDING ON data item out of range
```

Cause The data item referenced in an OCCURS DEPENDING clause is either less than the minimum or greater than the maximum number of occurrences.

Effect The run unit terminates abnormally.

Recovery Either change the maximum or minimum number of occurrences in the OCCURS clause or correct the program so that the DEPENDING data item has a legal value.

127

`CALL references an active program`

Cause A CALL statement is recursive—it references a program that has been called but has not executed an EXIT PROGRAM statement.

Effect The run unit terminates abnormally.

Recovery Correct the program.

128

`Reference modifier out of range`

Cause Either the leftmost character position is out of range (negative, zero, or greater than the number of characters in the data item), or the length is improper (negative, zero, or greater than the sum of leftmost character position minus the original item size plus 1).

Effect The run unit is terminated abnormally.

Recovery Correct the program.

129

`Improper context for RELEASE statement`

Cause A RELEASE statement was executed when a SORT input procedure was not executing.

Effect The run unit is terminated abnormally.

Recovery Correct the program.

130

`Improper context for RETURN statement`

Cause A RETURN statement was executed when a SORT or MERGE output procedure was not executing.

Effect The run unit is terminated abnormally.

Recovery Correct the program.

132

`SORT or MERGE statement executed while SORT or MERGE active`

Cause A SORT or MERGE statement was executed while a SORT or MERGE input or output procedure was being executed.

Effect The run unit terminates abnormally.

Recovery Correct the program.

133

`Subscript out of range`

Cause A subscript is zero, negative, or exceeds the maximum number of occurrences allowed for the data item.

Effect The run unit terminates abnormally.

Recovery Correct the program.

134

```
GO TO statement not initialized
```

Cause The program attempted to execute a paragraph consisting solely of a GO TO statement that does not specify a procedure name, but no ALTER statement had as yet been executed to establish the procedure name.

Effect The run unit terminates abnormally.

Recovery Correct the program.

135

```
ACCEPT or DISPLAY requested for an unsupported device
```

Cause The device associated with the mnemonic-name specified in an ACCEPT statement is not a process or a terminal; or the device associated with the mnemonic-name specified in a DISPLAY statement is not a process, terminal, operator console, printer, or existing disk file.

Effect The home terminal or the device specified as the EXECUTION-LOG is used instead. If the home terminal or the device specified as the EXECUTION-LOG cannot be used, the run unit terminates abnormally.

Recovery Change the program to assign a proper device. Use low-level debugging to trap changes to zero (see the *Inspect Manual*).

136

```
Input-output error nnn on ACCEPT or DISPLAY device
```

Cause A file system error with a Guardian error code of *nnn* occurred while the process was accepting or displaying data.

Effect For an ACCEPT statement, the run-time library returns spaces for a nonnumeric destination or zeros for a numeric destination, and execution continues. For a DISPLAY statement, the requested data is not displayed, and execution continues.

Recovery Type ERROR *nnn* to see the meaning of the Guardian error; it might indicate corrective action.

137

```
Called program not found
```

Cause The program referenced in a CALL statement with identifier specified could not be located. It must be present in the run unit or in the TNS or user library, and must be within the scope of the CALL statement.

Effect If the OVERFLOW or EXCEPTION clause is specified in the CALL statement, that path is taken. If not, the run unit is terminated abnormally.

Recovery If the program is not bound or linked into the run unit, and it should be, use BIND or a linker to be bind or link it in with the other programs. If the value of the data item is incorrect or the program being called is not within the scope of the called program, correct the program.

138

```
CANCEL references an active program
```

Cause The program referenced in a CANCEL statement has been called but has not executed an EXIT PROGRAM (it is still active).

Effect The run unit terminates abnormally.

Recovery Correct the program. Reduce the number of files that are open simultaneously.

139

```
Cancelled program not found
```

Cause The program referenced in a CANCEL statement with identifier specified could not be located. It must be present in the run unit or in the TNS or user library, and must be within the scope of the CANCEL statement.

Effect The run unit continues.

Recovery Correct the program. Reduce the number of files that are open simultaneously.

140

```
Record referenced in RELEASE statement not in SD of SORT or MERGE
statement
```

Cause The record-name referenced in a RELEASE statement is not associated with the file name that was referenced in the currently active SORT or MERGE statement.

Effect The run unit terminates abnormally.

Recovery Correct the source program logic.

141

```
File referenced in RETURN statement not in SD of SORT or MERGE statement
```

Cause The file name referenced in a RETURN statement is not the file name that was referenced in the currently active SORT or MERGE statement.

Effect The run unit terminates abnormally.

Recovery Correct the source program logic.

142

```
RELEASE operation failed - SORTMERGE message follows
```

Cause During the execution of a RELEASE statement in a SORT input procedure, the FastSort utility returned an error. The message from FastSort follows on the next line.

Effect The run unit terminates abnormally.

Recovery Action depends on the message FastSort returns. See the *FastSort Manual* for the meaning of these messages.

143

```
RETURN operation failed - SORTMERGE message follows
```

Cause During the execution of a RETURN statement in a SORT or MERGE output procedure, the FastSort utility returned an error. The message from FastSort follows on the next line.

Effect The run unit terminates abnormally.

Recovery Action depends on the message FastSort returns. See the *FastSort Manual* for the meaning of these messages.

144

```
SORT or MERGE operation failed before end - SORTMERGE message follows
```

Cause At the end of the execution of a SORT or MERGE operation, the FastSort utility returned an error. The message from FastSort follows on the next line.

Effect The run unit terminates abnormally.

Recovery Action depends on the message FastSort returns. See the *FastSort Manual* for the meaning of these messages.

145

```
SORT or MERGE operation failed at start - SORTMERGE message follows
```

Cause At the start of the execution of a SORT or MERGE operation, the FastSort utility returned an error.

The second line of the message is the text of the SORT error message returned by the FASTSORT procedure SORTERRORSUM.

The third line is

```
Sort error code = mm - System error code = nn
```

where *mm* is the SORT error code defined in the *FastSort Manual*, and.*nn* is the high-order 2 bytes of the error code return defined for SORTERRORSUM.

If the sort was a subsort, this text is appended to the message:

```
- Subsort index = oo, CPU,id = pp,qq
```

where *oo* is the *subsort-index* defined in the *FastSort Manual* under SORTERRORSUM, and *pp*,*qq* is the *subsort-id* defined in the *FastSort Manual* under SORTERRORSUM.

For some errors, one or more of these lines appear:

```
Error on following USING file file-name
Error on following GIVING file file-name
Error on following scratch file file-name
Error on the internal scratch file
Error on the internal free list file
Error in interprocess communication
```

where *file-name* is the name of the file where the error occurred. For “Error on following scratch file,” *file-name* is the one in the SELECT for the sort-merge file description.

Effect The run unit terminates abnormally.

Recovery Action depends on the message FastSort returns. See the *FastSort Manual* for the meaning of these messages.

147

```
Parameter mismatch for CALL identifier
```

Cause The number of parameters in the USING list of a CALL *identifier* statement does not match the number of parameters specified in the called program’s USING list in the Procedure Division.

Effect The run unit terminates abnormally.

Recovery Correct the USING list in either the CALL statement or Procedure Division.

148

```
PERFORM nesting too deep
```

Cause The PERFORM stack overflowed. The maximum nesting depth is 50.

Effect The run unit terminates abnormally.

Recovery Check PERFORM statements for proper termination or reduce the complexity of the program. It might be helpful to recompile the program with the directive PERFORM-TRACE (page 568).

148

```
PERFORM nesting too deep
```

Cause The PERFORM stack overflowed. The maximum nesting depth is 50.

Effect The run unit terminates abnormally.

Recovery Check PERFORM statements for proper termination or reduce the complexity of the program. It might be helpful to recompile the program with the directive PERFORM-TRACE (page 568).

149

```
Invalid Data Conversion
```

Cause Explicit or implicit conversions from DISPLAY numeric data to COMP or NATIVE contain invalid DISPLAY numeric data.

Effect The run unit terminates abnormally.

Recovery Use the IS NUMERIC class test to validate DISPLAY NUMERIC DATA.

NOTE: If the Common Run-Time Environment (CRE) DLL used is earlier than version T1269H03, the following run-time diagnostic message is displayed instead:

```
*** Run-time Error 149*** Unknown error ordinal
```

150

```
Alternate key not present in file
```

Cause During the execution of an OPEN statement, the description of an alternate key in the COBOL program does not correspond to any alternate key in the physical file that is assigned. That is, no key in the physical file matches the COBOL description in length and offset.

Effect The OPEN statement is unsuccessful with I-O status code “39.”

Recovery Correct either the program or the FUP CREATE or other process that created the file.

151

```
DUPLICATES specification in SELECT does not match file
```

Cause During the execution of an OPEN statement, the description of an alternate key in the COBOL program includes the DUPLICATES phrase, and the key in the physical file that is assigned does not allow duplicates, or the program does not include DUPLICATES and the file allows duplicates.

Effect The OPEN statement is unsuccessful with I-O status code “39.”

Recovery Correct either the program or the FUP CREATE or other process that created the file.

152

```
OPEN on a non-disk file that is specified with alternate keys
```

Cause The physical file assigned by the OPEN statement is not a structured disk file, but the COBOL program described it with alternate keys. Alternate keys are allowed only on structured disk files.

Effect The OPEN statement is unsuccessful with I-O status code “39.”

Recovery Either correct the program or assign the proper file.

153

Create of new file failed with error *nnn*

Cause During the attempt to create a new file during the execution of an OPEN statement, Guardian error *nnn* was returned by the operating environment.

Effect The OPEN statement is unsuccessful with I-O status code “30.”

Recovery Type ERROR *nnn* to see the meaning of the Guardian error; it might indicate corrective action.

154

```
Sequential DELETE must follow successful READ
```

Cause A DELETE statement was executed on a file with ACCESS MODE SEQUENTIAL, and the preceding statement was not a successful READ statement.

Effect The DELETE statement is unsuccessful with I-O status code "43."

Recovery Correct the program.

155

```
DELETE positioning failed with error nnn
```

Cause The file positioning request issued during the processing of a DELETE statement was rejected with Guardian error *nnn*.

Effect The DELETE statement is unsuccessful with I-O status code "30."

Recovery Type ERROR *nnn* to see the meaning of the Guardian error; it might indicate corrective action.

156

```
DELETE repositioning failed with error nnn
```

Cause The file positioning request issued after deleting a record during the processing of a DELETE statement was rejected with Guardian error *nnn*.

Effect The DELETE statement is unsuccessful with I-O status code "30".

Recovery Type ERROR *nnn* to see the meaning of the Guardian error; it might indicate corrective action.

157

```
Wrong open mode for DELETE
```

Cause A DELETE statement was executed, but the file was not open for I-O.

Effect The DELETE statement is unsuccessful with I-O status code "49."

Recovery Correct the program.

158

```
DELETE operation failed with error nnn
```

Cause The attempt to delete a record during the processing of a DELETE statement was rejected with Guardian error *nnn*.

Effect The DELETE statement is unsuccessful with I-O status code "30".

Recovery Type ERROR *nnn* to see the meaning of the Guardian error; it might indicate corrective action.

159

```
OPEN on an EDIT file and wrong open mode
```

Cause An EDIT file is opened for other than INPUT. HP COBOL cannot write to an EDIT file.

Effect The OPEN statement is unsuccessful with I-O status code "39."

Recovery Correct the program, or assign other than an EDIT file. You can write to an unstructured file or an entry sequenced file, then use the GET PUT operation of EDIT to produce an EDIT file.

160

`OPEN on an EDIT file described with a record size that is too big`

Cause The maximum record size for an EDIT file is 4095 bytes.

Effect The OPEN statement is unsuccessful with I-O status code “39.”

Recovery Correct the program, or assign other than an EDIT file.

161

`OPEN EXTEND positioning failed with error` *`nnn`*

Cause The attempt to position the file at the end during the processing of an OPEN … EXTEND statement failed with Guardian error *nnn*.

Effect The OPEN statement is unsuccessful with I-O status code “30.”

Recovery Type ERROR *nnn* to see the meaning of the Guardian error; it might indicate corrective action.

162

`OPEN on a nonexistent file that is not OPTIONAL`

Cause The file referenced in an OPEN … EXTEND, I-O, or INPUT statement does not exist, and OPTIONAL is not specified in the SELECT clause for the file.

Effect The OPEN statement is unsuccessful with I-O status code “35.”

Recovery If you want the file to be created for EXTEND or I-O, put OPTIONAL in the SELECT clause; otherwise, verify that the file exists when the program is run.

163

`OPEN page eject failed with error` *`nnn`*

Cause The attempt to eject the first page during the processing of an OPEN statement referencing a LINAGE file or during the process of COBOL_SPECIAL_OPEN_ failed with Guardian error *nnn*.

Effect The OPEN statement or call to COBOL_SPECIAL_OPEN_ is unsuccessful with I-O status code “30.”

Recovery Type ERROR *nnn* to see the meaning of the Guardian error; it might indicate corrective action.

164

`OPEN rewind failed with error` *`nnn`*

Cause The attempt to rewind the file during the processing of an OPEN statement failed with Guardian error *nnn*.

Effect The OPEN statement is unsuccessful with I-O status code “30.”

Recovery Type ERROR *nnn* to see the meaning of the Guardian error; it might indicate corrective action.

165

`OPEN requested for an unsupported device`

Cause The device assigned to the COBOL file is not a legal device for COBOL input or output.

Effect The OPEN statement is unsuccessful with I-O status code “39.”

Recovery Assign a file on a legal device.

166

```
OPEN requested for a locked file
```

Cause The file referenced in an OPEN statement is locked (CLOSE LOCK was executed). CLOSE LOCK is unrelated to LOCKFILE and UNLOCKFILE. A COBOL file name associated with a disk file can become locked even if it is never opened. This message appears when the disk file is also assigned to some other COBOL file name with the same ASSIGN name in the same run unit, and a CLOSE LOCK is executed on that file name.

Effect The OPEN statement is unsuccessful with I-O status code “38.”

Recovery Either do not close the file with lock or correct the program.

167

```
OPEN requested for an open file
```

Cause The file referenced in an OPEN statement is already open.

Effect The OPEN statement is unsuccessful with I-O status code “41.”

Recovery Correct the program.

168

```
LINAGE file not on printer or process
```

Cause The file referenced in an OPEN statement for a file with LINAGE specified is not assigned to a printer or a process (a spooler, for example).

Effect The OPEN statement is unsuccessful with I-O status code “39.”

Recovery Either assign the correct device or remove the LINAGE clause from the file description (FD).

169

`LOCK or UNLOCK operation failed with error` *`nnn`*

Cause The attempt to lock or unlock a file or record with a LOCK or UNLOCK statement failed with Guardian error *nnn*.

Effect The statement is unsuccessful with I-O status code “30.”

Recovery Type ERROR *nnn* to see the meaning of the Guardian error; it might indicate corrective action.

170

```
MULTIPLE FILE TAPE file not on tape
```

Cause The device assigned to a file name associated with a MULTIPLE FILE TAPE clause in the I-O CONTROL paragraph is not a magnetic tape device.

Effect The OPEN statement is unsuccessful with I-O status code “39.”

Recovery Either remove the MULTIPLE FILE TAPE clause or assign a magnetic tape to the file.

171

`OPEN positioning for MULTIPLE FILE TAPE failed with error` *`nnn`*

Cause The attempt to position to a file on a multiple file tape during the processing of an OPEN statement failed with Guardian error *nnn*.

Effect The OPEN statement is unsuccessful with I-O status code “30.”

Recovery Type ERROR *nnn* to see the meaning of the Guardian error; it might indicate corrective action.

172

```
OPEN on a nonexistent file and alternate keys specified
```

Cause A file with alternate keys specified does not exist when the attempt is made to open it. The file must have been created with a FUP CREATE or other means before it can be opened.

Effect The OPEN statement is unsuccessful with I-O status code "30."

Recovery Create the file correctly before running the program, assign the correct file, or remove the alternate key specifications.

173

```
Indexed file not defined as ORGANIZATION INDEXED
```

Cause The file assigned is an indexed (key sequenced) file, but the COBOL loadfile description does not specify ORGANIZATION INDEXED.

Effect The OPEN statement is unsuccessful with I-O status code "39."

Recovery Assign the correct file or correct the file description in the program.

174

```
OPEN INPUT when file not on input device
```

Cause A file is being opened for INPUT, but the device assigned is an illegal device or an output-only device (such as a printer).

Effect The OPEN statement is unsuccessful with I-O status code "37."

Recovery Assign the correct device or file.

175

```
Operation other than OPEN on file that is not open
```

Effect The program attempted an operation other than OPEN (such as READ or WRITE) on a file that is not open.

Effect The statement is unsuccessful with I-O status code "4*x*" where *x* is appropriate for the statement.

Recovery Correct the program.

176

```
File is not opened for timed I/O
```

Cause A TIME LIMIT operation (such READ … TIME LIMIT *nn*) was attempted on a file that was not opened with TIME LIMITS specified.

Effect The operation is unsuccessful with I-O status code "90."

Recovery Correct the program to do an OPEN … TIME LIMITS.

177

```
OPEN OUTPUT for file not on output device
```

Cause A file is being opened for OUTPUT, but the device assigned is an illegal device or an input-only device (such as a card reader).

Effect The OPEN statement is unsuccessful with I-O status code "37."

Recovery Assign the proper device or file.

178

```
Relative file not defined as ORGANIZATION RELATIVE
```

Cause The file assigned is a relative file, but the COBOL loadfile description does not specify ORGANIZATION RELATIVE.

Effect The OPEN statement is unsuccessful with I-O status code “39.”

Recovery Assign the correct file or correct the file description in the program.

179

```
Sequential file not defined as ORGANIZATION SEQUENTIAL
```

Cause The file assigned is a sequential file (entry sequenced or unstructured), but the COBOL loadfile description specifies an organization other than sequential.

Effect The OPEN statement is unsuccessful with I-O status code “39.”

Recovery Assign the correct file or correct the file description in the program.

180

```
Non-disk or unstructured file and not sequential organization
```

Cause The COBOL program describes the file as ORGANIZATION RELATIVE or INDEXED, but the assigned file is not a structured disk file. Only structured disk files can be assigned to RELATIVE or INDEXED COBOL files.

Effect The OPEN statement is unsuccessful with I-O status code “39.”

Recovery Assign the correct file or correct the file description in the program.

181

```
OPEN operation failed with error nnn
```

Cause A Guardian error *nnn* was returned during OPEN or COBOL_SPECIAL_OPEN_ processing.

Effect The OPEN statement or the call to COBOL_SPECIAL_OPEN_ is unsuccessful with I-O status code “30.”

Recovery Type ERROR *nnn* to see the meaning of the Guardian error; it might indicate corrective action.

182

```
Primary key offset in program does not match file
```

Cause During the execution of an OPEN statement, the prime key offset in the COBOL program does not correspond to the prime key in the physical file that is assigned.

Effect The OPEN statement is unsuccessful with I-O status code “39.”

Recovery Correct either the program or the FUP CREATE or other process that created the file.

183

```
Primary key size in program does not match file
```

Cause During the execution of an OPEN statement, the prime key size in the COBOL program does not correspond to the prime key in the physical file that is assigned.

Effect The OPEN statement is unsuccessful with I-O status code “39.”

Recovery Correct either the program or the FUP CREATE or other process that created the file.

184

```
Purge of file during OPEN failed with error nnn
```

Cause During the execution of an OPEN statement, the attempt to purge a file (the purge is due to some attribute conflicts) failed with Guardian error *nnn*.

Effect The OPEN statement is unsuccessful with I-O status code “30”.

Recovery Type ERROR *nnn* to see the meaning of the Guardian error; it might indicate corrective action. A value of “48,” for example, means that the program does not have permission to purge an existing file that has an improper record size and create a new one.

185

```
Purge data from file during OPEN failed with error nnn
```

Cause During the execution of an OPEN statement, the attempt to purge the data from the file (the purge is due to an OPEN OUTPUT on a file that contains data) failed with Guardian error *nnn*.

Effect The OPEN statement is unsuccessful with I-O status code “30.”

Recovery Type ERROR *nnn* to see the meaning of the Guardian error; it might indicate corrective action.

186

```
READ operation failed with error nnn
```

Cause During the execution of a READ statement, the system read routine failed with Guardian error *nnn*.

Effect The READ statement is unsuccessful with I-O status code “30.”

Recovery Type ERROR *nnn* to see the meaning of the Guardian error; it might indicate corrective action.

187

```
READ WITH LOCK on file with read ahead or an OSS file
```

Cause A READ … LOCK statement was attempted on an OSS file or a file that is reading ahead. This can happen if the file is opened for INPUT and the organization is sequential. A READ operation that is executed as part of a transaction (delimited by ENTER “BEGINTRANSACTION” and ENTER “ENDTRANSACTION” statements) requires such locking.

Effect The READ statement is unsuccessful with I-O status code “91.”

Recovery For a Guardian file, either remove the LOCK or open the file for I-O. For an OSS file, remove the LOCK.

188

```
READ positioning failed with error nnn
```

Cause An attempt to position the file for a READ statement failed with Guardian error *nnn*.

Effect The READ statement is unsuccessful with I-O status code “30.”

Recovery Type ERROR *nnn* to see the meaning of the Guardian error; it might indicate corrective action.

189

```
Sequential READ requested when current position is undefined
```

Cause The program attempted to execute a sequential READ statement when the last operation was not a successful READ, OPEN, or START. One common cause is attempting to do a READ after an at-end condition occurs.

Effect The READ statement is unsuccessful with I-O status code “46.”

Recovery Correct the program.

190

```
Wrong open mode for READ
```

Cause The program attempted to execute a READ statement, but the open mode for the file is not INPUT or I-O.

Effect The READ statement is unsuccessful with I-O status code “47.”

Recovery Correct the program.

191

```
Reel swap failed with error nnn
```

Cause An attempt to do a tape reel swap failed. If the failure was due to a system error, *nnn* is the Guardian error code. If the failure was due to the operator responding incorrectly to the swap requests, *nnn* is 0.

Effect The READ statement is unsuccessful with I-O status code “30.”

Recovery Correct the cause of the problem.

192

```
Sequential REWRITE must follow successful READ
```

Cause The program attempted to execute a REWRITE statement to a file for which ACCESS MODE SEQUENTIAL is specified, and the last operation on the file was not a successful READ statement (for example, an at-end condition resulted).

Effect The REWRITE statement is unsuccessful with I-O status code “43.”

Recovery Correct the program.

193

```
REWRITE positioning failed with error nnn
```

Cause An attempt to position the file for a REWRITE statement failed with Guardian error *nnn*.

Effect The REWRITE statement is unsuccessful with I-O status code “30.”

Recovery Type ERROR *nnn* to see the meaning of the Guardian error; it might indicate corrective action.

194

```
REWRITE repositioning failed with error nnn
```

Cause An attempt to reposition the file after the rewrite operation during execution of a REWRITE statement failed with Guardian error *nnn*.

Effect The REWRITE statement is unsuccessful with I-O status code “30.”

Recovery Type ERROR *nnn* to see the meaning of the Guardian error; it might indicate corrective action.

195

```
Wrong open mode for REWRITE
```

Cause The file referenced in a REWRITE statement is not opened in the I-O mode.

Effect The REWRITE statement is unsuccessful with I-O status code "49."

Recovery Correct the program.

196

```
Sequential REWRITE permitted only with same record size
```

Cause At attempt to REWRITE a record on a file with sequential organization failed because the new record is not the same size as the old one.

Effect The REWRITE statement is unsuccessful with I-O status code "44."

Recovery Correct the program.

197

`REWRITE operation failed with error` *`nnn`*

Cause The attempt to rewrite the record during execution of a REWRITE statement failed with Guardian error *nnn*.

Effect The REWRITE statement is unsuccessful with I-O status code "30."

Recovery Type ERROR *nnn* to see the meaning of the Guardian error; it might indicate corrective action.

198

`START operation failed with error` *`nnn`*

Cause The attempt to read the selected record during execution of a START statement failed with Guardian error *nnn*.

Effect The START statement is unsuccessful with I-O status code "30."

Recovery Type ERROR *nnn* to see the meaning of the Guardian error; it might indicate corrective action.

199

`START positioning failed with error` *`nnn`*

Cause The attempt to position the file during execution of a START statement failed with Guardian error *nnn*.

Effect The START statement is unsuccessful with I-O status code "30."

Recovery Type ERROR *nnn* to see the meaning of the Guardian error; it might indicate corrective action.

200

```
Wrong open mode for START
```

Cause The file referenced in a START statement is not open for INPUT or I-O.

Effect The START statement is unsuccessful with I-O status code "47."

Recovery Correct the program.

201

```
System node not available or does not exist
```

Cause During the processing of an OPEN statement referencing a file that is assigned to a system node (the ASSIGN phrase includes "*xxx*" in the file name), Guardian indicated that the system is not up or does not exist.

Effect The OPEN statement is unsuccessful with I-O status code "30."

Recovery Either assign to the correct system node or determine why the node is not responding.

202

`LOCKFILE, UNLOCKFILE or UNLOCKRECORD on a file that is not open`

Cause The file referenced in a LOCKFILE, UNLOCKFILE, or UNLOCKRECORD statement is not open.

Effect The statement is unsuccessful with I-O status code “42.”

Recovery Correct the program.

203

`OPEN on unstructured file described without fixed length records`

Cause An unstructured file is assigned to a COBOL file name whose RECORD CONTAINS clause in the file description indicates that the records are variable.

Effect The OPEN statement is unsuccessful with I-O status code “39.”

Recovery Either correct the RECORD CONTAINS clause or assign the correct file.

204

`Writing end of file failed with error` *`nnn`*

Cause The attempt to write an end of file on a tape during the execution of a CLOSE statement failed with Guardian error *nnn*.

Effect The CLOSE statement is unsuccessful with I-O status code “30.”

Recovery Type ERROR *nnn* to see the meaning of the Guardian error; it might indicate corrective action.

205

`Writing end of reel failed with error` *`nnn`*

Cause The attempt to write an end of reel on a tape during the execution of a CLOSE REEL or WRITE statement failed with Guardian error *nnn*.

Effect The statement is unsuccessful with I-O status code “30.”

Recovery Type ERROR *nnn* to see the meaning of the Guardian error; it might indicate corrective action.

206

`WRITE operation failed with error` *`nnn`*

Cause The attempt to write the record during the execution of a WRITE statement or the attempt to write a saved record during the execution of a CLOSE statement failed with Guardian error *nnn*.

Effect The statement is unsuccessful with I-O status code “30.”

Recovery Type ERROR *nnn* to see the meaning of the Guardian error; it might indicate corrective action. For example, a Guardian error value of 21 is a File System error that indicates “illegal count specified,” which could arise from a block size error in the file description.

207

`WRITE failed because file is full`

Cause The attempt to write the record during the execution of a WRITE statement or the attempt to write a saved record during the execution of a CLOSE statement failed because the file is full.

Effect The statement is unsuccessful with I-O status code “34.”

Recovery Correct the program if it is in a loop, or increase the number of extents allocated for the file. You can use a FUP CREATE command for this purpose.

208

```
WRITE positioning failed with error nnn
```

Cause The attempt to position the file during the execution of a random WRITE statement failed with Guardian error *nnn*.

Effect The statement is unsuccessful with I-O status code “30.”

Recovery Type ERROR `nnn` to see the meaning of the Guardian error; it might indicate corrective action.

209

```
WRITE repositioning failed with error nnn
```

Cause The attempt to reposition the file after writing a record during the execution of a random WRITE statement failed with Guardian error *nnn*.

Effect The statement is unsuccessful with I-O status code “30.”

Recovery Type ERROR `nnn` to see the meaning of the Guardian error; it might indicate corrective action.

210

```
Line skipping failed with error nnn
```

Cause The attempt to skip lines on a file during the execution of a random WRITE statement with ADVANCING specified failed with Guardian error *nnn*.

Effect The statement is unsuccessful with I-O status code “30.”

Recovery Type ERROR `nnn` to see the meaning of the Guardian error; it might indicate corrective action.

211

```
Wrong open mode for WRITE
```

Cause A WRITE statement was attempted on a relative or indexed on a file that does not meet any one of these criteria:

- Its open mode is OUTPUT.
- Its open mode is EXTEND and its access mode is sequential.
- Its open mode is I-O and its access mode is not sequential.

A WRITE statement was attempted on a sequential file that does not meet any of these criteria:

1. Its open mode is EXTEND or OUTPUT.
2. Its open mode is I-O and the device assigned is a process, a terminal or $RECEIVE.

Effect The statement is unsuccessful with I-O status code “48” except for Item 2 for sequential, in which the I-O status code is “90.”

Recovery Correct the program or assign the correct device.

212

```
Wrong length record specified for WRITE or REWRITE
```

Cause The record length specified in the program (either by an OCCURS DEPENDING clause in the record description of the record being written or by the DEPENDING identifier in the RECORD VARYING clause in the file description entry) is shorter than the minimum or larger than the maximum length specified in the RECORD clause.

Effect The WRITE or REWRITE statement is unsuccessful with I-O status code “44.”

Recovery Correct the program—in either the record length calculation or the RECORD clause.

213

```
OPEN EXTEND for file not on extend device
```

Cause A file is being opened for EXTEND, but the device assigned is an illegal device or an input-only device (such as a card reader).

Effect The OPEN statement is unsuccessful with I-O status code “37.”

Recovery Assign the proper device or file.

214

```
OPEN I-O for file not on input-output device
```

Cause A file is being opened for I-O, but the device assigned is an illegal device or an input- or output-only device (such as a magnetic tape).

Effect The OPEN statement is unsuccessful with I-O status code “37.”

Recovery Assign the proper device or file.

215

```
Wrong or missing LABELS attribute
```

Cause A LABELS attribute in a DEFINE command that refers to the file contains other than OMITTED, BYPASS, ANSI, or IBM.

Effect The OPEN statement is unsuccessful with I-O status code “90.”

Recovery Before running this program again, verify that the DEFINE in the environment is correct.

216

```
Wrong or missing USE attribute
```

Cause A USE attribute in a DEFINE command that refers to a file contains other than OPEN, IN, OUT, or EXTEND.

Effect The OPEN statement is unsuccessful with I-O status code “90.”

Recovery Before running this program again, verify that the DEFINE in the environment is correct.

217

```
Wrong or missing RECFORM attribute
```

Cause A RECFORM attribute in a DEFINE command that refers to a file contains other than F or U, or F is specified and variable length records are specified in the File Description for the file, or U is specified and fixed length records are specified for the file.

Effect The OPEN statement is unsuccessful with I-O status code “90.”

Recovery Before running this program again, verify that the DEFINE in the environment is correct or correct the file description.

218

```
Wrong or missing RECLEN attribute
```

Cause A RECLEN attribute in a DEFINE command that refers to a file contains more than 6 digits, is not correctly formed, is negative, or is not the record size for fixed length records.

Effect The OPEN statement is unsuccessful with I-O status code “90.”

Recovery Before running this program again, verify that the DEFINE in the environment is correct or correct the file's record description.

219

```
Wrong or missing BLOCKLEN attribute
```

Cause A BLOCKLEN attribute in a DEFINE command that refers to a file contains more than 6 digits, is not correctly formed, is negative, is omitted for tapes specified with LABEL RECORDS STANDARD, is greater than the record length when BLOCK CONTAINS is not specified, or does not match file File Definition.

Effect The OPEN statement is unsuccessful with I-O status code "90."

Recovery Before running this program again, verify that the DEFINE in the environment is correct or correct the file description.

220

```
Wrong or missing FILESEQ attribute
```

Cause A FILESEQ attribute in a DEFINE command that refers to a file contains more than 4 digits, is not correctly formed, is negative, or does not match that specified in the MULTIPLE FILE TAPE clause.

Effect The OPEN statement is unsuccessful with I-O status code "90."

Recovery Before running this program again, verify that the DEFINE in the environment is correct or correct the file's MULTIPLE FILE TAPE clause.

221

```
Wrong or missing DEVICE attribute
```

Cause A DEVICE attribute in a DEFINE command that refers to a file does not start with a dollar sign ($) or backslash (\) or is not alphabetic.

Effect The OPEN statement is unsuccessful with I-O status code "90."

Recovery Before running this program again, verify that the DEFINE in the environment is correct.

222

`A DEFINE procedure failed with error` *`nnn`*

Cause A request to get DEFINE information failed with Guardian error *nnn*.

Effect The OPEN statement is unsuccessful with I-O status code "90."

Recovery You can use the command interpreter ERROR command to discover the meaning of the Guardian error number. The message might suggest a corrective action.

223

```
DEFINE required for LABEL RECORDS STANDARD
```

Cause A DEFINE command is required if LABEL RECORDS STANDARD is specified in the File Definition.

Effect The OPEN statement is unsuccessful with I-O status code "90."

Recovery Before running this program again, verify that the DEFINE in the environment is correct or correct the file description.

224

`PositionEdit failed with error` *`nnn`*

Cause The call to PositionEdit made during the OPEN processing for an EDIT file failed with Guardian error *nnn*.

Effect The OPEN statement is unsuccessful with I-O status code “30.”

Recovery You can use the command interpreter ERROR command to discover the meaning of the Guardian error number. The message might suggest a corrective action.

225

```
Size of unstructured file opened EXTEND not multiple of record size
```

Cause The end of file on an unstructured file being opened for EXTEND is not a multiple of the record size.

Effect The OPEN statement is unsuccessful with an I-O status code “30.”

Recovery If an incorrect file was assigned, assign the correct one. Otherwise, you must take some action to repair the file.

226

```
File attributes don't match and file not opened OUTPUT or has alt keys
```

Cause The file attributes of the existing file conflict with the file description in the program and the file is not being opened for output, or (if it is being opened for output) the file description specifies alternate record keys (COBOL cannot create a file with alternate keys). The conflict is that the record sizes differ (the file description specifies a larger maximum record than the existing file’s attributes), and the COBOL program indicates that the records are fixed length.

Effect The OPEN statement is unsuccessful with I-O status code “30.”

Recovery Either correct the program descriptions or assign the correct file.

227

```
Loadclose failed with internal error mmm, GUARDIAN error nnn
```

Cause The HP COBOL Fast I-O routine loadclose returned the indicated error codes.

Effect The CLOSE statement is unsuccessful. If the Guardian error is 43, the I-O status code is "34"; otherwise it is "30".

Recovery If you can resolve the Guardian error, do so; otherwise report the error to your service provider.

228

```
SEGMENT_ALLOCATE_ for fast i/o failed with error return nnn
```

Cause An attempt to allocate an extended segment for use with HP COBOL Fast I-O failed with the indicated error.

Effect Instead of using HP COBOL Fast I-O (local buffering provided by HP COBOL), the program uses standard I-O processing.

Recovery The *Guardian Procedure Calls Reference Manual* documents the errors that ALLOCATESEGMENT returns. Take the appropriate action.

229

```
Loadopen failed with internal error mmm, GUARDIAN error nnn
```

Cause The HP COBOL Fast I-O routine loadopen returned the indicated error codes.

Effect The OPEN statement is unsuccessful with I-O status code “30.”

Recovery If you can resolve the Guardian error, do so; otherwise report the error to your service provider.

230

```
Loadwrite failed with internal error mmm, GUARDIAN error nnn
```

Cause The HP COBOL Fast I-O routine loadwrite returned the indicated error codes.

Effect The WRITE statement is unsuccessful with I-O status code "30."

Recovery If you can resolve the Guardian error, do so; otherwise report the error to your service provider.

231

```
Initnewdatablock (Fast i-o) failed
```

Cause The HP COBOL Fast I-O routine initnewdatablock failed.

Effect The READ statement is unsuccessful with I-O status code "30."

Recovery Report the error to your service provider.

232

```
An illegal operation was attempted on a fast i/o file
```

Cause The program attempted to execute a READ REVERSED, REWRITE, or START statement to a file that is using HP COBOL Fast I-O.

Effect The REWRITE statement is unsuccessful with I-O status code "40."

Recovery Change the program so that it does not attempt to use HP COBOL Fast I-O. A program attempts to use HP COBOL Fast I-O on a sequential file that meets these criteria:

- You are not creating an audited file (you can read an audited file, however).
- The file description includes a RESERVE clause with a `number` specifying the number of blocks to buffer. `number` must be greater than 2.
- The file description does not include a LINAGE clause or a CODE-SET clause.
- The file was not opened with time limits (as with the TIME LIMITS phrase in the OPEN statement).
- The program is not compiled with the NONSTOP directive.
- The specifications in the OPEN statement, or the attributes derived during the open operation by some other means (such as from an applicable command interpreter ASSIGN command), conform to:
  - — The open mode is INPUT or OUTPUT.
  - — The exclusion mode is EXCLUSIVE if OPEN OUTPUT is specified.
  - — The exclusion mode is PROTECTED if OPEN INPUT is specified.

233

```
OPEN OUTPUT SHARED specified for open disk file
```

Cause OPEN OUTPUT file name SHARED was specified, and another process had the file open (in any exclusion mode).

Effect The OPEN statement is unsuccessful with I-O status error 30 and GUARDIAN-ERR 12.

Recovery If you do not want to delete all of the information in the file, change the OPEN statement to

```
OPEN I-O file-name SHARED
```

and if necessary, specify OPTIONAL in the SELECT clause for the file.

If you do want to delete all of the information in the file, add a null Declarative Portion for this file to the program, recompile it (to allow the run unit to continue), and change the code to something like this:

```
PERFORM WITH TEST AFTER UNTIL file-status (1: 1) = "0"
     OPEN OUTPUT the-file SHARED
     IF file-status (1: 1) NOT = "0"
         IF file-status = "30"
           AND GUARDIAN-ERR = 12
             (count these, send a message, or take other
              appropriate action)
             ENTER TAL "DELAY" USING some-time
         ELSE
             (issue some other error and terminate
              program)
         END-IF
    END-IF
END-PERFORM
```

where *`file-status`* appears in the FILE STATUS clause in the SELECT clause, and *`some-time`* is the number of 0.01-second intervals to delay.

Each occurrence of this error produces a message on the home terminal or in the execution log.

234

`CLOSE operation failed with error nnn`

Cause The Guardian file system reported error *nnn* during execution of a close operation.

Effect The CLOSE statement is unsuccessful with I-O status "30."

Recovery Type ERROR *nnn* to see the meaning of the Guardian error; it may indicate corrective action.

235

`REVERSED not allowed for this file`

Cause The file name specified in a READ REVERSED statement is assigned to a physical file that cannot be read in reverse; that is, one of:

- A nondisk file
- A disk file for which preread was selected
- A disk file that is an edit file for ENV OLD
- A disk file that is an edit file for ENV COMMON or for native HP COBOL with the NONSTOP directive active
- An unstructured disk file that is blocked

Effect The READ statement is unsuccessful with I-O status "30."

Recovery One of:

- If the physical file is not a disk file, use a disk file instead.
- If preread is selected, cancel it (for example, add SHARED to the OPEN statement that opens the file, or open the file in I-O mode rather than INPUT mode).
- Use something other than an edit file.
- Remove BLOCK CONTAINS from the unstructured file FD.

A ASCII Character Set

This appendix contains two tables of the ASCII character set, both of which use these column headings:

Column Heading	Meaning
Ord.	Character's ordinal number in the ASCII character set
Octal	Character's octal representation (with left and right bytes)
Hex.	Character's hexadecimal representation
Dec.	Character's decimal representation
Char	Character code or character itself (such as *NULL* or *A*)
Meaning	Meaning of character code (such as "Null" or "Uppercase *A* ")

Table A-1 is in numeric order and Table A-2 is in alphabetic order.

ASCII Character Set in Numeric Order

Table A-1 presents the ASCII character set in numeric order; that is, these columns are in numeric order:

- Ord. (the character's ordinal number in the ASCII character set)
- Octal (the character's octal representation)
- Hex. (the character's hexadecimal representation)
- Dec. (the character's decimal representation)

If you know one of the preceding values for the character you want to look up, use Table A-1; if you know only the character code or the character itself (such as "NUL" or "A"), use Table A-2 instead.

Table A-1 ASCII Character Set in Numeric Order

	Octal					
Ord.	Left	Right	Hex.	Dec.	Char.	Meaning
1	000000	000000	00	0	NUL	Null
2	000400	000001	01	1	SOH	Start of heading
3	001000	000002	02	2	STX	Start of text
4	001400	000003	03	3	ETX	End of text
5	002000	000004	04	4	EOT	End of transmission
6	002400	000005	05	5	ENQ	Enquiry
7	003000	000006	06	6	ACK	Acknowledge
8	003400	000007	07	7	BEL	Bell
9	004000	000010	08	8	BS	Backspace
10	004400	000011	09	9	HT	Horizontal tabulation
11	005000	000012	0A	10	LF	Line feed
12	005400	000013	0B	11	VT	Vertical tabulation
13	006000	000014	0C	12	FF	Form feed
14	006400	000015	0D	13	CR	Carriage return

Table A-1 ASCII Character Set in Numeric Order *(continued)*

	Octal					
Ord.	Left	Right	Hex.	Dec.	Char.	Meaning
15	007000	000016	0E	14	SO	Shift out
16	007400	000017	0F	15	SI	Shift in
17	010000	000020	10	16	DLE	Data link escape
18	010400	000021	11	17	DC1	Device control 1
19	011000	000022	12	18	DC2	Device control 2
20	011400	000023	13	19	DC3	Device control 3
21	012000	000024	14	20	DC4	Device control 4
22	012400	000025	15	21	NAK	Negative acknowledge
23	013000	000026	16	22	SYN	Synchronous idle
24	013400	000027	17	23	ETB	End of transmission block
25	014000	000030	18	24	CAN	Cancel
26	014400	000031	19	25	EM	End of medium
27	015000	000032	1A	26	SUB	Substitute
28	015400	000033	1B	27	ESC	Escape
29	016000	016000	1C	28	FS	File separator
30	016400	000035	1D	29	GS	Group separator
31	017000	000036	1E	30	RS	Record separator
32	017400	000037	1F	31	US	Unit separator
33	020000	000040	20	32	SP	Space
34	020400	000041	21	33	!	Exclamation point
35	021000	000042	22	34	"	Quotation mark
36	021400	000043	23	35	#	Number sign
37	022000	000044	24	36	$	Dollar sign
38	022400	000045	25	37	%	Percent sign
39	023000	000046	26	38	&	Ampersand
40	023400	000047	27	39	'	Apostrophe
41	024000	000050	28	40	(	Opening parenthesis
42	024400	000051	29	41	)	Closing parenthesis
43	025000	000052	2A	42	*	Asterisk
44	025400	000053	2B	43	+	Plus
45	026000	000054	2C	44	,	Comma
46	026400	000055	2D	45	-	Hyphen (minus)
47	027000	000056	2E	46	.	Period (decimal point)
48	027400	000057	2F	47	/	Slash
49	030000	000060	30	48	0	Zero

Table A-1 ASCII Character Set in Numeric Order *(continued)*

	Octal					
Ord.	Left	Right	Hex.	Dec.	Char.	Meaning
50	030400	000061	31	49	1	One
51	031000	000062	32	50	2	Two
52	031400	000063	33	51	3	Three
53	032000	000064	34	52	4	Four
54	032400	000065	35	53	5	Five
55	033000	000066	36	54	6	Six
56	033400	000067	37	55	7	Seven
57	034000	000070	38	56	8	Eight
58	034400	000071	39	57	9	Nine
59	035000	000072	3A	58	:	Colon
60	035400	000073	3B	59	;	Semicolon
61	036000	000074	3C	60	<	Less than
62	036400	000075	3D	61	=	Equals
63	037000	000076	3E	62	>	Greater than
64	037400	000077	3F	63	?	Question mark
65	040000	000100	40	64	@	Commercial at sign
66	040400	000101	41	65	A	Uppercase `A`
67	041000	000102	42	66	B	Uppercase `B`
68	041400	000103	43	67	C	Uppercase `C`
69	042000	000104	44	68	D	Uppercase `D`
70	042400	000105	45	69	E	Uppercase `E`
71	043000	000106	46	70	F	Uppercase `F`
72	043400	000107	47	71	G	Uppercase `G`
73	044000	000110	48	72	H	Uppercase `H`
74	044400	000111	49	73	I	Uppercase `I`
75	045000	000112	4A	74	J	Uppercase `J`
76	045400	000113	4B	75	K	Uppercase `K`
77	046000	000114	4C	76	L	Uppercase `L`
78	046400	000115	4D	77	M	Uppercase `M`
79	047000	000116	4E	78	N	Uppercase `N`
80	047400	000117	4F	79	O	Uppercase `O`
81	050000	000120	50	80	P	Uppercase `P`
82	050400	000121	51	81	Q	Uppercase `Q`
83	051000	000122	52	82	R	Uppercase `R`
84	051400	000123	53	83	S	Uppercase `S`

Table A-1 ASCII Character Set in Numeric Order *(continued)*

	Octal					
Ord.	Left	Right	Hex.	Dec.	Char.	Meaning
85	052000	000124	54	84	T	Uppercase T
86	052400	000125	55	85	U	Uppercase U
87	053000	000126	56	86	V	Uppercase V
88	053400	000127	57	87	W	Uppercase W
89	054000	000130	58	88	X	Uppercase X
90	054400	000131	59	89	Y	Uppercase Y
91	055000	000132	5A	90	Z	Uppercase Z
92	055400	000133	5B	91	[	Opening bracket
93	056000	000134	5C	92	\	Backslash
94	056400	000135	5D	93	]	Closing bracket
95	057000	000136	5E	94	^	Circumflex
96	057400	000137	5F	95	_	Underscore
97	060000	000140	60	96	`	Grave accent
98	060400	000141	61	97	a	Lowercase a
99	061000	000142	62	98	b	Lowercase b
100	061400	000143	63	99	c	Lowercase c
101	062000	000144	64	100	d	Lowercase d
102	062400	000145	65	101	e	Lowercase e
103	063000	000146	66	102	f	Lowercase f
104	063400	000147	67	103	g	Lowercase g
105	064000	000150	68	104	h	Lowercase h
106	064400	000151	69	105	i	Lowercase i
107	065000	000152	6A	106	j	Lowercase j
108	065400	000153	6B	107	k	Lowercase k
109	066000	000154	6C	108	l	Lowercase l
110	066400	000155	6D	109	m	Lowercase m
111	067000	000156	6E	110	n	Lowercase n
112	067400	000157	6F	111	o	Lowercase o
113	070000	000160	70	112	p	Lowercase p
114	070400	000161	71	113	q	Lowercase q
115	071000	000162	72	114	r	Lowercase r
116	071400	000163	73	115	s	Lowercase s
117	072000	000164	74	116	t	Lowercase t
118	072400	000165	75	117	u	Lowercase u
119	073000	000166	76	118	v	Lowercase v

Table A-1 ASCII Character Set in Numeric Order *(continued)*

	Octal					
Ord.	Left	Right	Hex.	Dec.	Char.	Meaning
120	073400	000167	77	119	w	Lowercase *w*
121	074000	000170	78	120	x	Lowercase *x*
122	074400	000171	79	121	y	Lowercase *y*
123	075000	000172	7A	122	z	Lowercase *z*
124	075400	000173	7B	123	{	Opening brace
125	076000	000174	7C	124	|	Vertical line
126	076400	000175	7D	125	}	Closing brace
127	077000	000176	7E	126	~	Tilde
128	077400	000177	7F	127	DEL	Delete

ASCII Character Set in Alphabetic Order

Table A-2 presents the ASCII character set in alphabetic order—that is, alphabetic character codes (in the column labelled "Char.") are in alphabetic order.

Table A-2 ASCII Character Set in Alphabetic Order

			Octal			
Char.	Meaning	Ord.	Left	Right	Hex.	Dec.
^	Circumflex	95	057000	000136	5E	94
~	Tilde	127	077000	000176	7E	126
!	Exclamation point	34	020400	000041	21	33
"	Quotation mark	35	021000	000042	22	34
#	Number sign	36	021400	000043	23	35
$	Dollar sign	37	022000	000044	24	36
%	Percent sign	38	022400	000045	25	37
&	Ampersand	39	023000	000046	26	38
'	Apostrophe	40	023400	000047	27	39
(	Opening parenthesis	41	024000	000050	28	40
)	Closing parenthesis	42	024400	000051	29	41
*	Asterisk	43	025000	000052	2A	42
+	Plus	44	025400	000053	2B	43
,	Comma	45	026000	000054	2C	44
-	Hyphen (minus)	46	026400	000055	2D	45
.	Period (decimal point)	47	027000	000056	2E	46
/	Slash	48	027400	000057	2F	47
0	Zero	49	030000	000060	30	48
1	One	50	030400	000061	31	49
2	Two	51	031000	000062	32	50

Table A-2 ASCII Character Set in Alphabetic Order *(continued)*

			Octal			
Char.	**Meaning**	**Ord.**	**Left**	**Right**	**Hex.**	**Dec.**
3	Three	52	031400	000063	33	51
4	Four	53	032000	000064	34	52
5	Five	54	032400	000065	35	53
6	Six	55	033000	000066	36	54
7	Seven	56	033400	000067	37	55
8	Eight	57	034000	000070	38	56
9	Nine	58	034400	000071	39	57
:	Colon	59	035000	000072	3A	58
;	Semicolon	60	035400	000073	3B	59
<	Less than	61	036000	000074	3C	60
=	Equals	62	036400	000075	3D	61
>	Greater than	63	037000	000076	3E	62
?	Question mark	64	037400	000077	3F	63
@	Commercial at sign	65	040000	000100	40	64
[	Opening bracket	92	055400	000133	5B	91
\	Backslash	93	056000	000134	5C	92
]	Closing bracket	94	056400	000135	5D	93
_	Underscore	96	057400	000137	5F	95
`	Grave accent	97	060000	000140	60	96
A	Uppercase *A*	66	040400	000101	41	65
a	Lowercase *a*	98	060400	000141	61	97
ACK	Acknowledge	7	003000	000006	06	6
B	Uppercase *B*	67	041000	000102	42	66
b	Lowercase *b*	99	061000	000142	62	98
BEL	Bell	8	003400	000007	07	7
BS	Backspace	9	004000	000010	08	8
C	Uppercase *C*	68	041400	000103	43	67
c	Lowercase *c*	100	061400	000143	63	99
CAN	Cancel	25	014000	000030	18	24
CR	Carriage return	14	006400	000015	0D	13
D	Uppercase *D*	69	042000	000104	44	68
d	Lowercase *d*	101	062000	000144	64	100
DC1	Device control 1	18	010400	000021	11	17
DC2	Device control 2	19	011000	000022	12	18
DC3	Device control 3	20	011400	000023	13	19

Table A-2 ASCII Character Set in Alphabetic Order *(continued)*

			Octal			
Char.	**Meaning**	**Ord.**	**Left**	**Right**	**Hex.**	**Dec.**
DC4	Device control 4	21	012000	000024	14	20
DEL	Delete	128	077400	000177	7F	127
DLE	Data link escape	17	010000	000020	10	16
E	Uppercase `E`	70	042400	000105	45	69
e	Lowercase `e`	102	062400	000145	65	101
EM	End of medium	26	014400	000031	19	25
ENQ	Enquiry	6	002400	000005	05	5
EOT	End of transmission	5	002000	000004	04	4
ESC	Escape	28	015400	000033	1B	27
ETB	End of transmission block	24	013400	000027	17	23
ETX	End of text	4	001400	000003	03	3
F	Uppercase `F`	71	043000	000106	46	70
f	Lowercase `f`	103	063000	000146	66	102
FF	Form feed	13	006000	000014	0C	12
FS	File separator	29	016000	016000	1C	28
G	Uppercase `G`	72	043400	000107	47	71
g	Lowercase `g`	104	063400	000147	67	103
GS	Group separator	30	016400	000035	1D	29
H	Uppercase `H`	73	044000	000110	48	72
h	Lowercase `h`	105	064000	000150	68	104
HT	Horizontal tabulation	10	004400	000011	09	9
I	Uppercase `I`	74	044400	000111	49	73
i	Lowercase `i`	106	064400	000151	69	105
J	Uppercase `J`	75	045000	000112	4A	74
j	Lowercase `j`	107	065000	000152	6A	106
K	Uppercase `K`	76	045400	000113	4B	75
k	Lowercase `k`	108	065400	000153	6B	107
L	Uppercase `L`	77	046000	000114	4C	76
l	Lowercase `l`	109	066000	000154	6C	108
LF	Line feed	11	005000	000012	0A	10
M	Uppercase `M`	78	046400	000115	4D	77
m	Lowercase `m`	110	066400	000155	6D	109
N	Uppercase `N`	79	047000	000116	4E	78
n	Lowercase `n`	111	067000	000156	6E	110
NAK	Negative acknowledge	22	012400	000025	15	21

Table A-2 ASCII Character Set in Alphabetic Order *(continued)*

			Octal			
Char.	**Meaning**	**Ord.**	**Left**	**Right**	**Hex.**	**Dec.**
NUL	Null	1	000000	000000	00	0
O	Uppercase O	80	047400	000117	4F	79
o	Lowercase o	112	067400	000157	6F	111
P	Uppercase P	81	050000	000120	50	80
p	Lowercase p	113	070000	000160	70	112
Q	Uppercase Q	82	050400	000121	51	81
q	Lowercase q	114	070400	000161	71	113
R	Uppercase R	83	051000	000122	52	82
r	Lowercase r	115	071000	000162	72	114
RS	Record separator	31	017000	000036	1E	30
S	Uppercase S	84	051400	000123	53	83
s	Lowercase s	116	071400	000163	73	115
SI	Shift in	16	007400	000017	0F	15
SO	Shift out	15	007000	000016	0E	14
SOH	Start of heading	2	000400	000001	01	1
SP	Space	33	020000	000040	20	32
STX	Start of text	3	001000	000002	02	2
SUB	Substitute	27	015000	000032	1A	26
SYN	Synchronous idle	23	013000	000026	16	22
T	Uppercase T	85	052000	000124	54	84
t	Lowercase t	117	072000	000164	74	116
U	Uppercase U	86	052400	000125	55	85
u	Lowercase u	118	072400	000165	75	117
US	Unit separator	32	017400	000037	1F	31
V	Uppercase V	87	053000	000126	56	86
v	Lowercase v	119	073000	000166	76	118
VT	Vertical tabulation	12	005400	000013	0B	11
W	Uppercase W	88	053400	000127	57	87
w	Lowercase w	120	073400	000167	77	119
X	Uppercase X	89	054000	000130	58	88
x	Lowercase x	121	074000	000170	78	120
Y	Uppercase Y	90	054400	000131	59	89
y	Lowercase y	122	074400	000171	79	121
Z	Uppercase Z	91	055000	000132	5A	90
z	Lowercase z	123	075000	000172	7A	122

Table A-2 ASCII Character Set in Alphabetic Order *(continued)*

			Octal			
Char.	Meaning	Ord.	Left	Right	Hex.	Dec.
{	Opening brace	124	075400	000173	7B	123
|	Vertical line	125	076000	000174	7C	124
}	Closing brace	126	076400	000175	7D	125

B Data Type Correspondence

These tables contain the return value size generated by HP language compilers for each data type. Use this information when you need to specify values with the Accelerator ReturnValSize option. These tables are also useful if your programs use data from files created by programs in another language, or your programs pass parameters to programs written in callable languages.

Refer to the appropriate SQL/MP or SQL/MX manual for a complete list of SQL data type correspondence. Also note that the return value sizes given in these tables do not correspond to the storage size of SQL data types.

NOTE: COBOL includes COBOL 74, HP COBOL, and SCREEN COBOL unless otherwise noted.

If you are using the Data Definition Language (DDL) utility to describe your files, see the *Data Definition Language (DDL) Reference Manual* for more information.

Table B-1 Integer Types, Part 1

	8-Bit Integer	16-Bit Integer	32-Bit Integer
HP C and HP C++	char[1] unsigned char signed char	int in the 16-bit data model short unsigned	int in the 32-bit or wide data model long unsigned long
COBOL	Alphabetic Numeric DISPLAY Alphanumeric-Edited Alphanumeric Numeric-Edited	PIC S9(*n*) COMP or PIC 9(*n*) COMP without *P* or *V*, 1 <=*n* <= 4 Index Data Item[2] NATIVE-2[3]	PIC S9(*n*) COMP or PIC 9(*n*) COMP without *P* or *V*, 5 <=*n* <= 9 Index Data Item[2] NATIVE-4[3]
FORTRAN	--	INTEGER[4] INTEGER*2	INTEGER*4
Pascal	BYTE Enumeration, unpacked, <= 256 members Subrange, unpacked, *n* ...*m*, 0 <= *n* and *m* <= 255	INTEGER INT16 CARDINAL[1] BYTE or CHAR value parameter Enumeration, unpacked, > 256 members Subrange, unpacked, *n* ...*m*, -32768 <=*n* and *m* <= 32767, but at least *n* or *m* outside 0…255 range	LONGINT INT32 Subrange, unpacked *n*…*m*, –2147483648 <=*n* and *m* <= 2147483647, but at least *n* or *m* outside -32768…32767 range
SQL/MP or SQL/MX	CHAR	NUMERIC(1)… NUMERIC(4) PIC 9(1) COMP… PIC 9(4) COMP SMALLINT	NUMERIC(5)… NUMERIC(9) PIC 9(5) COMP … PIC 9(9) COMP INTEGER

TAL
pTAL
	STRING
	UNSIGNED(8)
		INT
		INT(16)
		UNSIGNED(16)
			INT(32)

Return Value Size (Words)	**1**	**1**	**2**

1 Unsigned integer
2 Index Data Item is a 16-bit integer in COBOL 74 and a 32-bit integer in HP COBOL
3 HP COBOL only
4 INTEGER is normally equivalent to INTEGER*2. The INTEGER*4 and INTEGER*8 compiler directives redefine INTEGER.

Table B-2 Integer Types, Part 2

	64-Bit Integer	Bit Integer of 1 to 31 Bits	Decimal Integer
HP C and HP C++	long long	--	--
COBOL	PIC S9(*n*) COMP or PIC 9(*n*) COMP without *P* or *V*, 10 <= *n* <= 18 NATIVE-8[1]	--	Numeric DISPLAY
FORTRAN	INTEGER*8	--	--
Pascal	INT64	UNSIGNED(*n*), 1 <= *n* <= 16 INT(*n*), 1 <= *n* <= 16	DECIMAL
SQL/MP or SQL/MX	NUMERIC(10)… NUMERIC(18) PIC 9(10) COMP… PIC 9(18) COMP LARGEINT	--	DECIMAL (*n*,*s*) PIC 9(*n*) DISPLAY
TAL pTAL	FIXED(0), INT(64)	UNSIGNED(*n*), 1 <= *n* <= 31	--
Return Value Size (Words)	**4**	**1 or 2 in TAL, 1 in other languages**	**1 or 2, depends on declared pointer size**

1 HP COBOL only

Table B-3 Floating, Fixed, and Complex Types

	32-Bit Floating	64-Bit Floating	64-Bit Fixed Point	64-Bit Complex
HP C and HP C++	float	double	--	--
COBOL	--	--	PIC S9(*n*–*s*)v9(*s*) COMP or PIC 9(*n*–*s*)v9(*s*) COMP, 10 <= *n* <= 18	--
FORTRAN	REAL	DOUBLE PRECISION	--	COMPLEX
Pascal	REAL	LONGREAL	--	--
SQL/MP or SQL/MX	--	--	NUMERIC (*n*,*s*) PIC 9(*n*-*s*)v9(*s*) COMP	--
TAL pTAL	REAL REAL(32)	REAL(64)	FIXED(*s*), -19 <= *s* <= 19	--
Return Value Size (Words)	**2**	**4**	**4**	**4**

Table B-4 Character Types

	Character	Character String	Varying Length Character String

HP C and HP C++	signed char
	unsigned char
		pointer to char	`struct {`
			`  int len;`
			`char val [n]`
			` };`

COBOL	Alphabetic Numeric DISPLAY Alphanumeric-Edited Alphanumeric Numeric-Edited	Alphabetic Numeric DISPLAY Alphanumeric-Edited Alphanumeric Numeric-Edited	01 name. 03 len USAGE IS NATIVE-2[1] 03 val PIC X(*n*).
FORTRAN	CHARACTER	CHARACTER array CHARACTER**n*	--
Pascal	CHAR or BYTE value parameter Enumeration, unpacked, <= 256 members Subrange, unpacked *n*...*m*, 0 <= *n* and *m* <= 255	PACKED ARRAY OF CHAR FSTRING(*n*)	STRING(*n*)
SQL/MP or SQL/MX	PIC X CHAR	CHAR(*n*) PIC X(*n*)	VARCHAR(*n*)
TAL pTAL	STRING	STRING array	--
Return Value Size (Words)	**1**	**1 or 2, depends on declared pointer size**	**1 or 2, depends on declared pointer size**

[1] HP COBOL only

Table B-5 Structured, Logical, Set, and File Types

	Byte-Addressed Structure	Word-Addressed Structure	Logical (true or false)	Boolean	Set	File
HP C and HP C++	--	struct	--	--	--	--
COBOL	--	01-level RECORD	--	--	--	--
FORTRAN	RECORD	--	LOGICAL[1]	--	--	--
Pascal	RECORD, byte-aligned	RECORD, word-aligned	--	BOOLEAN	Set	File
SQL/MP or SQL/MX	--	--	--	--	--	--
TAL pTAL	Byte-addressed standard STRUCT pointer	Word-addressed standard STRUCT pointer	--	--	--	--
Return Value Size (Words)	**1 or 2, depends on declared pointer size**	**1 or 2, depends on declared pointer size**	**1 or 2, depends on compiler directive**	**1**	**1**	**1**

[1] LOGICAL is normally defined as 2 bytes. The LOGICAL*2 and LOGICAL*4 compiler directives redefine LOGICAL.

Table B-6 Pointer Types

	Procedure Pointer	Byte Pointer	Word Pointer	Extended Pointer
HP C and HP C++	function pointer	byte pointer	word pointer	extended pointer
COBOL	--	--	--	--
FORTRAN	--	--	--	--
Pascal	Procedure pointer	Pointer, byte-addressed BYTEADDR	Pointer, byte-addressed WORDADDR	Pointer, extended-addressed EXTADDR
SQL/MP or SQL/MX	--	--	--	--
TAL	--	16-bit pointer, byte-addressed	16-bit pointer, word-addressed	32-bit pointer
pTAL	PROCPTR	16-bit pointer, byte-addressed	16-bit pointer, word-addressed	32-bit pointer
Return Value Size (Words)	**1 or 2, depends on declared pointer size**	**1 or 2, depends on declared pointer size**	**1 or 2, depends on declared pointer size**	**1 or 2, depends on declared pointer size**

Table B-7 Address Types[1]

	Procedure Pointer	Byte Address	Word Address	Extended Address
HP C and HP C++	--	--	--	--
COBOL	--	--	--	--
FORTRAN	--	--	--	--
Pascal	--	--	--	--
SQL/MP or SQL/MX	--	--	--	--
TAL	--	--	--	--
pTAL	PROCADDR	BADDR SGBADDR SGXBADDR CBADDR	WADDR SGWADDR SGXWADDR CWADDR	EXTADDR
Return Value Size (Words)	**1 or 2, depends on declared pointer size**	**1 or 2, depends on declared pointer size**	**1 or 2, depends on declared pointer size**	**1 or 2, depends on declared pointer size**

[1] Only the pTAL compiler supports address types

Index