Data Definition Language (DDL) Reference Manual

Abstract

This publication describes the DDL language syntax and the DDL dictionary database. The audience includes application programmers and database administrators.

Product Version

DDL D40
DDL H01

Supported Release Version Updates (RVUs)

This publication supports J06.03 and all subsequent J-series RVUs, H06.03 and all subsequent H-series RVUs, and G06.26 and all subsequent G-series RVUs, until otherwise indicated by its replacement publications.

Part Number	Published
529431-003	May 2010

Document History

Part Number	Product Version	Published
529431-002	DDL D40, DDL H01	July 2005
529431-003	DDL D40, DDL H01	May 2010

Legal Notices

Printed in the US

Data Definition Language (DDL) Reference Manual

2. DDL Language Elements (continued)

3. Running the DDL Compiler

4. Named Constants

5. Definitions and Records

5. Definitions and Records (continued)

6. Definition Attributes

6. Definition Attributes (continued)

7. SPI Tokens

8. Dictionary-Manipulation Statements

9. DDL Compiler Commands

9. DDL Compiler Commands (continued)

9. DDL Compiler Commands (continued)

10. Dictionary Maintenance

10. Dictionary Maintenance (continued)

A. DDL Messages

B. Sample Schemas

C. DDL Data Translation

D. Dictionary Database Structure

D. Dictionary Database Structure (continued)

E. Dictionary Reports

F. Syntax Summary

F. Syntax Summary (continued)

Examples

Examples (continued)

Examples (continued)

Examples (continued)

Examples (continued)

Examples (continued)

Examples (continued)

Figures

Figures (continued)

Tables

Tables (continued)

Tables (continued)

What's New in This Manual

Manual Information

Abstract

This publication describes the DDL language syntax and the DDL dictionary database. The audience includes application programmers and database administrators.

Product Version

DDL D40
DDL H01

Supported Release Version Updates (RVUs)

This publication supports J06.03 and all subsequent J-series RVUs, H06.03 and all subsequent H-series RVUs, and G06.26 and all subsequent G-series RVUs, until otherwise indicated by its replacement publications.

Part Number	Published
529431-003	May 2010

Document History

Part Number	Product Version	Published
529431-002	DDL D40, DDL H01	July 2005
529431-003	DDL D40, DDL H01	May 2010

New and Changed Information

Changes to the 529431-003 manual:

- Added J-series information to Supported Release Version Updates (RVUs).
- Added a note about DDL2 object on 1-3 and 5-1.
- Added a caution on 5-1.

Changes to the 529431-001 manual

This is a new manual.

About This Manual

This manual describes the Data Definition Language (DDL), an HP product for defining data objects and for translating object definitions to source code for programming languages and for other products on HP NonStop™ systems.

DDL data objects include:

- Constants
- Field definitions
- Group definitions
- Records
- Subsystem Programmatic Interface (SPI) token codes
- SPI token maps
- SPI token types

The DDL compiler can translate an object definition to source code for one or more of these HP products:

- HP C for NonStop Systems
- HP COBOL for NonStop Systems
- HP FORTRAN for NonStop Systems
- HP NonStop NET/MASTER Network Control Language (NCL)
- HP Pascal for NonStop Systems (D-series systems only)
- HP Portable Transaction Application Language (pTAL)
- HP Tandem Advanced Command Language (TACL)
- HP Transaction Application Language (TAL)
- File Utility Program (FUP)

In addition, the DDL compiler can generate its own source code from object definitions in a dictionary and produce reports on the contents of a dictionary.

This manual gives:

- An overview of DDL
- Instructions for using the DDL compiler
- Syntax descriptions, usage guidelines, and examples for all DDL statements and commands

Topics:

Audience

This manual is for application programmers and database administrators.

Application programmers can use DDL to:

- Add data definitions to Pathmaker catalogs
- Define data objects and translate them to host-language source code
- Define SPI message tokens

Database administrators can use DDL to:

- Generate FUP commands for creating databases
- Provide file type and access information for Enform Plus reports about databases
- Define Enscribe, HP NonStop SQL/MP, and HP NonStop SQL/MX databases

---

Note. For information about using DDL to define Enscribe, HP NonStop SQL/MP, or HP NonStop SQL/MP databases, see the *SQL/MP Reference Manual* and the *SQL/MX Reference Manual*.

Prerequisite Manuals

For all readers:

- *Guardian User's Guide*
- *Enscribe Programmer's Guide*
- *TACL Reference Manual*

For application programmers:

- *Guardian Programmer's Guide*
- *Introduction to Data Management*
- HP manuals for the "host languages" to which you want the DDL compiler to translate object definitions:
  - *C/C++ Programmer's Guide*
  - *COBOL Manual for TNS and TNS/R Programs*
  - *COBOL Manual for TNS/E Programs*
  - *FORTRAN Reference Manual*
  - *NET/MASTER Network Control Language (NCL) Reference Manual*
  - *Pascal Reference Manual* (D-series systems only)
  - *pTAL Reference Manual*
  - *TACL Reference Manual*
  - *TAL Reference Manual*

If you plan to use DDL to define SPI tokens:

- *Distributed Name Service (DNS) Management Programming Manual*
- *DSM Template Services Manual*
- *SPI Programming Manual*

Related Manuals

In addition to some of the prerequisite manuals, this manual refers to information in these HP manuals:

- *Enform Plus Reference Manual*
- *Event Management Service (EMS) Analyzer Manual*
- *File Utility Program (FUP) Reference Manual*
- *Software Internationalization Guide*
- *SQL/MP Reference Manual*
- *SQL/MX Reference Manual*
- *Pathmaker Programming Guide*
- *Guardian Procedure Calls Reference Manual*
- *Guardian Procedure Errors and Messages Manual*
- *TMF Management Programming Manual*

Notation Conventions

Hypertext Links

Blue underline is used to indicate a hypertext link within text. By clicking a passage of text with a blue underline, you are taken to the location described. For example:

> This requirement is described under Backup DAM Volumes and Physical Disk Drives on page 3-2.

General Syntax Notation

This list summarizes the notation conventions for syntax presentation in this manual.

UPPERCASE LETTERS. Uppercase letters indicate keywords and reserved words. Type these items exactly as shown. Items not enclosed in brackets are required. For example:

```
MAXATTACH
```

lowercase italic letters. Lowercase italic letters indicate variable items that you supply. Items not enclosed in brackets are required. For example:

```
file-name
```

computer type. `Computer type` letters within text indicate C and Open System Services (OSS) keywords and reserved words. Type these items exactly as shown. Items not enclosed in brackets are required. For example:

```
myfile.c
```

italic computer type. *`Italic computer type`* letters within text indicate C and Open System Services (OSS) variable items that you supply. Items not enclosed in brackets are required. For example:

```
pathname
```

[ ] Brackets. Brackets enclose optional syntax items. For example:

```
TERM [\system-name.]$terminal-name

INT[ERRUPTS]
```

A group of items enclosed in brackets is a list from which you can choose one item or none. The items in the list can be arranged either vertically, with aligned brackets on each side of the list, or horizontally, enclosed in a pair of brackets and separated by vertical lines. For example:

```
FC [ num  ]
   [ -num ]
   [ text ]

K [ X | D ] address
```

{ } Braces. A group of items enclosed in braces is a list from which you are required to choose one item. The items in the list can be arranged either vertically, with aligned braces on each side of the list, or horizontally, enclosed in a pair of braces and separated by vertical lines. For example:

```
LISTOPENS PROCESS { $appl-mgr-name }
                  { $process-name  }

ALLOWSU { ON | OFF }
```

| Vertical Line. A vertical line separates alternatives in a horizontal list that is enclosed in brackets or braces. For example:

```
INSPECT { OFF | ON | SAVEABEND }
```

… Ellipsis. An ellipsis immediately following a pair of brackets or braces indicates that you can repeat the enclosed sequence of syntax items any number of times. For example:

```
M address [ , new-value ]...

[ - ] {0|1|2|3|4|5|6|7|8|9}...
```

An ellipsis immediately following a single syntax item indicates that you can repeat that syntax item any number of times. For example:

```
"s-char..."
```

Punctuation. Parentheses, commas, semicolons, and other symbols not previously described must be typed as shown. For example:

```
error := NEXTFILENAME ( file-name ) ;

LISTOPENS SU $process-name.#su-name
```

Quotation marks around a symbol such as a bracket or brace indicate the symbol is a required character that you must type as shown. For example:

```
"[" repetition-constant-list "]"
```

Item Spacing. Spaces shown between items are required unless one of the items is a punctuation symbol such as a parenthesis or a comma. For example:

```
CALL STEPMOM ( process-id ) ;
```

If there is no space between two items, spaces are not permitted. In this example, no spaces are permitted between the period and any other items:

```
$process-name.#su-name
```

Line Spacing. If the syntax of a command is too long to fit on a single line, each continuation line is indented three spaces and is separated from the preceding line by a blank line. This spacing distinguishes items in a continuation line from items in a vertical list of selections. For example:

```
ALTER [ / OUT file-spec / ] LINE
   [ , attribute-spec ]...
```

!i and !o. In procedure calls, the !i notation follows an input parameter (one that passes data to the called procedure); the !o notation follows an output parameter (one that returns data to the calling program). For example:

```
CALL CHECKRESIZESEGMENT (  segment-id                    !i
                         , error        ) ;              !o
```

!i,o. In procedure calls, the !i,o notation follows an input/output parameter (one that both passes data to the called procedure and returns data to the calling program). For example:

```
error := COMPRESSEDIT ( filenum ) ;                    !i,o
```

!i:i. In procedure calls, the !i:i notation follows an input string parameter that has a corresponding parameter specifying the length of the string in bytes. For example:

```
error := FILENAME_COMPARE_ (  filename1:length         !i:i
                            , filename2:length ) ;     !i:i
```

!o:i. In procedure calls, the !o:i notation follows an output buffer parameter that has a corresponding input parameter specifying the maximum length of the output buffer in bytes. For example:

```
error := FILE_GETINFO_ (  filenum                      !i
                        , [ filename:maxlen ] ) ;      !o:i
```

Notation for Messages

This list summarizes the notation conventions for the presentation of displayed messages in this manual.

Bold Text. Bold text in an example indicates user input typed at the terminal. For example:

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

The user must press the Return key after typing the input.

Nonitalic text. Nonitalic letters, numbers, and punctuation indicate text that is displayed or returned exactly as shown. For example:

```
Backup Up.
```

lowercase italic letters. Lowercase italic letters indicate variable items whose values are displayed or returned. For example:

p-register

process-name

[ ] Brackets. Brackets enclose items that are sometimes, but not always, displayed. For example:

```
Event number = number [ Subject = first-subject-value ]
```

A group of items enclosed in brackets is a list of all possible items that can be displayed, of which one or none might actually be displayed. The items in the list can be arranged either vertically, with aligned brackets on each side of the list, or horizontally, enclosed in a pair of brackets and separated by vertical lines. For example:

```
proc-name trapped [ in SQL | in SQL file system ]
```

{ } Braces. A group of items enclosed in braces is a list of all possible items that can be displayed, of which one is actually displayed. The items in the list can be arranged either vertically, with aligned braces on each side of the list, or horizontally, enclosed in a pair of braces and separated by vertical lines. For example:

```
obj-type obj-name state changed to state, caused by
{ Object | Operator | Service }
```

```
process-name State changed from old-objstate to objstate
{ Operator Request. }
{ Unknown.          }
```

| Vertical Line. A vertical line separates alternatives in a horizontal list that is enclosed in brackets or braces. For example:

```
Transfer status: { OK | Failed }
```

% Percent Sign. A percent sign precedes a number that is not in decimal notation. The % notation precedes an octal number. The %B notation precedes a binary number. The %H notation precedes a hexadecimal number. For example:

```
%005400
```

```
%B101111
```

```
%H2F
```

```
P=%p-register E=%e-register
```

Notation for Management Programming Interfaces

This list summarizes the notation conventions used in the boxed descriptions of programmatic commands, event messages, and error lists in this manual.

UPPERCASE LETTERS. Uppercase letters indicate names from definition files. Type these names exactly as shown. For example:

```
ZCOM-TKN-SUBJ-SERV
```

lowercase letters. Words in lowercase letters are words that are part of the notation, including DDL keywords. For example:

```
token-type
```

!r. The !r notation following a token or field name indicates that the token or field is required. For example:

```
ZCOM-TKN-OBJNAME      token-type ZSPI-TYP-STRING.          !r
```

!o. The !o notation following a token or field name indicates that the token or field is optional. For example:

```
ZSPI-TKN-MANAGER      token-type ZSPI-TYP-FNAME32.         !o
```

Change Bar Notation

Change bars are used to indicate substantive differences between this manual and its preceding version. Change bars are vertical rules placed in the right margin of changed portions of text, figures, tables, examples, and so on. Change bars highlight new or revised information. For example:

> 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.

1 Introduction to DDL

The Data Definition Language (DDL) enables you to define data objects in Enscribe files and to translate these object definitions to source code for programming languages and other HP products.

The DDL language has statements to define data objects and commands to control how the statements are compiled. DDL data objects include:

- Constants
- Definitions (for single fields and groups of fields)
- Records
- Subsystem Programmatic Interface (SPI) token codes
- SPI token maps
- SPI token types

The DDL compiler compiles object definitions and generates any requested output from the compiled definitions. Depending on which commands you enter, the DDL compiler builds a dictionary from the definitions, translates the definitions to FUP commands, or generates object-definition source code in one or more programming languages.

A dictionary acts as a repository for the DDL definitions. It helps to maintain consistency so that the same data, regardless of where it is used, is described in the same way. Although commonly used to describe data in a database, a dictionary can be used to describe other types of data. For example, Transfer applications (only on G-series systems) generally use DDL to define and maintain units-of-work. An application can have more than one dictionary, or the application can maintain all of its data in a single dictionary. The only restriction is that only one dictionary can reside on a subvolume.

Subsystems that define SPI messages in a Distributed Systems Management (DSM) environment must define the SPI message tokens with DDL, optionally add them to a dictionary, and compile the definitions to C, COBOL, Pascal (on D-series systems), pTAL, TACL, or TAL code.

This section provides an overview of DDL functionality, including compiling and translating data definitions, using DDL definitions, creating a dictionary, creating a database, generating source code, maintaining a dictionary, and examining a dictionary.

Topics:

Compiling and Translating Data Definitions

The main functions of the DDL compiler are:

- Compiling statements that define data objects
- Translating compiled definitions to source code for host languages and FUP

Note. Starting H06.20/J06.09 RVUs, DDL is available with the DDL2 object file and the DDL object file. You can use either of the object files and create definitions. However, the DDL object file and the DDL2 object file are not compatible with each other. Therefore, when migrating from one DDL object file to another, you must generate the schema from the existing dictionary.

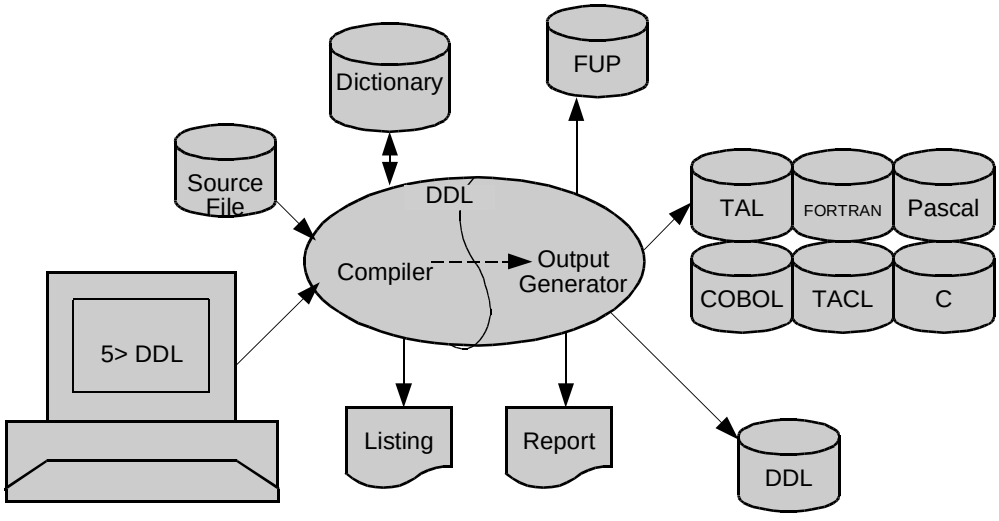
The DDL compiler also generates reports on the contents of a dictionary. All DDL statements and commands are passed as input to the DDL compiler input can be from a file or directly from a terminal. In either case, the DDL compiler checks the syntax and semantics of all statements and commands.

The DDL compiler translates the statements that define DDL objects to internal data definition format. If a dictionary is open for read and write access, the DDL compiler writes the compiled objects to the dictionary.

If output is requested, the DDL output generator retrieves the compiled object definitions from the dictionary (if open), translates the definitions to the appropriate source code, and writes the translated definitions to all open source code files and devices.

Figure 1-1 on page 1-3 is an overview of the DDL compiler.

Figure 1-1. DDL Compiler Overview



Using DDL Definitions

You use DDL statements to define, modify, delete, and display data in a dictionary. You use DDL commands to create and open DDL dictionaries and to generate files containing FUP commands, data definition source code in different languages, and report specifications. You can use DDL to perform these functions:

- Create a schema

  A DDL schema is composed of DDL statements that define the DDL objects. You can create a schema in an EDIT file and submit the file (called the *schema file*) to the DDL compiler, or you can run the DDL compiler and enter the definitions interactively.

- Create a dictionary

  A dictionary is a DDL database that contains the objects defined in a schema. When you run the DDL compiler to compile the schema, you can direct the DDL compiler to store the object definitions from the schema in the open dictionary.

- Create a database

  You can direct the DDL compiler to generate FUP commands from the record definitions in a schema or dictionary and to write these commands to an EDIT file. You can edit the commands, if needed, and then submit the command file to FUP to create your database files.

- Generate source code for programming languages

  You can direct the DDL compiler to translate the DDL object definitions from a schema or dictionary to C, COBOL, FORTRAN, Pascal (on D-series systems), pTAL, TACL, or TAL source code and to write the code to an EDIT file (called a *source code file*). You can edit this code, if needed, and then add it to your application program.

- Create messages

  You can use DDL to define messages for interprocess communication and store the message definitions in a dictionary. Having stored the messages, you can translate them to the appropriate programming language or languages.

  If you use SPI messages for interprocess communication in a DSM environment, you define the SPI message tokens with DDL and translate them to C, COBOL, Pascal (on D-series systems), pTAL, TACL, or TAL source code. SPI uses the token definitions to build the SPI messages.

- Maintain a dictionary

  You can use DDL commands and statements to add new object definitions to a dictionary, to delete or change existing definitions, and to build a schema from the dictionary and write the schema to an EDIT file. You can generate an entire backup schema, or you can write only selected sections of the dictionary to a DDL schema file.

- Examine a dictionary

  You can request a schema report that describes each object definition in a schema or dictionary, or that describes selected definitions. You can also use a set of Enform Plus queries provided by HP to produce more complex reports on all the objects in a dictionary, their structure, how they are linked, which objects are referenced by other objects, and so forth. You can, if you choose, modify the supplied queries or write your own.

Creating a Dictionary

A dictionary is a database consisting of 14 prenamed and predefined files. Because the files have fixed names, you can have only one dictionary on any subvolume. A dictionary can be created either by running the DDL compiler or by running Pathmaker, a NonStop Transaction Services/MP (NonStop TS/MP) application systems generator.

When you run the DDL compiler, you must open the dictionary with the DICT or DICTN command. This command creates a dictionary, if one does not already exist, or opens an existing dictionary. You can pass the DICT or DICTN command to the compiler as a parameter in the RUN DDL command.

When you add a Pathmaker project, The Pathmaker program creates a dictionary for you. The dictionary the Pathmaker program creates is part of a larger database that contains application design information. The Pathmaker program enters application design information to the dictionary.

Both Pathmaker dictionaries and dictionaries created from the DDL compiler can be written to by more than one user at the same time.

Once a dictionary is created, you can enter object definitions in it. If you have a lot of complex definitions, you probably want to present them in a schema file rather than entering them interactively.

The schema statements can be submitted to the DDL compiler as an input file in a noninteractive session or submitted with a SOURCE command in an interactive session. (You can also enter statements directly in an interactive session.) In each case, the DDL compiler compiles the object definition statements and, if a dictionary is open, writes the objects to the dictionary.

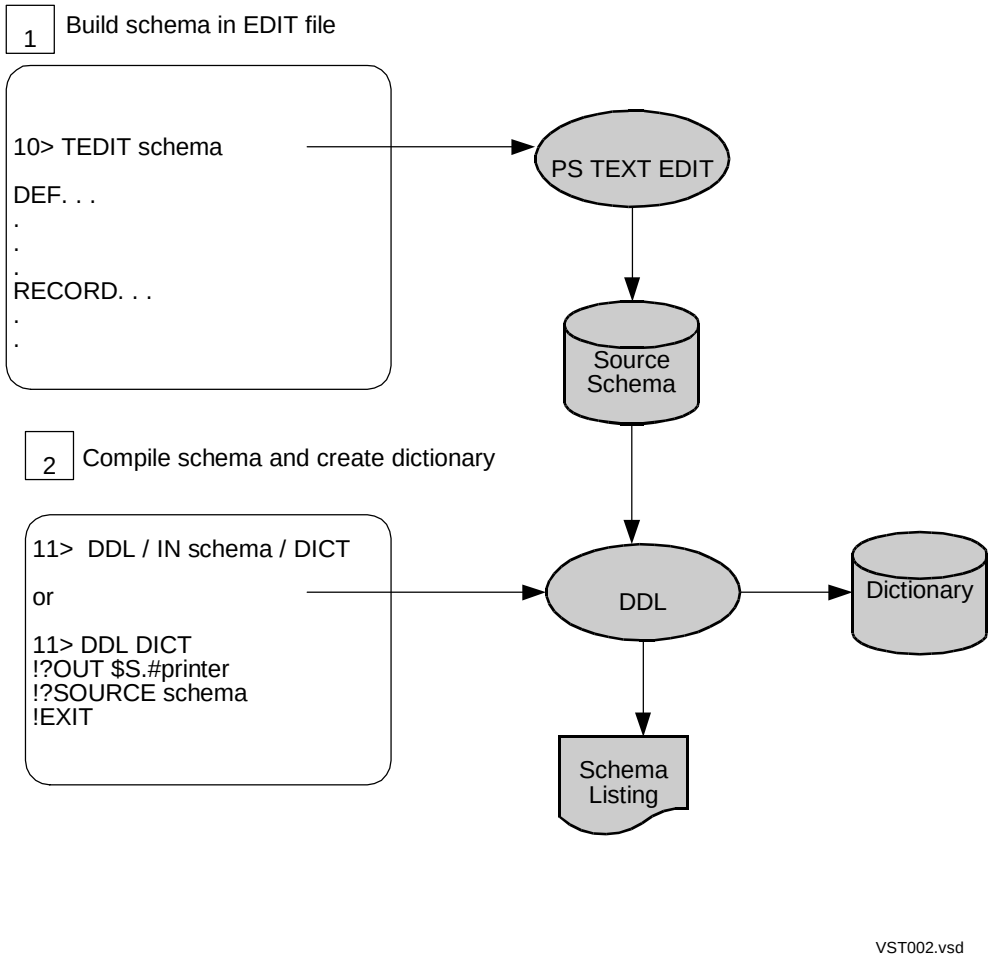
Unless suppressed by a NOLIST command, the DDL compiler automatically produces a compiler listing. By default, the listing is sent to the home terminal of the DDL compilation process. You can specify a different listing destination in the OUT run option of the RUN DDL command or in an OUT command.

A dictionary is not actively integrated with the database files or source code it describes. If you change a dictionary, the DDL compiler does not automatically change the associated database or source code. Conversely, if you change a database or source code directly, the associated dictionary is not affected.

Figure 1-2 on page 1-6 shows the steps for building a schema and compiling it into a dictionary:

1. Run a text editor program and enter DDL statements and, optionally, DDL commands into an EDIT file. This file contains your schema (it is your schema file).
2. Run the DDL compiler using the schema file as the input file, or run the DDL compiler interactively, submitting the schema file with a SOURCE command. In either case, use the DICT or DICTN command to open the dictionary on a specified volume and subvolume or on the default volume and subvolume. Optionally, you can specify a print device to receive the compiler listing.

Figure 1-2. Creating a Dictionary



Creating a Database

You use DDL RECORD statements to define database files. If you specify a FUP command in a schema file or interactively, the DDL compiler opens a FUP file, translates each subsequent RECORD statement to FUP file creation commands, and writes the commands to the open FUP file.

The FUP command file is an EDIT file. If you want to add to the FUP commands generated by the DDL compiler, you can close the FUP file and then modify it with a text editor. For example, you can create a partitioned file by adding the PART parameter to the FUP code. For information on file attributes that you cannot specify in DDL, see the *File Utility Program (FUP) Reference Manual*.

When you finish editing the file creation commands, you can run the FUP program using the FUP command file as the input file. FUP creates the files according to the file attributes originally specified in the DDL RECORD statements. For a full description of the FUP program and its commands, see the *File Utility Program (FUP) Reference Manual*.

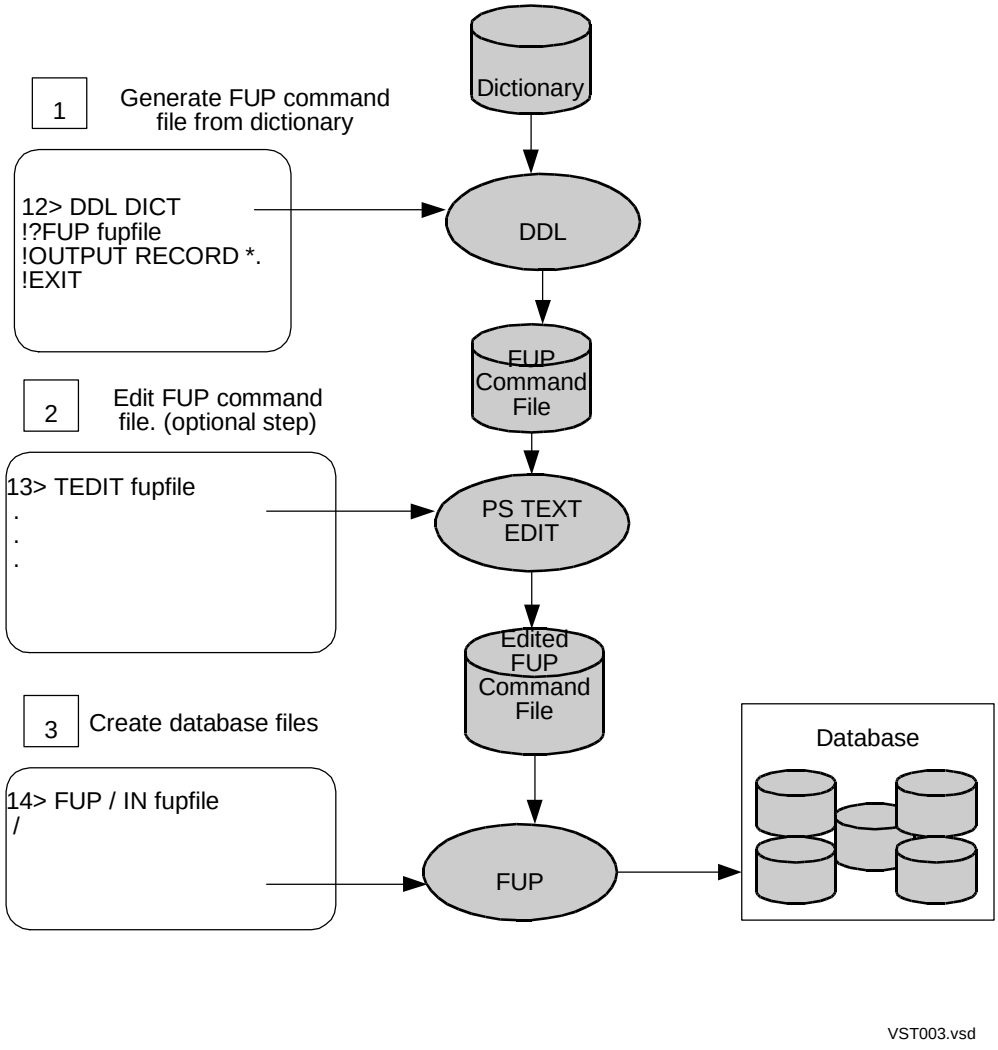
You can also generate FUP file-creation commands from records previously stored in a dictionary. To do this, open the dictionary and a FUP file, and use an OUTPUT statement to select record definitions from the dictionary for translation to FUP file creation commands.

Figure 1-3 on page 1-8 shows the steps for creating database files from DDL record definitions stored in a dictionary:

1. Run the DDL compiler interactively, open the dictionary, and open the FUP command file. Enter the OUTPUT RECORD statement, specifying records defined in the dictionary. The DDL compiler reads the record definitions from the dictionary and writes file creation commands for each specified record to the open FUP file.
2. (Optional Step). Exit from the DDL compiler and run the EDIT program to make any changes you want to the FUP commands in the FUP file; or stay in the DDL compiler, close the FUP file, use the EDIT command to modify the FUP file, and then exit from the DDL compiler.
3. Run FUP with the FUP command file as the input file. FUP creates the database files from the commands in the command file.

The files in a Guardian environment database are managed by the Enscribe record manager. For more information about file structures and access methods, see the *Enscribe Programmer's Guide*.

Figure 1-3. Creating Database Files



Generating Source Code

The DDL compiler can generate source code for definitions and records in any of the languages DDL supports: C, COBOL, FORTRAN, Pascal (on D-series systems), pTAL, TACL, or TAL. In addition, the DDL compiler can generate source code for constants and SPI token types, token codes, and token maps in C, COBOL, Pascal (on D-series systems), pTAL, TACL, or TAL. The DDL objects, once translated to source code, are used for a variety of application functions. DDL objects can describe the data in a database and the data local to an application; messages used for interprocess communication; units of work for the Transfer Delivery System (only on G-series systems); and SPI tokens used to build SPI messages.

For the DDL compiler to generate source code, the appropriate language source code file must be open. To open a language source code file, you can include one or more C, COBOL, FORTRAN, Pascal (on D-series systems), pTAL, TACL, or TAL commands in the schema. When the DDL compiler compiles the schema, it opens a file for each specified language, translates the subsequent object definition statements to source code for those languages, and writes the code to the language files.

You can also generate source code from an existing dictionary. To do this, you can add a language command to your schema and recompile the entire schema. Typically, however, you run the DDL compiler interactively, open the dictionary and a language source code file, and use the OUTPUT statement to specify the object definitions you want generated in the source language. This technique is particularly useful for writing selected definitions to the language source code file.

Figure 1-4 on page 1-10 shows two techniques for generating language source code. These techniques are:

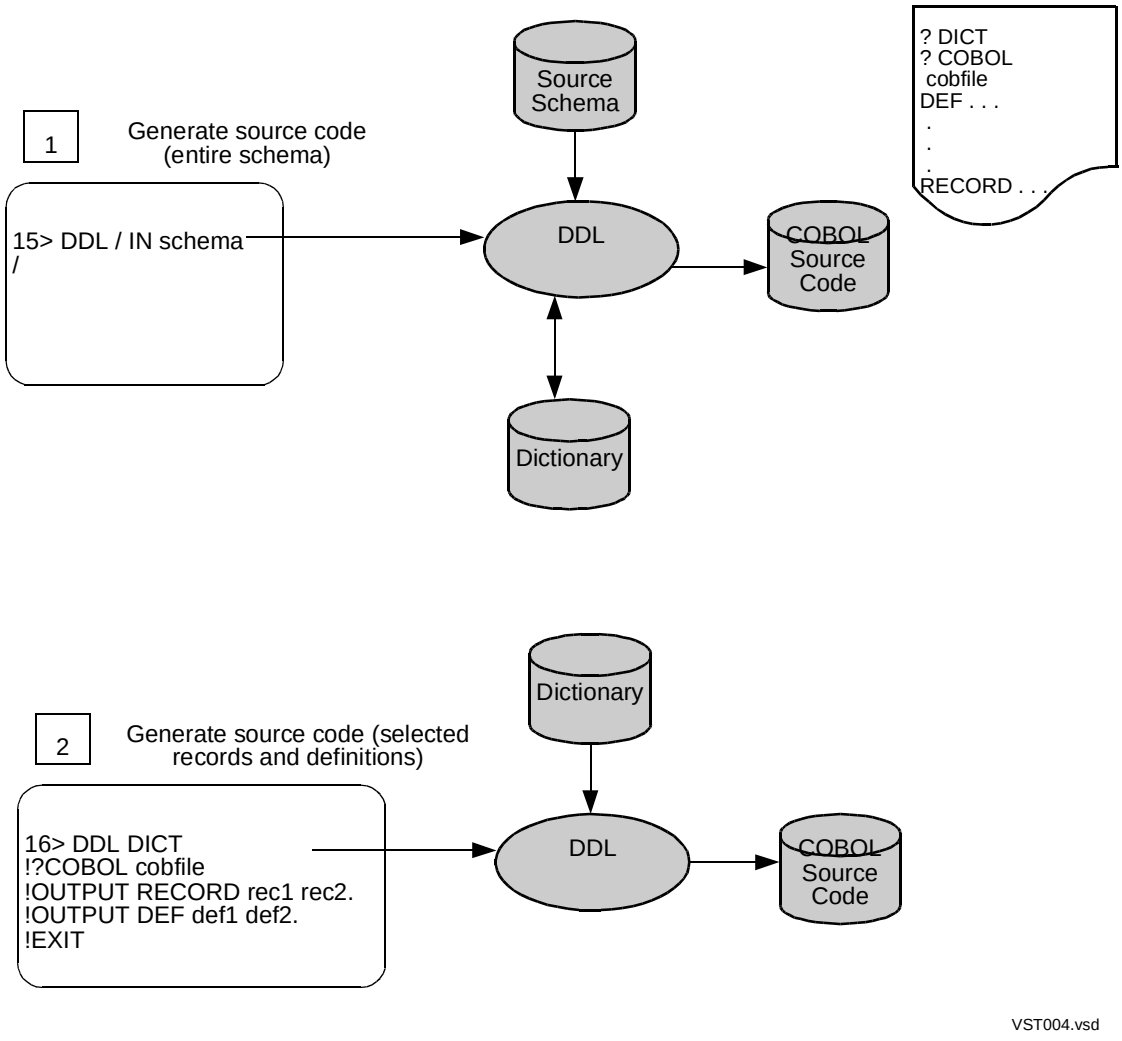
1. Generate source code for the entire schema.

   Enter the commands to open a dictionary and a language source code file in the schema before entering the definitions. Then run the DDL compiler to compile the schema. the DDL compiler builds the dictionary and then generates source code in any open language source code file.

2. Generate source code for selected definitions.

   Run the DDL compiler interactively; open the dictionary and a language source code file with the appropriate command; use an OUTPUT statement to specify the definitions you want the DDL compiler to translate to source code. The DDL compiler generates the code for the specified definitions and writes it to the open language source code file. Close the language source code file before doing any more interactive processing.

Figure 1-4. Generating Source Code



TNS and Native Compilers

The native compilers align data for optimal performance on TNS/R and TNS/E systems by default. This default alignment is different and incompatible with the default alignment generated by the TNS compilers.

Because of this data alignment incompatibility, the D40 DDL compiler was enhanced to generate source code that produces the same data alignment, regardless of whether the TNS compilers or native compilers are used. To ensure the same data alignment, the D40 DDL compiler emits `fieldalign shared2` pragmas for C and FIELDALIGN SHARED2 directives for TAL and pTAL.

Host-language source code files used by native programs and shared with TNS programs must be generated using a version D40 or later DDL compiler. Host-language source code files supplied by HP have already been generated by the correct version of the DDL compiler.

If your native programs do not share host-language source code files with TNS programs, you can direct the DDL compiler to align data optimally for the native TNS/R or TNS/E environment. To do so, specify the command FIELDALIGN_SHARED8 on page 9-58 when storing data in a dictionary. While DDL source code files generated with SHARED8 alignment can be used by TNS and native programs, the performance of TNS programs is degraded.

Maintaining a Dictionary

After a dictionary is created, change is inevitable. You might need to add new objects or to change or delete existing objects. DDL schema files help you perform these maintenance functions.

The DDL compiler can generate schema statements from the dictionary and write these statements to a DDL schema file. In its simplest role, a DDL schema file provides a backup schema for DDL dictionaries created from the DDL compiler. Suppose the dictionary but not the original schema has been changed, or suppose the original schema is lost. In either case, the DDL compiler can generate a new schema that accurately reflects the current dictionary and write this schema to a DDL schema file.

Do not attempt to back up a dictionary that is part of a Pathmaker catalog using this technique. Pathmaker dictionaries contain application design information that is not in generated DDL schemas.

The DDL compiler can compile the entire DDL schema file into a new dictionary just as it compiled the original schema. The DDL compiler can also compile selected sections of a schema file (or of any schema) and add them to an existing dictionary.

A DDL schema file is also useful for modifying dictionary objects that are used, or referenced, by other objects. To change or delete an object that is referenced, you must first delete all the objects that refer to that object, change the referenced object, and then reenter the deleted objects. The DDL compiler can generate all the source code necessary to perform these operations and write the source code to an open DDL schema file. To update the dictionary, you need only compile this file, first modifying it if necessary.

Figure 1-5 on page 1-13 shows two typical maintenance operations:

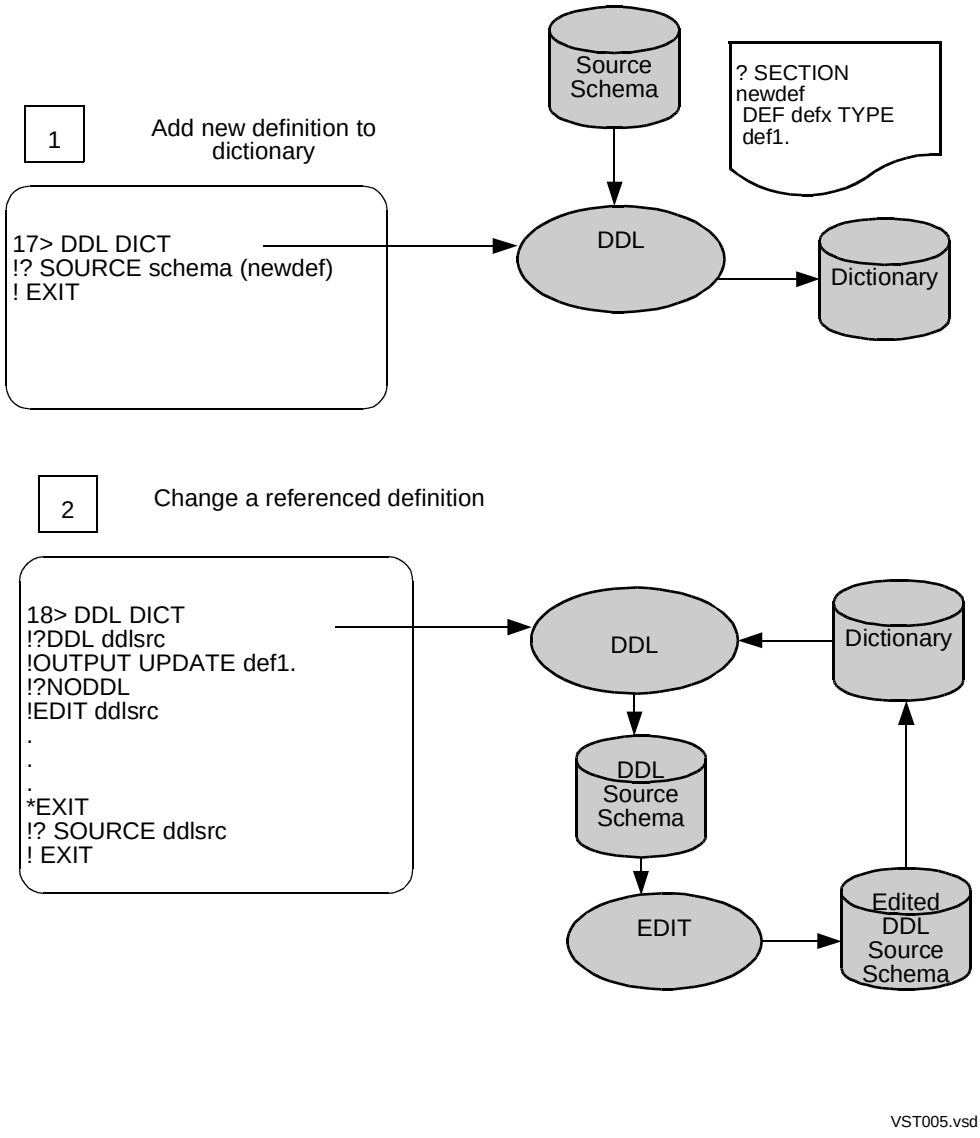
1. Add a new object to the dictionary.

   Define the new object in a schema file. If you add the new object definition to your original schema, precede the definition with a SECTION command. Then run the DDL compiler interactively, and use a SOURCE command to compile the definition in the schema and write the new object to the dictionary.

2. Change a referenced object.

   Open the dictionary and a new DDL schema file. Use an OUTPUT UPDATE statement to identify all objects that refer to the object you want to change and write the necessary statements to the open DDL schema file. Close the schema file and edit the DDL file, if necessary. Then compile the source statements into the open dictionary with a SOURCE command.

Figure 1-5. Maintaining a Dictionary



Examining a Dictionary

The DDL compiler produces a schema listing by default. In addition, the DDL compiler can produce a schema report that provides information about the object definitions in a schema. For each object, this report lists its type and size, its byte offset from the start of a group definition, and any definitions referenced by other objects.

You can also generate reports on the dictionary itself. HP supplies a set of Enform Plus queries that provide information about any dictionary. These reports are particularly useful for anyone acting as the administrator of a database. The reports:

- Show all the components of a dictionary.
- Tell when structures were last modified, which version of the DDL compiler produced the dictionary, which definitions are used by which other definitions, and where they are used.
- List file definitions and the key fields used by files.
- List all display text and comments stored in the dictionary.

If you want to produce your own dictionary reports, you can use the Enform Plus reports supplied by HP as templates, changing them or adding to them to suit your needs.

Figure 1-6 on page 1-15 shows how to produce a DDL schema report and how to produce dictionary reports using Enform Plus:

1. Produce schema report

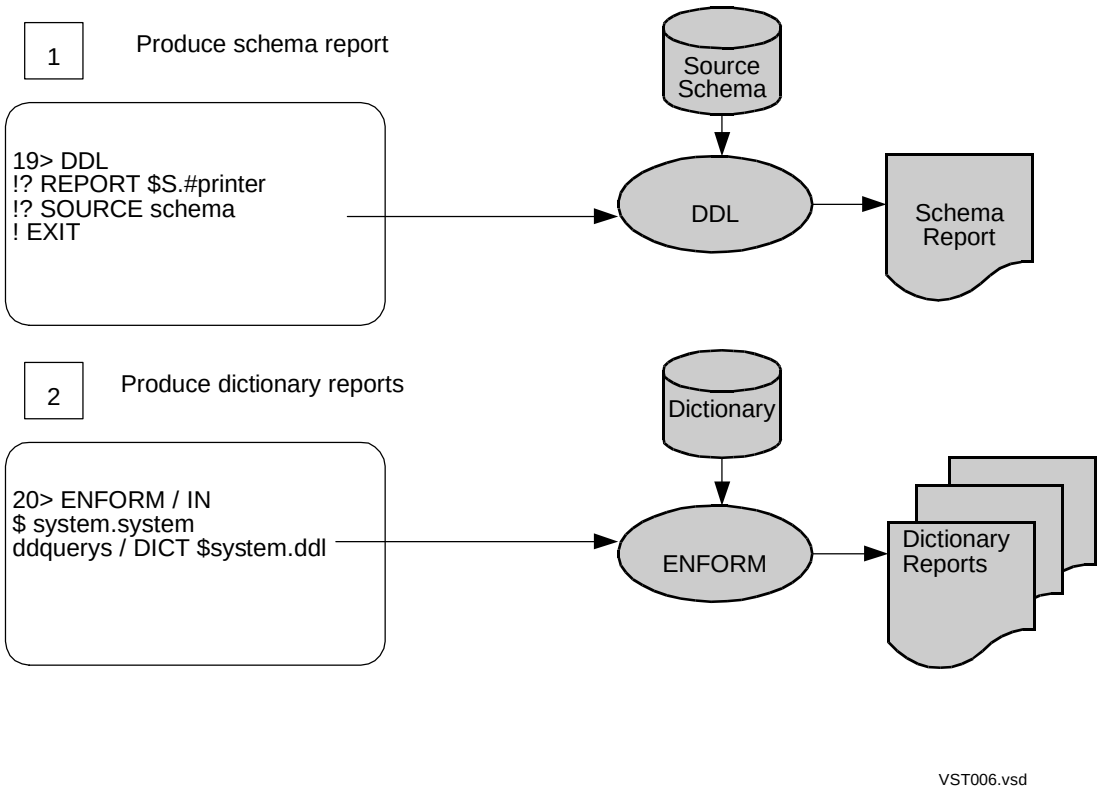
   Run the DDL compiler with the schema as the input file and include the REPORT command in the RUN DDL command. Or, run the DDL compiler interactively, compile a DDL schema file, and request the report with a REPORT command. The dictionary need not be open. By default, the DDL compiler sends the report to its home terminal; alternatively, you can specify a printer as the report destination. If you need information only on selected definitions in the dictionary, you can run the DDL compiler interactively with the REPORT command specified in the RUN DDL command and request reports using the OUTPUT statement.

2. Produce dictionary reports

   The dictionary must exist, but it need not be open. Run Enform Plus from the volume and subvolume on which the dictionary resides using the file $SYSTEM.SYSTEM.DDQUERYS as the input file. To select particular reports, you can run Enform Plus interactively and specify sections (R1 to R12) of the query file. Each section corresponds to a report.

For more information on producing dictionary reports, see Appendix E, Dictionary Reports.

Figure 1-6. Examining a Dictionary



2 DDL Language Elements

A DDL schema consists of DDL statements, DDL commands, and comments. You must enter statements, commands, and comments according to strict syntax.

This section briefly describes the DDL language elements common to statements and commands:

Names

DDL names are:

- Constant names
- Definition names
- Record names
- Field names
- Group names
- Condition-name item names (level 88)
- Enumeration item names (level 89)
- Renames item names (level 66)
- Token-type names (SPI only)
- Token-code names (SPI only)
- Token-map names (SPI only)

Topics:

Syntax

Every DDL name:

- Begins with either a letter (*A-Z* or *a-z*) or an underscore (_)
- Has a maximum of 30 ASCII characters, which are any of:
  - Letters
  - Decimal digits (*0-9*)
  - Hyphen (-)
  - Underscore
- Does not end with a hyphen

Uppercase letters are not distinguished from lowercase letters, and any underscores are part of the name.

You can make the name of a DDL elementary field unique by qualifying it with a record name or with one or more group names; for example:

Qualified Field Name	Example
record.field	CUSTOMER.CUSTNUM
group.group.field	CUSTINFO.ADDR.CITY

Example 2-1. DDL Names

```
A1
field-2
Employee-record-1
NEW-EMPLOYEE-NUMBER
_EMP
_EMP_
ZSPI-TKN-RETCODE
```

Restrictions

- DDL constant names cannot be DDL keywords (which are listed in Keywords on page 2-6).
- Other DDL names cannot be:
  - DDL reserved words (which are listed in Keywords on page 2-6)
  - Reserved words in the host language for which the DDL compiler is generating source code (which are listed in the host-language manuals in Prerequisite Manuals on page xxiv)

    If a DDL name is a host-language reserved word, the DDL compiler issues an error message and does not generate host-language source code for the object identified by the reserved word or containing an element identified by the reserved word. In this way, the DDL compiler avoids generating code that does not compile.

- If a DDL name in a RECORD statement is an Enform Plus reserved word, the DDL compiler warns you that you cannot use that record for an Enform Plus query. For a list of Enform Plus reserved words, see the *Enform Plus Reference Manual*.
- SPI variable names and other names defined by HP begin with the letter *Z*. To avoid conflict with a current or future HP name, do not begin a name with the letter *Z* unless you are referring to an existing SPI variable name, such as ZSPI-TKN-RETCODE.

File Names

The DDL compiler recognizes these types of file names:

- Local File Names on page 2-3
- Network File Names on page 2-4

Local File Names

Local file names identify files on a single system (or node). If the local file name is not in the current subvolume or volume, you must qualify the name with a specific subvolume or volume.

```
[$volume-name.][subvolume-name.]file-name
```

volume-name

is one alphabetic character followed by up to 6 alphanumeric ASCII characters.

Default: current default volume name

subvolume-name

is one alphabetic character followed by up to 7 alphanumeric ASCII characters.

Default: current default subvolume name

file-name

is one alphabetic character followed by up to 7 alphanumeric ASCII characters.

Network File Names

Network file names identify files that are accessed across a network of HP NonStop systems. Always identify a network file by a fully qualified file name that includes the system, volume, and subvolume.

```
\system-name.$volume-name.subvolume-name.file-identifier
```

system-name

is one alphabetic character followed by up to 6 alphanumeric ASCII characters.

volume-name

is one alphabetic character followed by up to 6 alphanumeric ASCII characters.

Note. The maximum number of characters in the volume name of a network file is less than the maximum for the volume name of a local file: only 6 characters plus the dollar sign rather than 7 characters plus the dollar sign for a local volume.

subvolume-name

is one alphabetic character followed by up to 7 alphanumeric ASCII characters.

file-identifier

is one alphabetic character followed by up to 7 alphanumeric ASCII characters.

Table 2-1. DDL File Names

File Name	Description
\DALLAS.$DATA.SALES.CUSTOMER	Network file name
$DATA.SALES.CUSTOMER	Fully qualified local file name
$DATA.CUSTOMER	Local file name, assumes current default subvolume
SALES.CUSTOMER	Local file name, assumes current default volume
CUSTOMER	Local file name, assumes current default volume and subvolume

Locale Names

The DDL compiler recognizes locale names for internationalization support. Using the clause LN on page 6-13, you can specify the language, territory, and character set for a text item. See Table 6-3, Supported Locale Names, on page 6-14.

Numbers

The DDL compiler recognizes both decimal and octal numeric values. An octal number is specified by a percent sign (%). An unsigned number is positive by default—a plus (+) sign is optional. A negative number is specified with a minus sign (-). Any plus or minus sign must immediately precede the number.

Examples:

Number	Description
39	Positive decimal value
-2	Negative decimal value
+8	Positive decimal value
-%10	Negative octal value (decimal -8)
+%17	Positive octal value (decimal 15)
%100	Positive octal value (decimal 64)

Strings

A DDL string is any combination of ASCII or national characters within quotation marks. The maximum length of a DDL string is 130 ASCII characters or 64 national characters.

If you use the COLUMNS command to specify the number of significant columns in an input line, the maximum string length is constrained by that number, *columns* :

Maximum ASCII characters in a string *columns* - 2

Maximum national characters in a string (*columns* - 4)/2

The maximum number of ASCII characters is the input line length minus 2 characters for the quotation marks. Because each national character requires 2 bytes, the maximum number of national characters is half of the following: the input line length minus 1 character for the letter *N* that precedes the string, 2 characters for the quotation marks, and 1 character to make the string an even number of bytes.

If the string has quotation marks, you must enter each quotation mark twice to distinguish the quotation mark character from the string delimiter. You can use either single or double quotation marks as string delimiters.

Example 2-2. DDL Strings

```
"C00"
"12.650"
"Enter a 2-character state code."
"M<(999) 999-9999>"
"Use quotes "" "" before and after the name."
'Use quotes '' '' to delimit a string.'
'alpha-string'
```

National Literals

```
{ N }{ "2-byte-character ..." }
{ n }{ ‘2-byte-character ...’ }
```

2-byte-character

must occupy two bytes internally. The first byte must not contain the binary equivalent of a quotation mark (“).

The opening quotation mark (" or ‘) must immediately follow *N* or *n*, with no intervening space or line break.

Keywords

A DDL keyword has a specific meaning when placed at a keyword position within a statement. Keywords must be spelled exactly as this manual shows.

DDL keywords cannot be defined as constant names.

Note. To distinguish keywords from variables, this manual capitalizes keywords in syntax descriptions. When you use keywords in DDL input, however, you do not need to capitalize them. DDL keywords are not case-sensitive.

DDL keywords:

A B C D E F H I J K L M N O P Q R S T U V Y Z

A

ALL
ALLOWED
ARE
AS
ASCENDING
ASSIGNED
AUDIT
AUDITCOMPRESS

B

BE
BEGIN
BINARY
BIT
BLOCK
BUFFERED

BUFFERSIZE
BY

C

CFIELDALIGN_MATCHED2
CHARACTER
C_MATCH_HISTORIC_TAL
CODE
COMP
COMP-3
COMPLEX
COMPRESS
COMPUTATIONAL
COMPUTATIONAL-3
CONSTANT
CRTPID
CURRENT

D

DATE
DATETIME
DAY
DCOMPRESS
DEF
DEFINITION
DELETE
DEPENDING
DESCENDING
DEVICE
DISPLAY
DUPLICATES

E

EDIT-PIC
END
ENTRY-SEQUENCED
ENUM
EXIT
EXT
EXTERNAL

F

FILE
FIELDALIGN_SHARED8
FILLER
FLOAT
FNAME
FNAME32
FOR
FRACTION

H

HEADING
HELP
HIGH-NUMBER
HIGH-VALUE
HOUR

I

ICOMPRESS
INDEX
INDEXED
INTERVAL
IS

J

JUST
JUSTIFIED

K

KEY
KEY-SEQUENCED
KEYTAG

L

LN
LOGICAL
LOW-NUMBER
LOW-VALUE
LOW-VALUES

M

MAXEXTENTS
MINUTE
MONTH
MUST

N

N
NO
NOT
NOVALUE
NOVERSION
NULL

O

OCCURS
ODDUNSTR
OF
ON
OUTPUT

P

PACKED-DECIMAL
PHANDLE
PIC
PICTURE

Q

QUOTE
QUOTES

R

RECORD
REDEFINES
REFRESH
RELATIVE
RENAMES
RIGHT

S

SECOND
SEQ
SEQUENCE
SERIALWRITES
SETLOCALENAME
SHOW
SPACE
SPACES
SPI-NULL
SQL
SQLNULL
SQL-NULLABLE
SSID
SUBVOL
SYSTEM

T

TACL
TALUNDERSCORE
TEMPORARY
THROUGH
THRU
TIME
TIMES
TIMESTAMP
TO
TOKEN-CODE
TOKEN-MAP
TOKEN-TYPE
TRANSID
TSTAMP
TYPE

U

UNSIGNED
UNSTRUCTURED
UPDATE
UPSHIFT
USAGE
USE
USERNAME

V

VALUE
VARCHAR
VARYING
VERIFIEDWRITESVERSION

Y

YEAR

Z

ZERO
ZEROES
ZEROS

Reserved Words

DDL reserved words are a subset of DDL Keywords on page 2-6. DDL reserved words cannot be defined as DDL data names.

DDL reserved words:

ARE
BINARY
CHARACTER
COMPLEX
END
ENUM
FILLER
FLOAT
IS
LOGICAL
OF
ON
THROUGH
THRU
TIME
TIMESTAMP

Special Characters

Table 2-2. DDL Special Characters

Name	Character	Function
Blank		Separates keywords, data names, and other language elements.
Percent Sign	%	Denotes an octal number.
Quote	“	Used as a delimiter for various language elements.
Apostrophe	‘	Can be used in place of a quotation mark.
Exclamation Point	!	The DDL interactive prompt; separates listing comments from statements and commands; after a file name in a DDL command, ! clears the file.
Asterisk	*	Denotes dictionary comment when * is the first character of a line; can be used as a wild-card character for dictionary references.
Question Mark	?	Denotes a command when ? is the first character of a line.
Comma	,	Separates multiple commands on the same line; separates multiple sections in a SOURCE command.
Period	.	Ends statements and parts of compound statements; used in qualification of field names.
Semicolon	;	Needed in previous DDL versions, but now interpreted as blank.

Comments

The DDL compiler supports two types of comments:

	Begins with:	Ends with:	Appears in:
Dictionary Comments	Asterisk (*) in first column of source line	End of source line	Dictionary and all open language source code files
Compiler Listing Comments	Exclamation point (!)	Another exclamation point or end of source line	Compiler listing only

The comment in Example 2-3 on page 2-12 begins with the asterisk (*) and ends when the line ends; that is, the entire line is a comment.

Example 2-3. dictionary Comment

```
* CUSTNUM is the primary key
```

In Example 2-4 on page 2-13:

- The first comment begins with the exclamation point (!) and ends when the line ends; that is, the entire line is a comment.
- The second comment begins with the first exclamation point and ends with the second exclamation point; that is, the second comment is "`!numeric key!`".

Example 2-4. DDL Compiler Listing Comments

```
!  The CUSTOMER record is on page 8
DEF custnum  !numeric key!  PIC 9(4).
```

Dictionary Comments

Dictionary comments describe a field or a group of fields within a data structure. The DDL compiler stores any dictionary comments associated with a data structure in the dictionary with that structure if the dictionary is open and if a COMMENTS command is specified.

DDL has two types of dictionary comments:

- User-Defined Dictionary Comments on page 2-13
- Dictionary Comments Generated by the DDL Compiler on page 2-14

Regardless of how they originate, all dictionary comments begin with an asterisk in the first character position of an input line and continue for the remainder of the line (as in Example 2-3 on page 2-12). Following the asterisk, a comment can consist of any ASCII characters.

User-Defined Dictionary Comments

You can precede any CONSTANT, DEFINITION, RECORD, TOKEN-CODE, TOKEN-MAP, or TOKEN-TYPE statement with a comment. The DDL compiler groups consecutive dictionary comment lines together as a single comment.

In Example 2-5 on page 2-13, the three dictionary comment lines form a single comment that the DDL compiler stores with the `empnum` dictionary entry.

Example 2-5. User-Defined Dictionary Comments

```
* Employee Number definition
* empnum uniquely identifies employees
* Possible values:  0 - 9999

DEFINITION  empnum   PIC 9(4).
```

A user-defined dictionary comment can precede a field or group description within a DEFINITION or RECORD statement, as in Example 2-6 on page 2-14.

Example 2-6. User-Defined Dictionary Comments

```
DEFINITION custinfo.

* This field is a unique customer identifier:
  02 custnum  PIC 9(4).
* This group has the customer name in sequence:
  02 custname.
     04 last-name    PIC X(12).
     04 first-name   PIC X(8).
     04 initial      PIC X(2).
END
```

You can put user-defined dictionary comments in:

- The Dictionary

  The DDL compiler stores user-defined dictionary comments in the dictionary only if the dictionary is open and comments are specifically requested by a COMMENTS command. The DDL compiler stores any comments that follow a COMMENTS command in the open dictionary. A NOCOMMENTS command causes the DDL compiler to stop storing comments in the dictionary.

- Source Code Files

  If dictionary comments are stored in the dictionary, the DDL compiler automatically reproduces these comments in any open C, COBOL, DDL, FORTRAN, Pascal (on D-series systems), TACL, or TAL source code file. You can suppress the writing of dictionary comments to any of these open source code files by entering the NOCLISTOUT command. You can resume reproducing comments with a CLISTOUT command.

- The Compiler Listing

  The DDL compiler puts all user-defined dictionary comments in its compiler listing by default, whether or not the dictionary is open and whether or not you specify a COMMENTS command. You can suppress the listing of dictionary comments by issuing the NOCLISTIN command, and you can resume listing dictionary comments by issuing a CLISTIN command.

Dictionary Comments Generated by the DDL Compiler

The DDL compiler generates dictionary comments that report the date and time the schema was first compiled and the date and time each definition and record is compiled. These timestamp comments are always added to the dictionary. They are also added to any open source code files unless suppressed with the NOTIMESTAMP command. A TIMESTAMP command causes the DDL compiler to add subsequent timestamp comments to any open source code files.

Compiler Listing Comments

The DDL compiler puts compiler listing comments only in its compiler listing, not in the dictionary or in host-language source code files. Like dictionary comments, DDL has two types of listing comments:

- User-Defined Compiler Listing Comments on page 2-15
- Production Comments on page 2-15

User-Defined Compiler Listing Comments

The DDL compiler always puts user-defined compiler listing comments in its compiler listing. You cannot suppress these comments as you can suppress dictionary comments.

A user-defined compiler listing comment begins with an exclamation point (!) and ends at the next exclamation point or at the end of the input line. You can include a listing comment on the same line as a DDL statement. You cannot include a listing comment on the same line as a command.

A listing comment can be on a line by itself or between clauses in a statement.

Example 2-7. User-Defined Compiler Listing Comment on Line by Itself

```
!Suppress the listing of dictionary comments
?NOCLISTIN
```

Example 2-8. User-Defined Compiler Listing Comments Between Clauses

```
DEFINITION empnum !employee #! PIC 9(4). ! used by EMPLOYEE
```

In Example 2-8 on page 2-15:

- The first comment is "`!employee #!`".
- The second comment is "`! used by EMPLOYEE`".

Production Comments

The DDL compiler always puts production comments in the compiler listing. Production comments follow the name of each compiled element to describe the compiler actions. Production comments include error and warning messages.

Like user-defined listing comments, production comments occur only in the compiler listing. You cannot suppress any production comments except warning messages, which you can suppress by issuing a NOWARN command.

Statements

Each of the DDL statements in Table 2-3, DDL Statements That Define or Replace Objects, on page 2-17:

- Defines or replaces an object in the open dictionary, where other DDL statements can use it
- If a source code file is open, translates the definition of the object to the language of the source code file and write the definition to the source code file

The DDL statements in Table 2-4, DDL Statements That Display Objects, on page 2-17 display objects that are in the open dictionary.

The statement DELETE on page 8-1 deletes specified objects from the open dictionary.

The statement EXIT on page 8-4:

- Ends the DDL session
- Closes any files that were opened in the session
- Returns control to the command interpreter

Syntax rules for DDL statements:

- Every simple statement except EXIT must end with a period.
- Every compound statement must end with END, optionally followed by a period. These are compound statements:
  - Field DEFINITION statements that include BEGIN
  - Group DEFINITION statements
  - RECORD statements
  - TOKEN-MAP statements
- An input line can include more than one statement.
- Statements can continue on succeeding input lines without any continuation character.

Table 2-3. DDL Statements That Define or Replace Objects

Statement	Object	How Other DDL Statements Can Use Object
CONSTANT on page 4-1	Constant	As a literal value
DEFINITION on page 5-1	Elementary or group data structure	To define: ● Other data structures ● Records ● Token types ● Token maps
RECORD on page 5-8	Disk file record	To define other records
TOKEN-CODE on page 7-8	SPI token code of a simple token	
TOKEN-MAP on page 7-13	SPI token code of an extensible structured token	
TOKEN-TYPE on page 7-2	SPI token type	To define SPI token codes

Table 2-4. DDL Statements That Display Objects

Statement	Description
OUTPUT on page 8-5	Reads objects from the open dictionary and writes them to any open DDL schema file, FUP source code file, REPORT file, or host-language source code file
OUTPUT UPDATE on page 8-7	Generates DDL source code that updates every referenced object in the open dictionary and writes this code to the open DDL source code file for subsequent compilation
SHOW USE OF on page 8-11	Lists the objects in the open dictionary that directly or indirectly refer to specified objects

Commands

DDL commands instruct the DDL compiler to perform specific actions. DDL commands consist of one or more keywords. Some commands also have one or more parameters to further control the action of the command. For syntax of individual DDL commands, see Section 9, DDL Compiler Commands.

Rules for DDL commands:

- A command or sequence of commands can be either part of a DDL schema or a parameter in a RUN DDL command.
- An input line that consists of a command or sequence of commands must begin with a question mark (?).
- More than one command can be specified on an input line or in a RUN DDL command. Multiple commands must be separated by commas. Only the first command on an input line is preceded by a question mark.
- A command input line cannot include comments or statements.
- If a command or sequence of commands continues on the next input line, the first character in the next line must be a question mark.
- A single command cannot end with a period or any other punctuation mark.

3 Running the DDL Compiler

You run the DDL compiler by using the RUN DDL command. You can run the DDL compiler either interactively, entering commands and source lines from the keyboard, or noninteractively, entering an entire schema from a file.

Running the DDL compiler interactively is recommended for functions that require only a few statements or commands, such as modifying an existing dictionary or generating source code from a dictionary.

Because errors are difficult to correct while you enter statements interactively, entering an entire schema interactively is not recommended. Instead, enter the schema in an EDIT file, where you can correct mistakes as you type. When the schema is correct, specify the name of the EDIT file either in a DDL SOURCE command or in the IN run option of the RUN DDL command.

When the DDL compiler stops its operation, it returns a completion code to the command interpreter that indicates the outcome of the DDL compilation. The completion code is accessible in the TACL variable _COMPLETION.

Topics:

RUN DDL Command

The RUN DDL command, an implied TACL RUN command, runs the DDL compiler.

```
[RUN] DDL [ / run-option [ , run-option ]... / ]
  [ compiler-command [ , compiler-command ]... ]
```

run-option

is any RUN option described in the *TACL Reference Manual*. The RUN options of most importance to the DDL compiler are:

IN *ddl-source-file*

specifies the name of a file that contains DDL statements and commands.

If you specify this option, see Running the DDL Compiler Noninteractively on page 3-3.

If you omit this option, see Running the DDL Compiler Interactively on page 3-4.

OUT [*listing-destination*]

determines whether the DDL compiler produces a listing, and if so, where.

If you omit this option, the DDL compiler sends the listing to its home terminal.

If you specify OUT but omit *listing-destination*, the DDL compiler does not produce a listing.

listing-destination

specifies the output device or disk file to which the DDL compiler writes its listing.

If *listing-destination* is a disk file name, but no disk file with that name exists, the DDL compiler creates a disk file with that name.

If *listing-destination* is the name of an existing file, the DDL compiler stops abnormally with a “file create” error.

NOWAIT

returns control immediately to the command interpreter. Without NOWAIT, the command interpreter suspends while the DDL compiler runs.

HIGHPIN { ON | OFF }

specifies the desired process identification number (PIN) range for the DDL compilation process.

ON

runs the DDL compiler at a high PIN if the HIGHPIN bit is on in the DDL object file and if the other conditions for running the new process at a high PIN are met.

OFF

runs the DDL compiler at a low PIN regardless of other considerations.

Without HIGHPIN, the PIN of the DDL compilation process depends on the HIGHPIN setting of the associated TACL process. If you access a D-series or G-series DDL compilation process from a terminal on a system running C-series software, the DDL compiler runs at a low PIN.

compiler-command

is any command described in Section 9, DDL Compiler Commands.

Run-time defaults for the DDL compiler:

- If you do not fully qualify a file name with volume and subvolume names, the DDL compiler qualifies the file name with the current default volume and subvolume names.
- The DDL compiler creates all files, including dictionary files and host-language source code files, with your default file-creation security. To change your default file-creation security, use the TACL DEFAULT command.

Running the DDL Compiler Noninteractively

To run the DDL compiler noninteractively, specify the run option IN *ddl-source-file* in the RUN DDL Command on page 3-1.

When run noninteractively, the DDL compiler:

- Accepts source input from *ddl-source-file*, which can contain any statements and commands described in this manual.
- Compiles the statements in *ddl-source-file* and writes a compiler listing to *listing-destination*, which you can specify in the RUN DDL command with the run option OUT.
- Performs the actions specified by compiler commands in both the RUN DDL command and *ddl-source-file*. The DDL compiler processes the commands in the RUN command first, then processes the commands in *ddl-source-file* as it encounters them.
- Stops the DDL compilation process after encountering either an end-of-file mark or an EXIT statement in *ddl-source-file*, and returns control to the command interpreter.

Example 3-1. Running the DDL Compiler Noninteractively

```
DDL /IN ddlsrc, OUT listfile/ DICT, COBOL cobsrc
```

Assuming that a dictionary exists on the current default volume and subvolume, the command in Example 3-1 on page 3-3 directs the DDL compiler to:

- Open the dictionary on the current default volume and subvolume.
- Open the COBOL source code file named `cobsrc`.
- Read statements and commands from the file `ddlsrc`.
- Compile the object-definition statements in `ddlsrc` in accordance with any commands in `ddlsrc` and add the compiled objects to the dictionary.
- Generate COBOL source code from the schema in `ddlsrc` and write the COBOL source code to `cobsrc`.
- Write the compiler listing to the file `listfile`.

Running the DDL Compiler Interactively

To run the DDL compiler interactively, use the RUN DDL Command on page 3-1 and either:

- Omit the run options IN and OUT.
- Specify the same interactive terminal for both of the run options IN and OUT.

When run interactively, the DDL compiler:

- Accepts all input from its home terminal
- Sends all output to its home terminal
- Executes any commands in the RUN DDL command before prompting you for input
- Prompts you for input with the exclamation point (!)

  You can enter any command or statement described in this manual. Begin a command with a question mark (?) and do not end it with a period (.).
- Exits interactive mode when you either enter the EXIT statement or press the Ctrl-Y key

In Example 3-2 on page 3-4, assume that the dictionary `dict` exists on the current default volume and subvolume.

Example 3-2. Interactive DDL Session: Adding Structures to Existing Dictionary

`2> DDL dict`	Open dictionary `dict`.
`!?COBOL cobsrc`	Open (or create) COBOL source code file `cobsrc` to receive COBOL source code.
`!?SOURCE newsrc`	Compile statements in schema file `newsrc`, add compiled objects to `dict`, and write generated COBOL source code to `cobsrc`.
`!EXIT`	Exit the DDL compiler, returning to the command interpreter.

Example 3-3. Interactive DDL Session: Adding Structures to New Dictionary

`27> DDL dict !`	If dictionary `dict` exists on the default subvolume, open `dict` for update access and delete all the dictionary objects it contains; otherwise, create `dict` on the current default volume and subvolume.

`!DEF cust-info.`
`!  02 name  PIC X(25).`
`!  02 addr  PIC X(40).`
	Parse the DEFINITION statement as it is entered, ending with END.

`!END`	Compile the DEFINITION statement and write production comments to the terminal.
`!EXIT`	Exit the DDL compiler, returning to the command interpreter.

Example 3-4. Interactive DDL Session: Writing From a Dictionary to a File

```
63> DDL
```

`!?DICT $data.sales`	Open a dictionary on the volume $DATA and the subvolume SALES.
`!?FUP fupsrc !`	Open the file `fupsrc`, clearing any contents.
`!OUTPUT RECORD customer.`	Retrieve the record `customer` from the dictionary and write the appropriate FUP file-creation commands for this record to `fupsrc`.
`!EXIT`	Exit the DDL compiler, returning to the command interpreter.

Completion Codes

When the DDL compiler stops its operation, it returns a completion code to the command interpreter that indicates the outcome of the DDL compilation.

Table 3-1. DDL Compiler Completion Codes

Code	Meaning
0	Normal termination. If warnings but no errors occurred, and the NOWARN command was in effect, any warnings that did occur were suppressed.
1	One or more warnings were reported, but no errors occurred.
2	One or more errors were reported (regardless of whether any warnings were reported).
3	The DDL compiler stopped before processing all input because the number of errors reached the limit specified in the command ERRORS on page 9-55.

The completion code is accessible in the TACL variable _COMPLETION.

4 Named Constants

A named constant is a dictionary object that has a name, a data type, and a value. You define named a constant in a CONSTANT statement, and you can refer to a named constant value by name in other DDL statements.

Topics:

CONSTANT

The CONSTANT statement defines a constant and adds it to the open dictionary. When the constant is in the dictionary, other DDL statements can use the constant as a literal value.

If a CONSTANT statement identifies a constant that is already in the dictionary and that is not referenced by any other object, the DDL compiler replaces the existing constant with the new constant. If the constant is referenced by another object, the DDL compiler issues an error message and does not add the constant to the dictionary

If a previous command opened a C, COBOL, Pascal (on D-series systems), pTAL, TACL, or TAL source code file, the DDL compiler translates any constant defined in a CONSTANT statement to the specified language and writes it to the open source code file.

```
CONSTANT constant-name { num-value-clause [ TYPE type ] }
                       { [ TYPE type ] num-value-clause }
                       { value-clause                   }.
```

constant-name

is the name of a constant.

num-value-clause

```
VALUE [ IS ] { { constant-number   } [ LN-clause ]... }
             { { national-literal  }                   }
             { { existing-constant }                   }
```

type

is the type of a numeric constant:

```
BINARY { [16] } [ UNSIGNED ]
       {  32  }
```

Default: BINARY 16

value-clause

```
VALUE [ IS ] { { { constant-number    } [ LN-clause ]... }
             { { { "string"           }                  }
             { { { national-literal   }                  }
             { { { existing-constant  }                  }
             {                                           }
             { VERSION "Lnn"                             }
```

constant-number

is a signed or unsigned decimal or octal integer (see Numbers on page 2-5) that is consistent with *type* (if *type* is specified).

"*string*"

is a string of from 1 to 130 ASCII characters (see Strings on page 2-5).

national-literal

is a national literal (see National Literals on page 2-6).

existing-constant

is the name of an existing constant whose value is consistent with the type of the constant being defined.

LN-clause

specifies the locale name for *value* (see LN on page 6-13).

"*Lnn*"

is a product version string.

L

is a letter. The DDL compiler treats *L* as uppercase whether you specify it as uppercase or lowercase.

nn

is a two-digit number.

Topics:

Numeric Constants

Each type of numeric constant has a different range of valid values, as Table 4-1 on page 4-3 shows.

Table 4-1. Ranges of Numeric Constant Values

Type	Lowest Value	Highest Value
BINARY 16	-32,768	32,767
BINARY 16 UNSIGNED	0	65,535
BINARY 32	-2,147,483,648	2,147,483,647
BINARY 32 UNSIGNED	0	4,294,967,295
BINARY 64	-9,223,372,036,854,775,808	9,223,372,036,854,775,807

The type of a numeric constant:

- Ensures that the specified value is consistent with the type, whether the type is explicitly specified or is BINARY 16 by default. For example, the value 40,000 is not consistent with type BINARY 16 because the value is too large, but it is consistent with type BINARY 16 UNSIGNED. If a value is not consistent with its type, the CONSTANT statement fails.
- Controls the types of COBOL, pTAL, and TAL data items generated for the constant. For example, type BINARY 32 causes the DDL compiler to generate a pTAL or TAL LITERAL with a double-word value and a COBOL NATIVE-4 two-word value. The DDL compiler translates binary numbers as TACL TEXT values rather than as binary data in a STRUCT so, the TYPE clause does not affect TACL output from DDL constants.

Product Version Constants

When specifying product version constants:

- You can use a product version constant only in the VERSION clause of a TOKEN-MAP statement, in the VALUE clause of a DEFINITION statement, or in the VALUE clause of a CONSTANT statement.
  - When a VALUE clause in a DEFINITION or CONSTANT statement includes a product version constant, the DDL compiler treats the product version constant as a BINARY 16 integer type.
  - When a VERSION clause in a TOKEN-MAP statement includes a product version number, the SPI product version compatibility mechanism uses the product version number to identify the structure of a particular product version of an extensible structured token.
- When generating output from a product version constant for host-language source code, the DDL compiler converts the product version string from the form *ann* to the numeric representation of a product version number returned by the TOSVERSION Guardian procedure.

You can compare product version numbers without decoding them.

To “decode” a product version number:

1. Obtain the letter of the product version by dividing the product version number by 256. The quotient is the ASCII decimal representation of the uppercase letter. Any remainder is ignored.
2. Obtain the number of the product version by multiplying the quotient from Step 1 by 256 and subtracting the answer from the product version number.

For example, for product version number 17162:

1. 17162 divided by 256 = 67.04
2. 67 is the ASCII decimal representation for the letter *C*
3. 67 multiplied by 256 = 17152, and 17162 - 17152 = 10
4. C10 is the product version

Existing Constants

When using the name of an existing constant as the value in a CONSTANT statement:

- You can specify a DDL constant name instead of a literal value in the VALUE clause of a CONSTANT statement whether the constant name identifies a string constant, a numeric constant, or a product version constant.
- When the name identifies a previously defined string constant, the new value is identical to the value of the string constant.
- When the name identifies a previously defined product version constant, the new value is identical to the value of the product version constant.
- When an existing numeric constant is named in the definition of another DDL numeric constant, certain rules apply:
  - If the CONSTANT statement does not include a TYPE clause, the constant being defined inherits the type of *existing-constant*.
  - If the CONSTANT statement has a TYPE clause, its specified type overrides the type of *existing-constant*.
  - If a TYPE clause in the CONSTANT statement overrides the type of *existing-constant*, the value of *existing-constant* must be consistent with the specified type.

C

When generating C source code from CONSTANT statements:

- If you request C source-code output, by giving the C command, the DDL compiler generates `#define`s for named constants.
- The DDL compiler converts any hyphen in the constant name to an underscore (_) in the `#define` name.
- The DDL compiler generates uppercase letters for names that follow #define.
- For a string constant, the DDL compiler generates a `#define` of this form:

  ```
  #define CONSTANT-NAME string-literal
  ```

- For a numeric constant, the DDL compiler generates a `#define` of this form:

  ```
  #define CONSTANT-NAME numeric-constant
  ```

- For a product version constant, the DDL compiler generates a `#define` that contains the product version number.

COBOL

When generating COBOL source code from CONSTANT statements:

- If you request COBOL source-code output, by giving the COBOL command, the DDL compiler generates a level-01 data description entry for each named constant.
- For a string constant, the DDL compiler generates a string value identical to the specified constant value.
- For a numeric constant, the DDL compiler generates a COBOL data type based on the type of the numeric constant:

Constant Type	COBOL Data Type
BINARY 16	NATIVE-2
BINARY 32	NATIVE-4
BINARY 64	NATIVE-8
BINARY 16 UNSIGNED	NATIVE-2
BINARY 32 UNSIGNED	NATIVE-4

  Unsigned binary constants are translated to COBOL signed data types.

- For a product version constant, the DDL compiler generates a COBOL NATIVE-2 elementary item that contains the product version number.

Pascal (D-series Systems Only)

When generating Pascal source code from CONSTANT statements:

- If you request Pascal source-code output, by giving the PASCAL command, the DDL compiler generates Pascal constants.
- The DDL compiler converts any hyphen in the constant name to an underscore (_) in the Pascal constant name.
- Pascal does not support the TYPE clause in the CONSTANT statement.
- For a string constant, the DDL compiler generates a Pascal FSTRING constant.
- For a numeric constant, the DDL compiler generates a Pascal numeric constant.
- For a product version constant, the DDL compiler generates the product version number.

TACL

When generating TACL source code from CONSTANT statements:

- If you request TACL source-code output, by giving the TACL command, the DDL compiler generates TACL TEXT variables for named constants.
- The DDL compiler converts any hyphen in the constant name to a circumflex (^) in the TACL TEXT variable name.
- For a string constant, the DDL compiler generates a TACL TEXT variable with a value derived from the DDL constant value.
  - The value of the TACL TEXT variable differs from the DDL constant if certain special characters are specified in the constant. The DDL compiler precedes these special characters with a tilde (~) in the variable:

    ```
    [ ] { } | ==
    ```

    For example, the value in this CONSTANT statement includes special characters:

    ```
    CONSTANT tacl-out VALUE IS "#OUTPUT [#NEXTFILENAME]".
    ```

    The resulting TACL source code is:

    ```
    ?Section TACL^OUT TEXT
    #OUTPUT ~[#NEXTFILENAME~]
    ```
  - The total number of bytes generated for a TACL string constant cannot exceed 130, including any generated tildes. If the value would be longer than 130 bytes, the DDL compiler does not generate the TACL constant.
- For a numeric constant, the DDL compiler generates a TACL TEXT variable with a value identical to the value of the DDL constant.
- For a product version constant, the DDL compiler generates a TACL TEXT variable that contains the product version number.
- The internal representation of a DDL constant in TACL differs from these representations:
  - The internal representation of DDL constants in pTAL, TAL, and COBOL.
  - The internal representation of all other DDL objects in TACL. For a definition, record, token code, token map, or token type, the DDL compiler generates a TACL STRUCT with the same internal representation as pTAL or TAL source code.

  This difference does not cause problems in messages because messages contain data structures, not constants. If you use #APPENDV to move a TACL representation of a DDL constant to a message for a program coded in another language, the value in the message will not match the same DDL constant in the other language.

TAL

When generating pTAL or TAL source code from CONSTANT statements:

- If you request pTAL or TAL source-code output by giving the TAL command, the DDL compiler generates pTAL or TAL source code for named constants.
- The DDL compiler converts any hyphen in the constant name to a circumflex (^) in the TAL DEFINE name.
- For a string constant, the DDL compiler generates a TAL DEFINE. Each DEFINE specifies a value that exactly matches the constant value. TAL limits the length of a string constant to 128 bytes, although the DDL compiler accepts string constants of up to 130 ASCII characters.
- For a numeric constant, the DDL compiler generates a pTAL or TAL literal based on the type of the numeric constant. A numeric constant with a value *n* results in a different pTAL or TAL literal for each DDL constant type:

Constant Type	pTAL or TAL Data Type
BINARY 16	*n*
BINARY 32	*n* D
BINARY 64	*n* F

  If the type is specified as unsigned, the DDL compiler generates the pTAL or TAL literal in octal representation.
- For a product version constant, the DDL compiler generates a pTAL or TAL literal that has a type equivalent to BINARY 16 UNSIGNED and that contains the product version number.

Examples

Example 4-1. CONSTANT Statements

```
CONSTANT prog-name              VALUE IS "MYPROG".
CONSTANT myprog                 VALUE IS prog-name.
CONSTANT zspi-val-tandem-owner VALUE IS "TANDEM  ".

CONSTANT hundred                VALUE IS 100.
CONSTANT double-hundred         VALUE 100    TYPE BINARY 32.
CONSTANT quad-num               VALUE 800000 TYPE BINARY 64.
CONSTANT zspi-val-msghdrsize    VALUE 6      TYPE BINARY 16
UNSIGNED.
```

In Example 4-2 on page 4-9, constant B inherits the type of constant A; that is, constant B is also type BINARY 16 UNSIGNED.

Example 4-2. Numeric Constant Defined by Existing Constant—Same Type

```
CONSTANT a VALUE 200 TYPE BINARY 16 UNSIGNED.
CONSTANT a VALUE b.  ! Type binary 16 unsigned
```

If the second CONSTANT statement includes a TYPE clause, the clause overrides the defining constant.

In Example 4-3 on page 4-9, the type specified for constant C overrides the type specified for constant A.

Example 4-3. Numeric Constant Defined by Existing Constant—New Type

```
CONSTANT a VALUE 200 TYPE BINARY 16 UNSIGNED.
CONSTANT c VALUE a TYPE BINARY 16.  ! Type binary 16
```

When the data types are not the same, the constant value must be compatible with each specified data type. In Example 4-4 on page 4-9, both statements are valid; the value 1000 is compatible with both type BINARY 16 and type BINARY 32.

Example 4-4. Numeric Constants With Compatible Types

```
CONSTANT thousand   VALUE 1000      TYPE BINARY 32.
CONSTANT max-value  VALUE thousand  TYPE BINARY 16.
```

In Example 4-5 on page 4-9, the value of HI-VAL is too large for type BINARY 16. The DDL compiler issues an error message and does not execute the CONSTANT statement.

Example 4-5. Numeric Constants With Incompatible Types

```
CONSTANT fifty-thou  VALUE 50000       TYPE BINARY 16 UNSIGNED.
CONSTANT hi-val      VALUE fifty-thou  TYPE BINARY 16.
*** ERROR *** Invalid value for value type
*** WARNING *** Errors detected - no output produced for HI-VAL
```

For examples using locale names, see Appendix B, Sample Schemas.

Standard SPI Constants

Subsystems that use DSM are provided with a set of CONSTANT statements to define standard values for use in SPI messages. For the names and descriptions of standard SPI constants, see the *SPI Programming Manual* and the *SPI Common Extensions Manual*.

5 Definitions and Records

Definitions and records are dictionary objects that describe data structures and disk-file record structures, respectively.

DEFINITION and RECORD statements:

- Define definitions and records, respectively
- Must conform to the syntax rules in Statements on page 2-16
- Have many syntax elements in common
- Are usually specified in a schema file that is used as the IN file when the DDL compiler is executed noninteractively

Topics:

DEFINITION

The DEFINITION statement defines an elementary or group data structure by specifying its name, data type, size, and other attributes.

The definition can be added to the open dictionary; referenced for defining other data structures, record structures, token types, or token maps; and compiled into a DDL or host-language source code file.

If a DEFINITION statement names a definition that is already in the open dictionary and no other object refers to the definition, the DDL compiler replaces the existing definition with the new definition. If another object refers to the existing definition, the DDL compiler issues an error message and does not add the new definition to the dictionary.

If a previous command opened any language source code files, the DDL compiler translates the definition to the specified language and writes it to the open source code files.

Note. For the DDL2 object file, if a definition is used only for the working storage, the length of the definition or any of its fields is limited to 2,097,152 bytes. This length is limited to 32,767 bytes in the DDL object file.

△ **Caution.** The DDL and DDL2 object files are not compatible with each other. The DDL2 object file cannot read or write to the dictionary created using the DDL object file. Similarly, the DDL object file cannot read or write to the dictionary created using the DDL2 object file. Therefore, you must generate the schema from the existing dictionary before deleting the dictionary.

The DEFINITION statement has three forms:

Form	Description
Field Definition on page 5-4	Defines a single field
Group Definition on page 5-5	Defines a group of fields or a group of groups
Reference Definition on page 5-7	Defines a field or group by referring to a previous definition

Topics:

Order of Clauses

The clauses in a DEFINITION statement can be in any order, with these exceptions:

- Any level-88 condition-name clauses and level-89 enumeration clauses must follow the first period in a field definition or description. A single-field definition that has one or more of these clauses must also have BEGIN before the first period and END after the last clause.
- The level-66 RENAMES clause must immediately precede END in a group definition.
- All clauses except level-88, level-89, and level-66 clauses must precede the first period in a definition or description.
- END must follow all clauses in a definition. A single-field definition that includes BEGIN must also include END; other single-field definitions cannot include END. All group definitions must include END.

Definition Length

A definition’s length must conform to these rules:

- If a definition is used only for working storage, its length or the length of any field within the definition is limited to 32,767 bytes.
- If a RECORD statement refers to a definition, the length of the definition is subject to the maximum record-length limitations.

- These languages further definition length:

Language	Maximum Definition Length (Bytes)
FORTRAN	255
TACL	5,000
Pascal (on D-series systems)	32,766
COBOL	4,096

Field Definition

This DEFINITION statement defines a single field.

```
DEF[INITION] def-name
    { PICTURE-clause | TYPE-clause }
    [ AS-clause ]
    [ BEGIN ]
    [ DISPLAY-clause ]
    [ EDIT-PIC-clause ]
    [ EXTERNAL-clause ]
    [ HEADING-clause ]
    [ HELP-clause ]
    [ JUSTIFIED-clause ]
    [ MUST-BE-clause ]
    [ NULL-clause ]
    [ SPI-NULL-clause ]
    [ SQLNULLABLE-clause ]
    [ TACL-clause ]
    [ UPSHIFT-clause ]
    [ USAGE-clause ]
    [ VALUE-clause ] .
    [ 88-condition-name-clause . ] ...
    [ 89-enumeration-clause . ] ...
    [ END [ . ] ]
```

For descriptions of clauses, see Syntax Elements on page 5-21.

Example 5-1. Field Definitions

```
DEF company-name  TYPE CHARACTER 30   NULL 0 .

DEF custnum       PIC 9(6)  HEADING "Customer/Number" .

DEF status TYPE ENUM BEGIN.
   89 no-error.
   89 read-error VALUE 3.
   89 write-error.
END.
```

Group Definition

This DEFINITION statement defines a group of fields or a group of groups.

```
DEF[INITION] def-name

   [ DISPLAY-clause ]

   [ EXTERNAL-clause ]

   [ HEADING-clause ]

   [ HELP-clause ]

   [ NULL-clause ]

   [ SQLNULLABLE-clause ]

   [ USAGE-clause ]

   [ VALUE-clause ] .

   line-item specification ...

   [ 66-RENAMES-clause . ] ...

END [ . ]
```

For descriptions of clauses, see Syntax Elements on page 5-21.

Each field or group within a group DEFINITION statement must be defined by at least a level number and a name. The level number must precede the group or field name. Other clauses can follow in any order.

A group DEFINITION statement can contain nested group descriptions, which must contain at least one field description.

Every field within a group DEFINITION statement must be described with a PICTURE or TYPE clause; a group description cannot have either clause.

The TYPE clause for a field within a group DEFINITION statement can refer to a field or group definition previously stored in the open dictionary. When a field is defined by referring to a group definition, the field effectively becomes a group.

A group's size is the total of the lengths of its member fields plus any FILLER fields generated by the DDL compiler.

Example 5-2. Group Definitions

```
DEF address.
    03 street-address.
       05 street-no  PIC X(8).
       05 street     PIC X(12).
       05 apt-no     PIC X(4).
    03 city          PIC X(14).
    03 state-cd      PIC X(2).
    03 zip           PIC X(5).
END.

DEF phone.           DISPLAY "n<(999) 999-9999>"
     03 area-cd      PIC 9(3).
     03 prefix       PIC 9(3).
     03 numb         PIC 9(4).
END.  ! Period is optional

DEF cust-info.
     03 company-name  TYPE *
                      HEADING "Company".
     03 cust-address  TYPE address.
                      HEADING "Address".
     03 cust-phone    TYPE phone.
                      HEADING "Phone".
END.

DEF customer.
    03 cust-name      TYPE cust-name.
    03 cust-id        PIC 9(6).  ! Level number must be < 04
END.
```

Reference Definition

This DEFINITION statement copies an existing definition, giving it a new name. The new definition can be given its own attributes, which can override all copied attributes except data type and size.

```
DEF[INITION] def-name-1 TYPE def-name-2

    [ AS-clause ]

    [ BEGIN ]

    [ DISPLAY-clause ]

    [ EDIT-PIC-clause ]

    [ EXTERNAL-clause ]

    [ HEADING-clause ]

    [ HELP-clause ]

    [ MUST-BE-clause ]

    [ NULL-clause ]

    [ SPI-NULL-clause ]

    [ TACL-clause ]

    [ UPSHIFT-clause ]

    [ USAGE-clause ]

    [ VALUE-clause ] .

    [ 88-condition-name-clause . ] ...

    [ END [ . ] ]
```

For descriptions of clauses, see Syntax Elements on page 5-21.

Example 5-3. Reference Definitions

```
DEF cust-name           TYPE company-name
                        HEADING "Customer" .

DEF home-phone          TYPE phone
                        HEADING "Employee/Home Phone".
```

Error Handling

When the DDL compiler encounters an error in a DEFINITION statement, it continues processing the statement to determine if there are other errors before processing the next statement. The DDL compiler does not add the definition to the dictionary, and if any source code files are open, the DDL compiler does not write the definition to those files.

An extra period in a group definition might cause the DDL compiler to not report any additional errors until it encounters END.

RECORD

The RECORD statement defines a disk file record, specifying the record's file name and type. If the file is structured, the RECORD statement also identifies the key fields and assigns a key specifier to any alternate keys.

If a dictionary is open, the DDL compiler stores the record in the dictionary. If a record of the same name already exists, the DDL compiler replaces the existing record with the new record.

Depending on which source code files are open, the DDL compiler writes the record to a DDL source code file, writes source code to describe the record to a host-language source code file, and writes the file creation commands to a FUP source code file.

```
RECORD record-name .

  [ file-creation ]

  { record-structure | record-reference }

  [ key-assignment ]

END [ . ]
```

record-name

is the name of the record to be added to, or replaced in, the open dictionary.

file-creation

specifies either the name or the type of the disk file that will store occurrences of the record (see File-Creation Syntax on page 5-10).

record-structure

specifies the data structure of the record and (optionally) identifies primary and alternate keys (see Record Structure Syntax on page 5-15).

record-reference

specifies the data structure of the record in terms of another, existing record and (optionally) identifies primary and alternate keys (see Record Reference Syntax on page 5-16).

key-assignment

specifies one or more fields or groups of fields as Enscribe keys, assigns key specifiers to key fields, and specifies that a file is to be sorted on a nonkey field or group of fields (see Key Assignment Syntax on page 5-17).

You can omit *key-assignment* if the record has no key fields or if you declare its key fields with the clause KEYTAG on page 6-12.

Note. The DDL compiler ignores *key-assignment* when generating TACL source code from a RECORD statement.

END [.]

ends the RECORD statement.

Topics:

File-Creation Syntax

In the statement RECORD on page 5-8, *file-creation* specifies either the name or the type of the disk file that will store occurrences of the record.

```
FILE IS { ["]file-name["] } [ creation-attribute ] ...
        { TEMPORARY       }
        { ASSIGNED        }
```

file-name

is the name of a disk file that is to contain occurrences of the record defined in the RECORD statement.

file-name can appear in more than one RECORD statement in the same dictionary. To avoid file name conflicts:

- Select one record structure to generate FUP file-creation commands.
- Define other record structures as TEMPORARY or ASSIGNED.

TEMPORARY

specifies that the disk file that will store occurrences of the record is a temporary file (created programmatically and purged when closed).

Note. FUP output is not generated for temporary files.

ASSIGNED

specifies that the record is a logical record with the same structure as one or more physical records.

You can include the logical record definition in a program and assign the logical record to a physical file with a TACL ASSIGN command before you run the program.

Note. FUP output is not generated for assigned files.

creation-attribute

is an attribute of the disk file that will store occurrences of the record (see Creation-Attribute Syntax on page 5-12).

Note. The DDL compiler ignores *creation-attribute* when generating TACL source code from a RECORD statement.

If you omit *file-creation* from the RECORD statement:

- The DDL compiler derives *file-name* from *record-name*: If *record-name* has a hyphen (-) within its first 8 characters, *file-name* is all of the characters up to the first hyphen; otherwise, *file-name* is the first 8 characters of *record-name*. Volume and subvolume names are undefined.
- The DDL compiler assigns a file type:

If the record has ...	File type is ...
A primary key	Key-sequenced
No keys (an unstructured file can have a SEQUENCE IS clause)	Unstructured
One or more alternate keys and a SEQUENCE IS clause but no primary key	Entry-Sequenced
One or more alternate keys but no primary key or SEQUENCE IS clause	Relative

If you do not specify a file type in FUP, FUP automatically creates the file as unstructured. DDL and FUP also have different default file attributes (see Table 5-1 on page 5-11).

Table 5-1. File Attributes for DDL and FUP

		Default Value	
File Attribute	**File Type**	**DDL**	**FUP**
BLOCK	Key-sequenced Relative Entry-sequenced	4096 bytes	4096 bytes
EXT	All	Primary: 4 pages Secondary: 32 pages	Determined by file type and block or buffer size (see the *File Utility Program (FUP) Reference Manual*)
MAXEXTENTS	All	100	16
NO ODDUNSTR	Unstructured	Odd	Even

Creation-Attribute Syntax

In the File-Creation Syntax on page 5-10, *creation-attribute* is an attribute of the disk file that will store occurrences of the record defined by the statement RECORD on page 5-8.

Note. The DDL compiler ignores *creation-attribute* when generating TACL source code from a RECORD statement.

```
{ KEY-SEQUENCED | RELATIVE | ENTRY-SEQUENCED | UNSTRUCTURED }

[ AUDIT ]

[ AUDITCOMPRESS]

[ BLOCK block-length ]

[ [NO]BUFFERED ]

[ BUFFERSIZE buffer-size ]

[ CODE file-code ]

{ COMPRESS | DCOMPRESS | ICOMPRESS }

[      { extent-size                                  } ]
[ EXT  {                                              } ]
[      { ( pri-extent-size [, sec-extent-size ] )     } ]

[ MAXEXTENTS maximum-extents ]

[ NO ODDUNSTR ]

[ REFRESS ]

[ SERIALWRITES ]

[ VERIFYWRITES ]
```

KEY-SEQUENCED
RELATIVE
ENTRY-SEQUENCED
UNSTRUCTURED

are Enscribe file types. The first three specify structured files that can have keys. For more information, see the *Enscribe Programmer's Guide*.

AUDIT

specifies AUDIT when generating FUP source code. AUDIT designates the file as audited by TMF. For more information, see the *TMF Management Programming Manual*.

AUDITCOMPRESS

compresses the file's audit trail. For more information, see the *TMF Management Programming Manual*.

BLOCK *block-length*

specifies the block size, in bytes, for both data and index blocks in a structured file. You can specify *block-length* either as an integer or as the name of a constant in the open dictionary. The value of *block-length* must be one of:

- 512
- 1,024
- 2,048
- 4,096

Default: 4,096 bytes

BUFFERED

writes to your file are buffered in the disk-process cache. BUFFERED is the default for audited files.

NOBUFFERED

writes to your file written to the disk. NOBUFFERED is the default for nonaudited files.

BUFFERSIZE *buffer-size*

specifies the buffer size, in bytes, for an unstructured file. You can specify *buffer-size* either as an integer or as the name of a constant in the open dictionary. The value of *buffer-size* must be one of:

- 512
- 1,024
- 2,048
- 4,096

Default: 4,096 bytes

CODE *file-code*

assigns a file code to a file. You can specify *file-code* either as an integer or as the name of a constant in the open dictionary. The value of *file-code* must be in either of these ranges:

- 0 through 99
- 1,000 through 65,535

File codes 100 through 999 are reserved for use by HP.

Default: Zero

{ COMPRESS | DCOMPRESS | ICOMPRESS }

are only for key-sequenced files.

COMPRESS

turns on both index compression and data compression.

DCOMPRESS

turns on data compression.

ICOMPRESS

turns on index compression.

```
    { extent-size                                  }
EXT {                                              }
    { ( pri-extent-size [, sec-extent-size ] )     }
```

sets the extent size in pages.

extent-size

specifies the total extent size. You can specify *extent-size* either as an integer or as the name of a constant in the open dictionary. The value *extent-size* must be an integer from 1 through 65,535.

For structured files, *extent-size* must be a multiple of the file's BLOCK value.

For unstructured files, *extent-size* must be a multiple of the file's BUFFERSIZE value.

pri-extent-size

specifies the primary extent size. You can specify *pri-extent-size* either as an integer or as the name of a constant in the open dictionary. The value *pri-extent-size* must be an integer from 1 through 65,535.

Default: 4 pages

sec-extent-size

specifies the secondary extent size. You can specify *sec-extent-size* either as an integer or as the name of a constant in the open dictionary. The value *sec-extent-size* must be an integer from 1 through 65,535.

Default: 32 pages

MAXEXTENTS *maximum-extents*

sets the maximum number of extents the file can have. You can specify *maximum-extents* either as an integer or as the name of a constant in the open dictionary. The value of *maximum-extents* must be an integer from 1 to *n*, where *n* is determined by the available free space in the file label.

Default: 100

NO ODDUNSTR

specifies that all unstructured files be processed as even-unstructured files.

Enscribe unstructured files can be even-unstructured or odd-unstructured. In even-unstructured files, an odd byte count given for reading, writing, or positioning is rounded upward. Odd-unstructured files have no upward rounding; you always read, write, or position at the byte count you give.

REFRESH

forces the operating system to copy the file's label to disk whenever the file's file control block is updated.

SERIALWRITES

specifies that writes to the mirror disk be done serially.

Default: Mirror disk writes are done in parallel.

VERIFIEDWRITES

specifies that disk writes be verified by the disk process. Disk writes are verified by comparing the data on the disk with the data in memory.

Default: Disk writes are not verified.

Record Structure Syntax

In the statement RECORD on page 5-8, *record-structure* specifies the data structure of the record and (optionally) identifies primary and alternate keys.

```
line-item specification ... [ 66-RENAMES-clause . ] ...
```

For descriptions of *line-item specification* and *66-RENAMES-clause*, see Syntax Elements on page 5-21.

A record structure must contain at least one field description. Every field description must have a PICTURE or TYPE clause.

A record structure can contain one or more group descriptions. A group description cannot have a PICTURE clause.

A TYPE clause for a field within a record structure can refer to a field or group definition previously stored in a dictionary. When a field is defined by referring to a group definition, it effectively becomes a group.

The size of a record structure is the total of the lengths of its member fields, plus any FILLER fields generated by the DDL compiler.

Maximum record length:

- Depends on file type:

	Maximum Length	
File Type	**Format 1**	**Format 2**
Unstructured	4,096 bytes	4,096 bytes
Entry-sequenced	4,072 bytes	4,048 bytes
Relative	4,072 bytes	4,048 bytes
Key-sequenced	4,062 bytes	4,040 bytes

- Is limited further by these languages:

Language	Maximum Record Length (Bytes)
FORTRAN	255
TACL	5,000
Pascal (on D-series systems)	32,766

Record Reference Syntax

In the statement RECORD on page 5-8, *record-reference* specifies the data structure of the record in terms of another, existing record and (optionally) identifies primary and alternate keys.

```
DEF[INITION] IS def-name
```

def-name

is the name of an existing definition in the open dictionary.

When you use record reference syntax, you must declare any key fields with a key assignment at the end of the RECORD statement. You cannot use a KEYTAG clause to declare key fields with a reference record structure.

Key Assignment Syntax

In the statement RECORD on page 5-8, *key-assignment* specifies specifies one or more fields or groups of fields as Enscribe keys, assigns key specifiers to key fields, and specifies that a file is to be sorted on a nonkey field or group of fields.

Note. The DDL compiler ignores *key-assignment* when generating TACL source code from a RECORD statement.

```
KEY key-specifier IS { group-name | field-name }

   [ FILE IS ["]file-name["] ]

   [ DUPLICATES [ NOT ] ALLOWED ] . ] ...

   [ UPDATE [ NOT ] ALLOWED ]

   [ SEQUENCE IS [ ASCENDING  ] { group-name }  ]
   [             [ DESCENDING ] { field-name } .]
```

KEY *key-specifier*

specifies a field or group of fields as an Enscribe key and assigns a key specifier to the key. You can specify *key-specifier* either as an integer from -32,768 through 32,767; as two ASCII characters enclosed in quotation marks; or as the name of a constant in the open dictionary. The value of the constant must be either an integer from -32,768 through 32,767 or a string of two ASCII characters.

You can omit *key-specifier* for a primary key, but if you include it, its value must be 0. A nonzero value for *key-specifier* indicates an alternate key.

{ *group-name* | *field-name* }

is the name of a group or field used as either a primary key, an alternate key, or a sequence field. If the name is not unique in the dictionary, the name must be qualified to make it unique.

file-name

is the file name of the alternate key file for the specified key.

Default: Primary file name with a number appended

DUPLICATES [NOT] ALLOWED

specifies whether to allow duplicate alternate key values. Do not specify DUPLICATES ALLOWED for a primary key field.

Default: DUPLICATES ALLOWED

`UPDATE [ NOT ] ALLOWED`

specifies whether to allow updates for an alternate key file. This clause affects FUP output generated for the alternate key.

Default: UPDATE ALLOWED

`SEQUENCE IS [ ASCENDING | DESCENDING ] {`*`group-name`* `|`*`field-name`* `}`

specifies that the file is to be sorted on a nonkey field or group by the application program. Only one field or group in a record can be used for this purpose.

`[ ASCENDING | DESCENDING ]`

specifies the sort order.

Default: ASCENDING

`{` *`group-name`* `|` *`field-name`* `}`

is the name of the sort field. If the name is not unique in the dictionary, the name must be qualified to make it unique.

Only key-sequenced records can have a *`key-specifier`* with a value of 0, indicating a primary key. Key-sequenced records must have one and only one primary key.

A key defined with a nonzero key specifier, such as “NM” or 32000, is an alternate key.

Unstructured file records cannot have alternate keys.

For COBOL, keys must be alphanumeric; that is, the PICTURE for a key must be either all *X*s, all *9*s (without a sign), or TYPE CHARACTER.

Key fields can overlap.

Error Handling

When the DDL compiler encounters an error in a RECORD statement, it continues processing the statement to determine if there are other errors before processing the next statement. The DDL compiler does not add the record to the dictionary, does not write any FUP source for it, and does not write the record to any open language source code files.

Examples

The RECORD statements in Example 5-5 on page 5-19 through Example 5-8 on page 5-20 refer to the definitions in Example 5-4 on page 5-19.

Example 5-4. Definitions Referenced in RECORD Statements

```
CONSTANT phone-heading  VALUE IS "Phone Number".
CONSTANT phone-display  VALUE IS "M<(999) 999-9999>".

DEF phone             HEADING phone-heading
                      DISPLAY phone-display.
  02 area-code        PIC 9(3).
  02 prefix           PIC 9(3).
  02 numb             PIC 9(4).
END

DEF addr.
  02  address        PIC X(22).
  02  city           PIC X(14).
  02  state          PIC X(12).
END

DEF custinfo.
  02  custnum        PIC 9(4).
  02  custname       PIC X(18).
  02  addr           TYPE *.
END
```

Example 5-5. Record Defined by Existing Definition

```
RECORD cust.
   FILE IS "$data.sales.customer".   ! File name
   KEY-SEQUENCED.                    ! File type
   DEF IS custinfo.                  ! Record structure
   KEY IS cust.custnum.              ! Primary key
   KEY "nm" IS cust.custname.        ! Alternate key
END.
```

Example 5-6. Record With Unique Alternate Key

```
RECORD supplier-info.
  FILE IS "$data.sales.supplier" KEY-SEQUENCED .

  02 suppnum   PIC 9(4).
  02 suppname  PIC X(18).
  02 addr      TYPE *.

  KEY IS suppnum.
  KEY "sn" IS suppname DUPLICATES NOT ALLOWED .
END
```

Example 5-7. Qualifying Alternate Key Fields Whose Names Are the Same

```
RECORD phones .
  FILE IS "\dallas.$data.sales.person"
  KEY-SEQUENCED.

    02 social-security    PIC 9(9).
    02 home-phone         TYPE phone.
    02 work-phone         TYPE phone.

  KEY IS social-security.
  KEY "hc" IS home-phone.area-code.
  KEY "wc" IS work-phone.area-code.
END
```

Example 5-8. Creating an Alternate Key File

DDL Source Code:

```
RECORD test-1.
  FILE IS "Test1".

  02 f-1 TYPE BINARY.
  02 f-2 PIC X(10).
  02 f-3 TYPE COMPLEX.

  KEY "KY" IS f-2 FILE IS "AltKy" UPDATE NOT ALLOWED.
END.
```

FUP Output:

```
RESET
   SET ALTKEY ("KY", KEYOFF 2, KEYLEN 10, FILE 0, NO UPDATE)
   SET NO ALTCREATE
   SET ALTFILE (0, AltKy)
   SET TYPE R
   SET REC 20
   SET BLOCK 4096
   SET EXT(4, 32)
   SET MAXEXTENTS 100
CREATE Test1
   RESET
   SET TYPE K
   SET KEYLEN 12
   SET REC 16
   SET BLOCK 4096
   SET IBLOCK 4096
   SET EXT(4, 32)
   SET MAXEXTENTS 100
CREATE AltKy
```

Syntax Elements

These syntax elements appear in one or more forms of DEFINITION and RECORD statements:

- Clauses on page 5-21
- Other Elements on page 5-23

Clauses

This topic lists clauses with level numbers first, in numerical order, followed by other clauses in alphabetical order.

66-RENAMES-clause

renames a previously defined field or group or set of fields or groups (see 66 RENAMES on page 6-79).

Note. The DDL compiler ignores this clause when generating source code for languages other than DDL and COBOL

88-condition-name-clause

associates a condition name with a value, list of values, or range of values, enabling you to refer to the value or values by the condition name (see 88 Condition-Name on page 6-81).

Note. The DDL compiler ignores this clause when generating source code for languages other than DDL and COBOL

89-enumeration-clause

associates a name and (optionally) a display string with an enumeration value (see 89 Enumeration on page 6-84).

A single-field definition that has one or more level-89 clauses must also have BEGIN before the first period and END after the last clause.

AS-clause

specifies a default display string for a field of type ENUM (see AS on page 6-3).

DISPLAY-clause

specifies the default format for field or group values listed in an Enform Plus report (see DISPLAY on page 6-4).

EDIT-PIC-clause

specifies the format in which Pathmaker-generated requesters display a field's data on a screen (see EDIT-PIC on page 6-5).

EXTERNAL-clause

writes the EXTERNAL clause to COBOL source code files (see EXTERNAL on page 6-6).

HEADING-clause

specifies a default field heading for values listed on Enform Plus reports or displayed on screens generated by ENABLE and Pathmaker (see HEADING on page 6-9).

HELP-clause

assigns help text, used by Pathmaker-generated requesters, to a group or elementary item (see HELP on page 6-10).

JUSTIFIED-clause

writes the JUSTIFIED RIGHT clause to COBOL source code files (see JUSTIFIED on page 6-11).

LN-clause

specifies the language, territory, and character set for a text item (see LN on page 6-13).

MUST-BE-clause

specifies the set of valid values that can be entered in a field (see MUST BE on page 6-15).

NULL-clause

assigns a null value to a field or group used as an alternate key (see NULL on page 6-19).

OCCURS-clause

repeats a field or group a fixed number of times (see OCCURS on page 6-20).

OCCURS-DEPENDING-ON-clause

repeats a field or group a variable number of times, depending on the current value of an integer variable (see OCCURS DEPENDING ON on page 6-23).

PICTURE-clause

specifies the data type and size of a field (see PICTURE on page 6-25).

REDEFINES-clause

assigns a new name and, optionally, a new structure to previously defined definition or record (see REDEFINES on page 6-31).

SPI-NULL-clause

specifies an SPI null value for a field or group in an SPI-extensible structured token or for a field or group within a group definition (see SPI-NULL on page 6-37).

SQLNULLABLE-clause

specifies whether a line item is to be treated as an SQL-nullable column (see SQLNULLABLE on page 6-39).

TACL-clause

specifies the TACL data type to which a DDL data item is to be converted when generating TACL source code (see TACL on page 6-44).

TYPE-clause

specifies the data type and size of a field (see PICTURE on page 6-25 and TYPE on page 6-48).

UPSHIFT-clause

upshifts ASCII characters entered in the field (see UPSHIFT on page 6-69).

USAGE-clause

either specifies computational storage allocation for a numeric group or field or identifies a COBOL as an index (see USAGE on page 6-70).

VALUE-clause

assigns or suppresses a DDL or COBOL field or group's initial value (see VALUE on page 6-75).

Other Elements

This topic lists elements in alphabetical order.

BEGIN

precedes any level-88 condition-name clauses and level-89 enumeration clauses in a DEFINITION statement for a single field. BEGIN must precede the first period in the definition. A DEFINITION statement that includes BEGIN must also include END and at least one level-88 or level-89 clause.

def-name

is the name of the data structure to be added to, or replaced in, the open dictionary.

def-name-1 TYPE *def-name-2*

defines a new data structure, *def-name-1*, by referring to a previously defined data structure, *def-name-2*. Both *def-name-1* and *def-name-2* are DLL names.

END [.]

ends either a group DEFINITION statement, a single-field DEFINITION statement that includes BEGIN, or a RECORD statement.

level-number { *field-name* | *group-name* | FILLER }

specifies a field or group of fields within a group definition.

level-number

is a two-digit number from 02 through 49 that establishes the hierarchy of fields or groups of fields within the definition or record.

Level number rules:

- The DEFINITION or RECORD statement does not have a level number; it is implicitly at level 01.
- Each group and field within a group DEFINITION statement or a RECORD statement has a level number to indicate its relationship to other groups and fields within the group. A group of level *nn* includes all following groups and fields with level numbers greater than *nn* up to the next group or field of level *nn* or less.
- Level numbers need not be assigned sequentially. For instance, an level-02 group can contain two level-05 fields with no intervening level-03 or level-04 fields.
- If a field is defined by a TYPE clause that refers to a group definition, the field's level number replaces the implicit level-01 number of the referenced definition, and the level numbers of the definition's member fields are adjusted accordingly.
- If a field is defined by a TYPE clause that refers to a previous definition, the level number of any element following the field must be less than or equal to the level number of the field.

field-name

is a name that uniquely identifies a field within the enclosing group description or definition.

group-name

is a name that uniquely identifies a group within the enclosing group description or definition.

FILLER

defines an unnamed field that is never referenced directly (see FILLER on page 6-7).

line-item specification

```
level-number { field-name | group-name | FILLER }
{ PICTURE-clause | TYPE-clause }
[ AS-clause ]
[ DISPLAY-clause ]
[ EDIT-PIC-clause ]
[ HEADING-clause ]
[ HELP-clause ]
[ JUSTIFIED-clause ]
[ LN-clause ] ...
[ MUST-BE-clause ]
[ NULL-clause ]
{ OCCURS-clause | OCCURS-DEPENDING-ON-clause }
[ REDEFINES-clause ]
[ SPI-NULL-clause ]
[ SQLNULLABLE-clause]
[ TACL-clause ]
[ USAGE-clause ]
[ VALUE-clause ] .
[ 88-condition-name-clause . ] ...
[ 89-enumeration-clause . ] ...
```

Clauses can be in any order, with this exception: definition attribute clauses must precede 88-condition-name clauses and 89 enumeration clauses.

6 Definition Attributes

Definition attributes are part of definitions and records, which are dictionary objects that describe data structures and disk-file record structures. Each definition and record includes attributes such as size, data type, and usage. The definition attributes are defined by clauses in DEFINITION statements or in the record structure portion of RECORD statements. Many of these clauses are similar to COBOL clauses of the same name.

Table 6-1. Definition and Record Clauses (page 1 of 2)

Clause	Function
AS *	Specifies a display string for a value of type ENUM
DISPLAY *	Specifies a default display format for field or group values listed on an Enform Plus report
EDIT-PIC *	Specifies the format in which Pathmaker-generated requesters display a field's data on a screen
EXTERNAL	Writes the EXTERNAL clause to COBOL source code files
FILLER	Defines an unnamed field that is never referenced directly
HEADING *	Specifies a default field heading for values listed on Enform Plus reports or displayed on screens generated by ENABLE and Pathmaker
HELP *	Assigns help text, used by Pathmaker-generated requesters, to a group or elementary item in a DEFINITION statement.
JUSTIFIED	Writes the JUSTIFIED RIGHT clause to COBOL source code files
KEYTAG	Specifies that a field or group is an Enscribe key field
LN	Specifies a locale name (language, territory, and character set) for a value in a CONSTANT statement, AS clause, HEADING clause, VALUE clause, or 88 condition-name clause
MUST BE *	Specifies the set of valid values for a field
NULL *	Assigns a null value to a field or group used as an Enscribe alternate key
OCCURS	Repeats a field or group a fixed number of times
OCCURS DEPENDING ON	Repeats a field or group a variable number of times (for COBOL and DDL source code only)
PICTURE	Specifies (using COBOL notation) the data type and size of a field
REDEFINES	Assigns a new name and, optionally, a new structure to a previously defined field or group

* The DDL compiler ignores this clause when generating host-language source code.

Table 6-1. Definition and Record Clauses (page 2 of 2)

Clause	Function
SPI-NULL	Defines an SPI null value for a field or a group in an SPI-extensible structured token or for a field or group within a group definition
[NOT]SQLNULLABLE	Specifies that a line item is [not] to be treated as an SQL-nullable column
TACL	Specifies the TACL data type to which a DDL data item is to be converted when generating TACL source code
TYPE	Specifies the data type and size of a data structure, either explicitly or by referring to a previously defined data structure
UPSHIFT	Upshifts ASCII characters entered in the field
USAGE	Either specifies computational storage allocation for a numeric group or field or identifies a COBOL as an index
[NO]VALUE	Assigns [suppresses] a DDL or COBOL field or group's initial value
66 RENAMES	Renames a previously defined DDL or COBOL field or group or set of fields or groups
88 Condition-Name	For COBOL source code, associates a condition name with a value, list of values, or range of values, enabling you to refer to the value or values by the condition name
89 Enumeration	Associates a name with a specified or default enumeration value and, optionally, specifies a display string for the value

* The DDL compiler ignores this clause when generating host-language source code.

AS

Note. The DDL compiler ignores this clause when generating host-language source code.

The AS clause specifies a display string.

Context	Effect
Field Definition on page 5-4	Specifies a display string for an enumeration value in a field of type ENUM
89 Enumeration on page 6-84	Specifies a default display string for a field of type ENUM. This default becomes the display string when the value of the field does not match any of the values specified by level-89 enumeration clauses in the field's definition or description.

```
AS display-string  [ LN-clause ]...
```

display-string

is either a string of ASCII or national characters (enclosed in quotation marks) or the name of a constant in the open dictionary. The value of the constant must be a string of ASCII or national characters.

LN-clause

specifies the locale name for *value* (see LN on page 6-13).

In Example 6-1 on page 6-3, a DDL definition uses an AS clause to specify a default display string.

Example 6-1. AS Clause

```
CONSTANT prts-obj-bolt      VALUE IS 1.
CONSTANT prts-obj-nut       VALUE IS 2.
CONSTANT prts-obj-pin       VALUE IS 3
CONSTANT prts-obj-screw     VALUE IS 4.
CONSTANT prts-obj-washer    VALUE IS 5.

DEF prts-ddl-object-type    TYPE ENUM BEGIN AS "Miscellaneous".
  89 prts-enm-bolt          VALUE IS prts-obj-bolt AS "Bolt".
  89 prts-enm-nut           VALUE IS prts-obj-nut AS "Nut".
  89 prts-enm-pin           VALUE IS prts-obj-pin AS "Pin".
  89 prts-enm-screw         VALUE IS prts-obj-screw AS "Screw".
  89 prts-enm-washer        VALUE IS prts-obj-washer AS "Washer".
END.
```

DISPLAY

Note. The DDL compiler ignores this clause when generating host-language source code.

The DISPLAY clause specifies a default display format for field or group values listed on an Enform Plus report.

```
DISPLAY display-format
```

display-format

is either a string (enclosed in quotation marks) or the name of a constant in the open dictionary. The value of *display-format* must be a string of repeatable edit descriptors, nonrepeatable edit descriptors, and modifiers, as described in the *Enform Plus Reference Manual*.

A display format specified in a DDL DISPLAY clause can be overridden by an Enform Plus AS clause.

The examples in Table 6-2 on page 6-4 show a commonly used format, the mask edit descriptor.

Table 6-2. Display Format Examples

Display Format	Value	Displayed Value
“M<99/99/99>”	012791	01/27/91
“M<Z,ZZZ.99>”	0.00	.00
“M<Z,ZZZ.99>”	1.499	1.50
“M<$ZZ,ZZ9.99>”	5246.95	$ 5,246.95
“M<(999) 999-9999>”	4084266974	(408) 426-6974
“M<99,999>”	524695	******* (overflow)

Example 6-2. Constant Names That Specify DISPLAY Formats

```
CONSTANT mdy-date-display   VALUE "M<mm/dd/yy>".
CONSTANT phone-display      VALUE "M<(999) 999-9999>".

DEF deliv-date   PIC 9(6)   DISPLAY mdy-date-display.
DEF custphone    PIC 9(10)  DISPLAY phone-display.
```

EDIT-PIC

Note. The DDL compiler ignores this clause when generating host-language source code.

The EDIT-PIC clause specifies the format in which Pathmaker-generated requesters display a field's data on a screen.

```
EDIT-PIC edit-picture-string
```

edit-picture

is either a string (enclosed in quotation marks) or the name of a constant in the open dictionary. The value of *edit-picture* must conform to the field's data type.

The EDIT-PIC clause does not replace the PICTURE clause. EDIT-PIC specifies a picture of a temporary item to which the value is moved for display.

If a field's data type and the edit picture are defined as two different data elements in the working-storage section of a COBOL program, moving the data from one picture to the other must be possible. The edit picture for an alphanumeric field must be alphanumeric, and the edit picture for a numeric field must be numeric.

The length of the data in an edit picture must conform to the length of the field's data type. To determine the data length in an edit picture, count only the digits or characters of data, not decorations. For example, the data length of $99.99 is 4.

For alphanumeric fields, these rules about data length apply:

- The length of the data in the edit picture must be less than or equal to the length of the field.
- The length of the data in the edit picture must be greater than 0.

For numeric fields, these rules apply:

- The length of the data to the left of the decimal point in the edit picture must be less than or equal to the length of the field to the left of the decimal point.
- The length of the data to the right of the decimal point in the edit picture must be less than or equal to the length of the field to the right of the decimal point.
- The length of the data in the edit picture must be greater than 0.

The maximum length of data in an EDIT-PIC clause is 32,767 bytes.

If an EDIT-PIC clause overrides an inherited edit picture, the edit picture specified in the clause must conform to the type of the referenced definition.

If an EDIT-PIC clause contains an invalid edit picture, the DDL compiler generates an error message and does not add the definition to the dictionary.

You cannot use an EDIT-PIC clause for data types not supported by COBOL.

In Example 6-3 on page 6-6:

- Although the first edit picture has more characters than the elementary item, the edit picture is valid because the extra characters leave room to display the minus sign and currency symbol at the beginning. Also, it is valid to display fewer characters to the right of the decimal point than the elementary item contains there.
- The data length of the second edit picture is invalid because it has too many minus signs. An edit picture can include 1 extra character for a minus sign, but the number of remaining minus signs must be equal to or less than the number of digits in the elementary item.
- The data length of the third edit picture is valid because the length on each side of the decimal point is less than or equal to the data length on that side in the elementary item.

Example 6-3. EDIT-PIC Clause

```
Elementary item:S999V99.

Valid:-$$$$.9
Invalid:-----.99
Valid:---.99
```

EXTERNAL

Note. The DDL compiler ignores this clause when generating host-language source code.

The EXTERNAL clause writes the EXTERNAL clause to COBOL source code files. The COBOL source code files can be part of a copy library that is shared among different program modules.

```
EXTERNAL
```

The EXTERNAL clause can be specified only on the object-name level.

If you specify the EXTERNAL clause in a definition statement, none of the line items in the definition or the record can have a VALUE clause or a REDEFINES clause.

The EXTERNAL clause is not inheritable.

The EXTERNAL clause cannot be used in combination with the FILLER clause.

FILLER

The FILLER clause defines an unnamed field that is never referenced directly.

```
FILLER
```

A FILLER field must have its data type and size specified with a PICTURE or TYPE clause.

A FILLER field can be repeated with an OCCURS clause.

A FILLER field is always part of a group definition or description, never a stand-alone field.

A FILLER field cannot be referenced directly, but it can be referenced indirectly as part of a group.

A FILLER field cannot be described with a DISPLAY, HEADING, HELP, KEYTAG, MUST BE, NULL, REDEFINES, or UPSHIFT clause.

Any noncomputational PICTURE clause or nonnumeric TYPE clause can be used to specify the length of a FILLER field. (In Example 6-4 on page 6-7, each FILLER field reserves a storage area of 6 bytes.)

Pascal (on D-series systems) and C do not have FILLER clauses. For these languages, the DDL compiler generates a unique name for each FILLER field; the name is of the form FILLER_*number*. The *number* portion of the name is incremented by 1 for each FILLER clause the DDL compiler encounters in the definition. For C, *number* starts at 0 for each new DDL definition. For Pascal, *number* starts at 1 for each new DDL definition.

If the generated name for a FILLER field would be the same as the name of an existing field or group at the same level, the DDL compiler uses the next integer that does not cause duplication.

Do not access the C or Pascal FILLER data items. Example 6-4 on page 6-7 shows a DDL definition containing FILLER clauses translated to C and Pascal source code.

Example 6-4. FILLER Clause

```
02 FILLER   PIC X(6) .
02 FILLER   TYPE CHARACTER 6 .
02 FILLER   PIC 9(6) .
```

Example 6-5. FILLER Clauses Translated to C and Pascal Source Code

DDL Definition

```
DEF name-struct.
  02 first-name    PIC X(10).
  02 FILLER        PIC X(6).
  02 second-name   PIC X(24).
  02 FILLER        PIC 9(6).
END.
```

C Code

```
#pragma section name_struct
#pragma fieldalign shared2_name_struct
struct name_struct_def
{
   char   first_name[10];
   char   filler_0[6];
   char   second_name[24];
   char   filler_1[6];
};
```

Pascal Code

```
?Section NAME_STRUCT
TYPE NAME_STRUCT_DEF = RECORD
   FIRST_NAME    : FSTRING[10];
   FILLER_1      : FSTRING[6];
   SECOND_NAME   : FSTRING[24];
   FILLER_2      : FSTRING[6];
END;
```

HEADING

Note. The DDL compiler ignores this clause when generating host-language source code.

The HEADING clause specifies a default field heading for values listed on Enform Plus reports or displayed on screens generated by ENABLE and Pathmaker.

```
HEADING report-heading  [ LN-clause ]...
```

report-heading

is either a string of ASCII or national characters (enclosed in quotation marks) or the name of a constant in the open dictionary. The value of the constant must be a string of ASCII or national characters.

LN-clause

specifies the locale name for *value* (see LN on page 6-13).

A slash (/) within *report-heading* indicates a line break in an Enform Plus heading. A slash within *report-heading* indicates a line break in a Pathmaker field only if the item format is tabular. If the item format is compressed or uncompressed, a slash within *report-heading* is replaced by a blank space.

If the HEADING clause is omitted for a field or group, the field or group name is the default heading.

A heading specified in the DDL HEADING clause can be overridden by the Enform Plus product or suppressed by the ENABLE or Pathmaker product.

Example 6-6 on page 6-9 and Example 6-7 on page 6-9 give the same result. The named constant in Example 6-7 on page 6-9 must be in the open dictionary.

Example 6-6. Multiline Heading in Enform Plus

HEADING Clause

```
DEF ordernum   PICTURE X(3)  HEADING "Order/Number" .
```

Heading Displayed

```
Order
Number
```

Example 6-7. Multiline Heading That Uses a Named Constant

```
CONSTANT ordernum-display VALUE "Order/Number".
DEF ordernum  PICTURE X(3) HEADING ordernum-display.
```

HELP

Note. The DDL compiler ignores this clause when generating host-language source code.

The HELP clause assigns help text, used by Pathmaker-generated requesters, to a group or elementary item in a DEFINITION statement.

```
HELP help-text [ [,] help-text ]...
```

help-text

is either a string of ASCII or national characters (enclosed in quotation marks) or the name of a constant in the open dictionary. The value of the constant must be a string of ASCII or national characters.

Single lines of text must not exceed 77 characters if you plan to generate a Pathmaker application; help text must be less than or equal to 77 characters to fit on a Pathmaker screen. If a single line of text does exceed 77 characters, you will receive a warning message.

The ASCII quotation mark character (") can be represented within a help text string by using two consecutive quotation marks ("").

Help text appears on DDL schema listings. Help text is displayed on the screen when an end user requests help from NonStop Transaction Services/MP (NonStop TS/MP) applications generated by the Pathmaker product.

Use of a comma between help text strings is optional. If you do not use a comma, you must delimit consecutive help strings by blanks or carriage returns. (Example 6-8 on page 6-11 shows both cases.)

Help text cannot be specified for any of these:

- Level-66 RENAMES clauses
- Level-88 condition-name clauses
- Record names
- File creation information

If a definition or record that includes help text refers to a definition that also includes help text, the help text in the referring definition or record overrides the help text in the referenced definition.

Example 6-8. HELP Clause

```
DEF address           HELP "This is a four-field",
                           "address consisting of street,"
                           "city, state code, and ZIP code.".
  02 street           PIC X(30).
  02 city             PIC X(20).
  02 state            PIC X(2).
  02 zip              PIC X(5).
END
```

If you use the same help text frequently, you can define a constant containing the help text and then use the constant name in the DEFINITION statement.

Example 6-9. Using a Constant for Frequently Used Help Text

```
CONSTANT mdy-date-display  VALUE IS "M<99/99/99>".
CONSTANT mdy-date-help     VALUE IS "date format: mm/dd/yy".

DEF mdy-date   PIC 9(6)    DISPLAY mdy-date-display
                           HELP "Enter date as", mdy-date-help.
```

You can combine a help text string with a constant in the same DEFINITION statement.

JUSTIFIED

Note. The DDL compiler ignores this clause when generating source code for languages other than COBOL.

The JUSTIFIED clause writes the JUSTIFIED RIGHT clause to COBOL source code files.

```
JUST[IFIED] RIGHT
```

The DDL compiler accepts JUST as an abbreviation for JUSTIFIED

The JUSTIFIED clause can appear only in an alphabetic or alphanumeric elementary item; it cannot appear in a group item.

An elementary item with a JUSTIFIED clause cannot be subordinate to a group item with a VALUE clause.

The JUSTIFIED clause is inheritable.

KEYTAG

Note. The DDL compiler ignores this clause when generating TACL source code.

The KEYTAG clause specifies that the field or group is an Enscribe key field.

```
KEYTAG key-specifier [ DUPLICATES [NOT] ALLOWED ]
```

key-specifier

is either as an integer from -32,768 through 32,767; two ASCII characters enclosed in quotation marks; or the name of a constant in the open dictionary. The value of the constant must be either an integer from -32,768 through 32,767 or a string of two ASCII characters.

You can omit *key-specifier* for a primary key, but if you include it, its value must be 0. A nonzero value for *key-specifier* indicates an alternate key.

DUPLICATES [NOT] ALLOWED

specifies whether to allow duplicate alternate key values. Do not specify DUPLICATES ALLOWED for a primary key field.

Default: DUPLICATES ALLOWED

If you use the KEYTAG clause to declare a record's key fields, you can omit *key-assignment* in the RECORD statement (as in Example 6-10 on page 6-12).

Key fields can overlap.

Example 6-10. KEYTAG Clause

```
CONSTANT partnum-heading  VALUE IS "Part/Number".
CONSTANT partnum-display  VALUE IS "M<ZZZ,ZZ9.99>".

RECORD partinfo.
  FILE IS "$data.sales.parts"  KEY-SEQUENCED.

  02 partnum   PIC 9(4)        KEYTAG   0
                               HEADING partnum-heading.
  02 partname  PIC X(18)       KEYTAG  "pn".
  02 inventory PIC 9(3)S.
  02 location  PIC X(3).
  02 price     PIC 9(6)V99     DISPLAY partnum-display.
END
```

LN

The LN clause specifies a locale name (language, territory, and character set) for:

- Specified *value-clause* items in CONSTANT on page 4-1
- *display-string* in AS on page 6-3
- *report-heading* in HEADING on page 6-9
- Specified *value* items in VALUE on page 6-75
- Specified *value* items in 88 Condition-Name on page 6-81

```
{ LN"language-code[_territory-code][.charset]" }
{ constant-name                                }
```

language-code

is a character string forming a language symbol.

territory-code

is a two-character string forming an ISO 3166:1988 Alpha-2 code entity name.

charset

is a string forming a HP internal character set name.

constant-name

is the name of a constant, previously defined in the dictionary. The constant name must be defined in the form *language-code*[*_territory-code*][*.charset*].

Example 6-11. DDL Locale Name and Components

locale-name	da_DK.ISO8859-1
language-code	da
territory-code	DK
charset	ISO8859-1

Table 6-3. Supported Locale Names

Locale Name	Description
POSIX	
C	
da_DK.ISO8859-1	Danish
de_CH.ISO8859-1	German, Switzerland
de_DE.ISO8859-1	German, Germany
el_GR.ISO8859-1	Greek
en_GB.ISO8859-1	English, UK
en_US.ISO8859-1	English, USA
es_ES.ISO8859-1	Spanish
fi_FI.ISO8859-1	Finnish
fr_BE.ISO8859-1	French, Belgium
fr_CA.ISO8859-1	French, Canada
fr_CH.ISO8859-1	French, Switzerland
fr_FR.ISO8859-1	French, France
is_IS.ISO8859-1	Icelandic
it_IT.ISO8859-1	Italian
en_JP.ISO8859-1	Japanese-English, Japan
ja_JP.AJEC	Japanese, EUC
ja_JP.SJIS	Japanese, SJIS
ko_KR.eucKR	Korean, EUC
nl_BE.ISO8859-1	Dutch, Belgium
nl_NL.ISO8859-1	Dutch, Netherlands
no_NO.ISO8859-1	Norwegian
pt_PT.ISO8859-1	Portuguese
sv_SE.ISO8859-1	Swedish
tr_TR.ISO8859-9	Turkish
zh_TW.eucTW	Taiwanese, EUC

If there is more than one literal specified with the same locale name for a text item, an error occurs. The literal with the duplicate locale name is ignored.

A text item is any text associated with an object. A text item has one of these types:

Type	Description
Number	ASCII representation of a numeric literal in a VALUE or MUST BE clause
String	Alphanumeric string in a COMMENT, DISPLAY, HEADING, HELP, MUST BE, PICTURE, or VALUE clause
Keyword	Keyword in a MUST BE or VALUE clause
Enumeration	Name of a value in a level 89 enumeration clause
National	National string in a MUST BE or VALUE clause
International	Internationalized text items in an 88 Condition-Name, AS, HEADING, or VALUE clause

A maximum of 32 internationalization (I18N) definitions are allowed per text item. If more than 32 I18N definitions are associated with one text item, an error occurs. The additional I18N definitions are not added to the dictionary.

Example 6-12. LN Clause

In CONSTANT Statement

```
CONSTANT Saga-Language VALUE "Icelandic" LN"is_IS.ISO8859-1"
```

In HEADING Clause (3)

```
DEFINITION custname PIC 9(4).
  HEADING "Finnish" LN"fi_FI.ISO8859-1"
          "Norwegian" LN"no_NO.ISO8859-1"
          "Danish" LN"da_DK.ISO8859-1".
```

MUST BE

Note. The DDL compiler ignores this clause when generating host-language source code.

The MUST BE clause specifies the set of valid values that can be entered in a field.

```
MUST BE { value                                  }
        { value-1 { THROUGH | THRU } value-2     }
```

value
value-1
value-2

is a value consistent with the data type of the field. *value-1* must be less than or equal to *value-2*.

```
{ "character-string"  }
{ constant-name       }
{ figurative-constant }
{ national-literal    }
{ number              }
{ symbolic-literal    }
{ value-name          }
```

character-string

is a string of ASCII characters.

constant-name

is the name of a constant in the open dictionary. The constant value must not be a figurative constant (see Table 6-4 on page 6-17) or symbolic literal (see Table 6-5 on page 6-17), and must be consistent with type of any associated level-88 data item (see 88 Condition-Name on page 6-81).

figurative-constant

is a figurative constant from Table 6-4 on page 6-17.

national-literal

is a national literal whose length is consistent with the length specified in the PICTURE clause for the national data item.

number

is one or more digits (*0* through *9*), an optional plus (+) or minus (-) sign, and an optional decimal point.

symbolic-literal

is a symbolic literal from Table 6-5 on page 6-17. Use symbolic literals only for numeric items.

The DDL compiler replaces *symbolic-literal* with the appropriate literal for COBOL output; therefore, the generated COBOL output does not contain a MUST BE clause.

value-name

is the *value-name* in the clause 89 Enumeration on page 6-84.

Table 6-4. Figurative Constants

Figurative Constant *	Value
LOW-VALUE LOW-VALUES	One or more of the lowest character in the ASCII or national collating sequence
HIGH-VALUE HIGH-VALUES	One or more of the highest character in the ASCII or national collating sequence
QUOTE QUOTES	One or more of the ASCII or national quotation mark character
SPACE SPACES	One or more of the ASCII or national space character (blank)
ZERO ZEROS ZEROES	Either the numeric value 0 or one or more of the ASCII or national zero character, depending on context
ALL *literal* *literal*	A repeated literal. The literal can be either an ASCII character string, a national literal, or a figurative constant other than ALL. When the literal is a figurative constant, the word ALL is unnecessary.

* Figurative constants in the same row are equivalent.

Table 6-5. Symbolic Literals

Symbolic Literal	Value
LOW-NUMBER	The minimum numeric value of the type specified for this field
HIGH-NUMBER	The maximum numeric value of the type specified for this field

A MUST BE clause cannot be specified for a group item. A MUST BE clause can be specified for individual fields within the group, as long as the group does not have an initial value.

For a data item declared with TYPE BINARY 64 UNSIGNED, the MUST BE clause supports a value range of only 0 to 9,223,372,036,854,775,807.

If a field described with a MUST BE clause also has a VALUE clause, the initial value specified in the VALUE clause must satisfy the MUST BE constraints.

If a field described with a MUST BE clause also has an UPSHIFT clause, the MUST BE values must be upshifted.

If a field described with a MUST BE clause is of type ENUM, the values in the clause can only be level-89 enumeration names.

You can specify only one MUST BE clause for a field.

You cannot specify a MUST BE clause in a field or group definition or description that includes a REDEFINES clause.

You cannot specify a MUST BE clause for fields of some SQL data types (see the *SQL/MP Reference Manual* and *SQL/MX Reference Manual*).

You cannot specify CURRENT, SYSTEM, or SQLNULL as a value for a MUST BE clause.

Requesters generated by the Pathmaker product enforce the MUST BE constraints; programs written by users must be coded to enforce these constraints as well.

In Example 6-13 on page 6-18, the MUST BE clause defines the acceptable ranges of values for days in a month and months in a year.

Example 6-13. MUST BE Clause

```
DEF date.
  02 day          PIC 9(2)
                  MUST BE 1 THROUGH 31.
  02 month        PIC 9(2).
                  MUST BE 1 THROUGH 12.
  02 year         PIC 9(2).
END
```

If you specify the same MUST BE values frequently, you can define the values as constants. You can also use the constant names in condition-name clauses associated with the definition.

Example 6-14. Defining MUST BE Values as Constants

```
CONSTANT sales       VALUE 1.
CONSTANT shipping    VALUE 2.
CONSTANT personnel   VALUE 3.

DEF company.
  02 department   TYPE BINARY 16
                  MUST BE sales,
                          shipping,
                          personnel.
  88 sales        VALUE sales.
  88 shipping     VALUE shipping.
  88 personnel    VALUE personnel.
END
```

NULL

Note. The DDL compiler ignores this clause when generating host-language source code.

The NULL clause assigns a null value to a field or group used as an Enscribe alternate key. If a record being inserted in the database has a null value in the alternate key field, the alternate key is not added to the alternate key file.

```
NULL { "character" | number  | constant-name }
```

character

is any ASCII character.

number

is any number from 0 through 255.

constant-name

is the name of a constant in the open dictionary. The constant value must be a valid *character* or *number* value.

Any alternate key can be assigned a null value. The most common null values are ASCII blank (%40) and binary zero. The null value used must fit in one byte.

When you generate FUP source code from the DDL definition, the FUP code specifies alternate key file information, including the octal representation of the null value you select.

The file system checks records as they are inserted in the file to see if the value in the alternate key field matches the null value. The effects of using a null value are:

- When records are inserted, if the record has an alternate key with a null value, the key is not added to the alternate key file.
- When records are updated, any alternate key with a null value is deleted from the alternate key file.
- If a file is read sequentially by an alternate key, any record with a null value for that alternate key is not found.

In Example 6-15 on page 6-20, if the employee does not have a spouse or dependents, the key is not added to the alternate key file.

Example 6-15. NULL Clause

```
RECORD employee.
FILE IS "employee" key-sequenced.

  02 empinfo.
     04 empid         PIC 9(4).
     04 empname       PIC X(22).
     04 dept          PIC X(4).

  02 taxinfo          NULL 0.
     04 spousename    PIC X(22).
     04 dependents    PIC 9(2).

KEY IS empid.
KEY "ti" is taxinfo.
END
```

You can also use a constant name to specify the NULL value.

Example 6-16. Specifying NULL Value With a Constant

```
CONSTANT null-0  VALUE 0.

  ...

   02 taxinfo          NULL null-0.
      04 spousename    PIC X(22).
      04 dependents    PIC 9(2).
```

OCCURS

The OCCURS clause repeats a field or group a fixed number of times.

```
OCCURS max [ TIMES ] [ INDEXED BY index-name ]
```

max

specifies the number of times the field or group repeats. You can specify *max* either as an integer or as the name of a constant in the open dictionary. The value of *max* must be an integer from 1 through 32,767.

index-name

is the name of a field to use as an index. The maximum length of *index-name* is 30 ASCII characters.

Note. Use INDEXED BY *index-name* only for COBOL.

These statements apply to both the OCCURS clause and the OCCURS DEPENDING ON clause except as explained in OCCURS DEPENDING ON on page 6-23:

- OCCURS cannot be specified for the first element of a RECORD or DEFINITION statement. OCCURS can be specified only at level number 02 or greater.
- A field that is described with an OCCURS clause or that is part of a group described with an OCCURS clause cannot have a VALUE clause unless the VALUE is associated with a level-88 condition-name clause.
- A field or group described with an OCCURS clause cannot be specified as a key field in a RECORD statement.
- OCCURS clauses can be nested. COBOL allows seven levels of nested OCCURS clauses.
- When OCCURS clauses are nested, a separate subscript is associated with each level of nesting; the subscripts are written in order from most inclusive to least inclusive.
- The form of the subscript depends on the language. For example, COBOL encloses subscripts in parentheses, and pTAL or TAL encloses them in brackets. Subscript bounds depend on the language accessing the data:
  - For Pascal (on D-series systems), COBOL, and FORTRAN, the subscript bounds are implicitly 1 and *max*.
  - For C and TACL, the subscript bounds are implicitly 0 and *max* -1.
  - The values of pTAL, TAL, or Pascal subscripts depend on the TALBOUND or PASCALBOUND command. TALBOUND 0 or PASCALBOUND 0 causes the subscript bounds to be 0 and *max* - 1. TALBOUND 1 or PASCALBOUND 1 causes the subscript bounds to be 1 and *max*.

    The DDL compiler compiles the TALBOUND and PASCALBOUND setting (0 or 1 for each) to the OCCURS definition. You can change this value only by replacing the definition.
- COBOL output for the INDEXED BY attribute is the direct translation of the attribute.
- If you specify an index name in the OCCURS clause, do not specify USAGE IS INDEX for the field of that name, because COBOL requires that all index names be unique throughout a program. The DDL compiler checks for the uniqueness of an index name you specify in the INDEXED BY attribute.
- A group can be repeated with an OCCURS clause, as in Example 6-18 on page 6-22.

In Example 6-17 on page 6-22, which declares storage for 52 paycheck values, one for each week of the year:

- TAL programs with TALBOUND 1 or with no TALBOUND clause access individual paycheck values like this:

  `PAYCHECK [1]` Paycheck value for the first week

  `PAYCHECK [52]` Paycheck value for the last week

- TAL programs with TALBOUND 0 access individual paycheck values like this:

  `PAYCHECK [0]` Paycheck value for the first week

  `PAYCHECK [51]` Paycheck value for the last week

Example 6-17. OCCURS Clause

```
DEF salary.
  02 paycheck     PIC 9999V99
                  OCCURS 52 TIMES.
END
```

Example 6-18. Repeating a Group With an OCCURS Clause

```
DEF paydate.
  02  dates OCCURS 12 TIMES.
    03  month PIC 99.
    03  day   PIC 99.
    03  year  PIC 99.
END
```

To refer to an individual field within a group, follow the field name with a subscript. For example, to refer to the tenth month within the `dates` group in Example 6-18 on page 6-22, a COBOL program uses the subscript 10:

```
month(10)
```

Example 6-19. Constant as OCCURS Value

```
CONSTANT pay-period   VALUE IS 24.

DEF bi-monthly-paydate.
  02 paydate  OCCURS pay-period TIMES.
     03 bi-month  PIC 99.
     03 day       PIC 99.
     03 year      PIC 99.
END
```

COBOL output for the INDEXED BY attribute:

DDL Code	COBOL Code

DEF xyz
  02 abc TYPE BINARY
  OCCURS 3 TIMES
  INDEXED BY ix.
END.
	01 XYZ.
	  02 ABC NATIVE-2
	  OCCURS 3 TIMES
	  INDEXED BY IX.

OCCURS DEPENDING ON

For DDL and COBOL source code, the OCCURS DEPENDING ON clause repeats a field or group a variable number of times, depending on the current value of an integer variable.

For source code in other languages, the OCCURS DEPENDING ON clause repeats a field or group the specified maximum number of times.

```
OCCURS min TO max TIMES DEPENDING ON field-name
  [ INDEXED BY index-name ]
```

min

is the minimum number of times the field or group can repeat. You can specify *min* either as an integer or as the name of a constant in the open dictionary. The value of *min* must be an integer from 0 through 32,767.

max

is the maximum number of times the field or group can repeat. You can specify *max* either as an integer or as the name of a constant in the open dictionary. The value of *max* must be a positive integer greater than or equal to the value of *min*.

field-name

is the name of a numeric field within the same definition. The value of *field-name* must be a positive integer.

index-name

is the name of a field to use as an index. The maximum length of *index-name* is 30 ASCII characters.

Note. Use INDEXED BY *index-name* only for COBOL.

The OCCURS DEPENDING ON clause differs from OCCURS on page 6-20 in that:

- For Pascal (on D-series systems), C, FORTRAN, pTAL, TACL, and TAL, the DDL compiler generates source code identical to the code it generates for OCCURS *max* TIMES.
- Only one OCCURS DEPENDING ON clause can be in a DEFINITION or RECORD statement, and the clause's subordinate fields or groups must be the last fields or groups in that statement.
- OCCURS DEPENDING ON clauses cannot be nested; however, a subordinate field or group can have an OCCURS clause.
- COBOL output for the INDEXED BY attribute is the direct translation of the attribute.
- If you specify an index name in the OCCURS clause, do not specify USAGE IS INDEX for the field of that name, because COBOL requires that all index names be unique throughout a program. The DDL compiler checks for the uniqueness of an index name you specify in the INDEXED BY attribute.

In Example 6-20 on page 6-24, the number of occurrences of DEP-NAME depends on the value of NUM-DEP. NUM-DEP must contain a positive integer value.

Example 6-20. OCCURS DEPENDING ON Clause (page 1 of 2)

```
DEF name.
  02 last-name      PIC X(12).
  02 first-name     PIC X(8).
  02 midinit        PIC X(2).
END

DEF addr.
  02 address        PIC X(22).
  02 city           PIC X(14).
  02 state          PIC X(2).
  02 zip            PIC 9(5).
END

DEF employee.
  02  emp-name      TYPE name.
  02  emp-addr      TYPE addr.
  02  num-dep       TYPE BINARY 16  MUST BE 0 THRU 12.
  02  dep-name      TYPE name
                    OCCURS 0 TO 12 TIMES
                    DEPENDING ON num-dep.
END
```

Example 6-20. OCCURS DEPENDING ON Clause (page 2 of 2)

DDL Code for DEPENDING ON Clause

```
DEF xyz.
  02 i      TYPE BINARY
  02 abc TYPE BINARY
     OCCURS 1 to 3

TIMES
     DEPENDING ON i
     INDEXED BY ix.
END.
```

COBOL Code for DEPENDING ON Clause

```
01 XYZ.
  02 I        NATIVE-2.
  02 ABC NATIVE-2
     OCCURS 1 TO 3

TIMES
     DEPENDING ON I OF

XYZ
     INDEXED BY IX.
```

PICTURE

The PICTURE clause specifies (using COBOL notation) the data type and size of a field or of a field.

```
PIC[TURE] { "{picture-string          }" }
          {  {national-picture-string }  }
          {                              }
          {  {picture-string          }  }
          {  {national-picture-string }  }
```

picture-string

specifies the data type and size of a field:

```
{ alphanumeric-string | numeric-string }
```

alphanumeric-string

```
{ A | X | 9 }...[(length)]
```

numeric-string

```
{ [S]9...[(length)[V[9...[(length)]]]] }
{ T[9...[(length)[V[9...[(length)]]]]] }
{ 9...[(length)[V[9...[(length)]]]]S   }
{ 9...[(length)[V[9...[(length)]]]]T   }
```

X

represents any ASCII character.

A

represents any lowercase or uppercase letter of the alphabet or an ASCII blank.

`9`

represents an ASCII digit, from *0* through *9*.

`length`

is a one-digit to five-digit integer that specifies the number of times the preceding symbol repeats.

You can omit *`length`* and specify the length by repeating the symbol (*X, A,* or *9*) once for each character position you want in the field.

`S`

represents a sign character in a signed numeric field.

`T`

represents a numeric character with an implied embedded sign.

Alone, the symbol *T* represents a one-byte numeric field.

`V`

represents an implied decimal point location in a numeric field.

`national-picture-string`

```
{ { N | n } [(length)] }
{ { N | n }...         }
```

`{ N | n }`

represents a national character.

`length`

is a one-digit to five-digit integer that specifies the number of times the preceding symbol repeats.

You can omit *`length`* and specify the length by repeating the symbol (*N* or *n*) once for each character position you want in the field.

Each national character occupies two bytes.

If *`picture-string`* has two or more of the symbols *X, A,* and *9*, the DDL compiler assumes the data type is alphanumeric (PIC X).

Example 6-21. PICTURE Clauses Describing ASCII Character Fields

```
DEF ascii-pictures
  02 alpha-field  PIC A(10).        ! 10 alphabetic characters
  02 alphanum-2   PIC X(10).        ! 10 alphanumeric characters
  02 alphanum-1   PIC AAX(4)9(4).   ! 10 alphanumeric characters
  02 nat-field    PIC N(5).         ! 5 2-byte national characters
  02 unsigned     PIC 9(5).         ! 5 unsigned digits
  02 signed-1     PIC S9(5).        ! 5 digits plus leading sign
  02 signed-1     PIC 9(5)S.        ! 5 digits plus trailing sign
  02 signed-2     PIC T9(5).        ! 5 digits plus embedded leading sign
  02 signed-3     PIC 9(5)T.        ! 5 digits plus embedded trailing sign
  02 imp-decimal  PIC 9(3)V9(2).    ! 5 digits with implied decimal point
END.
```

Example 6-22. PICTURE Clauses Describing Binary Fields

```
DEF binary-pictures
  02 binary-int      PIC 9(4)   COMP.  ! 2-byte unsigned integer
  02 binary-int-s    PIC 9S(4)  COMP.  ! 2-byte signed integer
  02 binary-int2     PIC 9(5)   COMP.  ! 4-byte unsigned integer
  02 binary-int2-s   PIC S9(5)  COMP.  ! 4-byte signed integer
  02 binary-int4     PIC 9(10)  COMP.  ! 8-byte unsigned integer
  02 binary-int4-s   PIC S9(10) COMP.  ! 8-byte signed integer
END.
```

Topics:

National Data Items

The only symbol you can specify in a national picture string is *N* or *n* (except for the parentheses and a number to specify the length, or number of repetitions).

The maximum length you can specify for a national data item is half of the maximum internal field length. For definitions, the maximum internal field length is 32,767 bytes. For records, the maximum length depends on the file type:

File Type	Record's Maximum Internal Field Length
Entry-sequenced	4,072 bytes
Key-sequenced	4,062 bytes
Relative	4,072 bytes
Unstructured	4,096 bytes

PIC N(16383) specifies the maximum length allowed for a field definition.

Only COBOL output for a national data item appears as defined in DDL. For other host-language output, the DDL compiler generates the equivalent number of characters. For example, PIC N(10) in DDL translates to:

Language	Output
C	`char name [20]`
FORTRAN	`CHARACTER*20`
Pascal (on D-series systems)	`FSTRING (20)`
TACL	`STRUCT name: BEGIN CHAR BYTE (0:19); END;`
pTAL or TAL	`STRUCT name: BEGIN STRING BYTE [1:20]; END;`

C

The DDL compiler translates alphanumeric and numeric PICTURE clauses, except numeric clauses described with USAGE IS COMPUTATIONAL, to C char types. The DDL compiler translates numeric PICTURE clauses with USAGE IS COMPUTATIONAL to C `short`, `long`, `double`, and `long long` types.

If a field described with USAGE IS COMPUTATIONAL has a PICTURE declaration of the form

```
PIC 9 ... [(length)] [V 9 ... [(length)]]
```

and the symbol *9* occurs 10 or more times, the item is declared as TYPE BINARY 64 UNSIGNED.

For more information, see Table C-1, Sample DDL/C Data Translation Table, on page C-1.

COBOL

DDL PICTURE clauses are translated to COBOL PICTURE clauses.

For a national picture string, the DDL compiler generates COBOL output as specified in the PICTURE clause.

The maximum field length depends on data type:

- If an alphanumeric or national field is used only in working storage (not in a record), the maximum field length is 32,767 bytes.
- If an alphanumeric or national field is defined in or referenced by a RECORD statement, the maximum field length is the maximum record length, which depends on file type:

File Type	Record's Maximum Length
Entry-sequenced	4,072 bytes
Key-sequenced	4,062 bytes
Relative	4,072 bytes
Unstructured	4,096 bytes

The maximum length of a numeric field is 18 digits.

The symbols *S* and *V* are not counted in the 18-digit COBOL limit on numeric fields; the symbols 9 and *T* are each counted as 1 digit in the 18-digit COBOL limit on numeric fields.

The DDL symbol *S* is not the same as the COBOL PICTURE *S*:

- In DDL, the symbol *S* represents a digit with a separate sign. the DDL compiler translates the symbol *S* to the COBOL PICTURE *S* and adds a COBOL SIGN clause with a SEPARATE phrase.
- If *S* is the first symbol in a numeric picture string, the DDL compiler adds SIGN LEADING SEPARATE.
- If *S* is the last symbol in a numeric picture string, the DDL compiler adds SIGN TRAILING SEPARATE.

For COBOL, the symbol *T* represents a digit that contains an embedded sign:

- DDL translates the symbol *T* to PICTURE S9 and adds the COBOL SIGN clause.
- If the *T* is the first character in the PICTURE string, the DDL compiler adds SIGN LEADING.
- If *T* is the last character, the DDL compiler adds SIGN TRAILING.
- If *T* is the only character in the PICTURE string, the DDL compiler translates the PICTURE clause to a PIC S9 SIGN IS LEADING clause for COBOL.
- The symbol *T* is counted as one digit in a numeric field.

For more information, see Table C-2, Sample DDL/COBOL Data Translation Table, on page C-3.

FORTRAN

Most alphanumeric and numeric PICTURE clauses are translated to FORTRAN character strings. The only exceptions are numeric fields described with USAGE IS COMP; these fields are translated to FORTRAN integers.

The maximum length of an alphanumeric or a numeric field is 255 bytes.

For more information, see Table C-3, Sample DDL/FORTRAN Data Translation Table, on page C-5.

Pascal (D-series Systems Only)

DDL translates alphanumeric, national, and numeric PICTURE clauses, except numeric clauses described with USAGE IS COMP, to Pascal FSTRING types. Numeric PICTURE clauses with USAGE IS COMP translate to Pascal integer types.

For more information, see Table C-4, Sample DDL/Pascal Data Translation Table, on page C-7.

pTAL and TAL

Alphanumeric, national, and numeric PICTURE clauses are translated to pTAL or TAL character strings, except for numeric fields described with USAGE IS COMP, which are translated to pTAL or TAL binary data types.

If a field described with USAGE IS COMPUTATIONAL has a PICTURE declaration of the form

```
PIC 9 ... [(length)] [V 9 ... [(length)]]
```

and the symbol *9* occurs 10 or more times, the item is declared as TYPE BINARY 64 UNSIGNED.

The maximum field length depends on where the field is used:

- If a numeric, national, or alphanumeric field is not defined in or referenced by a RECORD statement, the maximum field length is 32,767 bytes.
- If a numeric, national, or alphanumeric field is defined in or referenced by a RECORD statement, the maximum field length is the maximum record length, which depends on file type:

File Type	Record's Maximum Length
Entry-sequenced	4,072 bytes
Key-sequenced	4,062 bytes
Relative	4,072 bytes
Unstructured	4,096 bytes

For more information, see Table C-6, Sample DDL/pTAL and TAL Data Translation Table, on page C-11.

TACL

Most alphanumeric, national, and numeric PICTURE clauses are translated to TACL STRUCTs containing character strings. The only exception is numeric PICTURE clauses described with USAGE IS COMP. COMP numeric fields are translated to TACL STRUCTs containing binary data types.

If a TACL clause is specified, the resulting TACL STRUCT contains the high-level data type specified in the TACL clause rather than the standard TACL data type generated from a PICTURE clause.

The maximum length of a TACL STRUCT is 5,000 bytes. Any filler generated by the DDL compiler for alignment counts towards this maximum length.

For more information, see Table C-5, Sample DDL/TACL Data Translation Table, on page C-9.

REDEFINES

The REDEFINES clause assigns a new name and, optionally, a new structure to previously defined field or group.

```
REDEFINES { field-name | group-name }
```

field-name

is the name of the previous field in the definition or record currently being defined.

group-name

is the name of the previous group in the definition or record currently being defined.

Redefining structures must start at the same level as the structures they redefine.

A redefining structure must immediately follow the structure it redefines except in the case of multiple redefines where each redefining structure refers back to the same original structure.

A redefining field must not have a VALUE clause, a MUST BE clause, or an UPSHIFT clause.

Because the data type of a group is always alphanumeric, an attempt to redefine a group containing binary items can produce unpredictable results.

A redefining structure must not be larger than the structure it redefines.

Example 6-23 on page 6-32 defines storage for exempt employees and redefines it for nonexempt employees.

Example 6-23. REDEFINES Clause

```
DEF employee.
  02 emp-id        PIC 9(4).
  02 emp-name      PIC X(22).
  02 emp-type      PIC X.
  02 exmpt-emp.
     04 salary     PIC 9(6)V99.
  02 non-exmpt-emp REDEFINES exmpt-emp.  ! Redefines salary
    04 hrly-wage   PIC 9(3)V99.
    04 hrs-wrkd    PIC 9(3).
  02 dept          PIC 9(4).
  02 emp-sex       PIC X.
  02 spouse-name   PIC X(22).
END
```

Topics:

C

For C, the DDL compiler generates source code that combines the items of a REDEFINES clause to a union. The C structure containing such a union has one more item level than the corresponding DDL structure containing the REDEFINES clause. This situation causes the DDL compiler to issue a warning message unless you include the NOWARN command. The name of the union has the form u_*first_member_name*. If the union name generated by the DDL compiler is the same as any of its siblings defined in the same group, the DDL compiler issues an error message and does not generate output.

Example 6-24. REDEFINES Clause With C Output (page 1 of 2)

DDL Input

```
DEF a.
  02 b  PIC 9(4)
  02 c  PIC 9(6).
  02 d  PIC 9(6)  REDEFINES c.
END
```

Example 6-24. REDEFINES Clause With C Output (page 2 of 2)

DDL Output (C Code)

```
#pragma fieldalign shared2__a
typedef struct__a
{
char b[4];
union
  {
    char c[6];
    char d[6];
  } u_c;
} a_def;
```

COBOL

In COBOL, a redefining structure must not be smaller than the structure it redefines. When REDEFINES and OCCURS clauses are at the same level, then FILLER emitted results in an incompatible structure when compared with C, pTAL, or TAL output. To avoid this, split REDEFINES and OCCURS to separate groups. Whenever possible, the DDL compiler pads the smaller structure with FILLER fields to make it the same size as the structure it redefines.

FORTRAN

Example 6-25. REDEFINES Clause With FORTRAN Output (page 1 of 2)

DDL Input

```
DEF A.
    02 B          PIC 9(4).
    02 C          REDEFINES B.
       04 C-1     PIC 9(2).
       04 C-2     PIC 9(2).
    02 D.
       04 D-1     PIC X.
       04 D-2     PIC 9 REDEFINES D-1.
    02 E          PIC 9(5).
    02 F          REDEFINES E.
       04 F-1     PIC 9(3).
       04 F-2     PIC 9(2).
END.
```

Example 6-25. REDEFINES Clause With FORTRAN Output (page 2 of 2)

DDL Output (FORTRAN Code)

```
RECORD A
   CHARACTER*4 B
   RECORD C
     CHARACTER*2 C1
     CHARACTER*2 C2
   END RECORD
   EQUIVALENCE ( C, B )
   RECORD D
     CHARACTER*1 D1
     CHARACTER*1 D2
     EQUIVALENCE ( D2, D1 )
   END RECORD
   CHARACTER*5 E
   RECORD F
     CHARACTER*3 F1
     CHARACTER*2 F2
   END RECORD
   EQUIVALENCE ( F, E )
END RECORD
```

Pascal (D-series Systems Only)

For Pascal, the DDL compiler translates a REDEFINES clause to a variant record.

The variants within the record are the data items of the redefined structure and the data items of the redefining structure.

The DDL compiler generates integer case labels for each variant. For each REDEFINES clause, the integer case labeling begins at 1.

If the REDEFINES clause is not the last item in its group, the DDL compiler generates an anonymous record to contain the variant or variants. The DDL compiler then issues a warning. The DDL compiler generates the variant record name by prefixing a V_ to the name of the first structure being redefined. If the DDL-generated variant record name is the same as any of its siblings defined in the same group, the DDL compiler issues an error message and does not generate output.

Pascal does not do any run-time checking to enforce which variant is active at any given time.

Example 6-26 on page 6-35 shows the Pascal source code generated by the DDL compiler for a REDEFINES clause. The DDL compiler generates an anonymous record (V_B) for the REDEFINES B clause because this clause was not the last level-02 item in DEF A. The DDL compiler did not generate an anonymous record for the REDEFINES D clause because it was the last level-04 item in D, nor did the DDL compiler generate one for REDEFINES E because it was the last level-02 item in DEF A.

Example 6-26. REDEFINES Clause With Pascal Output

DDL Input

```
DEF A.
    02 B          PIC 9(4).
    02 C          REDEFINES B.
       04 C-1     PIC 9(2).
       04 C-2     PIC 9(2).
    02 D.
       04 D-1     PIC X.
       04 D-2     PIC 9 REDEFINES D-1.
    02 E          PIC 9(5).
    02 F          REDEFINES E.
       04 F-1     PIC 9(3).
       04 F-2     PIC 9(2).
END.
```

DDL Output (Pascal Code)

```
TYPE A_DEF = RECORD
   V_B                  : RECORD

   CASE INT16 OF
      01:( B            : FSTRING[4]   );
      02:( C            : RECORD
                C_1     : FSTRING[2];
                C_2     : FSTRING[2];
             END  );
   END;

   D                    : RECORD
      CASE INT16 OF
         01:( D_1       : CHAR  );
         02:( D_2       : CHAR  );
   END;

   CASE INT16 OF
      01:( E            : FSTRING[5]   );
      02:( F            : RECORD
                F_1     : FSTRING[3];
                F_2     : FSTRING[2];
             END  );
END;
```

pTAL or TAL

In pTAL or TAL, a redefining structure can be smaller than the structure it redefines.

TACL

Example 6-27. REDEFINES Clause With TACL Output

DDL Input

```
DEF A.
    02 B          PIC 9(4).
    02 C          REDEFINES B.
       04 C-1     PIC 9(2).
       04 C-2     PIC 9(2).
    02 D.
       04 D-1     PIC X.
       04 D-2     PIC 9 REDEFINES D-1.
    02 E          PIC 9(5).
    02 F          REDEFINES E.
       04 F-1     PIC 9(3).
       04 F-2     PIC 9(2).
END.
```

DDL Output (TACL Code)

```
?Section A Struct
Begin
STRUCT   B;
   BEGIN CHAR BYTE(0:3); END;
STRUCT   C REDEFINES B;
   Begin
   STRUCT   C^1;
      BEGIN CHAR BYTE(0:1); END;
   STRUCT   C^2;
      BEGIN CHAR BYTE(0:1); END;
   End;
STRUCT   D;
   Begin
   CHAR     D^1;
   CHAR     D^2 REDEFINES D^1;
   End;
STRUCT   E;
   BEGIN CHAR BYTE(0:4); END;
STRUCT   F REDEFINES E;
   Begin
   STRUCT   F^1;
      BEGIN CHAR BYTE(0:2); END;
   STRUCT   F^2;
      BEGIN CHAR BYTE(0:1); END;
   End;
End;
```

SPI-NULL

The SPI-NULL clause specifies an SPI null value for a field or group in an SPI-extensible structured token or for a field or group within a group definition.

Note. Use the SPI-NULL clause only if you plan to use SPI messages to communicate among processes in a Distributed Systems Management (DSM) environment.

```
SPI-NULL {"character" | number | constant-name }
```

character

is any ASCII character.

number

is any number from 0 through 255.

constant-name

is the name of a constant in the open dictionary. The constant value must be a valid *character* or *number* value.

The SPI-NULL value must fit in one byte.

A field or group with an SPI null value in every byte is considered to have unspecified data.

The SPI-NULL clause differs from the NULL clause in that:

- The SPI-NULL clause is used only to assign an SPI null value to a field or a group of fields that will be used in an SPI extensible structured token defined by a TOKEN-MAP statement.
- The NULL clause is used only to assign a null value to an alternate-key field referenced in a RECORD statement.

Every field in a extensible structured token must have an SPI null value, whether specified explicitly or implicitly by default.

For a field to contain an SPI null value, each byte of the field must contain the value specified in the SPI-NULL clause. You use the SPI SSNULL procedure to fill the field with the SPI null value specified in the SPI-NULL clause.

The SPI-NULL value for a bit field must be 255.

If SPI-NULL is not specified, the default SPI null value is 255; that is, SPI-NULL sets all bits to 1.

An explicit SPI-NULL clause for a field or for a group containing the field overrides the default SPI null value.

Do not specify a VALUE clause for a field or group used to define an extensible structured token. Every field in an extensible structured token is initialized to its SPI null value before it is used, so any initial value is overwritten.

If the field is used for other purposes, then you can specify a VALUE clause as well as an SPI-NULL clause. In this case, the field is not initialized to its SPI null value but is given the specified initial value. Because an initial value and an SPI null value are never used for the same purpose, they need not be the same value.

An SPI-NULL clause specified in a group definition, or in a group description within a group definition, is inherited by each of the fields within the group that has the clause. A field within a group defined with an SPI-NULL clause cannot have its own SPI-NULL clause.

When you refer to one definition from another:

- If you specify an SPI-NULL clause in the referring definition or in any group that includes the referring definition, the specified SPI null value overrides all SPI null values in the referenced definition.
- If you do not specify an SPI-NULL clause in the referring definition, the referring definition inherits the SPI null value of the referenced definition.

Example 6-28. SPI-NULL Clause For a Single Field

```
DEF assn-ddl-jobcode   TYPE BINARY 16  SPI-NULL 0.
```

Example 6-29. SPI-NULL Clause For a Group of Fields

```
DEF assn-ddl-jobinfo SPI-NULL 1
  02 jobcode   TYPE BINARY 16.     ! Inherits SPI-NULL value 1.
  02 priority  TYPE BINARY 16.     ! Inherits SPI-NULL value 1.
END.
```

When a definition refers to another definition and the referring definition contains one or more SPI-NULL clauses, these clauses override any SPI-NULL clauses in the referenced definition. If the referring definition does not have any SPI-NULL clauses, it inherits the SPI null value or values from the referenced definition.

Example 6-30. Inherited and Overridden SPI-NULL Values

```
DEF assn-ddl-jobcode SPI-NULL 0.
  02 prefix TYPE BINARY 16.  ! SPI-NULL 0 (inherited)
  02 code   TYPE BINARY 16.  ! SPI-NULL 0 (inherited)
END

DEF assn-ddl-jobinfo.
  02 jobcode  TYPE assn-ddl-jobcode.     ! SPI-NULL 0   (inherited)
  02 priority TYPE BINARY 16 SPI-NULL 1. ! SPI-NULL 1   (stated)
  02 location TYPE BINARY 16.            ! SPI-NULL 255 (default)
END

DEF assn-ddl-jobinfo-groups.
  02 jobinfo-1 TYPE assn-ddl-jobinfo.  ! Inherits jobinfo SPI-NULL values
  02 jobinfo-2 TYPE assn-ddl-jobinfo SPI-NULL 2. ! Overrides inherited value
END
```

If a field is sometimes used as an extensible structured token and sometimes for another purpose, you can define the field with both an SPI-NULL clause and a VALUE clause.

Example 6-31. Field Defined With SPI-NULL and VALUE Clauses

```
DEF jobclass TYPE BINARY 16  SPI-NULL 255
                             VALUE 0.
```

When JOBCLASS is used as an extensible structured token, the SSNULL procedure initializes JOBCLASS to the specified SPI null value. When JOBCLASS is used for any other purpose, the DDL compiler initializes it to the initial value specified in the VALUE clause.

SQLNULLABLE

The SQLNULLABLE clause specifies that a line item is to be treated as an SQL-nullable column. The NOTSQLNULLABLE clause specifies that a line item is not to be treated as an SQL-nullable column.

```
[NOT] SQLNULLABLE
```

In SQL, if a column is not explicitly specified as NOT NULL, it is a nullable column. Internally, a nullable SQL column is composed of the column itself and a numeric flag that indicates whether the column is null. The DDL compiler supports an SQL-nullable line item in the same way: an SQL-nullable line item consists of the line item itself and a numeric item that signals whether the item is null. Because of the presence of this additional numeric item, an SQL-nullable item is word aligned; the internal byte size of an SQL-nullable line item is the size specified plus 2.

Specifying SQLNULLABLE at the group level for definitions or records means that all subordinate line items in the group are SQL-nullable, except for those individual line items explicitly specified as NOT SQLNULLABLE. Specifying NOT SQLNULLABLE for a group means that all its subordinate line items, except for those explicitly specified as SQLNULLABLE, are not SQL-nullable; this condition also exists if no such specification is made for the group.

The DDL compiler outputs an SQL-nullable line item as a group with two elementary items in all of the supported host languages: Pascal (on D-series systems), C, COBOL, FORTRAN, pTAL, TACL, and TAL. The name of the group is derived from the name of the SQL-nullable line item. The names of the elementary items are *indicator* and *valu*. The data type of *indicator* is the data type within the specific language that corresponds to the DDL data type BINARY. The data type of *valu* is the language output for the data type specified in the SQL-nullable line item.

The value for the null indicator is usually determined at run time. If your application obtains Enscribe file layouts or SQL record schema from DDL, the recommended values for the indicator item are:

Value	Meaning
0	The value field contains meaningful data
-1	The data is null (not supplied)

The attributes SQLNULLABLE and NOT SQLNULLABLE applicable only to SQL, and are not output for any of the supported languages, which do not recognize the attributes in their syntax.

SQLNULLABLE or NOT SQLNULLABLE can be specified on a definition level, a group level, or an elementary line item.

SQLNULLABLE and NOT SQLNULLABLE cannot be specified concurrently on the same line item.

If the DDL clause NULL is specified for a line item, NOT SQLNULLABLE cannot be specified or implied for that item.

SQLNULLABLE and NOT SQLNULLABLE cannot be specified on a line item whose data type has been set by a previous definition, nor on a group or subgroup that contains such a line item.

An SQL-nullable line item is a word-aligned item regardless of its data type. An implicit filler of one byte is generated, when necessary, preceding the SQL-nullable line item.

The SQLNULLABLE or NOT SQLNULLABLE attribute is inheritable. That is, a line item that refers to a definition that is SQL-nullable becomes SQL-nullable as well; a line item that refers to a definition that is not SQL-nullable becomes itself not SQL-nullable.

A SQL-nullable line item can redefine another line item and can itself be redefined. If an SQL-nullable line item is redefined, the maximum byte size of the redefining line item is the specified size of the SQL-nullable line item plus 2.

If an EDIT-PIC clause or a literal string is specified on a null line item, the length of the string must be less than or equal to the specified size of the line item. Do not include the added numeric field as part of the available space for the string.

If an odd-byte length is specified on an SQL-nullable line item that has an OCCURS clause specified or implied, the internal total size of the line item is calculated by:

Occurrences specified * (a 2-byte numeric field + the byte length specified + a 1-byte padded filler)

Because an SQL-nullable line item is word-aligned, a filler is padded to align each repetition of an SQL-nullable line item with odd-byte length. A padded filler is not required for a repetition of an SQL-nullable line item specified with even-byte length.

The implicit filler emitted by The DDL compiler is generated explicitly in language outputs, but not for C or Pascal.

The maximum actual internal byte size of an SQL-nullable line item is 32,767 bytes in definitions; in records, it is:

File Type	Record's Maximum Length
Entry-sequenced	4,072 bytes
Key-sequenced	4,062 bytes
Relative	4,072 bytes
Unstructured	4,096 bytes

Because of this restriction, the maximum size that can be specified on an SQL-nullable line item is two bytes less than the numbers shown above. These two bytes are the indicator that shows whether the line item is null.

SQLNULLABLE cannot be specified for FILLER or BIT line items; these types of items can never be SQL-nullable. SQLNULLABLE cannot be specified on a group that contains a FILLER or BIT line item unless that line item is explicitly declared to be NOT SQLNULLABLE.

Neither SQLNULLABLE nor NOT SQLNULLABLE can be specified on an 88 condition-name line item or an 89 enumeration line item.

The dictionary fields that support SQL-nullable items are described in Appendix D, Dictionary Database Structure.

In Example 6-32 on page 6-42:

- Because DEF A has no specification regarding SQL-nullability, line items within the group are not SQL-nullable unless individually declared to be SQL-nullable.
- Because DEF B is specified as SQLNULLABLE, line items within that group are SQL-nullable unless individually not to be SQL-nullable.
- Semantically, A and B are equivalent.

Example 6-32. SQLNULLABLE Clause

```
DEF A.
  02 name PIC X(25).       ! Not nullable
  02 nickname PIC X(10) VALUE SQLNULL SQLNULLABLE.
  02 salary TYPE BINARY.   ! Not nullable
  02 hire-date TYPE DATE. ! Not nullable
END.

DEF B SQLNULLABLE.
  02 name PIC X(25).
  02 nickname PIC X(10) VALUE SQLNULL SQLNULLABLE.
  02 salary TYPE BINARY.
  02 hire-date TYPE DATE.
END.
```

Example 6-33. SQL-Nullable Output for C

DDL Type

```
DEF B SQLNULLABLE.

02 name     PIC X(25)
            NOT SQLNULLABLE.
02 nickname PIC X(10)
            VALUE SQLNULL
02 salary   TYPE BINARY
            NOT SQLNULLABLE.
02 hire-date TYPE SQL DATE
             NOT SQLNULLABLE.
END.
```

C Type

```
#pragma fieldalign shared2 __b
typedef struct __b
{
 char name[25];
 struct
 {
  short indicator;
  char  valu[10];
  } nickname;
  short    salary;
  char     hire_date[10];
 } b_def;
```

Example 6-34. SQL-Nullable Output for COBOL

DDL Type

```
DEF B SQLNULLABLE.
  02 name        PIC X(25)
                 NOT SQLNULLABLE.
  02 nickname    PIC X(10)
                 VALUE SQLNULL.
  02 salary      TYPE BINARY
                   NOT SQLNULLABLE.
  02 hire-date   TYPE SQL DATE
                   NOT SQLNULLABLE.
END.
```

COBOL Type

```
01 B.
  02 NAME        PIC X(25).
  02 FILLER      PIC X(1).
  02 NICKNAME.
    03 INDICATOR PIC S9(4) COMP.
    03 VALU      PIC X(10).
  02 SALARY      PIC S9(4) COMP.
  02 HIRE-DATE   PIC X(10).
```

Example 6-35. SQL-Nullable Output for FORTRAN

DDL Type

```
DEF B SQLNULLABLE.
 02 name         PIC X(25)
                 NOT SQLNULLABLE.
 02 nickname     PIC X(10)
                 VALUE SQLNULL.
 02 salary       TYPE BINARY
                 NOT SQLNULLABLE.
 02 hire-date    TYPE SQL DATE
                 NOT SQLNULLABLE.
END.
```

FORTRAN Type

```
RECORD B.
CHARACTER*25 NAME
FILLER*1
 RECORD NICKNAME.
   INTEGER*2 INDICATOR
   CHARACTER*10 VALU
 END RECORD
INTEGER*2 SALARY
CHARACTER*10 HIREDATE
END RECORD
```

Example 6-36. SQL-Nullable Output for Pascal (D-series Systems Only)

DDL Type

```
DEF B SQLNULLABLE
 02 name         PIC X(25)
                 NOT SQLNULLABLE.
 02 nickname     PIC X(10)
                 VALUE SQLNULL.
 02 salary       TYPE BINARY
                 NOT SQLNULLABLE.
 02 hire-date    TYPE SQL DATE
                 NOT SQLNULLABLE.
END.
```

Pascal Type

```
TYPE B_DEF = RECORD
 NAME        : FSTRING(25);
 NICKNAME    : RECORD
   INDICATOR : INT16;
   VALU      : FSTRING(10);
 END;
 SALARY      : INT16;
 HIRE_DATE   : FSTRING(10);
END;
```

Example 6-37. SQL-Nullable Output for pTAL or TAL

DDL Type

```
DEF B SQLNULLABLE.
 02 name         PIC X(25)
                 NOT SQLNULLABLE.
 02 nickname     PIC X(10)
                 VALUE SQLNULL.
 02 salary       TYPE BINARY
                 NOT SQLNULLABLE.
 02 hire-date    TYPE SQL DATE
                 NOT SQLNULLABLE.
END.
```

pTAL or TAL Type

```
STRUCT B^DEF (*) FIELDALIGN
(SHARED2);
 BEGIN
 STRUCT     NAME;
  BEGIN STRING BYTE[1:25]; END;
FILLER 1;
STRUCT     NICKNAME;
 BEGIN
 INT      INDICATOR;
 STRUCT   VALU;
  BEGIN STRING BYTE[1:10]; END;
 END;
INT         SALARY
STRUCT     HIRE^DATE;
 BEGIN STRING BYTE[1:10]; END;
END;
```

Example 6-38. SQL-Nullable Output for TACL

DDL Type

```
DEF B SQLNULLABLE.
 02 name         PIC X(25)
                 NOT SQLNULLABLE.
 02 nickname     PIC X(10)
                 VALUE SQLNULL.
 02 salary       TYPE BINARY
                 NOT SQLNULLABLE.
 02 hire-date    TYPE SQL DATE
                 NOT SQLNULLABLE.
END.
```

TACL Type

```
?Section B Struct
 Begin
STRUCT   NAME;
  BEGIN CHAR BYTE(0:24); END;
FILLER 1;
STRUCT   NICKNAME;
  Begin
  INT      INDICATOR;
  STRUCT   VALU;
    BEGIN CHAR BYTE(0:9); END;
  End;
INT        SALARY
STRUCT     HIRE^DATE;
  BEGIN CHAR BYTE(0:9); END;
End;
```

TACL

The TACL clause specifies the TACL data type to which a DDL data item is to be converted when generating TACL source code.

```
TACL type
```

type

is the TACL data type to which the DDL data type is to be converted.

```
{ CRTPID    }
{ DEVICE    }
{ ENUM      }
{ FNAME     }
{ FNAME32   }
{ PHANDLE   }
{ SSID      }
{ SUBVOL    }
{ TRANSID   }
{ TSTAMP    }
{ USERNAME  }
```

The TACL clause can be specified in a field or group DEFINITION statement.

The DDL compiler generates TACL data types only when a TACL command is specified; however, the DDL compiler checks that the length of the DDL data item matches the specified TACL data type whether the TACL command is specified or not. The DDL compiler issues an error message when the lengths do not match.

If the TACL clause is omitted, the DDL compiler translates the field or group to a TACL STRUCT that corresponds to the DDL data type. For a table showing the standard data-type translations for TACL, see Table C-5 on page C-9.

If a TACL data type is associated with a DDL data item defined with an OCCURS clause, each occurrence of the DDL data item must be the same length as the associated TACL data type.

The DDL compiler aligns on word boundaries all DDL data items associated with TACL data types.

If fields or groups associated with TACL data types are nested, all but the outermost TACL data type is ignored. Thus, if a TACL data type is specified for a group and a TACL data type is also specified for a field within the group, the DDL compiler uses only the group's TACL data type, ignoring the field's TACL data type.

If a DDL data item is defined by referring to an existing definition:

- When the referring definition does not include a TACL clause, it inherits any TACL data type specified in the referenced definition.
- When the referring definition includes a TACL clause, the specified TACL data type overrides any TACL data type specified in the referenced definition.
- The DDL field or group of fields must be the same length as the TACL high level data type. The length in bytes of each high-level TACL data type is shown in Table 6-6 on page 6-45.

Table 6-6. Lengths of TACL Data Types

TACL Type	Byte Length
CRTPID	8
DEVICE	8
ENUM	2
FNAME	24
FNAME32	32
PHANDLE	20
SSID	12
SUBVOL	16
TRANSID	8
TSTAMP	8
USERNAME	16

Example 6-39. TACL Clause

DEFINITION Statement With TACL Clause

```
?TACL
DEF term-id TYPE CHARACTER 8  TACL CRTPID.
```

Structure Generated for DEFINITION Statement with TACL Clause

```
?Section TERM^ID Struct
Begin
CRTPID   TERM^ID;
End;
```

DEFINITION Statement Without TACL Clause

```
?TACL
DEF term-id TYPE CHARACTER 8.
```

Structure Generated for DEFINITION Statement Without TACL Clause

```
?Section TERM^ID Struct
Begin
CHAR    BYTE (0:7)

End
```

A TACL clause at the group level overrides any TACL clauses specified for fields within the group.

Example 6-40. TACL Clause at Group Level

DDL Input

```
DEF fname-def TACL FNAME.
  02 volume   TYPE CHARACTER 8  TACL CRTPID.
  02 subvol   TYPE CHARACTER 8.
  02 file     TYPE CHARACTER 8.
END
```

DDL Output (TACL Code)

```
?Section TERM^ID Struct
Begin
CRTPID TERM^ID;  ! High-level TACL type: CRTPID
End;
```

If a definition or record is defined by referring to an existing definition and does not specify a TACL clause, the referring object assumes any TACL clause in the referenced definition. In Example 6-41 on page 6-47, the DEFINITION statement inherits the TACL data type specified for group FNAME-DEF in Example 6-40 on page 6-46.

Example 6-41. Inheriting TACL Clause From Referenced Definition

DDL Input

```
?TACL
DEF fname-2 TYPE fname-def.
```

DDL Output (TACL Code)

```
?Section FNAME^2 Struct
Begin
FNAME   FNAME^2;
End;
```

If the referring object is defined with a TACL clause, the referring TACL data type overrides any referenced TACL data type. In Example 6-42 on page 6-47, the TACL clause in the DEFINITION statement overrides the TACL clause in the definition of TERM-ID.

Example 6-42. Overriding Inheriting TACL Clause

DDL Input

```
?TACL
DEF trans-id TYPE term-id TACL TRANSID.
```

DDL Output (TACL Code)

```
?Section TRANS^ID Struct
Begin
TRANSID   TRANS^ID;
End;
```

TYPE

The TYPE clause specifies the data type and size of a data structure, either explicitly or by referring to a previously defined data structure.

```
TYPE { data-type | def-name  | * }
```

data-type

explicitly declares the data type of the data structure:

```
{ CHARACTER length                                          }
{                                                           }
{ BINARY { 8                    } [ UNSIGNED ]              }
{        { [ 16 [ , scale ] ]   }                            }
{        { 32 [ , scale ]       }                            }
{        { 64 [ , scale ]       }                            }
{                                                           }
{ FLOAT { [ 32 ] }                                          }
{       { 64     }                                          }
{                                                           }
{ COMPLEX                                                   }
{                                                           }
{ LOGICAL { 1      }                                        }
{         { [ 2 ]  }                                        }
{         { 4      }                                        }
{                                                           }
{ ENUM                                                      }
{                                                           }
{ SQL-data-type                                             }
{                                                           }
{ BIT bit-length [ UNSIGNED ] [ ENUM enum-name ]            }
```

CHARACTER *length*

represents a character string of *length* characters. The maximum values of *length* are:

Language	Maximum Value	Number of Characters
FORTRAN	255 bytes	255 ASCII characters or 127 national characters
TACL	5,000 bytes for an entire structure	5,000 ASCII characters or 2,500 national characters
COBOL pTAL TAL	Available address space, or, for part of a record, the record length	

```
BINARY { 8                    } [ UNSIGNED ]
       { [ 16 [ , scale ] ]   }
       { 32 [ , scale ]       }
       { 64 [ , scale ]       }
```

represents a two's complement binary number, whose size can be specified as 8, 16, 32, or 64 bits.

scale

is an integer that specifies the position of an implied decimal point.

Note. Use *scale* only for COBOL, pTAL, and TAL.

A scale of *n* multiplies the number by 10 to the power of *-n*; a scale of *-n* multiplies the number by 10 to the power of *n*. The value of *scale* depends on the BINARY item size:

BINARY Size	scale Size
16	-4 to 4
32	-9 to 9
64	-18 to 18 (for COBOL, -17 to 18)

UNSIGNED

declares an item of type BINARY as a positive binary integer.

BINARY Type	Lowest Value	Highest Value
16	-32,768	32,767
16 UNSIGNED	0	65,535
32	-2,147,483,648	2,147,483,647
32 UNSIGNED	0	4,294,967,295
64	-9,223,372,036,854,775,808	9,223,372,036,854,775,807
64 UNSIGNED*	0	18,446,744,073,709,551,615

* Use BINARY 64 UNSIGNED only for C, pTAL, and TAL—the DDL compiler issues an error message if you use BINARY 64 UNSIGNED for any other language.

```
FLOAT { [ 32 ] }
      { 64     }
```

represents a signed real number in binary scientific notation.

Note. Use FLOAT only for Pascal (on D-series systems), C, FORTRAN, pTAL, and TAL.

FLOAT Type	Length
32 (default)	4 bytes
64	8 bytes

COMPLEX

represents an 8-byte binary complex number.

Note. Use COMPLEX only for C and FORTRAN.

```
LOGICAL { 1      }
        { [ 2 ]  }
        { 4      }
```

represents a logical data type.

Note. Use LOGICAL only for Pascal (on D-series systems), C, FORTRAN, pTAL, and TAL.

The value of a logical data type is considered false if it is 0 and true if it is nonzero.

LOGICAL Type	Length
1*	1 byte
2 (default)	2 bytes
4	4 bytes

* Use LOGICAL 1 only for Pascal (on D-series systems) and C.

ENUM

represents an enumeration data type.

Note. Use ENUM only for Pascal (on D-series systems), C, FORTRAN, pTAL, and TAL.

SQL-*data-type*

is an SQL data type that DDL supports (see the *SQL/MP Reference Manual* and *SQL/MX Reference Manual*).

BIT *bit-length* [UNSIGNED] [ENUM *enum-name*]

represents the bit maps of the item.

bit-length

is an integer from 1 through 15 that specifies the size of the bit fields.

UNSIGNED

declares an item of type BIT as a positive number.

ENUM *enum-name*

specifies the enumeration definition that contains the values to use for the BIT item. The enumeration definition must be in the open dictionary, and the enumeration values in the enumeration definition must fit within the number of bits specified for the item, excluding any sign bit.

def-name

is the name of a previously defined data structure.

*

refers to a previously defined data structure that has the same name as the referring data structure.

In the TYPE clause, you can declare the type and size of an object, group, or field by one of these methods:

- Specifying TYPE data-type on page 6-51
- Specifying TYPE def-name on page 6-66
- Specifying TYPE * on page 6-67

Specifying TYPE data-type

data-type is one of:

- BINARY on page 6-52
- ENUM on page 6-53
- LOGICAL on page 6-54
- BIT on page 6-54
- The SQL data types in the *SQL/MP Reference Manual* or *SQL/MX Reference Manual*

Example 6-43. TYPE data-type Clauses

```
DEF type-clause-example.
  02 chr TYPE CHARACTER 8.             ! 8 alphanumeric characters
  02 bin-16 TYPE BINARY 16.            ! Signed integer
  02 bin-16-u TYPE BINARY 16 UNSIGNED. ! Signed integer
  02 bin-16-s TYPE BINARY 16,2  ! Signed integer, 2 decimal positions
  02 bin-32 TYPE BINARY 32.     ! Signed double integer
  02 bin-64 TYPE BINARY 64,16   ! Signed 4-word integer, 16 dec. positions
  02 flt TYPE FLOAT.            ! Signed 32-byte real number
  02 flt-64 TYPE FLOAT 64.      ! Signed 64-byte real number
  02 cmplx TYPE COMPLEX.        ! 8-byte complex binary number
  02 logicl TYPE LOGICAL.       ! 2-byte logical item
END
```

BINARY

- C

  BINARY 16, 32, and 64 data types are translated to types `short`, `long`, and `long long`, respectively. Any *`scale`* factor is ignored.

  For H06.03 and later RVUs, the HP C and HP C++ compilers support the data type `unsigned long long`, which corresponds to the DDL data type BINARY 64 UNSIGNED.

  The value of an item of type BINARY 64 UNSIGNED cannot be in octal form.

Example 6-44. C BINARY 64 and BINARY 64 UNSIGNED (H06.03 and Later RVUs)

DDL Code	C Code
`def def1 type binary 64 unsigned.`	`typedef unsigned long long def1_def;`

`def def2.`
`  02 f1 type binary 64`
`  02 f2 type binary 64 unsigned.`
`end.`
	`typedef struct __def2`
	`  {`
	`  long long f1;`
	`  unsigned long long f2;`
	`  } def2_def;`

`def def3 pic 9(10) comp.`	`typedef unsigned long long def3_def;`
`def def4 type binary 64,-18 unsigned.`	`typedef unsigned long long def4_def;`

- COBOL

  BINARY 16, 32, and 64 data types are translated to NATIVE-2, NATIVE-4, and NATIVE-8 data types, respectively.

  If *`scale`* is specified, the BINARY data type is translated to PIC [S]9(*n*)V9(*n*) COMP if the scale is positive, or to PIC [S]9(*n*)P(*n*) COMP if the scale is negative. The PICTURE clause includes *S* unless UNSIGNED is specified.

  UNSIGNED does not affect generation of COBOL code for a BINARY data type when *`scale`* is not specified.

- Pascal (D-series Systems Only)

  BINARY 16, 32, and 64 data types are translated to INT16, INT32, and INT64, respectively. Types BINARY 16 UNSIGNED and BINARY 32 UNSIGNED are translated to Pascal types CARDINAL and INT32, respectively. Pascal does not support scaling; any *`scale`* factor is ignored.

- TACL

  BINARY 16, 32, and 64 data types are translated to INT, INT2, and INT4, respectively. An UNSIGNED clause is ignored for TYPE BINARY 32, but is translated to UINT for TYPE BINARY 16. Any specified *`scale`* is ignored.

- pTAL or TAL

  For pTAL or TAL, the BINARY 16, 32, and 64 data types are translated to INT, INT(32), and FIXED data types, respectively. If UNSIGNED is specified, it is ignored in the TAL data type. Scale is ignored for BINARY 16 and BINARY 32; for BINARY 64, *scale* becomes the *fpoint* value of a FIXED data type.

  The value of an item of type BINARY 64 UNSIGNED cannot be in octal form.

Example 6-45. TAL BINARY 64 and BINARY 64 UNSIGNED

DDL Code	TAL Code
`def def1 type binary 64 unsigned.`	`FIXED DEF1;`

`def def2.`
`  02 f1 type binary 64`
`  02 f2 type binary 64 unsigned.`
`end.`
	`STRUCT DEF2^DEF (*) FIELDALIGN`
	`(SHARED2);`
	`BEGIN`
	`  FIXED F1;`
	`  FIXED F2;`
	`END;`

`def def3 pic S9(10) comp.`	`FIXED DEF3;`
`def def4 pic 9(10) comp.`	`FIXED DEF4;`
`def def5 type binary 64,-18 unsigned.`	`FIXED (-18) DEF5;`

ENUM

- C

  ENUM data type is translated to an enumeration type with the level-89 items included as literals in the C type.

- COBOL

  ENUM is translated to a NATIVE-2 item followed by level-88 items for the level-89 clauses.

- FORTRAN

  ENUM is unsupported, and the DDL compiler generates an INTEGER*2 followed by comments containing the level-89 items.

- Pascal (D-series Systems Only)

  ENUM is translated to constants followed by a type declaration of an INT16 item. When a type ENUM item is an elementary item of a group, the DDL compiler translates the ENUM item to an INT16 item and the level-89 clauses to constants preceding the group definition.

- TACL

  ENUM is translated to an ENUM with the level 89 items preceding the ENUM as TACL TEXT items.

- pTAL or TAL

  ENUM is translated to an INT with the level-89 items preceding the INT as literals. If you do not want to use an INT for a single-field definition, then use the NOTALALLOCATE command to generate the definition as a TAL DEFINE. For information about the NOTALALLOCATE command, see TALALLOCATE on page 9-108.

LOGICAL

For Pascal (on D-series systems), type LOGICAL 1 is translated to BOOLEAN. Types LOGICAL 2 and LOGICAL 4 are translated to INT16 and INT32, respectively.

BIT

A bit field inside a group structure that follows a nonbit field starts on a new 16-bit word. If you specify bit fields consecutively inside the group structure, the DDL compiler allocates the same 16-bit word for all contiguous bit fields that can fit in one 16-bit word. For the next bit field that cannot fit in the same 16-bit word, the DDL compiler allocates the next word.

Consecutive bit fields that occupy the same word have the same byte offset but different bit offset values in their records in the DICTOBL dictionary file.

A field that follows a bit field and has another data type starts on the next word.

A substructure containing only bit fields always starts and ends on a word boundary, padded with implicit bit fillers when necessary. Such a substructure is always an even number of bytes long, which is consistent with the way the C, Pascal (on D-series systems), and TAL compilers allocate spaces for structures containing bit maps.

TAL and Pascal support bit maps outside group structures; however, these bit maps are packed in pTAL or TAL and unpacked in Pascal. To ensure that bit maps outside group structures are compatible between languages, the DDL compiler generates 16-bit integer items for bit fields declared as field definitions, with warning messages in all language outputs except Pascal.

In languages that do not support bit maps, including COBOL, FORTRAN, and TACL, the DDL compiler generates a FILLER item for a bit map outside a group structure. The FILLER item has a number of words equivalent to the number of words required for such a bit field specified inside a group structure.

Note. A variable declared as a simple bit field can be a different size than an elementary item that is a bit field inside a structure (bit fields are packed within structures, but might or might not be in a simple bit field). Avoid variables of simple bit fields in COBOL, FORTRAN, or TACL (which do not support bit maps), or be certain you know what you are doing in handling such variables.

You can specify a bit map as a filler explicitly, the same way you specify a byte filler. Unlike a byte filler, a bit filler always starts at a new word if the bit filler follows a nonbit item.

When the definition of a group structure implies bit fillers, the DDL compiler generates the bit fillers implicitly, in the same way that the DDL compiler generates implicit byte fillers.

The SPI-NULL value for a bit field is 255 by default. Because all bit fields and bit fillers have the same SPI-NULL value, all bits are turned on in a byte containing bit items.

Bit fields that share the same byte must have the same product version number in a token map. The product version number applies to the entire byte. If a bit field extends across 2 bytes within a word, the product version number of that field applies to the entire word.

You specify product version numbers in the TOKEN-MAP statement

Example 6-46. Specifying Product Version Numbers

```
DEF bit-ddl-ex-a.
  02 bits-8          Type BIT 8.
  02 bits-3          Type BIT 3.
  02 bits-2          Type BIT 2.
  02 bits-10         Type BIT 10.
  02 bits-1          Type BIT 1.
  END.

TOKEN-MAP bit-map-ex-a VALUE 1 DEF bit-ddl-ex-a.
          VERSION "D20" FOR bits-8.
          VERSION "D30" FOR bits-3 THRU bits-2.
          VERSION "D40" FOR bits-10 THRU bits-1.
          END.
```

For information about product version numbers for bit fillers and more examples of specifying product versions, see the TOKEN-MAP on page 7-13.

Level-88 clauses following a bit map item are rejected by the DDL compiler, because level-88 clauses are meaningful only in COBOL, which does not support bit maps.

The DDL compiler does not generate language output for an ENUM clause specified with type BIT, because the DDL compiler emits the output for the enumeration when the clause is defined as type ENUM.

An OCCURS clause cannot apply to a bit map, because no compatible structure is available in the languages supported by DDL that have bit fields. C does not support arrays of bit fields. Pascal allocates one word for each signed bit field in an array and packs unsigned bit fields in an array. TAL supports only arrays of bit fields that are 1, 2, 4, or 8 bits long, packed inside an array.

A REDEFINES clause cannot apply directly to a bit map, because you cannot have an equivalent bit map item in TAL or a union of bit map items in C. A substructure containing bit maps can redefine another data item as long as such a REDEFINES clause follows DDL rules.

Bit fields in a record cannot be used as keys.

The DDL compiler generates:

Bit Maps for C

If bit-length is greater than 1, the output for field definition is SHORT or UNSIGNED SHORT. Group definition output is `short` *`fieldname:bit-length`* or `unsigned` *`fieldname:bit-length`*.

If *`bit-length`* is 1, the output for a bit map is `unsigned short`.

The DDL compiler does not generate C output for implicit bit fillers and therefore does not generate any C code for an implicit byte filler at the end of the substructure.

The output for an explicit byte filler is a bit field whose name is of the form filler_*n* ; that is, the output is the same as for other FILLER items. If any item at the same level as the bit filler has the same generated filler name, then the filler name ends with the next integer that does not cause the conflict. Do not try to access bit filler data items or reference the name of a bit filler.

Avoid defining level-89 clauses with the same name in different items. In C, two distinctive literals cannot have the same name, whether the literals are numeric constants or are in an enumeration item. When generating output for C, the DDL compiler does not check for level-89 clauses of the same name.

For a list of C data types that the TYPE *`data-type`* clause generates, see Table C-1, Sample DDL/C Data Translation Table, on page C-1.

Example 6-47. Bit Field Output for C

DDL Type	C Type
`DEF Bit-1 TYPE BIT 1`	`typedef unsigned short bit_1_def;`
`DEF New-Bit-1 TYPE Bit-1`	`typedef bit_1_def new_bit_1_def;`
`DEF Bit-10 TYPE BIT 10 UNSIGNED.`	`typedef unsigned short bit_10_def;`

`DEF Bit-Map.`
`  2 Bits-8 TYPE BIT 8.`
`  2 Bits-3 TYPE BIT 3 UNSIGNED.`
`  2 Bits-10 TYPE BIT 10.`
`End.`
	`#pragma fieldalign shared2 __bit_map`
	`typedef struct __bit_map`
	`  {`
	`  short bits_8:8;`
	`  unsigned short bits_3:3;`
	`  short bits_10:10;`
	`  } bit_map_def;`

`DEF Bit-Struct.`
`  2 Bits-0 TYPE Bit-1.`
`  2 Bits-1-To-10 TYPE Bit-10.`
`End.`
	`#pragma fieldalign shared2 __bit_struct`
	`typedef struct`
	`  {`
	`  unsigned short bits_0:1;`
	`  unsigned short bits_1_to_10:10;`
	`  } bit_struct_def;`

In Example 6-47 on page 6-57, a simple variable of type `bit_1_def` has a different size from the field `bits_0` in a variable having the type `bit_struct_def`.

Example 6-48. Bit Field Output for C (page 1 of 2)

DDL Type	C Type

`DEF Bit-Fillers`
 	
`  2 Field-1 TYPE CHARACTER 3.`
`  2 Filler  TYPE BIT 4.`
`  2 Bit-Field-1 TYPE BIT 5.`
`  2 Filler  TYPE BINARY.`
`  2 Field-2 TYPE BINARY 32.`
`  End.`
	`#pragma fieldalign shared2 __bit_fillers`
	`typedef struct`
	`  {`
	`  char field_1[3];`
	`  short filler_0:4;`
	`  short bit_field_1:5;`
	`  short filler_1[2];`
	`  long field_2;`
	`  } bit_fillers_def;`

`DEF Enum-Spec Begin TYPE ENUM.`
`  89 Val-1 Value 1.`
`  89 Val-2 Value 3.`
`  89 Val-3 Value 0.`
`  End.`
	`enum`
	`  {`
	`  val_1 = 1,`
	`  val_2 = 3,`
	`  val_3 = 0`

Example 6-48. Bit Field Output for C (page 2 of 2)

DDL Type

```
DEF Bits-With-Enums.

 02 Bit-Field-1 TYPE BIT 8
                ENUM Enum-Spec.
 02 Bit-Field-2 TYPE BIT 4.
 End.


DEF Reused-Bits.

 02 Data-Item Type Binary.
 02 Bits-Layout-1
          Redefines Data-Item.
    03 F-11 TYPE BIT 5.
    03 F-12 TYPE BIT 6.
    03 F-13 TYPE BIT 4.
 02 Bits-Layout-2
          Redefines Data-Item.
    03 F-21 TYPE BIT 4.
    03 F-22 TYPE BIT 3.
 End.
```

C Type

```
typedef short enum_spec_def;
#pragma fieldalign shared2
__bits_with_enums
typedef struct __bits_with_enums
 {
   short          bit_field_1:8;
   short          bit_field_2:4;
 } bits_with_enums_def;

#pragma fieldalign shared2 __reused_bits
typedef struct __reused_bits
 {
   union
     {
       short            data_item;
       struct
         {
           short       f_11:5;
           short       f_12:6;
           short       f_13:4;
           } bits_layout_1;
       struct
         {
           short       f_21:4;
           short       f_22:3;
         } bits_layout_2;
      } u_data_item;
 } reused_bits_def;
```

Bit Maps for COBOL

The output for a bit map declared as a field definition is NATIVE-2. the DDL compiler ignores *bit-length*.

The output for a bit map declared in a group structure is FILLER. Both named bit fields and filler bit fields have the same number of words as the bit map.

For a list of COBOL data types that the TYPE *data-type* clause generates, see Table C-2, Sample DDL/COBOL Data Translation Table, on page C-3.

Example 6-49. Bit Field Output for COBOL

DDL Type

```
DEF Bit-1 TYPE BIT 1.

DEF New-Bit-1 TYPE Bit-1.

DEF Bit-10 TYPE BIT 10 UNSIGNED.

DEF Bit-Map.
   2 Bits-8 TYPE BIT 8.
   2 Bits-3 TYPE BIT 3 UNSIGNED.
   2 Bits-10 TYPE BIT 10.
   End.

DEF Bit-Struct.
   2 Bits-0 TYPE Bit-1.
   2 Bits-1-To-10 TYPE Bit-10.
   End.

DEF Bit-Fillers.
   2 Field-1 Type Character 3.
   2 Filler  Type Bit 4.
   2 Bit-Field-1 Type Bit 5.
   2 Filler  Type Binary 16.
   2 Field-2 Type Binary 32.
   End.

DEF Enum-Spec Begin
         TYPE ENUM.
   89 Val-1 Value 1.
   89 Val-2 Value 3.
   89 Val-3 Value 0.
   End.

DEF Bits-With-Enums.
   02 Bit-Field-1 TYPE BIT 8
                  ENUM Enum-Spec.
   02 Bit-Field-2 TYPE BIT 4.
   End.

DEF Reused-Bits.
   02 Data-Item Type Binary.
   02 Bits-Layout-1
            Redefines Data-Item.
      03 F-11 TYPE BIT 5.
      03 F-12 TYPE BIT 6.
      03 F-13 TYPE BIT 4.
   02 Bits-Layout-2
            Redefines Data-Item.
      03 F-21 TYPE BIT 4.
      03 F-22 TYPE BIT 3.
   End.
```

COBOL Type

```
01 BIT-1 NATIVE-2.

01 NEW-BIT-1 NATIVE-2.

01 BIT-10 NATIVE-2.

01 BIT-MAP.
   02 FILLER NATIVE-2.
   02 FILLER NATIVE-2.

01 BIT-STRUCT.
   02 FILLER NATIVE-2.

01 BIT-FILLERS.
   02 FIELD-1 PIC X(3).
* the following filler is implicit
   02 FILLER PIC X(1).
* the following filler is bit maps
   02 FILLER NATIVE-2.
* the following filler is explicit
   02 FILLER NATIVE-2.
   02 FIELD-2 NATIVE-4.

01 ENUM-SPEC NATIVE-2.
   88 VAL-1 VALUE IS 1.
   88 VAL-2 VALUE IS 3.
   88 VAL-3 VALUE IS 0.

01 BITS-WITH-ENUMS.
   02 FILLER NATIVE-2.

01 REUSED-BITS.
   02 DATA-ITEM NATIVE-2.
   02 BITS-LAYOUT-1
      REDEFINES DATA-ITEM
      03 FILLER NATIVE-2.
   02 BITS-LAYOUT-2
      REDEFINES DATA-ITEM
      03 FILLER NATIVE-2.
```

Bit Maps for FORTRAN

The output for a bit map declared as a field definition is INTEGER*2. the DDL compiler ignores *bit-length*.

The output for a bit map declared in a group structure is FILLER. Both named bit fields and filler bit fields have the same number of words as the bit map.

For a list of FORTRAN data types that the TYPE *data-type* clause generates, see Table C-3, Sample DDL/FORTRAN Data Translation Table, on page C-5.

Example 6-50. Bit Field Output for FORTRAN (page 1 of 2)

DDL Type	FORTRAN Type
`DEF Bit-1 TYPE BIT 1`	`INTEGER*2 BIT1`
`DEF New-Bit-1 TYPE Bit-1.`	`INTEGER*2 NEWBIT1`
`DEF Bit-10 TYPE BIT 10 UNSIGNED.`	`INTEGER*2 BIT10`

`DEF Bit-Map.`
`  2 Bits-8 TYPE BIT 8.`
`  2 Bits-3 TYPE BIT 3 UNSIGNED.`
`  2 Bits-10 TYPE BIT 10.`
`  End.`
	`RECORD BITMAP`
	`  FILLER*2`
	`  FILLER*2`
	`END RECORD`

`DEF Bit-Struct.`
`  2 Bits-0 TYPE Bit-1.`
`  2 Bits-1-To-10 TYPE Bit-10.`
`  End.`
	`RECORD BITSTRUCT`
	`FILLER*2`
	`END RECORD`

`DEF Bit-Fillers.`
`  2 Field-1 Type Character 3.`
`  2 Filler  Type Bit 4.`
`  2 Bit-Field-1 Type Bit 5.`
`  2 Filler  Type Binary 16.`
`  2 Field-2 Type Binary 32.`
`  End.`
	`  RECORD BITFILLERS`
	`  CHARACTER*3 FIELD1`
	`C the following filler is implicit`
	`  FILLER*1`
	`C the following filler is bit maps`
	`  FILLER*2`
	`C the following filler is explicit`
	`  FILLER*2`
	`  INTEGER*4 FIELD2`
	`  END RECORD`

`DEF Enum-Spec Begin`
`  TYPE ENUM.`
`  89 Val-1 Value 1.`
`  89 Val-2 Value 3.`
`  89 Val-3 Value 0.`
`  End.`
	`INTEGER*2 ENUMSPEC`
	`C  VAL-1 = 1`
	`C  VAL-2 = 3`
	`C  VAL-3 = 0`

`DEF Bits-With-Enums.`
`  02 Bit-Field-1 TYPE BIT 8`
`  ENUM Enum-Spec.`
`  02 Bit-Field-2 TYPE BIT 4.`
`  End.`
	`RECORD BITSWITHENUMS`
	`  FILLER*2`
	`END RECORD`

Example 6-50. Bit Field Output for FORTRAN (page 2 of 2)

DDL Type

```
DEF Reused-Bits.
   02 Data-Item Type Binary.
   02 Bits-Layout-1
            Redefines Data-Item.
      03 F-11 TYPE BIT 5.
      03 F-12 TYPE BIT 6.
      03 F-13 TYPE BIT 4.
   02 Bits-Layout-2
            Redefines Data-Item.
      03 F-21 TYPE BIT 4.
      03 F-22 TYPE BIT 3.
   End.
```

FORTRAN Type

```
RECORD REUSEDBITS
  INTEGER*2 DATAITEM
  RECORD BITSLAYOUT1
    FILLER*2
  END RECORD
  EQUIVALENCE (BITSLAYOUT1,DATAITEM)
  RECORD BITSLAYOUT2
    FILLER*2
  END RECORD
  EQUIVALENCE (BITSLAYOUT2,DATAITEM)
```

Bit Maps for Pascal (D-series Systems Only)

The output for a bit map declared as a field definition is INT (*bit-length*) or UNSIGNED (*bit-length*). The Pascal compiler allocates the whole 16-bit word for the bit fields and treats the unused leading bits as bit fillers.

The output for a bit map declared in a group definition or record is INT (*bit-length*) or UNSIGNED (*bit-length*) inside a packed record.

The output for a bit filler is a bit field whose name is of the form FILLER_*n*; that is, the output is the same as for other FILLER items. If any item at the same level as the bit filler has the same generated filler name, then the filler name ends with the next integer that does not cause the conflict. Do not try to access bit filler data items or reference the name of a bit filler.

Avoid defining level-89 clauses with the same name in different items. In Pascal, two distinctive literals cannot have the same name, whether the literals are numeric constants or are in an enumeration item. When generating output for Pascal, the DDL compiler does not check for level-89 clauses of the same name.

For a list of Pascal data types that the TYPE *data-type* clause generates, see Table C-4, Sample DDL/Pascal Data Translation Table, on page C-7.

Example 6-51. Bit Field Output for Pascal

DDL Type	Pascal Type
`DEF Bit-1 TYPE BIT 1.`	`TYPE BIT_1_DEF = INT(1);`
`DEF New-Bit-1 TYPE Bit-1.`	`TYPE NEW_BIT_1_DEF = BIT_1_DEF;`
`DEF Bit-10 TYPE BIT 10 UNSIGNED.`	`TYPE BIT_10_DEF = UNSIGNED(10);`

`DEF Bit-Map.`
`  2 Bits-8 TYPE BIT 8.`
`  2 Bits-3 TYPE BIT 3 UNSIGNED.`
`  2 Bits-10 TYPE BIT 10.`
`  End.`
	`TYPE BIT_MAP_DEF = PACKED RECORD`
	`  BITS_8  : INT(8);`
	`  BITS_3  : UNSIGNED(3);`
	`  FILLER_1 : INT(5);`
	`  BITS_10  : INT(10);`
	`  FILLER_2 : INT(6);`
	`  END;`

`DEF Bit-Struct.`
`  2 Bits-0 TYPE Bit-1.`
`  2 Bits-1-To-10 TYPE Bit-10.`
`  End.`
	`TYPE BIT_STRUCT_DEF = PACKED RECORD`
	`  BITS_0  : BIT_1_DEF;`
	`  BITS_1_TO_10  : BIT_10_DEF;`
	`  FILLER_1  : INT(5);`
	`  END;`

`DEF Bit-Fillers.`
`  2 Field-1 Type Character 3.`
`  2 Filler  Type Bit 4.`
`  2 Bit-Field-1 Type Bit 5.`
`  2 Filler  Type Binary 16.`
`  2 Field-2 Type Binary 32.`
`  End.`
	`TYPE BIT_FILLERS_DEF = PACKED RECORD`
	`  FIELD_1  : FSTRING(3);`
	`  { the following is implicit }`
	`  FILLER_1  : CHAR;`
	`  FILLER_2  : INT(4);`
	`  BIT_FIELD_1  : INT(5);`
	`  { the following is implicit }`
	`  FILLER_3  : INT(7);`
	`  FILLER_4  : INT16;`
	`  FIELD_2  : INT32;`
	`  END;`

In Example 6-51 on page 6-62, a simple variable of type BIT_1_DEF has a different size from the field BITS_0 in a variable of the record type BIT_STRUCT_DEF.

Example 6-52. Bit Field Output for Pascal (page 1 of 2)

DDL Type	Pascal Type

`DEF Enum-Spec Begin`
`  TYPE ENUM.`
`  89 Val-1 Value 1.`
`  89 Val-2 Value 3.`
`  89 Val-3 Value 0.`
`  End.`
	`CONST VAL_1 = 1;`
	`CONST VAL_2 = 3;`
	`CONST VAL_3 = 0;`
	`TYPE ENUM_SPEC_DEF = INT16;`

`DEF Bits-With-Enums.`
`  02 Bit-Field-1 TYPE BIT 8`
`  ENUM Enum-Spec.`
`  02 Bit-Field-2 TYPE BIT 4.`
`  End.`
	`TYPE BITS_WITH_ENUMS_DEF = PACKED RECORD`
	`  BIT_FIELD_1  : INT(8);`
	`  BIT_FIELD_2  : INT(4);`
	`  FILLER_1  : INT(4);`
	`  END;`

Example 6-52. Bit Field Output for Pascal (page 2 of 2)

DDL Type

```
DEF Reused-Bits.
   02 Data-Item Type Binary.
   02 Bits-Layout-1
            Redefines Data-Item.
      03 F-11 TYPE BIT 5.
      03 F-12 TYPE BIT 6.
      03 F-13 TYPE BIT 4.
   02 Bits-Layout-2
            Redefines Data-Item.
      03 F-21 TYPE BIT 4.
      03 F-22 TYPE BIT 3.
   End.
```

Pascal Type

```
TYPE REUSED_BITS_DEF = RECORD
CASE INT16 OF
01: ( DATA_ITEM : INT16 );
02: ( BITS_LAYOUT_1 :
          PACKED RECORD
        F_11     : INT(5);
        F_12     : INT(6);
        F_13     : INT(4);
        FILLER_1 : INT(1);
      END );
03: ( BITS_LAYOUT_2 :
          PACKED RECORD
        F_21     : INT(4);
        F_22     : INT(3);
        FILLER_2 : INT(9);
      END );
END;
```

Bit Maps for TACL

The output for a bit map declared as a field definition is INT or UINT. The DDL compiler ignores *bit-length*.

The output for a bit map declared in a group structure is FILLER. Both named bit fields and filler bit fields have the same number of words as the bit map.

For a list of TACL data types that the TYPE *data-type* clause generates, see Table C-5, Sample DDL/TACL Data Translation Table, on page C-9.

Example 6-53. Bit Field Output for TACL (page 1 of 2)

DDL Type

```
DEF Bit-1 TYPE BIT 1.
```

TACL Type

```
?Section BIT^1 Struct
Begin
  INT   BIT^1;
End;
```

DDL Type

```
DEF New-Bit-1 TYPE Bit-1.
```

TACL Type

```
?Section NEW^BIT^1 Struct
Begin
  INT   NEW^BIT^1;
End;
```

DDL Type

```
DEF Bit-10 TYPE BIT 10 UNSIGNED.
```

TACL Type

```
?Section BIT^10 Struct
Begin
  UINT  BIT^10;
  End;
```

Example 6-53. Bit Field Output for TACL (page 2 of 2)

DDL Type

```
DEF Bit-Map.
   2 Bits-8 TYPE BIT 8.
   2 Bits-3 TYPE BIT 3 UNSIGNED.
   2 Bits-10 TYPE BIT 10.
   End.
```

```
DEF Bit-Struct.
   2 Bits-0 TYPE Bit-1.
   2 Bits-1-To-10 TYPE Bit-10.
   End.
```

```
DEF Bit-Fillers.
   2 Field-1 Type Character 3.
   2 Filler  Type Bit 4.
   2 Bit-Field-1 Type Bit 5.
   2 Filler  Type Binary 16.
   2 Field-2 Type Binary 32.
   End.
```

```
DEF Enum-Spec Begin
        TYPE ENUM.
   89 Val-1 Value 1.
   89 Val-2 Value 3.
   89 Val-3 Value 0.
   End.
```

```
DEF Bits-With-Enums.
   02 Bit-Field-1 TYPE BIT 8
                  ENUM Enum-Spec.
   02 Bit-Field-2 TYPE BIT 4.
   End.
```

```
DEF Reused-Bits.
   02 Data-Item Type Binary.
   02 Bits-Layout-1
            Redefines Data-Item.
      03 F-11 TYPE BIT 5.
      03 F-12 TYPE BIT 6.
      03 F-13 TYPE BIT 4.
   02 Bits-Layout-2
            Redefines Data-Item
      03 F-21 TYPE BIT 4.
      03 F-22 TYPE BIT 3.
   End.
```

TACL Type

```
?Section BIT^MAP Struct
Begin
  FILLER   2;
  FILLER   2;
  End;
```

```
?Section BIT^STRUCT Struct
Begin
  FILLER   2;
  End;
```

```
?Section BIT^FILLERS Struct
Begin
  STRUCT FIELD^1;
  BEGIN CHAR BYTE(0:2); END;
  FILLER   1;
  FILLER   2;
  FILLER   2;
  INT2     FIELD^2;
  End;
```

```
?Section VAL^1 Text
1
?Section VAL^2 Text
3
?Section VAL^3 Text
0
?Section ENUM^SPEC Struct
Begin
  ENUM     ENUM^SPEC;
  End;
```

```
?Section BITS^WITH^ENUMS Struct
Begin
  FILLER   2;
  End;
```

```
?Section REUSED^BITS Struct
Begin
  INT      DATA^ITEM;
  STRUCT BITS^LAYOUT^1
         REDEFINES DATA^ITEM;
     Begin
       FILLER   2;
       End;
  STRUCT BITS^LAYOUT^2
         REDEFINES DATA^ITEM;
     Begin
       FILLER   2;
       End;
End;
```

Bit Maps for pTAL and TAL

The output for a bit map declared as a field definition is INT. The DDL compiler ignores *bit-length*.

The output for a bit map declared in a group definition or a record is UNSIGNED (*bit-length*) in a STRUCT template.

The output for a bit filler is BIT_FILLER *bit-length* in a STRUCT template.

Avoid defining level-89 clauses with the same name in different items. In pTAL and TAL, two distinctive literals cannot have the same name, whether the literals are numeric constants or are in an enumeration item. When generating output for these languages, the DDL compiler does not check for level-89 clauses of the same name.

For a list of pTAL and TAL data types that the TYPE *data-type* clause generates, see Table C-6, Sample DDL/pTAL and TAL Data Translation Table, on page C-11.

Example 6-54. Bit Field Output for pTAL and TAL

DDL Type	pTAL or TAL Type
`DEF Bit-1 TYPE BIT 1.`	`INT BIT^1;`
`DEF New-Bit-1 TYPE Bit-1.`	`INT NEW^BIT^1;`
`DEF Bit-10 TYPE BIT 10 UNSIGNED.`	`INT BIT^10;`

`DEF Bit-Map.`
`  2 Bits-8 TYPE BIT 8.`
`  2 Bits-3 TYPE BIT 3 UNSIGNED.`
`  2 Bits-10 TYPE BIT 10.`
`  End.`
	`STRUCT BIT^MAP^DEF (*) FIELDALIGN (SHARED2);`
	`  BEGIN`
	`  UNSIGNED(8) BITS^8;`
	`  UNSIGNED(3) BITS^3;`
	`  BIT_FILLER 5;`
	`  UNSIGNED(10) BITS^10;`
	`  BIT_FILLER 6;`
	`  END;`

`DEF Bit-Struct.`
`  2 Bits-0 TYPE Bit-1.`
`  2 Bits-1-To-10 TYPE Bit-10.`
`  End.`
	`STRUCT BIT^STRUCT^DEF (*) FIELDALIGN(SHARED2);`
	`BEGIN`
	`  UNSIGNED(1) BITS^0;`
	`  UNSIGNED(10) BITS^1^TO^10;`
	`  BIT_FILLER 5;`
	`END;`

`DEF Bit-Fillers.`
`  2 Field-1 Type Character 3.`
`  2 Filler Type Bit 4.`
`  2 Bit-Field-1 Type Bit 5.`
`  2 Filler Type Binary 16.`
`  2 Field-2 Type Binary 32.`
`  End.`
	`STRUCT BIT^FILLERS^DEF (*) FIELDALIGN(SHARED2);`
	`BEGIN`
	`  STRUCT FIELD^1;`
	`  BEGIN STRING BYTE[1:3]; END;`
	`  FILLER 1;`
	`  BIT_FILLER 4;`
	`  UNSIGNED(5) BIT^FIELD^1;`
	`  BIT_FILLER 7;`
	`  FILLER 2;`
	`  INT(32) FIELD^2;`
	`END;`

In Example 6-54 on page 6-65, the simple variable BIT^1 has a different size from the field BITS^0 in a variable having the structure BIT^STRUCT^DEF.

Example 6-55. Bit Field Output for pTAL and TAL

DDL Type

```
DEF Enum-Spec Begin
         TYPE ENUM.
   89 Val-1 Value 1.
   89 Val-2 Value 3.
   89 Val-3 Value 0.
   End.

DEF Bits-With-Enums.
   02 Bit-Field-1 TYPE BIT 8
                  ENUM Enum-Spec.
   02 Bit-Field-2 TYPE BIT 4.
   End.

DEF Reused-Bits.
   02 Data-Item Type Binary.
   02 Bits-Layout-1
            Redefines Data-Item.
      03 F-11 TYPE BIT 5.
      03 F-12 TYPE BIT 6.
      03 F-13 TYPE BIT 4.
   02 Bits-Layout-2
            Redefines Data-Item.
      03 F-21 TYPE BIT 4.
      03 F-22 TYPE BIT 3.
   End.
```

pTAL or TAL Type

```
LITERAL VAL^1 = 1,
        VAL^2 = 3,
        VAL^3 = 0;
INT   ENUM^SPEC;

STRUCT BITS^WITH^ENUMS^DEF (*) FIELDALIGN
(SHARED2);
BEGIN
  UNSIGNED(8) BIT_FIELD_1;
  UNSIGNED(4) BIT_FIELD_2;
  BIT_FILLER  4;
END;

STRUCT REUSED^BITS^DEF (*) FIELDALIGN
(SHARED2);
BEGIN
  INT      DATA^ITEM;
  STRUCT   BITS^LAYOUT^1 = DATA^ITEM;
  BEGIN
    UNSIGNED(5)    F^11;
    UNSIGNED(6)    F^12;
    UNSIGNED(4)    F^13;
    BIT_FILLER     1;
    STRUCT   BITS^LAYOUT^2 = DATA^ITEM;
    BEGIN
      UNSIGNED(4)    F^21;
      UNSIGNED(3)    F^22;
      BIT_FILLER     9;
    END;
END;
```

Specifying TYPE def-name

Specifying TYPE *def-name* refers to an existing definition that has a different name from the object, group, or field that you are defining.

The DDL compiler reads *def-name* from the dictionary and then places the entire definition at the level of the referring data element.

The level number of a data element immediately following the TYPE *def-name* data element and in the same DEFINITION statement must be equal to or less than the level number of the TYPE *def-name* data element.

If you use HEADING, DISPLAY, or VALUE in a definition that refers to another definition, the new heading, display format, or initial value replaces the original heading, display format, or initial value–but only if the original value was at the outermost level in the referenced definition.

If you use NULL or UPSHIFT in a definition that refers to another definition, the referring definition inherits the null value and upshift attribute from the referenced definition.

You can use TYPE *def-name* to specify the length of a FILLER field. The FILLER field assumes the total length of the referenced definition.

If you add comments, the new comments replace the original comments in a definition referenced by another definition. CLISTOUTDETAIL lists the original comments along with the new ones. For more information, see the CLISTOUT on page 9-21.

Specifying TYPE *

Specifying TYPE * refers to an existing definition that has the same name as the object, group, or field that you are defining.

The DDL compiler reads the definition with the same name as the subject of the TYPE * clause from the dictionary and then places the entire referenced definition at the level of the referring object.

The level number of a data element immediately following the TYPE * element in the same DEFINITION statement must be equal to or less than the level number of the TYPE * data element.

Any HEADING, DISPLAY, NULL, or VALUE clauses in the referring data element override any equivalent clauses in the referenced data element–but only if the clauses are at the outermost level of the referenced definition.

If you add comments, the new comments replace the original comments in a definition referenced by another definition. CLISTOUTDETAIL lists the original comments along with the new ones. For more information, see the CLISTOUT on page 9-21.

Example 6-56. TYPE def-name and TYPE * Clauses (page 1 of 2)

Definitions in Dictionary

```
DEF name.
  02 last-name  PIC X(10).
  02 first-name PIC X(20).
END
DEF ordernum    PIC 9(3)                        HEADING
                                                "Order #".
DEF orddate     TYPE SQL DATETIME YEAR TO DAYHEADING
   "OrderDate".
DEF ordinterval TYPE SQL INTERVAL MONTH 2    HEADING "Order
    Interval".
DEF deldate     TYPE SQL DATE                HEADING"Delivery
    Date".
```

Example 6-56. TYPE def-name and TYPE * Clauses (page 2 of 2)

Definition That Refers to Dictionary Definitions

```
DEF orderinfo.
  02 employee   TYPE name.
  02 ordernum   TYPE *          HEADING
       "Order/Number".
  02 orderdt    TYPE orddate    HEADING "Order Date".
  02 orderint   TYPE ordintervalHEADING "Order
         Interval".
  02 delivdate  TYPE deldate    HEADING "Deliv Date".
  02 salesman   PIC 9(4)        HEADING "Salesman #".
  02 custnum    PIC 9(4)        HEADING "Customer #".
END
```

If the TYPE clauses in the `orderinfo` definition in Example 6-56 on page 6-67 were replaced by the structures they represent, the definition look like Example 6-57 on page 6-68.

Example 6-57. Equivalent to Example 6-56 on page 6-67

```
DEF  orderinfo.
  02 employee.
    03 last-name  PIC X(10).
    03 first-name PIC X(20).
  02 ordernum  PIC  9(3)                      HEADING "Order
                                                      Number".
  02 orderdt   TYPE SQL DATETIME YEAR TO DAY HEADING "Order
                                                      Date".
  02 orderint  TYPE SQL INTERVAL MONTH 2     HEADING "Order
                                                      Interval".
  02 delivdate TYPE SQL DATE                 HEADING "Deliv
                                                      Date".
  02 salesman  PIC 9(4)                      HEADING "Salesman
                                                      #".
  02 custnum   PIC 9(4)                      HEADING "Customer
                                                      #".
END
```

The definition `orderinfo`, referenced by TYPE *, keeps its name in the `orderinfo` record, but its implicit level-01 is changed to 02, and a new heading overrides its original heading.

New headings specified for `ordernum`, `orderdt`, and `delivdate` override the heading declared in `orddate` and `deldate`.

UPSHIFT

The UPSHIFT clause upshifts ASCII characters entered in the field.

```
UPSHIFT
```

Requesters generated by the Pathmaker product translate lowercase characters entered in this field to uppercase characters; user-written programs must be coded to enforce UPSHIFT.

You cannot use the UPSHIFT clause for numeric or computational fields. If a definition or description for such a field contains UPSHIFT, the DDL compiler sends an error message and does not enter the definition or record in the dictionary.

UPSHIFT can be associated only with elementary items in RECORD or DEFINITION statements.

A field can have a MUST BE clause and an UPSHIFT clause. If these clauses are used together, the MUST BE string must be upshifted.

A field can have both a VALUE clause and an UPSHIFT clause. If these clauses are used together, you must specify any alphabetic characters in the VALUE clause as uppercase.

An UPSHIFT clause cannot be specified in a definition or record that includes the REDEFINES clause.

If a definition refers to a definition that includes the UPSHIFT clause, the referring definition inherits the UPSHIFT attribute.

You cannot specify the upshift clause for a national data item.

Example 6-58. UPSHIFT Clause

```
DEF name.
  02 first      PIC X(20)
                UPSHIFT .
  02 middle     PIC X(15)
                UPSHIFT .
  02 last       PIC X(20)
                UPSHIFT .
END
```

USAGE

The USAGE clause either specifies computational storage allocation for a numeric group or field or identifies a COBOL as an index.

```
[ USAGE [ IS ] ] { COMP[UTATIONAL]    }
                 { INDEX              }
                 { COMP[UTATIONAL]-3  }
                 ( PACKED-DECIMAL     }
```

`COMP[UTATIONAL]`

specifies that the field or group is a numeric item that is to be stored as a computational value.

`INDEX`

specifies that a field is to be used as an index for COBOL only.

`COMP[UTATIONAL]-3`

specifies that the field or group is a numeric item that is stored in decimal form, but one digit takes one half-byte. The sign is stored separately as the rightmost half-byte, regardless of whether *S* is specified in the PICTURE declaration. See Example 6-59 on page 6-71.

`PACKED-DECIMAL`

Specifies that the field or group is a numeric item that is stored in decimal form, but one digit takes one-half byte. The sign is stored separately as the rightmost half byte, regardless of whether *S* is specified in the PICTURE declaration.

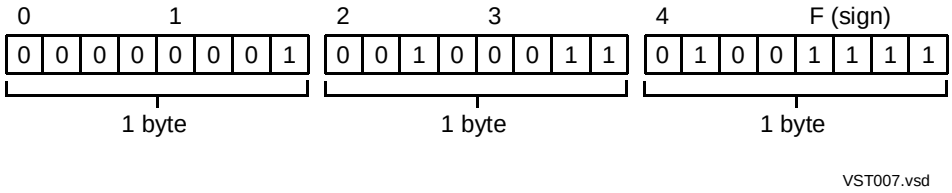
Note. Use PACKED-DECIMAL only for COBOL.

Example 6-59. USAGE COMPUTATIONAL Clause

```
DDL
!?DICT
!DEF EMP.
!02  F1  PIC 9999 PACKED-DECIMAL VALUE 1234.
!END.
!?COBOL
!OUTPUT *.

$ADE101 JYOTI 4> DDL
DDL Compiler T9100ABQ - (15NOV99)  SYSTEM \BOMBAY
COPYRIGHT TANDEM COMPUTERS INCORPORATED 1978, 1979, 1981, 1982, 1986-1999
!?DICT
 Audited dictionary created on subvol $ADE101.JYOTI.
 Dictionary opened on subvol $ADE101.JYOTI for update access.
!DEF EMP.
!02  F1 PIC 9999 PACKED-DECIMAL VALUE 1234.
!END.
 Definition EMP size is 3 bytes.
 Definition EMP added to dictionary.
!?COBOL
* SCHEMA PRODUCED DATE - TIME : 8/01/2000 - 11:20:29
 Output source for COBOL is opened on $ZTN1.#PTPJHYV
!OUTPUT *.
 Loading Definition EMP
?SECTION EMP,TANDEM
* Definition EMP created on 08/01/2000 at 11:20
01 EMP.
   02 F1                      PIC 9999       COMP-3
                                             VALUE 1234.
COBOL output produced for EMP.
```

For the PACKED-DECIMAL data type and a PICTURE 9999 declaration, the number +1234 is stored like this:



All fields declared as TYPE BINARY are COMPUTATIONAL items by default.

A field can be declared as COMPUTATIONAL if the associated PICTURE declaration is of the form:

```
PIC [ S ] 9 ... [ (length) ] [ V 9 ... [ (length) ] ]
```

The symbol 9 can occur a maximum of 18 times in a picture for an item declared as COMPUTATIONAL. If the symbol 9 occurs more than 10 times, the picture must include the symbol S.

When a group is declared as COMPUTATIONAL, each member of the group is also COMPUTATIONAL. All elements of the group must either be declared TYPE BINARY or have a picture compatible with TYPE BINARY. Reference definitions (TYPE * or TYPE *def-name*) are accepted if they refer to an element declared as COMPUTATIONAL or TYPE BINARY.

For TAL and FORTRAN source code, the DDL compiler translates the COMPUTATIONAL clause to the type and scale appropriate to the language. The data type for translation is based on the number of *9*s in the PICTURE:

Number of 9s	Type
1 to 4	BINARY 16
5 to 9	BINARY 32
10 to 18	BINARY 64

See TYPE on page 6-48 for the TAL and FORTRAN data types that correspond to the BINARY types.

For TACL source code, the DDL compiler translates COMP data types to binary data types corresponding to the data types generated for TAL, unless scale is specified; the DDL compiler ignores scale when generating TACL binary data types.

If the PICTURE of a COMPUTATIONAL item includes the symbol *V*, the DDL compiler calculates the appropriate *scale*.

For COBOL source code, translation is not needed unless the usage is computational by default; that is, the item is described as TYPE BINARY.

For C source code, the DDL compiler translates COMP data types to short, unsigned short, long, unsigned long, or double C data types.

For Pascal source code (on D-series systems), the DDL compiler translates COMP data types to INT16, CARDINAL, INT32, or INT64.

Appendix C, DDL Data Translation, has tables showing the host-language data types generated from the DDL COMP data types.

You can specify INDEX only for a field definition or a field description.

COBOL output for USAGE IS INDEX is the direct translation of the DDL source code, without generation of the storage specification or of any COBOL attributes supported by DDL for the field definition or description.

The DDL compiler verifies the size of the field against the target language before generating the COBOL output for the field. To match the COBOL storage allocation for index names, the field must be a 4-byte computational item.

You cannot specify INDEX for a noncomputational picture storage or character type.

A reference definition can refer to a field defined with INDEX, but the DDL compiler does not generate COBOL output for the USAGE IS INDEX clause from the reference definition.

You cannot specify a USAGE clause for a national data item.

A field can be declared as COMP-3 if the associated PICTURE declaration is of the form:

```
PIC   [S] 9.....[(length)]  [ v 9.....[(length)] ]
```

The symbol “9” can occur a maximum of 18 times in a PICTURE clause for an item declared as COMP-3.

When a group is declared as COMP-3, each member is also COMP-3. All elements of the group must have a PICTURE declaration compatible with COMP-3.

A COMP-3 item can refer to another item (TYPE * or TYPE *def-name*) provided the referenced item has been declared as COMP-3. If the referenced definition is a group item, then either the item itself or all of its constituent elements must have been declared as COMP-3.

The number of bytes required by a data item that has been declared with USAGE as COMP-3, COMPUTATIONAL-3, or PACKED-DECIMAL, depends upon the number of 9s specified in the picture clause of that item.

When a DDL item contains any PACKED-DECIMAL field (declared with a USAGE clause COMP-3, COMPUTATIONAL-3, or PACKED-DECIMAL value), then DDL only supports the generation of output for COBOL.

Although source code can be produced for computational items in each language, problems can occur when data is stored in such items. Consider a field described as:

```
PIC 9(7) COMP.
```

A COBOL program can enter only 7 digits in the field, but a TAL program can enter a much larger value in the INT(32) field generated from the description. Problems can occur if this larger value is accessed by the COBOL program.

You can avoid such problems in COBOL by using TYPE BINARY *n*, instead of PIC and COMP, in the DDL source code. A BINARY data type translates to a COBOL NATIVE-2, NATIVE-4, or NATIVE-8 data type.

Example 6-60 on page 6-74 shows the COBOL output that the DDL compiler generates for fields defined with USAGE IS INDEX. Error messages result when the size of the field definition or description does not match the storage allocation for index names in the target language.

Example 6-60. USAGE IS INDEX Output for COBOL

DDL	COBOL
`DEF abc TYPE BINARY 32` `  USAGE IS INDEX.`	`01 ABC USAGE IS INDEX.`
`DEF xyz TYPE BINARY` `  USAGE IS INDEX.`	`Invalid - ERROR`
`DEF tst TYPE abc.`	`01 TST NATIVE-4.`

`DEF grp.`
`  02 item-1 TYPE xyz.`
 	
`  02 item-2 PIC 9(10).`
`END.`
	`01 GRP.`
	`  02 ITEM-1 NATIVE-2.`
	
	`  02 ITEM-2 PIC 9(10).`

Example 6-61. USAGE IS PACKED-DECIMAL Output for COBOL

DDL	COBOL
`DEF def1 PIC 9(4) COMP-3.`	`01 def1 PIC 9(4) COMP-3.`
`DEF def2 PIC 9(4) COMPUTATIONAL-3.`	`01 def2 PIC 9(4) COMP-3.`
`DEF def3 PIC 9(4) PACKED-DECIMAL.`	`01 def3 PIC 9(4) COMP-3.`

`DEF grp1 COMP-3.`
`  02 fld1 PIC 99.`
`  02 fld2 PIC 99.`
`END.`
	`01 grp1 USAGE IS COMP-3`
	`  02 fld1 PIC 99.`
	`  02 fld2 PIC 99.`

`DEF grp2.`
`  02 g1  TYPE grp1.`
`END.`
	`01 grp2.`
	`  02 g1.`
	`  03 fld1 PIC 99 COMP-3.`
	`  03 fld2 PIC 99 COMP-3.`

`DEF grp3.`
`  02 h1  TYPE grp1 COMP-3`
`END.`
	`01 grp3.`
	`  02 h1 USAGE is COMP-3.`
	`  03 fld1 pic 99.`
	`  03 fld2 pic 99.`

When the DDL compiler receives a request for the items in Example 6-61 on page 6-74 in languages other than COBOL, the DDL compiler issues an error message and does not generate the source output.

VALUE

For DDL and some COBOL source code, the VALUE clause assigns an initial value to a field or group and the NOVALUE clause suppresses any VALUE clause in an item referenced by a TYPE clause.

For other languages and some COBOL source code:

Language	DDL compiler ...
C	Translates initial values to comments
COBOL	Translates initial values to comments if a value is declared for a data type that COBOL does not support
FORTRAN	Translates initial values to comments
Pascal (on D-series systems)	Ignores the VALUE clause
pTAL or TAL	Translates initial values to comments
TACL	Ignores the VALUE clause

```
{ VALUE [ IS ] value }
{ NOVALUE            }
```

value

is a literal value stored in the associated definition or record:

```
{ { "character-string" } [ LN clause ]... }
{ { constant-name      }                   }
{ { national-literal   }                   }
{ { number             }                   }
{                                          }
{ figurative-constant                      }
{ sql-datetime-literal                     }
{ sql-interval-literal                     }
{ symbolic-literal                         }
{ value-name                               }
```

character-string

is a string of ASCII characters.

constant-name

is the name of a constant in the open dictionary. The constant value must not be a figurative constant (see Table 6-4 on page 6-17) or symbolic literal (see Table 6-5 on page 6-17), and must be the same type as the associated data item.

national-literal

is a national literal whose length is consistent with the length specified in the PICTURE clause for the national data item.

number

is 1 or more digits (0 through 9), an optional leading plus or minus sign, and an optional decimal point.

LN-clause

specifies the locale name for *value* (see LN on page 6-13).

figurative-constant

is a figurative constant from Table 6-4 on page 6-17.

sql-datetime-literal

is a DATETIME, DATE, TIME, or TIMESTAMP value in ANSI, USA, or EUROPEAN format. For details, see the *SQL/MP Reference Manual* or *SQL/MX Reference Manual*.

sql-interval-literal

is a character string that conforms to the rules for a NonStop SQL/MP interval literal. For details, see the *SQL/MP Reference Manual* or *SQL/MX Reference Manual*.

symbolic-literal

is a symbolic literal from Table 6-5 on page 6-17. Use symbolic literals only for numeric items.

The DDL compiler replaces *symbolic-literal* with the appropriate literal for COBOL output.

value-name

is the *value-name* in the clause 89 Enumeration on page 6-84.

NOVALUE

suppresses any VALUE clauses in a definition referenced by another definition.

Note. Use NOVALUE only for a field or group defined with a TYPE clause.

An initial value must be compatible with the data type of the field or group for which it is declared.

An initial value declared at the group level must be alphanumeric.

A numeric value must be in the range of values specified by the receiving PICTURE string.

If used with MUST BE, an initial value must be in the range of values specified by the MUST BE string.

If a field is described as signed (the PICTURE clause includes the symbol *S*), you can include a sign in the numeric value. The sign must be leading in all cases, regardless of whether the PICTURE clause specifies a leading or trailing sign. An initial value cannot be:

- Used with a REDEFINES or OCCURS entry
- Composed of a null character string ("")

The only figurative constant that can be used to assign a value to a numeric data type is ZERO (or ZEROS or ZEROES).

A field can have both a VALUE clause and an UPSHIFT clause. If these clauses are used together, you must specify any alphabetic characters in the VALUE clause as uppercase.

If an initial value is specified at a group level, no other initial value can be specified within the group.

If a DEFINITION statement that includes an initial value is referenced by a statement that also includes an initial value, the DDL compiler overrides the referenced value with the value in the referring statement.

When you specify a constant name in a VALUE clause, the constant value must be a valid value for the data item and of the same type. A numeric constant can be used only with numeric-type data items, a string constant can be used only with character-type data items, and a national literal can be used only with national data items.

If you specify an initial value for a national data item, the value must be a national literal or a figurative constant.

The length of the national literal must agree with the length specified in the PICTURE clause for the national data item.

If you specify a *datetime-literal* or an *interval-literal* for a character field, the DDL compiler treats the literal as a regular character string. In such a case, the DDL compiler does not check the syntax and semantics of the string.

If you specify a *datetime-literal* or an *interval-literal* for a numeric field, the DDL compiler returns an error.

VALUE ZERO and VALUE ZEROES cannot be specified for some SQL data types (see the *SQL/MP Reference Manual* and *SQL/MX Reference Manual*).

SYSTEM is valid only for elementary fields of any type. If the type was previously defined, the definition must be a field definition.

When the VALUES clause is specified and the DDL compiler is generating source code for C, FORTRAN, pTAL, or TAL, the compiler translates any initial values to comments.

When the DDL compiler is generating Pascal (on D-series systems) or TACL source code, it ignores the VALUES clause.

For C and Pascal, a NOVALUES clause on a group definition has no effect on subgroups defined by reference to other groups.

SQLNULL is valid only for SQL-nullable elementary line items (that is, SQL items that are not specified as NOT NULL). If the type was previously defined, the definition must be an SQL-nullable field definition.

The values SYSTEM, CURRENT, and SQLNULL cannot be specified in a MUST BE clause, an 88 condition-name clause, or an 89 enumeration clause

If you specify a MUST BE clause and VALUE SYSTEM for the same item, the DDL compiler does not check the value SYSTEM against the specified MUST BE constraint. Similarly, if you specify a MUST BE clause and VALUE SQLNULL for the same item, the DDL compiler does not check the value SQLNULL against the specified MUST BE constraint.

Example 6-62. Assigning Initial Values With VALUE Clauses

```
DEF price        PIC 9(5)V99  VALUE IS ZERO.

DEF name                      VALUE SPACES.
  02 last        PIC X(20).
  02 first       PIC X(12).
  02 midinit     PIC X(2).
END
```

The initial values assigned in Example 6-62 on page 6-78 are overridden and suppressed by the statements in Example 6-63 on page 6-78.

Example 6-63. Overriding and Suppressing VALUE Clauses

```
DEF base-price TYPE price
  VALUE is 20.00.  ! Overrides initial value

DEF cust-name TYPE name
  NOVALUE.         ! Suppresses initial value
```

Example 6-64. Enumeration Values in VALUE Clauses

```
DEF prts-ddl-object-type TYPE ENUM BEGIN  AS "Miscellaneous".
   88 bolt               VALUE prts-enm-bolt.
   88 nut                VALUE prts-enm-nut.
   88 pin                VALUE prts-enm-pin.
   88 screw              VALUE prts-enm-screw.
   88 washer             VALUE prts-enm-washer.
   89 prts-enm-bolt      VALUE IS prts-obj-bolt AS "Bolt".
   89 prts-enm-nut       VALUE IS prts-obj-nut AS "Nut".
   89 prts-enm-pin       VALUE IS prts-obj-pin AS "Pin".
   89 prts-enm-screw     VALUE IS prts-obj-screw AS "Screw".
   89 prts-enm-washer    VALUE IS prts-obj-washer AS "Washer".
END.
```

Example 6-65. National-Literal Values in VALUE Clauses

```
DEF sample-type  PIC NN.

DEF language-info.
  02 language    TYPE sample-type    VALUE N"ab".
END.
```

Example 6-66. SQL-Literal Values in VALUE Clauses

```
DEF birthday TYPE SQL DATETIME year to day.

DEF family-birthday.
  02 father TYPE birthday                  VALUE "1945-12-12".
  02 mother TYPE birthday                  VALUE "1948-08-14".
  02 sister TYPE birthday                  VALUE "1980-01-13".
END.

DEF job-schedule.
  02 task1 TYPE SQL INTERVAL day 2             VALUE "12".
     ! An interval of 12 days
  02 task2 TYPE SQL INTERVAL minute TO second  VALUE "5:30".
     ! An interval of 5 minutes and 30 seconds
END.
```

66 RENAMES

Note. The DDL compiler ignores this clause when generating source code for languages other than DDL and COBOL.

For DDL and COBOL source code, the level-66 RENAMES clause renames a previously defined field or group or set of fields or groups.

For other languages:

Language	DDL compiler ...
C	Translates a field with a RENAMES clause to a comment
FORTRAN	Ignores the RENAMES clause
Pascal (on D-series systems)	Ignores the RENAMES clause
TACL	Ignores the RENAMES clause

```
66 renames-name RENAMES
   { field-name [ { THROUGH } field-name ] }
   {            [ { THRU    }            ] }
   {                                       }
   { group-name [ { THROUGH } group-name ] }
   {            [ { THRU    }            ] }
```

66

is the level number of the RENAMES clause.

renames-name

is a unique name.

field-name

is the name of a previously defined field in the dictionary. If *field-name* is not unique, qualify it with *group-name* and *def-name*.

group-name

is the name of a previously defined group in the dictionary. If *group-name* is not unique, qualify it with *group-name* and *def-name*.

A RENAMES clause does not redefine the characteristics of the field or group it renames; thus, no other clauses can be used with RENAMES.

If field and group names need to be qualified to make them unique, use the DDL (not COBOL) rules for qualifying names. For instance, to refer to the field STREET in the group ADDRESS in the definition EMPLOYEE, use:

```
employee.address.street         ! DDL qualification
```

Do not use:

```
street of address of employee   ! COBOL qualification
```

If the THROUGH option is used, the definition of the first named field or group must precede that of the second named field or group.

In Example 6-67 on page 6-80, ORDER-DETAIL renames the definition ODETAIL.

Example 6-67. RENAMES Clause

```
DEF odetail.
  02 primkey.
    03 ordernum       TYPE *.
    03 partnum        TYPE *.
  02 quantity         PIC 9(3).
  66 order-detail RENAMES  primkey THRU quantity.
END
```

88 Condition-Name

Note. The DDL compiler ignores this clause when generating source code for languages other than DDL and COBOL.

For DDL and COBOL source code, a level-88 condition-name clause associates a condition name with a value, list of values, or range of values, enabling you to refer to the value or values by the condition name.

```
88 condition-name { VALUE [ IS ]    }
                  { VALUES [ ARE ] }

  { value                      } [, value                      ]
  {                            } [                             ]
  { value { THROUGH } value    } [, value { THROUGH } value    ]
  {       { THRU    }          } [        { THRU    }          ] ...
```

`88`

is the level number of the condition-name clause.

condition-name

is a unique name.

value

```
{ { "character-string" } [ LN clause ] }
{ { constant-name      }                }
{ { national-literal   }                }
{ { number             }                }
{                                       }
{ figurative-constant                   }
{ sql-datetime-literal                  }
{ sql-interval-literal                  }
{ symbolic-literal                      }
{ value-name                            }
```

character-string

is a string of ASCII characters.

constant-name

is the name of a constant in the open dictionary. The constant value must not be a figurative constant (see Table 6-4 on page 6-17) or symbolic literal (see Table 6-5 on page 6-17), and must be a valid *condition-name* value.

national-literal

is a national literal whose length is consistent with the length specified in the PICTURE clause for the national data item.

number

is 1 or more digits (0 through 9), an optional plus or minus sign, and an optional decimal point.

LN-clause

specifies the locale name for *value* (see LN on page 6-13).

figurative-constant

is any figurative constant listed with the clause VALUE on page 6-75.

sql-datetime-literal

is a DATETIME, DATE, TIME, or TIMESTAMP value in ANSI, USA, or EUROPEAN format. For details, see the *SQL/MP Reference Manual* or *SQL/MX Reference Manual*.

sql-interval-literal

is a character string that conforms to the rules for a NonStop SQL/MP interval literal. For details, see the *SQL/MP Reference Manual* or *SQL/MX Reference Manual*.

symbolic-literal

is any symbolic literal listed with the clause VALUE on page 6-75.

value-name

is the *value-name* in the clause 89 Enumeration on page 6-84.

The syntax for a DDL level-88 clause differs from a COBOL level-88 clause only in its punctuation; DDL requires commas between values or sets of values, whereas COBOL does not.

The rules for the VALUE clause apply to the VALUE portion of a level-88 clause.

One or more condition-name clauses can follow the definition attribute clauses in a field definition or description. Condition-name clauses cannot directly follow a group definition or description.

Values of different condition names can overlap, so it is possible for several condition names to have the same value.

A single-field definition that has one or more level-88 clauses must also have BEGIN before the first period and END after the last clause.

In Example 6-68 on page 6-83, the values in ADDR-CODE are associated with condition names.

Example 6-68. Condition-Name Clauses

```
DEF cust-addr-cd.
  02 addr-code       TYPE BINARY 16.
    88 corp-hdq      VALUE 01.
    88 shipping      VALUE 02, 03.
    88 billing       VALUE 04 THRU 07.
    88 sales         VALUE 11 THRU 13, 15.
END
```

A COBOL program can use the construct in Example 6-68 on page 6-83 to determine the appropriate customer address: for example:

```
IF shipping PERFORM A00-send-ship-list.
```

Example 6-69. Condition-Name Values as Constants

```
CONSTANT corp-hdq  VALUE 01.
CONSTANT shipping1 VALUE 02.
CONSTANT shipping2 VALUE 03.
CONSTANT billing1  VALUE 04.
CONSTANT billing2  VALUE 05.
CONSTANT billing3  VALUE 06.
CONSTANT billing4  VALUE 07.
CONSTANT sales1    VALUE 11.
CONSTANT sales2    VALUE 12.
CONSTANT sales3    VALUE 13.
CONSTANT sales7    VALUE 17.

DEF cust-addr-cd.
  02 addr-code   TYPE BINARY 16.
    88 corp-hdq    VALUE corp-hdq.
    88 shipping    VALUE shipping1 THRU shipping2.
    88 billing     VALUE billing1 THRU billing4.
    88 sales       VALUE sales1 THRU sales3, sales7.
END
```

Example 6-70. Condition-Names as Enumeration Values

```
DEF prts-ddl-object-type TYPE ENUM BEGIN  AS "Miscellaneous".
   88 bolt               VALUE prts-enm-bolt.
   88 nut                VALUE prts-enm-nut.
   88 pin                VALUE prts-enm-pin.
   88 screw              VALUE prts-enm-screw.
   88 washer             VALUE prts-enm-washer.
   89 prts-enm-bolt      VALUE IS prts-obj-bolt AS "Bolt".
   89 prts-enm-nut       VALUE IS prts-obj-nut AS "Nut".
   89 prts-enm-pin       VALUE IS prts-obj-pin AS "Pin".
   89 prts-enm-screw     VALUE IS prts-obj-screw AS "Screw".
   89 prts-enm-washer    VALUE IS prts-obj-washer AS "Washer".
END.
```

89 Enumeration

In a field of type ENUM, a level-89 enumeration clause associates a name and (optionally) a display string with an enumeration value.

```
89 value-name [ VALUE value ] [ AS-clause ]
```

`89`

is the level number of the enumeration clause.

`value-name`

is a name that uniquely identifies the enumeration value.

`VALUE { value | constant-name }`

specifies a value to associate with *`value-name`*. You can specify *`value-name`* either as an integer or as the name of a constant in the open dictionary. The value of *`value-name`* must be an integer from -32,768 through 32,767.

Enumeration clauses for the same field cannot specify the same value.

Default values:

- For the first enumeration clause: zero
- For any subsequent enumeration clause: 1 more than the previous value

`AS-clause`

specifies a display string that represents the enumeration value (see AS on page 6-3).

Default display string: *`value-name`*

A single-field definition that has one or more level-89 enumeration clauses must also have BEGIN before the first period and END after the last clause.

One or more level-89 clauses can follow the definition attribute clauses in a field definition or description. Level-89 clauses cannot directly follow a group definition or description.

For C, the level-89 enumeration clauses for a field of type ENUM are translated to literals included in a C enumeration type. If the type of a single-field definition is ENUM, the DDL compiler generates a `typedef enum`. If the type of a field in a group definition is ENUM, the DDL compiler generates an `enum` embedded in a `typedef`:

```
enum
  {
     value-name1 = enumeration-value1,
     value-name2 = enumeration-value2,
     ...
  };
typedef short def-name_def;
```

If the type of a field in a group definition is ENUM, the DDL compiler generates a separate enumeration outside a `typedef struct`:

```
enum
  {
     value-name1 = enumeration-value1,
     value-name2 = enumeration-value2,
     ...
  };
typedef struct __group-name
  {
     char     first-element;
     short    enumeration-element
  } group-name_def;
```

Because the C compiler is case sensitive, the DDL compiler generates all lowercase letters for C source code.

For COBOL, the level-89 enumeration clauses for a field of type ENUM are translated to level-88 items. These items follow the source code for the ENUM field, a NATIVE-2 clause.

For FORTRAN, the level-89 enumeration clauses for a field of type ENUM are translated to comments. These comments follow the source code for the ENUM field, an INTEGER*2 type declaration.

For Pascal (on D-series systems), the level-89 enumeration clauses for a field of type ENUM are translated to constants. These constants precede the type declaration for the definition or record within the same section.

For pTAL or TAL, the level-89 enumeration clauses for a field of type ENUM are translated to LITERALs. If the type of a single-field definition is ENUM, and you do not specify NOTALALLOCATE, the DDL compiler generates LITERALs followed by an INT for the definition; for example:

```
LITERAL ENUMERATION-NAME1 = ENUMERATION-VALUE1,
        ENUMERATION-NAME2 = ENUMERATION-VALUE2,
           ...  ;
INT     DEF-NAME;
```

If the type of a field in a group definition is ENUM, the DDL compiler generates LITERALs followed by a STRUCT template.

For TACL, the level-89 enumeration clauses for a field of type ENUM are translated to SECTION directives of type TEXT followed by an ENUM for the item with which the level-89 clauses are associated:

```
?Section ENUMERATION-NAME1 Text
 ENUMERATION-VALUE1
?Section ENUMERATION-NAME2 Text
 ENUMERATION-VALUE2
    ...
?Section DEF-NAME Struct
 Begin
 ENUM  DEF-NAME;
 End;
```

Example 6-71. Enumeration Clause Output for C

DDL Type

```
DEF status TYPE ENUM BEGIN.

   89 no-error.
   89 read-error.
   89 write-error VALUE 6.
END.


DEF old-status TYPE status
               VALUE no-error.

DEF cpu.
   2 state TYPE ENUM.
      89 stop.
      89 pause.
      89 running.
END.


DEF system-state.


   2 cpu0 TYPE cpu.
   2 cpu1 TYPE cpu.
END.
```

C Type

```
#pragma section status
enum
      {
         no_error = 0,
         read_error = 1,
         write_error = 6
      };
typedef short status_def;

typedef status_def old_status_def;
/*value is no_error*/

#pragma section cpu
  enum
  {
     stop = 0,
     pause = 1,
     running = 2
  };
#pragma fieldalign shared2 __cpu
typedef struct __cpu
  {
     short  state;
  } cpu_def;

#pragma section system state
#pragma fieldalign shared2 __system_state
typedef struct __system_state
  {
    cpu_def     cpu0;
    cpu_def     cpu1;
  } system_state_def;
```

Example 6-72. Enumeration Clause Output for FORTRAN

DDL Type

```
DEF status TYPE ENUM BEGIN.
   89 no-error.
   89 read-error.
   89 write-error VALUE 3.
END.

DEF old-status TYPE status
               VALUE no-error.

DEF cpu.
   2 state TYPE ENUM.
     89 stop.
     89 pause.
     89 running.
END.

DEF system-state.
   2 cpu0 TYPE cpu.
   2 cpu1 TYPE cpu.
END.
```

FORTRAN Type

```
  INTEGER*2 STATUS
C  NO-ERROR = 0
C  READ-ERROR = 1
C  WRITE-ERROR = 3

  INTEGER*2 OLDSTATUS
C  Initial value is NO-ERROR

  RECORD CPU
    INTEGER*2 STATE
C     STOP = 0
C     PAUSE = 1
C     RUNNING = 2
  END RECORD

RECORD SYSTEMSTATE
  RECORD CPU0
    INTEGER*2 STATE
  END RECORD
  RECORD CPU1
    INTEGER*2 STATE
  END RECORD
END RECORD
```

Example 6-73. Enumeration Clause Output for Pascal (D-series Systems Only)

DDL Type

```
DEF status TYPE ENUM BEGIN.
   89 no-error.
   89 read-error VALUE 3.
   89 write-error.
END.

DEF old-status TYPE status
               VALUE no-error.

DEF cpu.
   2 state TYPE ENUM.
      89 stop.
      89 pause.
      89 running VALUE 4.
END.

DEF system-state.
   2 cpu0 TYPE cpu.
   2 cpu1 TYPE cpu.
END.
```

Pascal Type

```
CONST NO_ERROR = 0;
CONST READ_ERROR = 3;
CONST WRITE_ERROR = 4;
TYPE STATUS_DEF = INT16;

TYPE OLD_STATUS_DEF = STATUS_DEF;

CONST STOP = 0;
CONST PAUSE = 1;
CONST RUNNING = 4;
TYPE CPU_DEF = RECORD
  STATE : INT16;
END;

TYPE SYSTEM_STATE_DEF = RECORD
  CPU0  : CPU_DEF;
  CPU1  : CPU_DEF;
END;
```

Example 6-74. Enumeration Clause Output for TACL

DDL Type

```
DEF status TYPE ENUM BEGIN.
   89 no-error.
   89 read-error.
   89 write-error VALUE 6.
END.

DEF old-status TYPE status
               VALUE no-error.

DEF cpu.
   2 state TYPE ENUM.
      89 stop.
      89 pause.
      89 running.
END.

DEF system-state.
   2 cpu0 TYPE cpu.
   2 cpu1 TYPE cpu.
END.
```

TACL Type

```
?Section NO^ERROR Text
0
?Section READ^ERROR Text
1
?Section WRITE^ERROR Text
6
?Section STATUS STRUCT
Begin
ENUM  STATUS;
End;

?Section OLD^STATUS Struct
Begin
ENUM  OLD^STATUS;
End;

?Section STOP Text
0
?Section PAUSE Text
1
?Section RUNNING Text
2
?Section CPU Struct
Begin
ENUM  STATE;
End;

?Section SYSTEM^STATE Struct
Begin
STRUCT CPU0;
  Begin
  ENUM  STATE;
  End;
STRUCT CPU1;
  Begin
  ENUM  STATE;
  End;
End;
```

Example 6-75. Enumeration Clause Output for pTAL or TAL

DDL Type

```
DEF status TYPE ENUM BEGIN.
   89 no-error.
   89 read-error.
   89 write-error VALUE 6.
   89 status-error.
END.

DEF old-status TYPE status
               VALUE no-error.

DEF cpu.
   2 state TYPE ENUM.
      89 stop.
      89 pause.
      89 running.
END.

DEF system-state.
   2 cpu0 TYPE cpu.
   2 cpu1 TYPE cpu.
END.
```

pTAL or TAL Type

```
LITERAL NO^ERROR = 0,
        READ^ERROR = 1,
        WRITE^ERROR = 6,
        STATUS^ERROR = 7;
        INT     STATUS;

INT OLD^STATUS^DEF;
Value is NO^ERROR

LITERAL STOP = 0,
        PAUSE = 1,
        RUNNING = 2;
STRUCT CPU^DEF (*) FIELDALIGN (SHARED2);
   BEGIN
   INT   STATE;
   END;

STRUCT SYSTEM^STATE^DEF(*) FIELDALIGN
(SHARED2);
   BEGIN
     STRUCT   CPU0;
     END.
   BEGIN
     INT      STATE;
     END;
   STRUCT   CPU1;
   BEGIN
     INT      STATE;
     END;
   END;
```

7 SPI Tokens

SPI tokens are the smallest accessible units in an SPI message buffer. You can use token definitions provided by HP, and you can define your own tokens using DDL. HP supplies standard token definitions in C, COBOL, Pascal (on D-series systems), TACL, pTAL, and TAL. When you define your own tokens, you first define the tokens with the DDL statements described in this section and then generate token definitions in a host language, using the source output commands described in Section 9, DDL Compiler Commands.

You need the statements described in this section only if you plan to write your own subsystem using Subsystem Programmatic Interface (SPI) messages in a Distributed Systems Management (DSM) environment.

If you are writing a management application that communicates with HP subsystems using SPI messages, you use the token definitions supplied by HP. In such a case, this section can help you understand the DDL excerpts in the manuals that describe SPI programmatic interfaces.

This section describes the statements that define token types, token codes, and token maps. For information about building and using SPI messages, see the *Distributed Name Service (DNS) Management Programming Manual*.

Topics:

Defining SPI Tokens

An SPI token has two parts:

- An identifying code
- A token value

A token value is referenced by its token code rather than by its position in the buffer or by its address.

There are two forms of SPI tokens:

Token Form	Values	Defined By	Identifying Code
Simple	Single fields or fixed structures	Token type (which determines data type and size)	Token code
Extensible structured	Extensible (new fields can be added to the token in subsequent product versions to provide new features)	The standard token type that all extensible structured tokens have	Token map

Tokens are defined by these dictionary objects:

Object	Definition	Statement That Defines Object
Token type	Data type and size of one or more tokens	TOKEN-TYPE on page 7-2
Token code	Identifying code of a simple token	TOKEN-CODE on page 7-8
Token map	Identifying code of an extensible structured token	TOKEN-MAP on page 7-13

TOKEN-TYPE

The TOKEN-TYPE statement defines a token type and adds the definition to the open dictionary.

If a TOKEN-TYPE statement identifies a token type that already exists in the open dictionary and that is not referenced by another object, the DDL compiler replaces the existing token type with the new token type. If the existing token type is referenced by another object, the DDL compiler issues an error message and does not add the new token type to the dictionary.

If the appropriate source code files are open, the DDL compiler generates C, COBOL, Pascal (on D-series systems), pTAL, TACL, or TAL token-type structures when it executes the TOKEN-TYPE statement.

```
TOKEN-TYPE type-name

  VALUE [ IS ] token-data-type

  { DEF [ IS ] def-name [ OCCURS number TIMES ] }
  {                                             }
  { OCCURS { VARYING [ DEF [ IS ] def-name ] }  }
  {        { 0 TIMES                         }  }
```

type-name

is the name of a token type.

VALUE [IS] *token-data-type*

identifies a token type. You can specify *token-data-type* either as an integer or as the name of a constant in the open dictionary. The value of *token-data-type* must be a positive integer from 1 through 254 that SPI has defined as a token-data-type identifier. (HP supplies a set of predefined constants, defined in the file ZSPIDEF.ZSPIDDL.)

DEF [IS] *def-name* [OCCURS *number* TIMES]

defines the structure and length of the token by referring to an existing definition in the open dictionary. The total length of the token must be less than or equal to 254 bytes.

def-name

is the name of an existing definition in the open dictionary.

number

specifies the number of occurrences of the definition that defines the token. You can specify *number* either as a positive integer or as the name of a constant in the open dictionary. The value of the constant must be a positive integer.

Default: 1

```
OCCURS { VARYING [ DEF [ IS ] def-name ] }
       { 0 TIMES                         }
```

OCCURS VARYING [DEF [IS] *def-name*]

indicates that the length of the token varies. This clause sets the token length to its maximum of 255 bytes.

`DEF [ IS ] def-name`

documents the token structure, not altering the host-language output, but enabling SPI-buffer-display software to interpret the fields of tokens defined using the token type.

`OCCURS 0 TIMES`

indicates that the token length is 0 (there is no token value).

Every simple token in an SPI message has a token type to define its data type and length. You must specify the token type of a simple token in a TOKEN-TYPE statement.

You cannot specify the token type of an extensible structured token because SPI defines the token type of all extensible structured tokens as the DDL token type ZSPI-TYP-MAP.

The token data type specified in the VALUE clause of the TOKEN-TYPE statement must be a token data type that has been defined by SPI. If you specify the token data type as a number, this number must be the value of a predefined token-data-type constant. The token-data-type constants are defined in the file ZSPIDEF.ZSPIDDL. Always refer to one of these constants for the *token-data-type* value.

The length of a simple token is determined by one of:

- A definition referenced in a DEF IS clause, optionally repeated by an OCCURS *number* TIMES clause
- An OCCURS VARYING clause
- An OCCURS 0 TIMES clause

Topics:

- TOKEN-TYPE Statement Output on page 7-5
- Standard SPI TOKEN-TYPE Definitions on page 7-5

TOKEN-TYPE Statement Output

If you request C, COBOL, Pascal (on D-series systems), pTAL, TACL, or TAL source code, the DDL compiler generates compatible data structures for the requested language. The SPI routines can use these token-type data structures to identify a token and its type.

The data structures the DDL compiler generates for token types in each language are:

Language	Data Structure
C	`#define TYPE_NAME value`
COBOL	`01 type-name NATIVE-2 VALUE IS value`
Pascal (on D-series systems)	`CONST type-name = value`

TACL	`? Section type^name Struct`
	`BEGIN`
	`INT value`
	`END;`

pTAL or TAL	`Literal type^name = value`

The DDL compiler replaces any hyphen in a DDL name with a circumflex (^) in a TAL LITERAL name or in a TACL STRUCT name, and with an underscore (_) in a C `#define` name or in a Pascal defined constant name.

Standard SPI TOKEN-TYPE Definitions

SPI defines a set of standard token types. The names of standard SPI token types have the format:

`Zsss-TYP-name`

In this format, the letter *Z* indicates that the token type is defined by HP, *sss* is a subsystem name or is SPI for a standard SPI name, and *name* identifies the token type.

The file ZSPIDEF.ZSPIDDL on the disk volume chosen for your system contains the DDL definitions of the standard SPI token types. To use the standard SPI definitions, compile this file into your dictionary, using the DDL SOURCE command. For a complete description of the standard SPI token types, see the *SPI Programming Manual* and the *SPI Common Extensions Manual*.

You can use the standard token types where applicable, or you can define your own token types. When you define a token type specifically for your own subsystem, do not begin its name with the letter Z; this ensures that your token-type name will not be the same as a current or future name supplied by HP.

Example 7-1. Standard SPI Token Definition for Simple Token With 16-Bit Integer Values

```
TOKEN-TYPE zspi-typ-int  ! Token name
  VALUE IS zspi-tdt-int  ! Token data type
  DEF IS zspi-ddl-int.   ! Token definition
```

The definition `zspi-ddl-int` specifies the structure of all simple tokens of the token type `zspi-typ-int` in Example 7-1 on page 7-6:

```
DEF zspi-ddl-int    TYPE BINARY 16   SPI-NULL 0.
```

These definitions are in the file ZSPIDEF.ZSPIDDL.

Example 7-2 on page 7-6 defines two token types you might use for your own subsystem:

- The first token type, `assn-typ-status`, is defined by reference to the standard definition in Example 7-1 on page 7-6. This token type is identical to the token type `zspi-typ-int` except for its name. You can use `zspi-typ-int` instead of defining your own token type, but redefining a standard token type allows you to give it a name that is meaningful to your subsystem.
- The second token-type, `assn-typ-variable-token`, contains a varying number of two-word integers. The DEF IS clause is included for documentation only; it does not determine the token length of the token type. Because the token type is defined as variable-length with OCCURS VARYING, the token length is set to 255 by default. The token's structure is equivalent to this definition:

  ```
  DEF assn-ddl-variable-token.
    02  size        TYPE BINARY 16.
    02  data-table  TYPE BINARY 32
                    OCCURS 1 TO 100 TIMES DEPENDING ON size.
  END
  ```

Example 7-2. Possible Subsystem Token Types

```
TOKEN-TYPE assn-typ-status
  VALUE IS zspi-tdt-int
  DEF IS zspi-ddl-int.

TOKEN-TYPE assn-typ-variable-token
  VALUE IS zspi-tdt-int2        ! 2-word integer token data type
  OCCURS VARYING
  DEF IS assn-ddl-variable-token.  ! For documentation only
```

From the definitions in Example 7-2 on page 7-6, the DDL compiler generates the source code in Example 7-3 on page 7-7 through Example 7-7 on page 7-8.

Example 7-3. COBOL Source Code Generated for Example 7-2 on page 7-6

```
01 ZSPI-TYP-INT NATIVE-2 VALUE IS 514.
01 ASSN-TYP-STATUS NATIVE-2 VALUE IS 514.
01 ASSN-TYP-VARIABLE-TOKEN NATIVE-2 VALUE IS 1023.
```

The DDL compiler generates the token-type value in Example 7-3 on page 7-7 by left-shifting the token length in the second (low-order) byte and combining it with the token data type in the first (high-order) byte.

Example 7-4. TAL Source Code Generated for Example 7-2 on page 7-6

```
Literal ZSPI^TYP^INT = 2 '<<' 8 + 2;
Literal ASSN^TYP^STATUS = 2 '<<' 8 + 2;
Literal ASSN^TYP^VARIABLE^TOKEN = 3 '<<' 8 + 255;
```

The generated values in Example 7-4 on page 7-7 are identical to the values generated for COBOL or TACL source-code output from the same TOKEN-TYPE statements.

Example 7-5. TACL Source Code Generated for Example 7-2 on page 7-6

```
?Section ZSPI^TYP^INT Struct
BEGIN
UINT     TOKEN^TYPE  VALUE 514;
END;

?Section ASSN^TYP^STATUS Struct
BEGIN
UINT     TOKEN^TYPE  VALUE 514;
END;

?Section ASSN^TYP^VARIABLE^TOKEN Struct
BEGIN
UINT     TOKEN^TYPE  VALUE 1023;
END;
```

The generated values in Example 7-5 on page 7-7 are identical to the values generated for pTAL, TAL, or COBOL source-code output from the same TOKEN-TYPE statements.

Example 7-6. C Source Code Generated for Example 7-2 on page 7-6

```
#pragma section zspi_typ_int
#define ZSPI_TYP_INT 514U

#pragma section assn_typ_status
#define ASSN_TYP_STATUS 514U

#pragma section assn_typ_variable_token
#define ASSN_TYP_VARIABLE_TOKEN 1023U
```

The generated values in Example 7-6 on page 7-7 are identical to the values generated for COBOL, TAL, or TACL source-code output from the same TOKEN-TYPE statements.

Example 7-7. Pascal Source Code Generated for Example 7-2 on page 7-6

```
?Section ZSPI_TYP_INT
CONST ZSPI_TYP_INT =  514;

?Section ASSN_TYP_STATUS
CONST ASSN_TYP_STATUS =  514;

?Section ASSN_TYP_VARIABLE_TOKEN
CONST ASSN_TYP_VARIABLE_TOKEN =  1023;
```

The generated values in Example 7-7 on page 7-8 are identical to the values generated for COBOL, pTAL, TAL, TACL, or C source-code output from the same TOKEN-TYPE statements.

TOKEN-CODE

The TOKEN-CODE statement defines a token code for a particular simple token and stores the definition in the open dictionary.

If the TOKEN-CODE statement identifies a token code that already exists in the open dictionary, the DDL compiler replaces the existing token code with the new token code.

If the appropriate source code files are open, the DDL compiler generates C, COBOL, Pascal (on D-series systems), pTAL, TACL, or TAL token-code structures when it executes the TOKEN-CODE statement.

```
TOKEN-CODE token-name

  VALUE [ IS ] token-number

  TOKEN-TYPE [ IS ] type-name

  [ SSID subsystem-id ]

  [ HEADING label ]

  [ DISPLAY display-format ]
```

token-name

is the name of a simple token.

VALUE [IS] *token-number*

identifies a simple token. You can specify *token-number* either as an integer or as the name of a constant in the open dictionary. The value of *token-number* must be an integer in the range -32768 through 32767.

For subsystems that you write, *token-number* must be in the range from 1 through 9998. Numbers outside this range are reserved by HP or are previously defined by SPI.

type-name

is the name of a token type in the open dictionary.

SSID *subsystem-id*

identifies the subsystem to which the token belongs. You can specify *subsystem-id* either as an ASCII character string (enclosed in quotation marks) or as the name of a constant in the open dictionary. The value of *subsystem-id* must conform to the valid external format for a subsystem ID, which consists of 1 to 8 alphanumeric characters and hyphens specifying the subsystem owner, a period, a subsystem number or name, another period, and a product version number; for example:

```
"TANDEM.43.1245"
"TANDEM.XYZ.0"
```

If *subsystem-id* is invalid, the DDL compiler rejects the token.

If you omit the SSID clause, DSM Template Services does not keep track of the information in the TOKEN-CODE statement's HEADING and DISPLAY clauses.

HEADING *label*

specifies a label that identifies the token in DSM Template Services. DSM Template Services uses only the first 40 characters of the heading. You can specify *label* either as an ASCII character string (enclosed in quotation marks) or as the name of a constant in the open dictionary. The value of *label* must be an ASCII character string.

Default label: *token-name*

DISPLAY *display-format*

specifies the display format for the token. You can specify *display-format* either as an ASCII character string (enclosed in quotation marks) or as the name of a constant in the open dictionary. The value of *display-format* must be a format code described in the *DSM Template Services Manual*.

Default display format: display format for the data type specified in the definition

A token code is a 2-word structure that consists of a token type defined in a prior TOKEN-TYPE statement and the token number specified in the TOKEN-CODE statement.

Every token code is implicitly or explicitly qualified by an SPI subsystem ID. Two tokens of the same token type but qualified by different subsystem IDs can have identical token numbers and still be differentiated by SPI.

Within a subsystem, tokens must be differentiated by their token numbers.

Topics:

TOKEN-CODE Statement Output

If you request C, COBOL, Pascal (on D-series systems), pTAL, TACL, or TAL source code, the DDL compiler generates compatible data structures for the requested language. The SPI routines use these token-code data structures to identify a token and its data type.

The data structures the DDL compiler generates for token codes in each language are:

Language	Data Structure
C	#define TOKEN_NAME *value*
COBOL	01 *token-name* NATIVE-4 VALUE IS *value*
Pascal (on D-series systems)	CONST *token_name = value*;

TACL	?Section *token^name* Struct
	BEGIN
	INT2 TOKEN^CODE VALUE *value*;
	END;

pTAL or TAL	Literal *token^name* = *value*;

The DDL compiler replaces any hyphen in a DDL name with a circumflex (^) in a pTAL or TAL LITERAL name or in a TACL STRUCT name, and with an underscore (_) in a C #define name or a Pascal defined constant name.

Standard SPI TOKEN-CODE Definitions

SPI supplies a set of standard token codes to satisfy needs that are common to most programmatic interfaces. The standard token-code names have the format:

Z*sss*-TKN-*name*

In this format, the letter *Z* indicates that the token code is defined by HP, *sss* is a subsystem name or a standard SPI name, and *name* is the token name.

The file ZSPIDEF.ZSPIDDL on the disk volume chosen for your system contains the DDL definitions of the standard token codes. To use the standard SPI definitions, compile this file into your dictionary, using the DDL SOURCE command. For a complete description of the standard SPI token codes, see the *SPI Programming Manual* and the *SPI Common Extensions Manual*.

Note. When you define a token code specifically for your own subsystem, do not begin its name with the letter Z; this ensures that your token-code name is not the same as any current or future name supplied by HP.

Example 7-8. Definition of Standard Return Token

```
TOKEN-CODE zspi-tkn-retcode   VALUE IS 0
                              TOKEN-TYPE IS zspi-typ-enum.
```

The definition of ZSPI-TKN-RETCODE and the standard token-type definition to which it refers, ZSPI-TYP-ENUM, are in the file ZSPIDEF.ZSPIDDL.

When writing your own subsystem, you often need to define your own token codes. For example, you might need tokens to pass status information to and from your subsystem. Example 7-9 on page 7-11 shows the TOKEN-CODE statements to define two such token codes and the TOKEN-TYPE statement to define their token type.

Example 7-9. Possible Subsystem Token Codes

```
TOKEN-TYPE assn-typ-status     VALUE IS zspi-tdt-enum
                               DEF   IS zspi-ddl-enum.

TOKEN-CODE assn-tkn-my-status  VALUE IS 101
                               TOKEN-TYPE IS assn-typ-status.

TOKEN-CODE assn-tkn-stat-reply VALUE IS 102
                               TOKEN-TYPE IS assn-typ-status.
```

In Example 7-9 on page 7-11, the token type is the same in both TOKEN-CODE statements. Any number of tokens can be of the same token type.

From the definitions in Example 7-9 on page 7-11, the DDL compiler generates the source code in Example 7-10 on page 7-11 through Example 7-14 on page 7-12.

Example 7-10. COBOL Source Code Generated for Example 7-9 on page 7-11

```
01 ASSN-TKN-MY-STATUS     NATIVE-4    VALUE IS 184680549.
01 ASSN-TKN-STAT-REPLY    NATIVE-4    VALUE IS 184680550.
```

The DDL compiler generates the value of the token code in Example 7-10 on page 7-11 from the values specified for the token data type in the referenced TOKEN-TYPE statement and for the token number in the VALUE clause of the TOKEN-CODE statement—the two token codes differ only in their token numbers. The DDL compiler performs an unsigned left-shift on each of these values to generate the single COBOL NATIVE-4 value shown in the example.

Example 7-11. TAL Source Code Generated for Example 7-9 on page 7-11

```
Literal ASSN^TKN^MY^STATUS  = 11D '<<' 24 + 2D '<<' 16 + 101D;
Literal ASSN^TKN^STAT^REPLY = 11D '<<' 24 + 2D '<<' 16 + 102D;
```

The value of the pTAL or TAL representation of the token code is in Example 7-11 on page 7-12 identical to the value generated for COBOL source-code output from the same TOKEN-CODE statement.

Example 7-12. TACL Source Code Generated for Example 7-9 on page 7-11

```
?Section ASSN^TKN^MY^STATUS Struct
BEGIN
  INT2    TOKEN^CODE VALUE 184680549;
END

?Section ASSN^TKN^STAT^REPLY Struct
BEGIN
  INT2    TOKEN^CODE VALUE 184680550;
END
```

The generated value in Example 7-12 on page 7-12 is identical to the value of a pTAL or TAL literal or a COBOL data item generated from the same TOKEN-CODE statement.

Example 7-13. C Source Code Generated for Example 7-9 on page 7-11

```
#pragma section assn_tkn_my_status
#define ASSN_TKN_MY_STATUS 184680549LU

#pragma section assn_tkn_stat_reply
#define ASSN_TKN_STAT_REPLY 184680550LU
```

The value of the TOKEN-CODE statement generated in Example 7-13 on page 7-12 is the same as that generated for COBOL, pTAL, TAL, or TACL.

Example 7-14. Pascal Source Code Generated for Example 7-9 on page 7-11

```
?Section ASSN_TKN_MY_STATUS
CONST ASSN_TKN_MY_STATUS =  184680549;

?Section ASSN_TKN_STAT_REPLY
CONST ASSN_TKN_STAT_REPLY =  184680550;
```

The value of the TOKEN-CODE statement generated in Example 7-14 on page 7-12 is the same as that generated for the other host languages.

TOKEN-MAP

The TOKEN-MAP statement defines a token map and stores the definition in the open dictionary.

If a TOKEN-MAP statement identifies a token map that already exists in the open dictionary, the DDL compiler replaces the existing token map with the new token map.

If the appropriate source code file is open, the DDL compiler generates C, COBOL, Pascal (on D-series systems), pTAL, TACL, or TAL token-map structures when it executes the TOKEN-MAP statement.

```
TOKEN-MAP map-name

  VALUE [ IS ] token-number

  DEF [ IS ] def-name

  [ SSID subsystem-id ]

  [ HEADING label ]

  { { VERSION { number        } }                        }
  { {         { "Lnn"         } }                        }
  { {         { constant-name } }                        }
  { {                          }                         }
  { { NONVERSION               }                         }
  {                                                      }
  {     FOR { field-name [ { THROUGH } field-name ] } }  }
  {         {              [ {   THRU  }            ] } }  }
  {         {                                         } }  }
  {         { group-name [ { THROUGH } group-name ] } }  }
  {         {              [ { THRU    }            ] }. } ...

  END [ . ]
```

map-name

is the name of an extensible structured token.

VALUE [IS] *token-number*

identifies a simple token. You can specify *token-number* either as an integer or as the name of a constant in the open dictionary. The value of *token-number* must be an integer in the range -32768 through 32767.

For subsystems that you write, *token-number* must be in the range from 1 through 9998. Numbers outside this range are reserved by HP or are previously defined by SPI.

def-name

specifies the definition (in the open dictionary) that defines the fields in the extensible structured token.

SSID *subsystem-id*

identifies the subsystem to which the token belongs. You can specify *subsystem-id* either as an ASCII character string (enclosed in quotation marks) or as the name of a constant in the open dictionary. The value of *subsystem-id* must conform to the valid external format for a subsystem ID, which consists of 1 to 8 alphanumeric characters and hyphens specifying the subsystem owner, a period, a subsystem number or name, another period, and a product version number; for example:

```
"TANDEM.43.1245"
"TANDEM.XYZ.0"
```

If *subsystem-id* is invalid, the DDL compiler rejects the token.

If you omit the SSID clause, DSM Template Services does not keep track of the information in the TOKEN-CODE statement's HEADING and DISPLAY clauses.

HEADING *label*

specifies a label that identifies the token in DSM Template Services. DSM Template Services uses only the first 40 characters of the heading. You can specify *label* either as an ASCII character string (enclosed in quotation marks) or as the name of a constant in the open dictionary. The value of *label* must be an ASCII character string.

Default label: *token-name*

```
{ VERSION   { number        } }
{           { "Lnn "        } }
{           { constant-name } }
{                             }
{ NONVERSION                  }
```

specifies whether or not a field or group of fields in the definition is associated with a product version number. Every elementary field must be defined with either a VERSION or a NOVERSION clause. If VERSION or NOVERSION is specified for a group, the clause applies to each field within that group. You can specify only one product version number for a field.

number

is an integer in the range 1 through 65,535.

Lnn

is a product version string.

L

is a letter. The DDL compiler treats *L* as uppercase whether you specify it as uppercase or lowercase.

nn

is a two-digit number.

constant-name

is the name of a constant in the open dictionary. The constant name must be a valid *number* or *Lnn* value.

```
FOR { field-name [ { THROUGH } field-name ] } }
    {              [ {   THRU  }             ] } }
    {                                          } }
    { group-name [ { THROUGH } group-name ]    } }
    {              [ { THRU    }             ] }.} ...
```

specifies one or more fields or groups within the definition identified by *def-name*.

field-name

is the name of a field within the definition.

group-name

is the name of a group within the definition.

A token map is a special type of token code used to identify an extensible structured token to which new fields can be added in subsequent product versions. You identify a token map by its *token-name*.

You do not specify the token type of a token map. The token type of every token map, ZSPI-TYP-MAP, is defined by SPI; it consists of the token data type ZSPI-TDT-MAP and a token length of 255.

You define the structure of the extensible structured token by referring to an existing definition, *def-name*, in the TOKEN-MAP statement.

You must specify a VERSION or NOVERSION clause in the TOKEN-MAP statement for every field or group of fields in the referenced definition.

- The product version number in a VERSION clause specifies the subsystem product version in which the field or group of fields was introduced.
- A NOVERSION clause is used for fields whose presence is indicated by the value of another field in the structure–an *is-present* field.

When VERSION or NOVERSION is specified for a group:

- Every field in the group inherits the product version specified for the group.
- No field within the group can have a VERSION or NOVERSION clause.

If you specify a VERSION or NOVERSION clause for a range of fields or groups, you must not specify a VERSION or NOVERSION clause for any field or group within the range; this can result in a field having more than one product version.

An extensible structured token must be extended only by adding new fields to the end of the token. As new fields are added, new VERSION or NOVERSION clauses must be added to the token map for the new fields in the extensible structured token.

For more information on using product versions in extensible structured tokens, see the *SPI Programming Manual* and the *SPI Common Extensions Manual*.

Every field in the referenced definition must have an SPI null value to which the field is initialized by SPI before actual values are placed in the field. SPI null values indicate the presence or absence of a value in the field. A field with an SPI null value is effectively not present. The SPI null value can be:

- Explicitly specified with the clause SPI-NULL on page 6-37.
- Derived from the SPI null value of a group definition that contains the elementary item.
- Implicitly specified by default; the default value for SPI-NULL is 255.

The null value specification for a group of bit fields that share the same byte or word is generated as one contiguous block having an SPI-NULL value of 255 following the product version number.

You can specify a REDEFINES clause in the definition of an extensible structured token, but redefined fields have the same SPI null value as the fields they overlay.

If you include comments in your token map definition, the DDL compiler issues a warning message and does not save the comments.

SPI considers a field to contain an SPI null value if every byte in the field contains the SPI null value for the field. You use the SSNULL operating system procedure to set each field of the structure to its specified SPI null value. For a description of the SSNULL procedure, see the *Distributed Name Service (DNS) Management Programming Manual*.

For the SPI null value to indicate the presence or absence of a value in its associated field, the SPI null value must not be a legitimate value for the field. If every possible value of a field is legitimate, then an SPI null value cannot be used to indicate the presence or absence of a value. In such a case, you have two alternatives:

- Indicate the presence or absence of a non-null value in the field by an *is-present* field. An *is-present* field is a Boolean field that can be set to -1 to indicate that the field has a value (is present), or set to 0 to indicate the field value is null (is not present). The field must still have an explicit or implicit null value.

- Make the field larger. For example, if a field is a 16-bit integer and all 16-bit values are valid for the field, you can define the field as a 32-bit integer. Lengthening the field enables you to choose an SPI null value that creates a value in the 32-bit integer that is not one of the valid values for the 16-bit integer.

For more information on using product versions in extensible structured tokens, see the *SPI Programming Manual* and the *SPI Common Extensions Manual*.

Topics:

Product Versions for Bit Fields

Bit fields that share the same byte must have the same product version number in the token map. The product version number applies to the entire byte. If a bit field extends across two bytes within a word, the product version number of that field applies to the entire word.

The product version number for a bit filler depends on the filler's position in a word and the length of the filler.

- If a bit filler or group of contiguous bit fillers is less than a byte long and is contained within one of the two bytes of a word, the filler or group of fillers assumes the same product version number as all other bit fields in the containing byte.
- If a bit filler or group of contiguous bit fillers is a byte long or longer and fills either the upper or lower byte of a word, the filler or group of fillers is a NOVERSION field. The remaining part of the bit filler or fillers, if any, assumes the same product version number as all other bit fields in the byte that contains the remaining part.
- If a bit filler extends across two bytes of the same word but does not fill up either byte, the part of the bit filler on either side of the byte boundary assumes the same product version number as all other bit fields in the byte that contains the filler.

The DDL compiler allocates words for bit fields according to these rules:

- A bit field inside a group structure that follows a nonbit field item starts on a new 16-bit word. If you specify bit fields consecutively inside the group structure, the DDL compiler allocates the same 16-bit word for all contiguous bit fields that can fit in the word. For the next bit field that cannot fit in the same 16-bit word, the DDL compiler allocates the next word.
- Consecutive bit fields that occupy the same word have the same byte offset value but different bit offset values in their records in the DICTOBL file. An elementary item of another data type that follows a bit field item starts on the next word.

- A substructure containing only bit fields always starts and ends on a word boundary, padded with implicit bit fillers when necessary. Such a substructure is always an even number of bytes long, which conforms to how the C, Pascal (on D-series systems), pTAL, and TAL compilers allocate space for structures containing bit maps.

TOKEN-MAP Statement Output

If you request C, COBOL, Pascal (on D-series systems), pTAL, TACL, or TAL source-code output, the DDL compiler generates compatible data structures for the token map for the requested language. The SPI routines use these data structures to construct and access the specified extensible structured token.

The DDL compiler replaces any hyphen in the DDL map name with a circumflex (^) in a pTAL, TAL, or TACL map name, and an underscore (_) in a C or Pascal (on D-series systems) map name.

Table 7-1. DDL Data Structures Generated for Token Maps

Language	Data Structure
C	`static int map_name = {v1,v2,...,vn};` For the C data structure, each element in the static integer array is the value of one word in the token map.

COBOL	`01 map-name.`
	`  02 FILLER NATIVE-2 VALUE v1.`
	`  02 FILLER NATIVE-2 VALUE v2.`
	`  ...`
	`  02 FILLER NATIVE-2 VALUE vn.`
	`END`
	For the COBOL data structure, each FILLER element specifies the value of one word in the token map.

Pascal (on D-series systems)	`VAR map_name : ARRAY [1..n] OF INT16 := [v1, v2, ..., vn];` For the Pascal data structure, each element in the integer array is the value of one word in the token map.

Table 7-1. DDL Data Structures Generated for Token Maps

Language	Data Structure

TACL	`?Section map^name Struct`
	`  BEGIN`
	`  INT2 CODE VALUE v1v2;`
	`  INT  MAP (0:n-3) VALUE v3, ..., vn;`
	`  END;`
	For the TACL data structure, the first STRUCT variable is a double-word integer specifying the token code in the first 2 words of the token map. The rest of the STRUCT is an integer array containing one value for each remaining word in the token map.

pTAL or TAL	`DEFINE map^name = [v1, v2, ..., vn]#;`
	`  LITERAL map^name^WLN = n;`
	For the pTAL or TAL data structure, each constant in the DEFINE list specifies the value of one word in the token map. The LITERAL map^name^WLN specifies the total number of words in the token map.

For a description of how to use these definitions in a subsystem that accepts SPI programmatic commands, see the *Distributed Name Service (DNS) Management Programming Manual*.

Standard SPI Definitions in Token-Map Definitions

SPI does not provide standard definitions for token maps; subsystems must define their own token maps. When you define a token map, do not prefix the *map-name* with the letter Z; this ensures that your token-map name will not be the same as a current or future name supplied by HP.

SPI does provide a standard token type for all token maps. The standard token type for token maps is ZSPI-TYP-MAP, which consists of the standard token data type ZSPI-TDT-MAP and a token length of 255. You never refer to these definitions when you define a token map.

Each field in an extensible structured token must have a size, a type, and an SPI null value. It is generally good practice to specify the field type by referring to one of the standard SPI definitions. The names of these definitions have the form:

`ZSPI-DDL-`*name*

In this form, *name* uniquely identifies the definition.

The file ZSPIDEF.ZSPIDDL on the disk volume chosen for your system contains the DDL definitions for any standard SPI definitions you need. To refer to the standard definitions, compile this file into your dictionary, using the DDL SOURCE command.

Example 7-15 on page 7-20 describes an extensible structured token with three fields and a total byte length of 12. Each field is defined by reference to a definition in the standard SPI definition file ZSPIDEF.ZSPIDDL. The referenced definition determines the data type of the field and specifies a default SPI null value for that field. The SPI null value “X” explicitly specified for the field LOCATION overrides the standard SPI null value for ZSPI-DDL-CHAR8, which is a set of empty quotes (“ ”). The token map assigns product version “C00” to each of these fields–any subsystem of product version C00 or later can access the entire structured token.

Example 7-15. Extensible Structured Token

```
DEF assn-ddl-jobinfo.  ! Defines fields in extensible structure
  02 jnumber TYPE zspi-ddl-int.
  02 priority TYPE zspi-ddl-int.
  02 location TYPE zspi-ddl-char8 SPI-NULL "X".
END.

CONSTANT assn-tnm-jobinfo        VALUE IS 3.
TOKEN-MAP jobinfo-map  VALUE IS assn-tnm-jobinfo
                       DEF is assn-ddl-jobinfo.
VERSION "C00" FOR jnumber THRU location.
END
```

From the definitions in Example 7-15 on page 7-20, the DDL compiler generates the source code in Example 7-16 on page 7-20 through Example 7-20 on page 7-21.

Example 7-16. COBOL Source Code Generated for Example 7-15 on page 7-20

```
01 JOBINFO-MAP.
   02 FILLER NATIVE-2 VALUE 2303.   ! Token type ZSPI-TYPE-MAP
   02 FILLER NATIVE-2 VALUE 3.      ! Token number
   02 FILLER NATIVE-2 VALUE 12.     ! Token byte length
   02 FILLER NATIVE-2 VALUE 17152.  ! Product version "C00"
   02 FILLER NATIVE-2 VALUE 1024.
   02 FILLER NATIVE-2 VALUE 2136.
END.
```

Example 7-17. pTAL or TAL Source Code Generated for Example 7-15 on page 7-20

```
DEFINE JOBINFO^MAP = [2303, 3, 12, 17152, 1024, 2136]#;.
LITERAL JOBINFO^MAP^WLN = 6;  ! Number of words in token map
```

Example 7-18. TACL Source Code Generated for Example 7-15 on page 7-20

```
?Section JOBINFO^MAP Struct
BEGIN INT2  CODE VALUE 150929411;        ! Value generated from token code
INT  MAP (0:3) VALUE 12 17152 1024 2136; ! Values for rest of map
```

Example 7-19. C Source Code Generated for Example 7-15 on page 7-20

```
#pragma section jobinfo_map
static int      jobinfo_map[] = {2303,3,12,17152,1024,2136};
```

Example 7-20. Pascal Source Code Generated for Example 7-15 on page 7-20

```
?Section JOBINFO_MAP
VAR JOBINFO_MAP : Array [1..6 ] of INT16 := [2303,3,12,17152,
                                             1024,2136];
```

Example 7-21 on page 7-21 shows the DEFINITION and TOKEN-MAP statements when the JOBINFO token is extended to add new fields associated with product version “C10.”

Example 7-21. Extending an Extensible Token

```
DEF assn-ddl-jobinfo.  ! Defines fields in extensible structure
    02 jnumber TYPE zspi-ddl-int.
    02 priority TYPE zspi-ddl-int.
    02 location TYPE zspi-ddl-char8 SPI-NULL "X".
    02 jobclass-is-present TYPE zspi-ddl-boolean.
    02 jobclass TYPE zspi-ddl-int.
    02 jobusername TYPE zspi-ddl-username.
END

TOKEN-MAP assn-map-jobinfo VALUE IS assn-tnm-jobinfo
                           DEF IS assn-ddl-jobinfo.
  VERSION "C00" FOR jnumber THRU location.
  VERSION "C10" FOR jobclass-is-present.
  NOVERSION     FOR jobclass.
  VERSION "C10" FOR jobusername.
END
```

Example 7-21 on page 7-21 assumes that every possible value of the integer field `jobclass` is legitimate so that an SPI null value cannot be used to indicate its presence or absence. In this case, the Boolean `jobclass-is-present` field indicates whether there is a `jobclass` value. When this technique is used, a product version is specified for `jobclass-is-present` and NOVERSION is specified for `jobclass`. NOVERSION removes the `jobclass` field from consideration when determining its product version; SPI assumes that the product version number of `jobclass-is-present` indicates the correct product version for `jobclass`.

For a description of using an *`is-present`* field, see the *Distributed Name Service (DNS) Management Programming Manual*.

Example 7-22. Specifying Product Version Numbers for Bit Fields

```
DEF bits-layout-x.
  02 x-1              Type BIT 5.
  02 x-filler         Type BIT 3.
  02 x-2              Type BIT 7.
  END

TOKEN-MAP map-bits-x VALUE is 1 DEF is bits-layout-x.
          VERSION "D40" FOR x-1 THRU x-2.
          END

DEF bits-layout-y.
  02 y-1              Type BIT 4.
  02 y-2              Type BIT 8.
  02 y-3              Type BIT 6.
  END

   TOKEN-MAP map-bits-y VALUE is 32740 DEF is bits-layout-y.
             VERSION 10000 FOR y-1 THRU y-2.
             VERSION 15000 FOR y-3.
             END.

DEF bit-ddl-ex-a.
  02 bits-8           Type BIT 8.
  02 bits-3           Type BIT 3.
  02 bits-2           Type BIT 2.
  02 bits-10          Type BIT 10.
  02 bits-1           Type BIT 1.
  END.

TOKEN-MAP bit-map-ex-a VALUE 1 DEF bit-ddl-ex-a.
          VERSION "D20" FOR bits-8.
          VERSION "D30" FOR bits-3 THRU bits-2.
          VERSION "D40" FOR bits-10 THRU bits-1.
          END.

DEF bit-ddl-ex-c.
  02 char-3           Type CHARACTER 3 SPI-NULL 255.
  02 bits-8           Type BIT 8.
  02 bits-3           Type BIT 3.
  02 FILLER           Type BIT 10.
  02 bits-2           Type BIT 2.
  02 bits-5           Type BIT 5.
  02 FILLER           Type BIT 4.
  02 bits-4           Type BIT 4.
  02 bits-7           Type BIT 7.
  END.

TOKEN-MAP bit-map-ex-c VALUE 111 DEF bit-ddl-ex-c.
          VERSION "C00" FOR char-3 THRU bits-3.
          VERSION "C10" FOR bits-2.
          VERSION "C20" FOR bits-5 THRU bits-4.
          NOVERSION FOR bits-7.
          END.
```

Example 7-23. pTAL or TAL Output for Example 7-22 on page 7-22 (page 1 of 2)

```
?SECTION BITS^LAYOUT^X
STRUCT       BITS^LAYOUT^X^DEF (*);
   BEGIN
   UNSIGNED(5)  X^1;
   UNSIGNED(3)  X^FILLER;
   UNSIGNED(7)  X^2;
   BIT_FILLER   1;
   END;

?SECTION MAP^BITS^X
DEFINE MAP^BITS^X = [ 2303, 1, 2, 17152, 767 ]#;
LITERAL MAP^BITS^X^WLN = 5;

?SECTION BITS^LAYOUT^Y
STRUCT       BITS^LAYOUT^Y^DEF (*);
   BEGIN
   UNSIGNED(4)  Y^1;
   UNSIGNED(8)  Y^2;
   BIT_FILLER   4;
   UNSIGNED(6)  Y^3;
   BIT_FILLER   10;
   END;

?SECTION MAP^BITS^Y
DEFINE MAP^BITS^Y = [2303,32740,4,10000,767,1,15000,511,1,0,
                     511 ]#;
LITERAL MAP^BITS^Y^WLN = 11;

?SECTION BIT^DDL^EX^A
STRUCT       BIT^DDL^EX^A^DEF (*);
   BEGIN
   UNSIGNED(8)  BITS^8;
   UNSIGNED(3)  BITS^3;
   UNSIGNED(2)  BITS^2;
   BIT_FILLER   3;
   UNSIGNED(10) BITS^10;
   UNSIGNED(1)  BITS^1;
   BIT_FILLER   5;
   END;

?SECTION BIT^MAP^EX^A
DEFINE BIT^MAP^EX^A = [2303,1,4,17152,511,1,17162,511,1,
                       17172,767 ]#;
LITERAL BIT^MAP^EX^A^WLN = 11;
```

Example 7-23. pTAL or TAL Output for Example 7-22 on page 7-22 (page 2 of 2)

```
?SECTION BIT^DDL^EX^C
STRUCT       BIT^DDL^EX^C^DEF (*);
   BEGIN
   STRUCT       CHAR^3;
      BEGIN STRING BYTE [0:2]; END;
   FILLER       1;
   UNSIGNED(8)  BITS^8;
   UNSIGNED(3)  BITS^3;
   BIT_FILLER   5;
   BIT_FILLER   10;
   UNSIGNED(2)  BITS^2;
   BIT_FILLER   4;
   UNSIGNED(5)  BITS^5;
   BIT_FILLER   4;
   UNSIGNED(4)  BITS^4;
   BIT_FILLER   3;
   UNSIGNED(7)  BITS^7;
   BIT_FILLER   9;
   END;

?SECTION BIT^MAP^EX^C
DEFINE BIT^MAP^EX^C = [2303,111,12,17152,1023,1,0,511,1,
                       17152,767,1,0,511,1,17162,511,1,17172,
                       767,1,0,767]#;
LITERAL BIT^MAP^EX^C^WLN = 23;
```

Table 7-2 on page 7-24 shows a further breakdown of the token map `BIT^MAP^EX^C`.

Table 7-2. Structure of a Bit Map

Value in Word	Byte 1	Byte 2	Meaning of Value
2303	8	255	Token type
111	0	111	Token number
12	0	12	Byte length of structure
17152	C	00	Product version C00 for first field
1023	3	255	Null value for char-3
1	0	1	New product version specification follows
0	0	0	NOVERSION
511	1	255	Null value for byte FILLER
1	0	1	New product version specification follows
17152	C	00	Product version “C00”
767	2	255	Null value for bits-8 and bits-3
1	0	1	New product version specification follows
0	0	0	NOVERSION
511	1	255	Null value for bit FILLER

Table 7-2. Structure of a Bit Map

Value in Word	Byte 1	Byte 2	Meaning of Value
1	0	1	New product version specification follows
17162	C	10	Product version C10
511	1	255	Null value for bits-2
1	0	1	New product version specification follows
17172	C	20	Product version C20
767	2	255	Null value for bits-5 and bits-4
1	0	1	New product version specification follows
0	0	0	NOVERSION
767	2	255	Null value for bits-7 and bit FILLER

Example 7-24. Incorrect Use of SPI-NULL Value for Bit Fields

```
DEF bits-layout-z.
   02 z-1            Type BIT 4 SPI-Null 0.
   02 z-2            Type BIT 3 SPI-Null 1.
   02 z-3            Type BIT 10.
   02 z-4            Type BIT 4.
   END.
*** ERROR *** SPI-NULL value on a bit field must be 255 - Z-1
*** ERROR *** SPI-NULL value on a bit field must be 255 - Z-2
```

In Example 7-25 on page 7-25, NOVERSION is incorrect for field `z-2` because the DDL compiler puts `z-2` in the same byte as field `z-1`, and bit fields in the same byte must have the same product version number.

Example 7-25. Incorrect Use of Product Version Numbers for Bit Fields

```
DEF bits-layout-z.
   02 z-1            Type BIT 4 SPI-Null 255.
   02 z-2            Type BIT 3 SPI-Null 255.
   02 z-3            Type BIT 10.
   02 z-4            Type BIT 4.
   END.

TOKEN-MAP map-bits-z VALUE is 20 DEF is bits-layout-z.
          VERSION "C00" FOR z-1.
          NOVERSION FOR z-2.
          VERSION "C10" FOR z-3 THRU z-4.
          END.
*** ERROR *** Inconsistent VERSION within byte - Z-2
```

In Example 7-26 on page 7-26, the field `bits-2` cannot have a product version number because the DDL compiler puts `bits-2` in the same byte as `bits-3`, for which NOVERSION is specified:

Example 7-26. Incorrect Use of Version Numbers for Bit Fields

```
DEF bit-ddl-ex-b.
  02 bits-8          Type BIT 8.
  02 bits-3          Type BIT 3 SPI-NULL 255.
  02 bits-2          Type BIT 2.
  02 bits-10         Type BIT 10.
  02 bits-1          Type BIT 1 SPI-NULL 255.
  END.

TOKEN-MAP bit-map-ex-b VALUE 1 DEF bit-ddl-ex-b.
          VERSION "C00" FOR bits-8.
          NOVERSION FOR bits-3.
          VERSION "C10" FOR bits-2.
          VERSION "C20" FOR bits-10 THRU bits-1.
          END.
*** ERROR *** Inconsistent VERSION within byte - BITS-2
```

8

Dictionary-Manipulation Statements

Table 8-1. Dictionary-Manipulation Statements

Statement	Function
DELETE on page 8-1	Deletes specified objects from the open dictionary
EXIT on page 8-4	Ends the DDL session, closes any files that were opened in the session, and returns control to the command interpreter
OUTPUT on page 8-5	Reads objects from the open dictionary and writes them to any open DDL schema file, FUP source code file, REPORT file, or host-language source code file
OUTPUT UPDATE on page 8-7	Generates DDL source code that updates every referenced object in the open dictionary and writes this code to the open DDL source code file for subsequent compilation
SHOW USE OF on page 8-11	Lists the objects in the open dictionary that directly or indirectly refer to specified objects

DELETE

The DELETE statement deletes specified objects from the open dictionary.

```
DELETE { CONSTANT constant-name ... }
       { DEF[INITION] def-name ...  }
       { RECORD record-name ...     }
       { TOKEN-CODE token-name ...  }
       { TOKEN-MAP map-name ...     }
       { TOKEN-TYPE type-name ...   }
```

constant-name

is the name of a constant in the open dictionary. You can specify *constant-name* up to 50 times.

def-name

is a name that uniquely identifies an existing definition in the open dictionary. You can specify *def-name* up to 50 times.

record-name

is a name that uniquely identifies an existing record in the open dictionary. You can specify *record-name* up to 50 times.

token-name

is a name that uniquely identifies an existing token code in the open dictionary. You can specify *token-name* up to 50 times.

map-name

is a name that uniquely identifies an existing token map in the open dictionary. You can specify *map-name* up to 50 times.

type-name

is a name that uniquely identifies an existing token type in the open dictionary. You can specify *type-name* up to 50 times.

Before using the DELETE statement, open the dictionary on the appropriate subvolume with the DICT command.

The DELETE statement deletes a DDL object only from the dictionary; it does not delete the corresponding entries from any DDL, FUP, or language source code files.

Before you can delete an object that is referenced by other objects, you must first delete all the objects that refer to it.

Deleting an object that is referenced by another object is more complicated than deleting an object that is not referenced. For example, deleting a definition can be more complicated than deleting a record because a record is never referenced by another record or by a definition. Similarly, deleting a constant or a token type can be more complicated than deleting a token map or a token code because token maps and token codes are never referenced by another object.

Deleting a constant is particularly complicated because constants are usually referenced by a number of different objects.

When deleting a constant, a definition, or a token map, use the SHOW USE OF statement to display all the objects that refer to the object you want to delete. You can use an OUTPUT UPDATE statement to produce DDL source code that can be used to delete the objects that refer to an object you want to delete.

If you do not use the OUTPUT UPDATE statement, you must delete every object that refers to a specified object before you can delete that object. This includes not only direct references, in which object B refers to object A directly, but also indirect references, in which, for example, object B refers to object A and object C refers to object A by referring to object B.

Example 8-1. Deleting a Record Interactively

`03> VOLUME $data.sales`	Go to subvolume with dictionary.
`04> DDL`	Run DDL compiler.
`!?DICT`	Open dictionary.
`!DELETE RECORD employee.`	Delete record.

Example 8-2 on page 8-3 deletes a definition called `zip-cd` that is referenced by two other definitions (`addr` and `custinfo`), one of which (`addr`) is referenced by two records (`customer` and `supplier`).

Example 8-2. Deleting a Record Interactively

`!?DICT $data.sales`	Open dictionary in its subvolume.
`!DELETE RECORD customer supplier.`	Delete records that refer to `addr`.
`!DELETE DEF addr custinfo.`	Delete definitions that refer to `zip-cd`.
`!DELETE DEF zip-cd.`	Delete `zip-cd`.

You can enter this code interactively (as in Example 8-2 on page 8-3) or you can place the code in a file and use the SOURCE command to pass the code to the DDL compiler, as in Example 8-3 on page 8-3.

Example 8-3. Deleting a Record Interactively

```
05> DDL
!?SOURCE del-file    DICT command and DELETE statements are in del-file
!EXIT
```

To delete an SPI token type that is referenced by SPI token codes, first delete the token codes, as in Example 8-4 on page 8-3.

Example 8-4. Deleting an SPI Token Type That SPI Token Codes References

```
06> VOLUME $spi.tokens
07> DDL DICT
!DELETE TOKEN-CODE assn-tkn-my-status, assn-tkn-stat-reply.
!DELETE TOKEN-TYPE assn-typ-status.
!EXIT
```

EXIT

The EXIT statement ends the DDL session, closes any files that were opened in the session, and returns control to the command interpreter.

```
EXIT [ . ]
```

When you run the DDL compiler interactively, an EXIT statement stops the DDL compiler and returns control to the command interpreter.

When you run the DDL compiler noninteractively, an EXIT statement within the schema stops the DDL compiler at that point and returns control to the command interpreter. Use of the EXIT statement within a schema is optional, because reaching the end of the file performs the same function.

EXIT closes any files that were opened in the session.

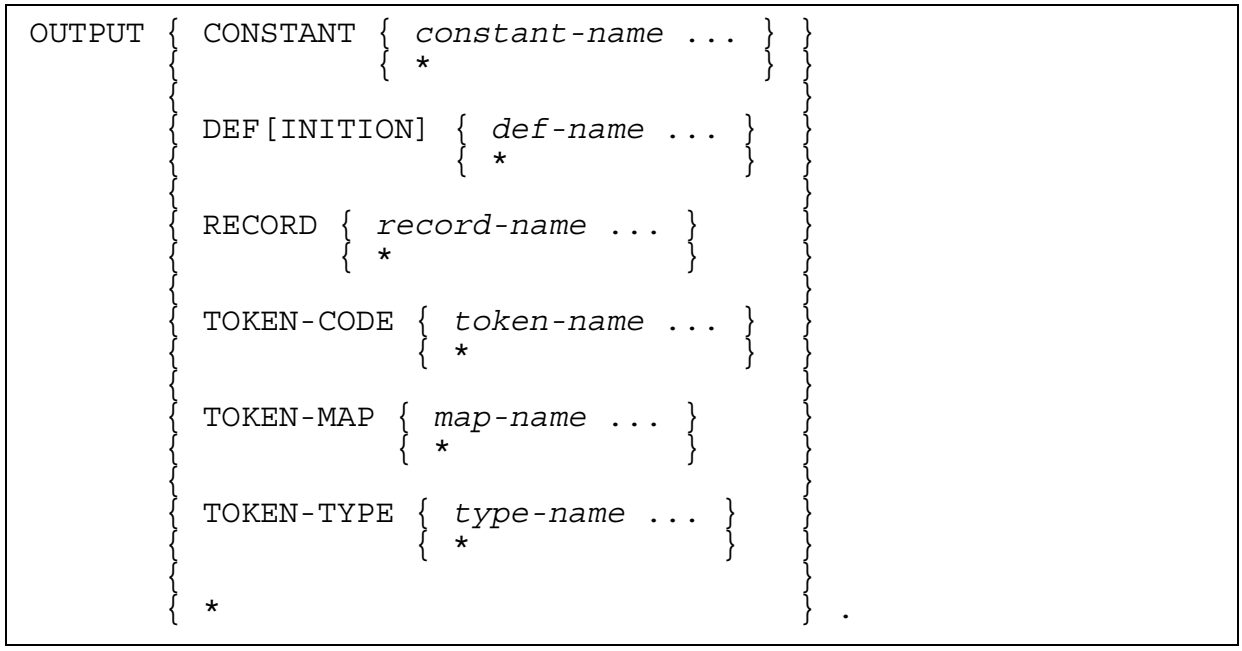
Entering Ctrl-y at the terminal has the same effect as an end-of-file mark. If you type Ctrl-y at the DDL prompt, the DDL compiler displays EOF! and ends the session.

Example 8-5. EXIT Statement in Interactive DDL Session

`10> VOLUME $data.sales`	Go to subvolume with dictionary.
`11> DDL`	Run DDL compiler.
`!?DICT`	Open dictionary.
`!?FUP fupsrc`	Open FUP source code file.
`!OUTPUT RECORD customer`	Write record to `fupsrc`.
`!EXIT`	Return to command interpreter.

OUTPUT

The OUTPUT statement reads objects from the open dictionary and writes them to any open DDL schema file, FUP source code file, REPORT file, or host-language source code file.



constant-name

is the name of a constant in the open dictionary. You can specify *constant-name* up to 50 times.

def-name

is a name that uniquely identifies an existing definition in the open dictionary. You can specify *def-name* up to 50 times.

record-name

is a name that uniquely identifies an existing record in the open dictionary. You can specify *record-name* up to 50 times.

token-name

is a name that uniquely identifies an existing token code in the open dictionary. You can specify *token-name* up to 50 times.

map-name

is a name that uniquely identifies an existing token map in the open dictionary. You can specify *map-name* up to 50 times.

type-name

is a name that uniquely identifies an existing token type in the open dictionary. You can specify *type-name* up to 50 times.

*

specifies either all objects of the given type or, when no type is specified, all objects.

OUTPUT is used to write DDL objects from the open dictionary to any open DDL, FUP, REPORT or language source code files.

OUTPUT cannot be used to generate output for Pathmaker objects. Pathmaker objects (servers, services, requesters, and screens) are added to and deleted from a dictionary by the Pathmaker product, not by the DDL compiler.

If a DDL source code file is open, the OUTPUT statement causes DDL to retrieve the specified objects from the open dictionary and generate DDL statements to define the objects in the specified DDL source code file.

Any objects written to a DDL source code file with the OUTPUT statement are listed on the DDL compiler listing.

If C, COBOL, Pascal (on D-series systems), pTAL, TACL, or TAL source code files are open, the OUTPUT statement retrieves any of the specified objects (constant, definition, record, token code, token map, or token code) from the open dictionary and generates the appropriate source code in each open source code file.

If FORTRAN source code files are open, the OUTPUT statement retrieves the specified definitions and records from the open dictionary and generates the appropriate source code in each open source code file.

If a FUP source code file is open, the OUTPUT statement retrieves the data structure and file attributes for any specified records from the dictionary and writes FUP file creation commands to the open FUP file.

In Example 8-6 on page 8-6, an OUTPUT RECORD statement causes the DDL compiler to retrieve the data structure and file attributes of the record `order-info` from the open dictionary and write the resulting source to open COBOL source code files and FUP source code files.

Example 8-6. OUTPUT RECORD Statement

`!?DICT $data.sales`	Open dictionary.
`!?COBOL $data.sales.cobsrc`	Open COBOL source code file.
`!?FUP $data.sales.fupsrc`	Open FUP source code file.
`!OUTPUT RECORD order-info.`	Write source for `order-info`.

If you have changed your dictionary and want to ensure that your source code files correspond exactly to the changed dictionary, use an OUTPUT * statement.

Example 8-7. OUTPUT * Statement

`15>DDL`	Run DDL compiler.
`!?DICT`	Open dictionary.
`!?DDL ddlsrc`	Open and clear DDL source code file.
`!OUTPUT *.`	Write all entries from dictionary to `ddlsrc`.
`!?NODDL`	Close DDL source code file.
`!?FUP fupsrc`	Open and clear FUP source code file.
`!OUTPUT RECORD *.`	Write all records from dictionary to `fupsrc`.

The statements in Example 8-8 on page 8-7 cause the DDL compiler to generate COBOL data descriptions for all constants in the dictionary, one token type, and two token codes. These descriptions are written to the open COBOL source code file, `spitkn`.

Example 8-8. OUTPUT Statements

```
16> DDL DICT $spi.tokens
!?COBOL spitkn
!OUTPUT CONSTANT *.
!OUTPUT TOKEN-TYPE assn-typ-status.
!OUTPUT TOKEN-CODE assn-tkn-my-status, assn-tkn-stat-reply.
!EXIT
```

Rather than specify a list of the particular constants needed by the token type and token code, Example 8-8 on page 8-7 generates source code for all the constants in the dictionary.

OUTPUT UPDATE

The OUTPUT UPDATE statement generates DDL source code that updates every referenced object in the open dictionary and writes this code to the open DDL source code file for subsequent compilation.

```
OUTPUT UPDATE { CONSTANT constant-name ...     }
              { [ DEF[INITION] ] def-name ... }
              { TOKEN-TYPE type-name ...       } .
```

constant-name

is the name of a constant in the open dictionary. You can specify *constant-name* up to 50 times.

def-name

is a name that uniquely identifies an existing definition in the open dictionary. You can specify *def-name* up to 50 times.

type-name

is a name that uniquely identifies an existing token type in the open dictionary. You can specify *type-name* up to 50 times.

The OUTPUT UPDATE statement is useful only when you want to modify or delete an object that might be referenced by one or more other dictionary objects.

The dictionary and a DDL source code file must both be open before you specify OUTPUT UPDATE.

Pathmaker objects (servers, services, requesters, and screens) that refer to the specified definition are ignored by OUTPUT UPDATE. If a Pathmaker object refers to a definition that has changed, the Pathmaker product makes the changes to the Pathmaker object, issuing an error message if appropriate.

OUTPUT UPDATE searches the dictionary for all DDL objects that refer to an object specified in the statement. The DDL compiler then generates source code that can be used to delete any objects that refer to the specified object, to update the definition of the specified object, and to redefine the referring objects. The DDL compiler writes this source code to the previously opened DDL source code file.

The DDL compiler generates these sections of source code for each object specified in the OUTPUT UPDATE statement:

Section	Contents
1	DELETE statements to delete any objects that directly or indirectly refer to the specified object
2	A statement to redefine the specified object
3 and greater	One section for each statement needed to rebuild the objects deleted in the first section–those objects that refer to the specified object

To update a specified object, close the DDL source-update file and edit the second section of the file to make the changes you want to the object definition. Then, use the SOURCE command to compile the entire DDL source-update file.

To delete a specified object, close the DDL source-update file and use the SOURCE command to compile only the first section. This instructs the DDL compiler to delete all referring objects. Then use a DELETE statement to delete the specified object.

Assume that your dictionary contains the objects defined in the database schema from Appendix B, Sample Schemas, and that you want to change the size of the `zip-cd` definition from 5 to 9 digits. Because other definitions and records refer either directly or indirectly to `zip-cd`, you cannot simply change its definition.

To change the definition of `zip-cd` and the records and definitions that refer to it, use OUTPUT UPDATE as in Example 8-9 on page 8-9.

Example 8-9. OUTPUT UPDATE Statement

`20>DDL dict`	Run DDL compiler, opening dictionary in current subvolume.
`!?DDL myfile`	Open DDL source code file.
`!OUTPUT UPDATE zip-cd.`	Write update source to `myfile`.
`!?NODDL`	Close `myfile` before editing it.
`!?EDIT myfile`	
`...`	Change definition of `zip-cd`.
`*EXIT`	Exit from the editor.
`!?SOURCE myfile`	Compile the contents of `myfile` into the dictionary.

Example 8-10. Contents of myfile After Example 8-9 on page 8-9 (page 1 of 2)

```
?Section ZIP-CD-DELETES             ! First section deletes all
Delete Record SUPPLIER.             ! objects that refer to
Delete Record CUSTOMER.             ! ZIP-CD.
Delete Definition SUPPINFO.
Delete Definition CUSTINFO.
Delete Definition ADDR.

?Section ZIP-CD                     ! Second section defines new
Definition ZIP-CD    PIC "9(9)".    ! ZIP-CD with new length.

?Section ADDR.                      ! Subsequent sections contain
Definition ADDR.                    ! definitions to rebuild
  02 ADDRESS         PIC "X(22)".   ! deleted objects.
  02 CITY            PIC "X(14)".
  02 STATE           PIC "X(12)".
  02 ZIP             TYPE ZIP-CD.
  End.

?Section CUSTINFO.
Definition CUSTINFO.
  02 CUSTNUM         TYPE *.
  02 CUSTNAME        TYPE NAME.
  02 ADDR            TYPE *.
  End.

?Section SUPPINFO.
Definition SUPPINFO.
  02 SUPPNUM         TYPE *.
  02 SUPPNAME        TYPE NAME.
  02 ADDR            TYPE *.

  End.
```

Example 8-10. Contents of myfile After Example 8-9 on page 8-9 (page 2 of 2)

```
?Section CUSTOMER.
Record CUSTOMER.
  File is "$data.sales.customer" Key-sequenced.
                                 Audit.

  Definition is CUSTINFO.

  Key is CUSTNUM duplicates not allowed.
  Key "cn" is CUSTNAME.
  End.

?Section SUPPLIER.
Record SUPPLIER.
  File is "$data.sales.supplier" Key-sequenced.
                                 Audit.

  Definition is SUPPINFO.

  Key is SUPPNUM duplicates not allowed.
  Key "su" is SUPPNAME.
  End.
```

Note. The order in which the objects are deleted and added is important. Any other order can cause the DDL compiler to issue an error message.

Example 8-11 on page 8-10 deletes the constant `mdy-date-display` and all the objects that refer to that constant.

Example 8-11. OUTPUT UPDATE Deleting a Constant and Objects That Refer to It

```
20>DDL dict

!?DDL ddlout

!OUTPUT UPDATE CONSTANT mdy-date-display.

!?NODDL

!?SOURCE ddlout (mdy-date-display-deletes)    Execute DELETE statements
                                              generated by OUTPUT UPDATE
                                              statement

!DELETE CONSTANT mdy-date-display             Delete mdy-date-display

!EXIT
```

SHOW USE OF

The SHOW USE OF statement lists the objects in the open dictionary that directly or indirectly refer to specified objects.

```
SHOW USE OF { CONSTANT constant-name [, constant-name ]... }
            { [ DEF[INITION] ] def-name [, def-name ]...   }
            { TOKEN-TYPE type-name [, type-name ]...       }
```

constant-name

is the name of a constant in the open dictionary. You can specify *constant-name* up to 50 times.

def-name

is a name that uniquely identifies an existing definition in the open dictionary. You can specify *def-name* up to 50 times.

type-name

is a name that uniquely identifies an existing token type in the open dictionary. You can specify *type-name* up to 50 times.

The dictionary must be open for SHOW USE OF to execute successfully.

If you want to modify or delete a referenced object, you can use the SHOW USE OF statement to list all references to the object you plan to modify or delete; however, HP recommends using OUTPUT UPDATE to make such changes.

SHOW USE OF generates a listing that shows which objects use the specified objects. Objects can refer to other objects:

Object	Can refer to ...
Constant	Other constants
Definition	Other definitions Constants
Record	Definitions Constants
Token type	Definitions Constants
Token code	Token types
Token map	Definitions Constants

The SHOW USE OF listing includes a number to indicate the nesting level of the objects it displays. Any objects that refer directly to a specified object are at nesting level 1; any objects that refer directly to an object at level 1 are at nesting level 2; and so forth.

If more than one object refers to a specified object, the listing is sequenced first by the order in which objects are specified in the SHOW USE OF statement, second by the order in which referring objects are in the dictionary, and third by the nesting level.

For the definitions in Example 8-12 on page 8-12 and Example 8-13 on page 8-13, see the sample schema in Appendix B, Sample Schemas.

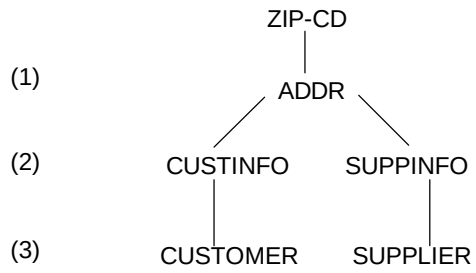
Example 8-12. SHOW USE OF Nesting Levels

`!?DICT`	Open dictionary.
`!SHOW USE OF DEF zip-cd`	Display objects that refer to the definition `zip-cd` (for the definition of `zip-cd` and the objects that refer to it, see the sample schema in Appendix B, Sample Schemas).

```
(1) Definition ZIP-CD      used by Definition ADDR
(2) Definition ADDR        used by Definition CUSTINFO
(3) Definition CUSTINFO    used by Record     CUSTOMER

(2) Definition ADDR        used by Definition SUPPINFO
(3) Definition SUPPINFO    used by Record     SUPPLIER
```

Nesting levels for ZIP-CD:



VST811.vsd

Example 8-13. SHOW USE OF Listing Sequence

```
?DICT
SHOW USE OF DEF custnum, name.

(1) Definition CUSTNUM          used by Definition CUSTINFO
(2) Definition CUSTINFO         used by Record     CUSTOMER

(1) Definition CUSTNUM          used by Definition ORDERINFO
(2) Definition ORDERINFO        used by Record     ORDERS

(1) Definition NAME             used by Definition CUSTINFO
(2) Definition CUSTINFO         used by Record     CUSTOMER

(1) Definition NAME             used by Definition EMPINFO
(2) Definition EMPINFO          used by Record     EMPLOYEE

(1) Definition NAME             used by Definition PARTSINFO
(2) Definition PARTSINFO        used by Record     PARTS

(1) Definition NAME             used by Definition SUPPINFO
(2) Definition SUPPINFO         used by Record     SUPPLIER
```

9 DDL Compiler Commands

DDL compiler commands instruct the DDL compiler to perform specific actions.

Commands can be:

- Placed anywhere in a DDL source code file
- Passed to the DDL compiler as part of the RUN DDL command
- Entered at your terminal when you run DDL interactively

For rules governing how you enter DDL commands, see Commands on page 2-18.

These tables list the commands according to the general functions they perform:

In the tables' command descriptions:

Symbol	Means
(A)	Acts immediately
(S)	Sets a condition flag that controls subsequent action

Many commands have a second form that begins with NO. You can set and reset these commands as necessary.

Table 9-1. Dictionary Commands

Command	Description
[NO]DICT	Opens [closes] a dictionary (A) Starts [stops] writing object definitions to the dictionary (S)
DICTN	Creates and opens a nonaudited dictionary or opens an existing dictionary (A)
	Writes subsequent object definitions to the dictionary (S)
DICTR	Opens an existing dictionary for read-only access (A)
[NO]SAVE	Saves [purges] the open dictionary when that dictionary is closed (S)

Table 9-2. Compilation Commands

Command	Description
COLUMNS	Specifies the number of significant columns (character positions) on DDL input lines (S)
[NO]COMMENTS	Includes [excludes] subsequent user-defined dictionary comments in [from] the open dictionary (S)
ERRORS	Specifies the number of errors allowed before compilation stops (S)
SECTION	Names a section of a DDL schema (without affecting the section headings in host-language source code files) (S)
SOURCE	Compiles all or part a specified DDL schema (A)
WARNINGS	Specifies the number of warnings allowed before compilation stops (S)

Table 9-3. C Source Output Commands (page 1 of 2)

Command	Description
[NO]C	Opens [closes] a C source code file (A)
	Starts [stops] writing translated DDL object definitions to the C source code file (S)
[NO]C00CALIGN	Generates C code according to C00 [pre-C00] alignment rules (S)
[NO]CCHECK	Performs [suppresses] C syntax checks on subsequent DDL object definitions without generating code (S)
[NO]CDEFINEUPPER	Generates C `#define` names in uppercase [lowercase] letters (S)
CFIELDALIGN MATCHED2	Generates C structures that are compatible with pTAL and TAL structures (S)
CIFDEF, CIFNDEF, and CENDIF	Generate the compiler directives `#ifdef`, `#ifndef`, and `#endif`, respectively (A)
[NO]CPRAGMA	Includes [excludes] `#pragma`-generating code (S)

Table 9-3. C Source Output Commands (page 2 of 2)

Command	Description
[NO]CTOKENMAP_ASDEFINE	Generates TOKEN MAP output as `#define` statements [a static `int array`] (S)
CUNDEF	Generates a `#undef` compiler directive (A)
[NO]C_DECIMAL	Generates decimal [`char`] output for subsequent C simple numeric items (S)
[NO]C_MATCH_HISTORIC_TAL	Generates [suppresses] C data structures that are equivalent to pTAL, TAL, and COBOL data structures
[NO]EXPANDC	Generates a C referenced type definition inline [as a structure name] (A)

Table 9-4. COBOL Source Output Commands

Command	Description
[NO]ANSICOBOL	Generates COBOL output in ANSI [TANDEM] format (S)
[NO]COBCHECK	Performs [suppresses] COBOL syntax checks on subsequent DDL object definitions without generating code (S)
COBLEVEL	Specifies a level-numbering scheme for COBOL output (S)
[NO]COBOL	Opens [closes] a COBOL source code file (A)
	Starts [stops] writing translated DDL object definitions to the COBOL source code file (S)
[NO]VALUES	Includes [excludes] initial values from DEFINITION and RECORD statements in [from] DDL or COBOL source code (S)

Table 9-5. FORTRAN Source Output Commands

Command	Description
[NO]FORCHECK	Performs [suppresses] FORTRAN syntax checks on subsequent DDL object definitions without generating code (S)
[NO]FORTRAN	Opens [closes] a FORTRAN source code file (A)
	Starts [stops] writing translated DDL object definitions to the FORTRAN source code file (S)
[NO]FORTRANUNDERSCORE	Replaces with underscores [deletes] hyphens in DDL names for FORTRAN output (S)

Table 9-6. File Utility Program (FUP) Source Output Commands

Command	Description
[NO]FUP	Opens [closes] a FUP source code file (A) Starts [stops] writing translated DDL object definitions to the FUP source code file (S)
NEWFUP_FILEFORMAT	Specifies file format 2 for all FUP source code files and all FUP alternate key files (S)
OLDFUP_FILEFORMAT	Specifies file format 1 for all FUP source code files and all FUP alternate key files (S)
NOFILEFORMAT	Specifies no file format for all FUP source code files and all FUP alternate key files (S)

Table 9-7. Pascal Source Output Commands (D-Series Systems Only)

Command	Description
[NO]PASCAL (D-Series Systems Only)	Opens [closes] a Pascal source code file (A) Starts [stops] writing translated DDL object definitions to the Pascal source code file (S)
PASCALBOUND (D-Series Systems Only)	Sets the lower bound for Pascal arrays (S)
[NO]PASCALCHECK (D-Series Systems Only)	Performs [suppresses] Pascal syntax checks on subsequent data descriptions without generating code (S)
[NO]PASCALNAMEDVARIANT (D-Series Only)	Generates the REDEFINES clause in the last element as a named [anonymous] variant record in Pascal output (S)

Table 9-8. pTAL and TAL Output Commands

Command	Description
DO_PTAL_ON[OFF]	Generates code that cannot [can] be compiled by older pTAL or TAL compilers that do not recognize FIELDALIGN clauses (S)
[NO]TAL	Opens [closes] a pTAL or TAL source code file (A) Starts [stops] writing translated DDL object definitions to the pTAL or TAL source code file (S)
[NO]TALALLOCATE	Causes [suppresses] memory allocation in pTAL or TAL for single-field definitions when the TAL command is in effect (S)
TALBOUND	Sets the lower bound for pTAL or TAL arrays (S)
[NO]TALCHECK	Performs [suppresses] pTAL or TAL syntax checking on subsequent data descriptions without generating code (S)
[NO]TALUNDERSCORE	Replaces hyphens with underscores [circumflexes] in DDL names for pTAL or TAL output (S)

Table 9-9. TACL Source Output Commands

Command	Description
[NO]TACL	Opens [closes] a TACL source code file (A)
	Starts [stops] writing translated DDL object definitions to the TACL source code file (S)
TACLGEN	Specifies a TACL source code generation product version

Table 9-10. DDL Other Source Output Commands

Command	Description
[NO]DDL	Opens [closes] a DDL schema file (A)
	Starts [stops] writing DDL object definitions to the DDL schema file (S)
FIELDALIGN SHARED8	Stores data structures in the dictionary with SHARED8 alignment (S)
FILLER	Specifies the algorithm for generating filler bytes for source code (S)
[NO]NCLCONSTANT	Opens [closes] an NCL source code file (A)
	Starts [stops] writing translated DDL constant definitions to the NCL source code file (S)
[NO]OUTPUT SENSITIVE	Generates case-sensitive [case-insensitive] output (S)
SETLOCALENAME	Specifies the language, territory, and character set for output of text items (S)
SETSECTION	Determines SECTION headings for all open source code files except TACL source code files (A) (S)
[NO]TIMESTAMP	Includes [excludes] data and time comments in [from] source code listings (S)
[NO]VALUES	Includes [excludes] initial values from DEFINITION and RECORD statements in [from] DDL or COBOL source code (S)

Table 9-11. Listing Commands

Command	Description
[NO]CLISTIN	Includes [excludes] subsequent user-defined comments in [from] the compiler listing (S)
[NO]DEFLIST	Includes in [excludes from] the compiler listing a description of each definition that is referenced by a DEFINITION or RECORD statement (S)
LINECOUNT	Specifies the number of lines for each page for all source code files (S)
[NO]LIST	Includes [excludes] subsequent DDL source lines in [from] the compiler listing (S)
OUT	Specifies the destination for compiler output (source lines, warnings, and error messages) (A)
PAGE	Writes the next line of the compiler listing at the top of the next page (A)
	Optionally specifies a page title (S)
[NO]REPORT	Opens [closes] a report file (A)
	Starts [stops] writing a schema report to the report file (S)
SPACING	Specifies the number of blank lines to insert between lines of a printed report (S)
[NO]WARN	Includes [excludes] warnings in [from] the compiler listing (S)

Table 9-12. Other DDL Commands

Command	Description
EDIT	Suspends compilation, starts an EDIT process, opens the specified file, executes the specified commands, closes the file, and resumes compilation when the EDIT process stops (A)
HELP	Briefly describes a specified command or all commands (A)
RESET	Stops compiling the current statement and returns to the state before compilation of that statement began (A)
TEDIT	Suspends compilation, starts a PS Text Edit process, opens the specified file, executes the specified commands, closes the file, and resumes compilation when the TEDIT process stops (A)

ANSICOBOL

The [NO]ANSICOBOL command generates COBOL output in ANSI [TANDEM] format.

```
[NO]ANSICOBOL
```

Default: NOANSICOBOL

`ANSICOBOL`

generates COBOL output in ANSI format.

`NOANSICOBOL`

generates COBOL output in TANDEM format.

ANSI and TANDEM formats are described in the *COBOL Manual for TNS and TNS/R Programs* and the *COBOL Manual for TNS/E Programs*.

In Table 9-1 on page 9-2, the DDL compiler opens a COBOL source code file, COBSRC, and adds the definition for NAME to that file.

Example 9-1. ANSICOBOL Command

DDL Input

```
?ANSICOBOL
?COBOL COBSCR
DEF name.
  02 last-name PIC X(12).
  02 first-name PIC X(8).
  02 midinit PIC X(2).
END.
```

DDL Output (COBOL Code)

```
      * SCHEMA PRODUCED DATE - TIME : 9/14/2004 - 18:22:07
?SECTION NAME,ANSI
      * Definition NAME created on 09/14/2004 at 18:22
         01 NAME.
           02 LAST-NAME              PIC X(12).
           02 FIRST-NAME             PIC X(8).
           02 MIDINIT                PIC X(2).
```

In Example 9-2 on page 9-8, the DDL compiler opens a COBOL source code file, COBSRC1, and adds the definition for `name` to that file.

Example 9-2. NOANSICOBOL Command

DDL Input

```
?NOANSICOBOL
?COBOL COBSCR1
?OUTPUT DEF name.
```

DDL Output (COBOL Code)

```
* SCHEMA PRODUCED DATE - TIME : 9/14/2004 - 18:07:27
?SECTION NAME,TANDEM
* Definition NAME created on 09/14/2004 at 18:07
  01 NAME.
    02 LAST-NAME                    PIC X(12).
    02 FIRST-NAME                   PIC X(8).
    02 MIDINIT                      PIC X(2).
```

C

The [NO]C command:

- Opens [closes] a C source code file
- Starts [stops] writing translated DDL object definitions to the C source code file

```
{ C [ c-source-file [ ! ] ] }
{ NOC                       }
```

Default: NOC

C

closes any open C source code file, opens *c-source-file*, translates subsequent DDL objects defined by statements or specified in OUTPUT statements to C source code statements, and writes the C source code statements to *c-source-file*.

c-source-file

is the name of the C source code file to be created, if necessary, and opened. The file must be one of:

- EDIT file
- Unstructured file
- Sequential device (such as a terminal, spooler, or process)

If the file exists but is not one of these types, the DDL compiler issues an error message and does not open the file.

If *c-source-file* is an EDIT file, and it exceeds 99,999 lines, the DDL compiler issues FILE ERROR - filename - Edit file line number too large (537) on page A-17.

Default: home terminal

`!`

purges the contents of *`c-source-file`* before opening it, if it exists. If *`c-source-file`* does not exist, the exclamation point has no effect.

Without the exclamation point, the DDL compiler appends the new C source code statements to the end of *`c-source-file`*, and does not replace any existing objects.

`NOC`

closes any open C source code file and stops translating DDL object definitions to C source code statements.

For the data types that the DDL compiler generates for C source code, see Table C-1 on page C-1.

The compiler can translate DDL objects specified in an OUTPUT statement only if the dictionary containing these objects is open.

Each DDL object translated to C source code is written to the C source code file in a separate section that has the same name as the DDL object it contains. To suppress the generation of individual section headings with the SETSECTION command.

With the exception of the TOKEN-MAP statement, the DDL compiler does not generate C data definitions that allocate space. Instead, the DDL compiler generates C `typedef`s for scalar types and structure templates for multiple-element DDL records.

The DDL compiler replaces any hyphen in a DDL name with an underscore (_) before writing the name to the C source code file.

The DDL compiler appends the characters `_def` to the tag for all C typedefs and structures generated by the DDL compiler; therefore, the maximum length for the name of a DDL definition or record that is going to be translated to C is 27 ASCII characters, not the standard DDL length of 30 characters.

All C identifiers generated by the DDL compiler are in lowercase letters, except #define names, which are in uppercase letters by default. You can use the NOCDEFINEUPPER command to specify lowercase letters for `#define` names.

The C source code for a definition or record compiled with EXPANDC contains the fillers added by the DDL compiler as specified by the alignment algorithm in effect when the definition or record was compiled.

The DDL compiler performs all of the syntax checks listed under the CCHECK command before writing the C source output. If the DDL compiler finds a syntax error, it does not write the source output for the object with the error; it does write source output for an object if only a warning is issued.

All C arrays have a lower bound of 0.

When generating C source code, the DDL compiler ignores these clauses:

- DISPLAY
- HEADING
- HELP
- INDEXED BY
- MUST BE
- NULL
- OCCURS DEPENDING ON
- TACL
- UPSHIFT
- USAGE IS INDEX
- 88 condition-name

In some cases, the DDL compiler ignores the NOVALUES clause. (For more information, see VALUES on page 9-115.)

In Example 9-3 on page 9-10, the DDL compiler retrieves the record CUSTOMER and the objects that the record refers to, directly and indirectly, from the open dictionary. Then the DDL compiler translates the record and the referenced objects to C source code and appends the source code to the open C file, $DATA.SALES.CSRC. For the definition of the CUSTOMER record and the objects it refers to, see Appendix B, Sample Schemas.

Example 9-3. C Command (page 1 of 2)

DDL Input

```
27> DDL
!?DICT
!?C $data.sales.csrc
!OUTPUT CONSTANT custnum-heading.
!OUTPUT DEF custnum zip-cd name addr custinfo.
!OUTPUT RECORD customer.
!EXIT
```

Example 9-3. C Command (page 2 of 2)

DDL Output (C Code)

```
/* SCHEMA PRODUCED DATE - TIME :11/02/1995 14:49:35 */
#pragma section custnum_heading
/* Constant CUSTNUM-HEADING created on 11/02/1995 at 14:37 */
#define CUSTNUM_HEADING "Customer/Number"
#pragma section custnum
/* Definition CUSTNUM created on 11/02/1995 at 14:37 */
typedef char                              custnum_def[4];
#pragma section zip_cd
/* Definition ZIP-CD created on 11/02/1995 at 14:37 */
typedef char                              zip_cd_def[5];
#pragma section name
/* Definition NAME created on 11/02/1995 at 14:37 */
#pragma fieldalign shared2 __name
typedef struct
{
   char                                   last_name[12];
   char                                   first_name[8];
   char                                   midinit[2];
} name_def;
#pragma section addr
/* Definition ADDR created on 11/02/1995 at 14:37 */
#pragma fieldalign shared2 __name
typedef struct
{
   char                                   address[22];
   char                                   city[14];
   char                                   state[2];
   zip_cd_def                             zip;
} addr_def;
#pragma section custinfo
/* Definition CUSTINFO created on 11/02/1995 at 14:37 */
typedef struct
{
   custnum_def                            custnum;
   name_def                               custname;
   addr_def                               addr;
} custinfo_def;
#pragma section customer
/* Record CUSTOMER created on 11/02/1995 at 14:37 */
typedef custinfo_def                      customer_def;
```

C00CALIGN

The [NO]C00CALIGN command generates C code according to C00 [pre-C00] alignment rules.

```
[NO]C00CALIGN
```

Default: C00CALIGN

`C00CALIGN`

generates C code according to C00 alignment rules.

`NOC00CALIGN`

generates C code according to pre-C00 alignment rules.

When using rules prior to C00, all substructures must be word aligned and an even number of bytes in length.

When using earlier rules, substructures that contain only byte data can be byte aligned or odd length.

The C00CALIGN and CFIELDALIGN_MATCHED2 commands cannot be in effect at the same time. The DDL compiler uses the value of the last command that was specified (C00CALIGN, NOC00CALIGN, or CFIELDALIGN_MATCHED2). See Example 9-5 on page 9-16.

For more information about alignment rules, see Appendix H, DDL Alignment Rules for C.

CCHECK

The [NO]CCHECK command performs [suppresses] C syntax checks on subsequent DDL object definitions without generating code.

```
[NO]CCHECK
```

Default: CCHECK if a C source code file is open, otherwise NOCCHECK

`CCHECK`

performs C syntax checks as though C source code were being produced.

`NOCCHECK`

suppresses C syntax checks.

If a C source code file is open, the compiler performs the C checks whether or not CCHECK is set.

You can stop C syntax checking by specifying NOCCHECK; you can restart checking with a subsequent CCHECK.

The DDL compiler does not perform the lengthy testing performed by the C compiler. The DDL compiler tests the DDL statements to ensure that they follow the rules specified by C:

- A name cannot be longer than 31 ASCII characters. A name might become longer because the DDL compiler appends `_def` to the name of a definition or record.
- C reserved words cannot be DDL names.

If you compile DDL data structures for C, you must maintain word alignment throughout. Be sure that all members of a structure containing character or filler items have an even number of characters, and that a substructure within a structure starts on a word boundary. Odd-length character fields must be followed by fields that are naturally word aligned. If you use the C_MATCH_HISTORIC_TAL or CFIELDALIGN_MATCHED2 command, these restrictions change. For more information, see CFIELDALIGN_MATCHED2 on page 9-14.

Example 9-4. CCHECK Command

```
?CCHECK
DEFINITION orderinfo.
  02 ordernum                          PIC 9(3)
                                       HEADING ordernum-heading.
  02 orderdate                         TYPE mdy-date.
  02 deldate                           TYPE mdy-date.
  02 salesperson                       TYPE empnum
                                       HEADING salesperson-
heading.
  02 custnum                           TYPE *.
END
Definition ORDERINFO size is 23 bytes.
Definition ORDERINFO added to dictionary.
***WARNING*** C OUTPUT DIAGNOSTICS:
***ERROR***Structure alignment in C is incompatible with
   DDL orderdate
?NOCCHECK
```

When CCHECK is in effect, the DDL compiler issues this message for each DDL object that passes the syntax check:

```
C CHECK completed for name
```

In the message, *name* is the name of the object checked by CCHECK.

CDEFINEUPPER

The [NO]CDEFINEUPPER command generates C `#define` names in uppercase [lowercase] letters.

```
[NO]CDEFINEUPPER
```

Default: CDEFINEUPPER

`CDEFINEUPPER`

generates C `#define` names in uppercase letters.

`NOCDEFINEUPPER`

generates C `#define` names in lowercase letters.

CFIELDALIGN_MATCHED2

The CFIELDALIGN_MATCHED2 command generates C structures that are compatible with pTAL and TAL structures.

```
CFIELDALIGN_MATCHED2
```

C output is generated starting on an odd byte for:

- A structure that contains a substructure beginning on an odd-byte boundary
- A structure that contains a substructure ending on an odd-byte boundary, followed by a user-defined item that the DDL compiler would allocate starting on that odd byte

The DDL compiler allocates data starting on an odd byte for character data only.

The CFIELDALIGN_MATCHED2 command allows members of a structure to be assigned a byte or word address consecutively. If the remaining byte in a two-byte word is not large enough for the next member, then the DDL compiler assigns the next word aligned address. This guideline also applies to substructures that are declared inline, using the first member of the substructure.

The DDL compiler word-aligns substructures that refer to other group definitions, and makes their length even.

An item following a referenced `struct` must be word aligned. If the referenced struct has an odd byte length, the DDL compiler adds a filler to the dictionary after the reference to the struct.

The DDL compiler word-aligns substructures declared by template with a `typedef` or structure tag. The DDL compiler adds a filler if needed to word align a referenced definition.

The C source code for a struct generated with CFIELDALIGN_MATCHED2 set is preceded by the statement:

```
#pragma fieldalign shared2 __struct-name
```

where *__struct-name* is the name of the struct.

When generating C source with MATCHED2 alignment, the DDL compiler allows substructures to start on an odd-byte boundary. Without matched2 alignment, C substructures must start on a word boundary.

The DDL source code for a definition or record compiled with matched2 alignment is preceded by the command CFIELDALIGN_MATCHED2.

Pascal (on D-series systems) does not support the CFIELDALIGN_MATCHED2 command.

The C00CALIGN and CFIELDALIGN_MATCHED2 commands cannot be in effect at the same time. The DDL compiler uses the value of the last command that was specified (C00CALIGN, NOC00CALIGN, or CFIELDALIGN_MATCHED2). C00CALIGN is the default.

To reset the CFIELDALIGN_MATCHED2 command, specify one of:

- The C00CALIGN command, which generates default C output.
- The NOC00CALIGN command, which does not generate C output for certain structures. For more information, see C00CALIGN on page 9-12.

When you set CFIELDALIGN_MATCHED2, the DDL compiler ignores any FILLER specification. Instead, the DDL compiler uses an extended FILLER 1 algorithm, adding fillers as described previously. For more information about FILLER 1, see FILLER on page 9-59.

To suppress the `#pragma` fieldalign matched2 statements, set the NOCPRAGMA command.

When compiling a definition or record with matched2 alignment, all referenced definitions must have been compiled with matched2 alignment; otherwise, the DDL compiler returns an error.

For more information about alignment rules, see Appendix H, DDL Alignment Rules for C.

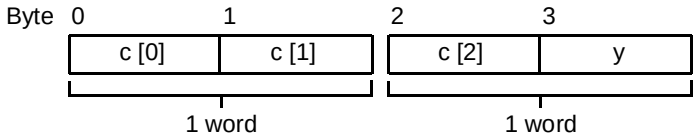
Example 9-5. CFIELDALIGN_MATCHED2 and C00CALIGN Commands

C Structure

```
struct s1
{
   struct
   {
      char c[3];
   } ss2;
   char y;
} s1;
```

Alignment of s1 with CFIELDALIGN_MATCHED2

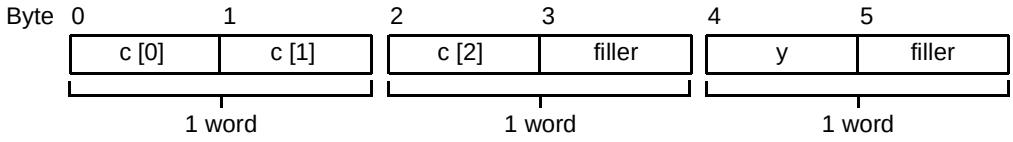
Members of s1 can start on odd-byte boundaries.



VST008.vsd

Alignment of s1 with C00CALIGN

All structures and substructures must begin and end on even-byte boundaries.



VST009.vsd

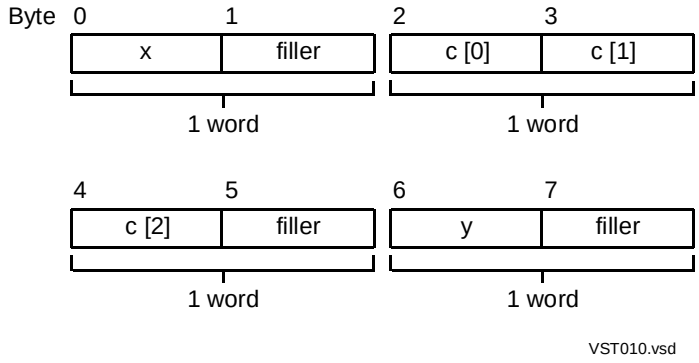
Example 9-6. CFIELDALIGN_MATCHED2 Command

C Structure

```
struct s3
{
   char x;
   struct
   {
      char c[3];
   } ss4;
   char y;
} s3;
```

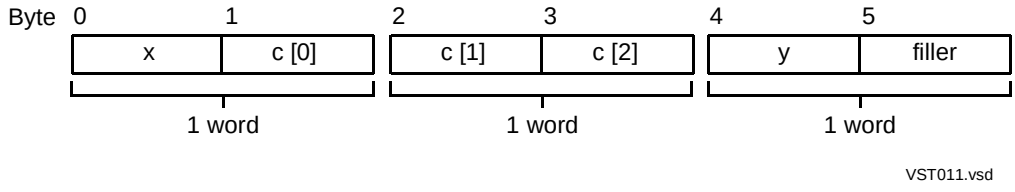
Alignment of s3 with Default Alignment

All structures and substructures must begin and end on even-byte boundaries.



Alignment of s3 with CFIELDALIGN_MATCHED2:

Members of the structure can start on odd-byte boundaries.



CIFDEF, CIFNDEF, and CENDIF

The CIFDEF, CIFNDEF, and CENDIF commands generate the compiler directives `#ifdef`, `#ifndef`, and `#endif`, respectively, for C output.

```
{ CIFNDEF } identifier_name
{ CIFDEF  }

CENDIF
```

identifier_name

is the name of the identifier affected by the command.

The DDL compiler does not store *identifier-name* in the dictionary.

Each CIFNDEF or CIFDEF command have a corresponding CENDIF command.

After closing a C source code file, the DDL compiler checks whether the CIFNDEF and CIFDEF commands match the CENDIF commands. If not, the DDL compiler issues a warning.

Before generating C output for CENDIF command, the DDL compiler checks for the corresponding CIFDEF or CIFNDEF command. If the DDL compiler does not find the corresponding command, then it issues a warning and does not produce output for the CENDIF command.

You can nest CIFNDEF and CIFDEF commands

Example 9-7. CIFNDEF, CIFDEF and CENDIF commands

```
10> DDL

!?C CSRC
!?CIFNDEF EMP
!CONSTANT EMP VALUE "JYOTI".
!?CIFDEF  EMP
!CONSTANT EMP VALUE "RAM".
!?NOC

!?C CSRC
Output source for C is opened on $ADE101.BUG.CSRC
!?CIFNDEF EMP
!CONSTANT EMP VALUE "JYOTI".
Constant EMP defined.
C output produced for EMP.
!?CIFDEF EMP
!CONSTANT EMP VALUE "RAM".
Constant EMP defined.
C output produced for EMP.
!?CENDIF
!?CENDIF
!?NOC
Output source for C is closed.

The 'C' source code file csrc contains the following.

/* SCHEMA PRODUCED DATE - TIME : 3/10/2000 - 19:39:53 */
#ifndef EMP
#pragma section emp
* Constant EMP created on 03/10/2000 at 19:40 */
#define EMP "JYOTI"
#ifdef EMP
#pragma section emp
/* Constant EMP created on 03/10/2000 at 19:41 */
#define EMP "RAM"
#endif
#endif
```

CLISTIN

The [NO]CLISTIN command includes [excludes] subsequent user-defined dictionary comments in [from] the compiler listing.

```
[NO]CLISTIN
```

Default: CLISTIN

`CLISTIN`

includes subsequent user-defined dictionary comments in the compiler listing.

`NOCLISTIN`

excludes subsequent user-defined dictionary comments from the compiler listing.

You can suppress comments on the output listing with NOCLISTIN and subsequently resume listing comments with CLISTIN.

Regardless of the setting of CLISTIN, the compiler listing always includes any production comments. The DDL compiler generates production comments to provide such information as the total length of records and definitions and to document such compiler actions as adding a record to the dictionary.

CLISTIN and NOCLISTIN work independently of the COMMENTS and NOCOMMENTS commands that control output of comments to the dictionary and of the CLISTOUT and NOCLISTOUT commands that control reproduction of comments on source code files.

Example 9-8. CLISTIN and NOCLISTIN Commands (page 1 of 2)

DDL Input

`*Comment for AA.` `DEF aa PIC X(24).`	List commands by default.

`?NOCLISTIN`
`*Comment for BB`
`DEF bb PIC X(10).`
	Stop listing comments.

`?CLISTIN`
`*Comment for CC`
`DEF cc PIC 9(6).`
	Start listing comments again.

Example 9-8. CLISTIN and NOCLISTIN Commands (page 2 of 2)

DDL Compiler Listing

```
*Comment for AA                    Comment from source code file
DEF aa PIC X(24).
Definition AA size is 24 bytes.    Production comment

?NOCLISTIN
DEF bb PIC X(10).
Definition BB size is 10 bytes.    Production comment

?CLISTIN
*Comment for CC                    Comment from source code file
DEF cc PIC 9(6).
Definition CC size is 24 bytes.    Production comment
```

CLISTOUT

The CLISTOUT command includes user-defined dictionary comments in (or excludes them from) source code files.

```
{ [NO]CLISTOUT | CLISTOUTDETAIL }
```

Default: CLISTOUT

`CLISTOUT`

includes user-defined dictionary comments in source code files.

`CLISTOUTDETAIL`

includes in source code files any user-defined dictionary comments on referenced definitions that immediately precede the referring definition or record.

`NOCLISTOUT`

excludes user-defined dictionary comments from source code files.

CLISTOUT and CLISTOUTDETAIL reproduce comments only if the command COMMENTS on page 9-29 is also specified.

CLISTOUT reproduces comments on any open C, COBOL, DDL, FORTRAN, FUP, Pascal (on D-series systems), pTAL, or TAL source code file.

You can suppress comments with NOCLISTOUT and then enable them with a subsequent CLISTOUT.

CLISTOUT does not reproduce comments for referenced objects. To reproduce comments associated with definitions referenced by another definition or by a record, specify CLISTOUTDETAIL.

CLISTOUTDETAIL causes the DDL compiler to reproduce any comments previously associated with a referenced definition. If a definition or record refers to a definition that has a comment and CLISTOUTDETAIL is in effect, the DDL compiler reproduces the comment in the source code just before the referenced definition. CLISTOUTDETAIL does not reproduce comments for definitions referenced by a token map or a token type.

Even if NOCLISTOUT is specified, a DDL timestamp comment, preceded by an asterisk, is included before every definition and record in a source code file. You can suppress this comment with a NOTIMESTAMP command (see the TIMESTAMP on page 9-113).

Example 9-9. CLISTOUT, NOCLISTOUT and CLISTOUTDETAIL Commands

DDL Input

```
?DICT
?COMMENTS                    Add comments to dictionary.
?COBOL cobsrc
?NOTIMESTAMP                 Suppress timestamp.
?NOCLISTOUT                  Suppress comments.
*Comment for aa
DEF aa PIC X(8).             Start including comments again.
?CLISTOUT
*Comment for bb
DEF bb PIC 9(6).
?CLISTOUTDETAIL              Include comments for aa and bb as well as yy.
*Comment for yy
DEF yy
  02 y1 TYPE aa.
  02 y2 TYPE bb.
END
?NOCLISTOUT                  Stop including comments.
```

DDL Output (COBOL Code)

```
?SECTION AA,TANDEM
  01 AA                                PIC X(8).
?SECTION BB,TANDEM
* comment for bb
  01 BB                                PIC 9(6).
?SECTION YY,TANDEM
* comment for yy
  01 YY.
* comment for aa
    02 Y1                              PIC X(8).
* comment for bb
    02 Y2                              PIC 9(6).
```

COBCHECK

The [NO]COBCHECK command performs [suppresses] COBOL syntax checks on subsequent DDL object definitions without generating code.

```
[NO]COBCHECK
```

Default: COBCHECK if a COBOL source code file is open, otherwise NOCOBCHECK

`COBCHECK`

performs COBOL syntax checks as though COBOL source code were being produced.

`NOCOBCHECK`

suppresses COBOL syntax checks.

If a COBOL source code file is open, the compiler performs the COBOL checks whether or not COBCHECK is set.

You can stop COBOL syntax checking by specifying a NOCOBCHECK command; you can restart checking with a subsequent COBCHECK.

The DDL compiler does not perform the lengthy syntax testing performed by the COBOL compiler. The DDL compiler tests the DDL statements to ensure that they follow the rules specified by COBOL:

- The number of alphabetic characters in the PICTURE literal cannot exceed 30 ASCII characters.
- The maximum numeric PICTURE size is 18 words.
- An elementary or group field with either an OCCURS or OCCURS DEPENDING ON clause cannot be redefined by another field or group.
- An elementary or group field with a REDEFINES clause cannot be larger than the field or group it redefines.
- COBOL reserved words cannot be used as DDL names.
- The object does not contain any of these unsupported types:
  - TYPE BINARY 8
  - TYPE FLOAT
  - TYPE COMPLEX
  - TYPE LOGICAL
- A TYPE BINARY 64 declaration cannot specify a scale factor of -18 (or less); the range is restricted to -17 through 18.
- A data item must not have the same name as a group or record that can be used to qualify the data item.

- No more than 3 levels of nested OCCURS can be in a COBOL data-description entry.
- An item specified as a key in a RECORD statement must be alphanumeric. To use a numeric field as a key, enclose it within a group and specify the group as the key; a group's data type is assumed to be alphanumeric regardless of the data types of its member fields.

Example 9-10. COBCHECK and NOCOBCHECK Commands

```
?COBCHECK
RECORD customer.
FILE IS "$data.sales.customer" KEY-SEQUENCED.
  02 custnum PIC S9(4) KEYTAG 0.
  02 custname PIC X(18) KEYTAG "cn".
  02 custaddr TYPE addr.
END
Record CUSTOMER size is 70 bytes.
*** WARNING *** COBOL OUTPUT DIAGNOSTICS:
*** ERROR *** Non-alphanumeric key element - CUSTNUM
?NOCOBCHECK
```

When COBCHECK is in effect, the DDL compiler issues this message for each DDL object statement that passes the syntax check:

```
COBOL CHECK completed for name
```

COBLEVEL

The COBLEVEL command specifies a level-numbering scheme for COBOL output.

```
COBLEVEL [ base [ , increment ] ]
```

base

is the starting level number.

Default: 1

increment

is the number of levels to skip.

Default: 1

The formula for calculating COBLEVEL level numbers is:

```
cobol-level := base + (increment * ( level - 1 ))
```

Here, *level* is the level number of the item within the dictionary; it can be any value from 0 to 49.

The COBLEVEL level numbers are used only for COBOL output; the DDL compiler does not keep these level numbers in the dictionary. If you need to rebuild your COBOL source code files and want to keep the COBLEVEL level numbers, you must specify the COBLEVEL command before issuing the OUTPUT statement.

Example 9-11. COBLEVEL Command

DDL Input

```
?COBLEVEL 5,3
DEF aa.
02 bb.
03 cc  PIC X.
END
```

DDL Output (COBOL Code)

```
05 AA.
  08 BB.
    11 CC           PIC X.
```

COBOL

The [NO]COBOL command:

- Opens [closes] a COBOL source code file
- Starts [stops] writing translated DDL object definitions to the COBOL source code file

```
{ COBOL [ cobol-source-file [ ! ] ] }
{ NOCOBOL                           }
```

Default: NOCOBOL

COBOL

closes any open COBOL source code file, opens *cobol-source-file*, translates subsequent DDL objects defined by statements or specified in OUTPUT statements to COBOL source code statements, and writes the COBOL source code statements to *cobol-source-file*.

cobol-source-file

is the name of the COBOL source code file to be created, if necessary, and opened. The file must be one of:

- EDIT file
- Unstructured file
- Sequential device (such as a terminal, spooler, or process)

If the file exists but is not one of these types, the DDL compiler issues an error message and does not open the file.

If *cobol-source-file* is an EDIT file, and it exceeds 99,999 lines, the DDL compiler issues FILE ERROR - filename - Edit file line number too large (537) on page A-17.

Default: home terminal

!

purges the contents of *cobol-source-file* before opening it, if it exists. If *cobol-source-file* does not exist, the exclamation point has no effect.

Without the exclamation point, the DDL compiler appends the new COBOL source code statements to the end of *cobol-source-file*.

NOCOBOL

closes any open COBOL source code file and stops translating DDL object definitions to COBOL source code statements.

For the data types that the DDL compiler generates for COBOL, see Table C-2 on page C-3.

The compiler can translate DDL objects specified in an OUTPUT statement only if the dictionary containing these structures is open.

Only one COBOL source code file can be open at a time. If you use the COBOL command when you already have a COBOL source code file open, the DDL compiler closes the current source code file before opening the new source code file.

The specified COBOL source code file can be an EDIT file, an unstructured file, or a sequential device such as a terminal, a spooler, or a process. If the file exists but is not one of these types, the DDL compiler issues an error message and does not open the file.

If the COBOL source code file already exists and the exclamation point is omitted, the DDL compiler appends the generated source code to the end of the file's original contents. The DDL compiler does not replace any existing structures.

Each DDL object translated to COBOL source code is written to the source code file in a separate section with the same name as the DDL object it contains. You can suppress the default section headings with the SETSECTION command.

The DDL compiler performs all of the syntax checks listed under the COBCHECK command before writing source output. If it finds a syntax error, the DDL compiler issues an error message and does not write the source output for the DDL object statement with the error; it does write source output for the DDL object if only a warning is issued.

In Example 9-12 on page 9-27, the DDL compiler opens a COBOL source code file, COBSRC, on the subvolume `$data.sales` and adds the definition for `name` to that file. COBOL does not recognize the UPSHIFT clause and is not included in the COBOL source code.

Example 9-12. COBOL Command (page 1 of 2)

DDL Input

```
?COBOL $data.sales.cobsrc
DEF name.
  02 last-name PIC X(12)
               UPSHIFT.
  02 first-name PIC X(8)
               UPSHIFT.
  02 midinit PIC X(2)
               UPSHIFT.
END
```

Example 9-12. COBOL Command (page 2 of 2)

DDL Output (COBOL Code)

```
?SECTION NAME,TANDEM
  01 NAME.
    02 LAST-NAME                    PIC X(12).
    02 FIRST-NAME                   PIC X(8).
    02 MIDINIT                      PIC X(2).
```

In Example 9-13 on page 9-28, the DDL compiler retrieves the record, CUSTOMER, from the open dictionary, translates it to COBOL source code, and appends the source code to the open COBOL file.

Example 9-13. COBOL Command

DDL Input

```
28> DDL                             Run DDL compiler.
!?DICT $data.sales                  Open dictionary.
!?COBOL $data.sales.cobsrc
!?OUTPUT RECORD customer.           Append customer record to COBSRC.
!EXIT
```

DDL Output (COBOL Code)

```
?SECTION CUSTOMER,TANDEM
  01 CUSTOMER.
    02 CUSTNUM                      PIC X(4).
    02 CUSTNAME.
      03 LAST-NAME                  PIC X(12).
      03 FIRST-NAME                 PIC X(8).
      03 MIDINIT                    PIC XX.
    02 CUSTADDR.
      03 ADDRESS                    PIC X(22).
      03 CITY                       PIC X(14).
      03 STATE                      PIC X(2).
      03 ZIP                        PIC 9(5).
```

For the DDL definition of the CUSTOMER record, see the sample database schema in Appendix B, Sample Schemas.

COLUMNS

The COLUMNS command specifies the number of significant columns (character positions) on DDL input lines.

```
COLUMNS num
```

num

is an integer from 12 through 132 that specifies the number of significant columns (character positions) on DDL input lines.

Default: 132

Changing the value of COLUMNS also changes the maximum string length for DDL.

COMMENTS

The [NO]COMMENTS command includes [excludes] subsequent user-defined dictionary comments in [from] the open dictionary.

```
[NO]COMMENTS
```

Default: NOCOMMENTS

COMMENTS

includes subsequent user-defined dictionary comments in the open dictionary.

NOCOMMENTS

excludes subsequent user-defined dictionary comments from the open dictionary.

After comments have been stored in the dictionary, they can be selectively passed to any open DDL, REPORT, or host-language source code files with the CLISTOUT command.

The comments associated with an object can be more than one line long.

Any comments that immediately precede the definition of an object are associated with that object.

Any comments that immediately precede the definition of an element in a group definition or record are associated with that element.

For more information on comments, see CLISTOUT on page 9-21, and Comments on page 2-12.

In Example 9-14 on page 9-30, the two comment lines preceding the group definition of NAME are stored as a single comment associated with NAME in the open dictionary, and three starred comment lines are each associated with an element within the group definition NAME. The comments are also written to the open COBOL source code file COBSRC (preceding NAME and CUSTNAME), where they are inherited by the definition CUSTNAME, which refers to NAME.

Example 9-14. COMMENTS Command

DDL Input

```
?DICT
?COBOL cobsrc
?COMMENTS
?CLISTOUT
?NOTIMESTAMP
* An expanded name in the following sequence:
* Last name, First name, Middle initial
DEF name.
* Last name
  02 last-name     PIC X(12).
* First name
  02 first-name    PIC X(8).
* Middle initial
  02 midinit       PIC X(2).
END

DEF custname       TYPE name.
```

DDL Output (COBOL Code)

```
* An expanded name in the following sequence:
* Last name, First name, Middle initial
  01 NAME.
* Last name
    02 LAST-NAME      PIC X(12).
* First name
    02 FIRST-NAME     PIC X(8).
* Middle initial
    02 MIDINIT        PIC X(2).
* An expanded name in the following sequence:
* Last name, First name, Middle initial
  01 CUSTNAME.
* Last name
    02 LAST-NAME      PIC X(12).
* First name
    02 FIRST-NAME     PIC X(8).
* Middle initial
    02 MIDINIT        PIC X(2).
```

In Example 9-15 on page 9-31, a comment on a TOKEN-TYPE statement is inherited in the C, COBOL, Pascal (on D-series systems), TACL, and pTAL or TAL source code generated for the token type.

Example 9-15. COMMENTS Command

DDL Input

```
?DICT
?COBOL
?TAL
?TACL
?COMMENTS
?CLISTOUT
?NOTIMESTAMP
*Token type for enumerated tokens
TOKEN-TYPE zspi-typ-enum      VALUE IS zspi-tdt-enum
                              DEF IS zspi-ddl-enum.
```

DDL Output (C Code)

```
/* Token type for enumerated tokens */
#pragma section zspi_typ_enum
#define ZSPI_TYP_ENUM 2818u
```

DDL Output (COBOL Code)

```
*Token type for enumerated tokens

    01 ZSPI-TYP-ENUM  NATIVE-2 VALUE IS 2818.
```

DDL Output (Pascal Code—D-Series Systems Only)

```
{ Token type for enumerated tokens }
?Section ZSPI_TYP_ENUM
CONST ZSPI_TYP_ENUM = 2818;
```

DDL Output (TACL Code)

```
?Section ZSPI-TYP-ENUM Struct

==Token type for enumerated tokens
BEGIN
UINT    ZSPI^TYP^ENUM IS 2818;
END;
```

DDL Output (pTAL or TAL Code)

```
!Token type for enumerated tokens
Literal ZSPI^TYP^ENUM = 11 '<' 8 + 2;
```

CPRAGMA

The [NO]CPRAGMA command includes [excludes] `#pragma`-generating code in C output.

△ **Caution.** The memory layout of the other machine might be different from the layout on a HP NonStop system.

```
[NO]CPRAGMA
```

Default: CPRAGMA

`CPRAGMA`

includes `#pragma`-generating code in C output.

`NOCPRAGMA`

encloses `#pragma`-generating code within the C comment characters. This allows you to use the C code on systems whose C compilers do not support `#pragmas`.

CTOKENMAP_ASDEFINE

The CTOKENMAP_ASDEFINE command generates TOKEN MAP output as `#define` statements [a `static int` array] in an open C source code file.

```
[NO]CTOKENMAP_ASDEFINE
```

Default: NOCTOKENMAP_ASDEFINE

`CTOKENMAP_ASDEFINE`

generates the C output of every subsequent TOKEN-MAP as a `#define`.

`NOCTOKENMAP_ASDEFINE`

generates the C output of every subsequent TOKEN-MAP as a `static int` array.

If the C output of a TOKEN-MAP as a #define exceeds one line, a continuation character "`\`" is appended to the end of all lines except the last one as required by the C syntax for a `#define`.

The rules governing CDEFINEUPPER and NOCDEFINEUPPER also apply to the C output of a TOKEN-MAP as a `#define`.

The CTOKENMAP_ASDEFINE and NOCTOKENMAP_ASDEFINE directives only affect the C output of a TOKEN-MAP. The output in any other language is unaffected. The generation of TOKEN-CODE and TOKEN-TYPE remain unchanged in all languages.

Example 9-16. CTOKENMAP_ASDEFINE Command (page 1 of 4)

```
$ADE101 JYOTI1 51> DDL
DDL Compiler T9100ABQ - (15NOV99)  SYSTEM \BOMBAY
COPYRIGHT TANDEM COMPUTERS INCORPORATED 1978, 1979, 1981, 1982, 1986-1999
!?DICT
Audited dictionary created on subvol $ADE101.JYOTI1.
Dictionary opened on subvol $ADE101.JYOTI1 for update access.

!?C
/* SCHEMA PRODUCED DATE - TIME : 8/02/2000 - 15:03:17 */
Output source for C is opened on $ZTN1.#PTPJHZ4
!?CTOKENMAP_ASDEFINE
!DEF EMP.
!02  F1 PIC XX.
!END.

Definition EMP size is 2 bytes.
Definition EMP added to dictionary.
#pragma section emp
/* Definition EMP created on 08/02/2000 at 15:03 */
#pragma fieldalign shared2 __emp
typedef struct __emp
{
   char                             f1[2];
} emp_def;

#define emp_def_Size 0
C output produced for EMP.
!TOKEN-MAP MAP1 VALUE 1 DEF EMP.
!VERSION "C00" FOR F1.
!END.
Token Map MAP1 defined.
Token Map MAP1 added to dictionary.
#pragma section map1
/* Token Map MAP1 created on 08/02/2000 at 15:03 */
#define MAP1 { 2303, 1, 2, 17152, 767 }
C output produced for MAP1.

!TOKEN-MAP MAP2 VALUE 20 DEF EMP.
!VERSION "C00" FOR F1.
!END.
Token Map MAP2 defined.
Token Map MAP2 added to dictionary.
#pragma section map2
/* Token Map MAP2 created on 08/02/2000 at 15:04 */
#define MAP2 { 2303, 20, 2, 17152, 767 }
C output produced for MAP2.

!DEF EMP1.
!02  F2  PIC X(500).
!END.
Definition EMP1 size is 500 bytes.
Definition EMP1 added to dictionary.
#pragma section emp1
/* Definition EMP1 created on 08/02/2000 at 15:05 */
#pragma fieldalign shared2 __emp1
typedef struct __emp1
{
   char                             f2[500];
} emp1_def;
```

Example 9-16. CTOKENMAP_ASDEFINE Command (page 2 of 4)

```
#define emp1_def_Size 0
C output produced for EMP1.
!TOKEN-MAP MAP3 VALUE 150 DEF EMP1.
!VERSION "C00" FOR F2.
!END.
Token Map MAP3 defined.
Token Map MAP3 added to dictionary.
#pragma section map3
/* Token Map MAP3 created on 08/02/2000 at 15:05 */
#define MAP3 { 2303, 150, 500, 17152, -1, -2561 }
C output produced for MAP3.

!?CDEFINEUPPER
!OUTPUT *.
Loading Definition EMP
#pragma section emp
/* Definition EMP created on 08/02/2000 at 15:03 */
#pragma fieldalign shared2 __emp
typedef struct __emp
{
   char                             f1[2];
} emp_def;

#define emp_def_Size 0
C output produced for EMP.
Loading Token-Map MAP1
#pragma section map1
/* Token Map MAP1 created on 08/02/2000 at 15:03 */
#define MAP1 { 2303, 1, 2, 17152, 767 }
C output produced for MAP1.

Loading Token-Map MAP2
#pragma section map2
/* Token Map MAP2 created on 08/02/2000 at 15:04 */
#define MAP2 { 2303, 20, 2, 17152, 767 }
C output produced for MAP2.
Loading Definition EMP1
#pragma section emp1
/* Definition EMP1 created on 08/02/2000 at 15:05 */
#pragma fieldalign shared2 __emp1
typedef struct __emp1
{
   char                             f2[500];
} emp1_def;
#define emp1_def_Size 0
C output produced for EMP1.

Loading Token-Map MAP3
#pragma section map3
/* Token Map MAP3 created on 08/02/2000 at 15:05 */
#define MAP3 { 2303, 150, 500, 17152, -1, -2561 }
C output produced for MAP3.
!?NOCDEFINEUPPER
!OUTPUT *.
Loading Definition EMP
#pragma section emp
/* Definition EMP created on 08/02/2000 at 15:03 */
#pragma fieldalign shared2 __emp
typedef struct __emp
{
   char                             f1[2];
} emp_def;
#define emp_def_Size 0
C output produced for EMP.
```

Example 9-16. CTOKENMAP_ASDEFINE Command (page 3 of 4)

```
Loading Token-Map MAP1
#pragma section map1
/* Token Map MAP1 created on 08/02/2000 at 15:03 */
#define map1 { 2303, 1, 2, 17152, 767 }
C output produced for MAP1.

Loading Token-Map MAP2
#pragma section map2
/* Token Map MAP2 created on 08/02/2000 at 15:04 */
#define map2 { 2303, 20, 2, 17152, 767 }
C output produced for MAP2.

Loading Definition EMP1
#pragma section emp1
/* Definition EMP1 created on 08/02/2000 at 15:05 */
#pragma fieldalign shared2 __emp1
typedef struct __emp1
{
   char                             f2[500];
} emp1_def;
#define emp1_def_Size 0
C output produced for EMP1.

Loading Token-Map MAP3
#pragma section map3
/* Token Map MAP3 created on 08/02/2000 at 15:05 */
#define map3 { 2303, 150, 500, 17152, -1, -2561 }
C output produced for MAP3.

!?NOCTOKENMAP_ASDEFINE
!OUTPUT *.
Loading Definition EMP
#pragma section emp
/* Definition EMP created on 08/02/2000 at 15:03 */
#pragma fieldalign shared2 __emp
typedef struct __emp
{
   char                             f1[2];
} emp_def;
#define emp_def_Size 0
C output produced for EMP.

Loading Token-Map MAP1
#pragma section map1
/* Token Map MAP1 created on 08/02/2000 at 15:03 */
static short   map1[] = { 2303, 1, 2, 17152, 767 };
C output produced for MAP1.

Loading Token-Map MAP2
#pragma section map2
/* Token Map MAP2 created on 08/02/2000 at 15:04 */
static short   map2[] = { 2303, 20, 2, 17152, 767 };
C output produced for MAP2.

Loading Definition EMP1
#pragma section emp1
/* Definition EMP1 created on 08/02/2000 at 15:05 */
#pragma fieldalign shared2 __emp1
typedef struct __emp1
{
   char                             f2[500];
} emp1_def;
#define emp1_def_Size 0
C output produced for EMP1.
```

Example 9-16. CTOKENMAP_ASDEFINE Command (page 4 of 4)

```
Loading Token-Map MAP3
#pragma section map3
/* Token Map MAP3 created on 08/02/2000 at 15:05 */
static short   map3[] = { 2303, 150, 500, 17152, -1, -2561 };
C output produced for MAP3.
!
```

CUNDEF

The CUNDEF command generates a `#undef` compiler directive for C output.

```
CUNDEF identifier_name
```

identifier_name

is the name of the identifier affected by the `#undef` directive.

The DDL compiler generates a `#undef` statement in C output without checking whether the identifier name was previously defined.

It is your responsibility to use proper identifiers with the CUNDEF command.

Example 9-17. CUNDEF Command (page 1 of 2)

DDL Input

```
11> DDL
!?C CSRC
!CONSTANT EMP VALUE "JYOTI".
!?CIFDEF  EMP
!?CUNDEF  EMP
!?CENDIF
!?NOC

!?C CSRC
Output source for C is opened on $ADE101.BUG.CSRC
!CONSTANT EMP VALUE "JYOTI".
Constant EMP defined.
C output produced for EMP.

!?CIFDEF EMP
!?CUNDEF EMP
!?CENDIF
!?NOC
Output source for C is closed.
```

Example 9-17. CUNDEF Command (page 2 of 2)

DDL Output (C Code)

```
/* SCHEMA PRODUCED DATE - TIME : 3/10/2000 - 20:05:28 */
#pragma section emp
/* Constant EMP created on 03/10/2000 at 20:05 */
#define EMP "JYOTI"
#ifdef EMP
#undef EMP
#endif
```

C_DECIMAL

The [NO]C_DECIMAL command generates decimal [`char`] output for subsequent C simple numeric items.

```
[NO]C_DECIMAL
```

Default: NOC_DECIMAL

`C_DECIMAL`

generates decimal output for subsequent C simple numeric items.

`NOC_DECIMAL`

generates `char` output for subsequent C simple numeric items.

The C_DECIMAL command is used to produce decimal values in C output for simple numeric fields.

For computational numeric fields, the DDL compiler ignores the effect of the C_DECIMAL command.

Example 9-18. C_DECIMAL and NOC_DECIMAL Commands (page 1 of 3)

```
12> DICT
!DEF    EMP.
!02     ITEM1   PIC 9(5).
!02     ITEM2   PIC 9(6).
!END.
```

Example 9-18. C_DECIMAL and NOC_DECIMAL Commands (page 2 of 3)

```
!?C
!OUTPUT *.
!?C_DECIMAL
!OUTPUT *.
!?NOC_DECIMAL
!OUTPUT *.
!DEF     EMP1.
!02      ITEM1   PIC 9(5).
!02      ITEM2   PIC 9(5)   COMP.
!END.
!?C_DECIMAL
!OUTPUT  DEF EMP1.

!?DICT
Audited dictionary created on subvol $ADE101.BUG.
Dictionary opened on subvol $ADE101.BUG for update access.
!DEF  EMP.
!02   ITEM1   PIC 9(5).
!02   ITEM2   PIC 9(6).
!END.
Definition EMP size is 11 bytes.
Definition EMP added to dictionary.

!?C
 /* SCHEMA PRODUCED DATE - TIME : 3/06/2000 17:14:32 */
Output source for C is opened on $ZTN0.#PTS3ZAW
!OUTPUT *.
Loading Definition EMP
#pragma section emp
/* Definition EMP created on 03/06/2000 at 17:13 */
#pragma fieldalign shared2 __emp
typedef struct __emp
{
char                            item1[5];
char                            item2[6];
} emp_def;
C output produced for EMP.

!?C_DECIMAL
!OUTPUT *.
Loading Definition EMP
#pragma section emp
/* Definition EMP created on 03/06/2000 at 17:13 */
#pragma fieldalign shared2 __emp
typedef struct __emp
{
decimal                         item1[5];
decimal                         item2[6];
} emp_def;
C output produced for EMP.
```

Example 9-18. C_DECIMAL and NOC_DECIMAL Commands (page 3 of 3)

```
!?NOC_DECIMAL
!OUTPUT *.
Loading Definition EMP
#pragma section emp
/* Definition EMP created on 03/06/2000 at 17:13 */
#pragma fieldalign shared2 __emp
typedef struct __emp
{
char                            item1[5];
char                            item2[6;
} emp_def;

C output produced for EMP.
!DEF EMP1.
!02  ITEM1  PIC 9(5).
!02  ITEM2  PIC 9(5)   COMP.
!END.
Filler emitted at level 2 after ITEM1
Definition EMP1 size is 10 bytes.
Definition EMP1 added to dictionary.
#pragma section emp1

/* Definition EMP1 created on 03/06/2000 at 17:24 */
#pragma fieldalign shared2 __emp1
typedef struct __emp1
{
char                            item1[5];
unsigned long                   item2;
} emp1_def;
C output produced for EMP1.

!?C_DECIMAL
!OUTPUT DEF EMP1.
Loading Definition EMP1
#pragma section emp1
/* Definition EMP1 created on 03/06/2000 at 17:24 */
#pragma fieldalign shared2 __emp1
typedef struct __emp1
{
decimal                         item1[5];
unsigned long                   item2;
} emp1_def;
C output produced for EMP1.
!
```

C_MATCH_HISTORIC_TAL

The [NO]C_MATCH_HISTORIC_TAL command generates [suppresses] C data structures that are equivalent to pTAL, TAL, and COBOL data structures.

```
[NO]C_MATCH_HISTORIC_TAL
```

Default: NOC_MATCH_HISTORIC_TAL

`C_MATCH_HISTORIC_TAL`

generates C data structures that start on odd bytes (equivalent to TAL and COBOL data structures) for:

- Any structure that contains a substructure beginning on an odd byte boundary
- Any structure that contains a substructure ending on an odd-byte boundary, followed by a user-defined item that the DDL compiler would allocate starting on the odd byte

The DDL compiler allocates data starting on an odd byte for character data only.

The source code for a generated `struct` is preceded by the statement:

`#pragma fieldalign shared2 __`*`struct-name`*

`NOC_MATCH_HISTORIC_TAL`

resets the option.

The C_MATCH_HISTORIC_TAL command does not affect the dictionary.

The C_MATCH_HISTORIC_TAL command allows members of structures to be aligned to a byte or word boundary. If the remaining byte in a two-byte word is not large enough for the next member, then the DDL compiler assigns the next word aligned address. This guideline also applies to substructures that are declared inline, using the first member of the substructure.

The DDL compiler word-aligns substructures declared by template with a `typedef` or `structure` tag. The DDL compiler adds a filler if needed to word-align a referenced definition.

The DDL compiler word-aligns substructures that refer to other group definitions to make their length even.

The C source code for a struct generated with matched2 alignment set is preceded by the statement:

#pragma fieldalign matched2 __*struct-name*

where __*struct-name* is the name of the struct.

To suppress the `#pragma fieldalign matched2` statements, set the NOCPRAGMA command.

Example 9-19 on page 9-41 shows the C source generated for the given DDL source with C_MATCH_HISTORIC_TAL in effect which allows substructures to start and end on odd-byte boundaries. If the C_MATCH_HISTORIC_TAL command is not in effect, C source will not be generated for `def f` because substructure `j` starts on an odd-byte boundary. The DDL compiler emits a filler at level 2 after `k` because the following data items will not fit in the remaining byte.

Example 9-19. C_MATCH_HISTORIC_TAL Command (page 1 of 2)

DDL Input

```
def a.
  02 b type character 1.
  02 c type character 1.
  02 d type character 1.
end.

def e type character 1.

def f.
  02 g type binary 16.
  02 h.
     03 i type e.
     03 j type a.
  02 k type character 1.
  02 l type binary 16.
end.
```

Example 9-19. C_MATCH_HISTORIC_TAL Command (page 2 of 2)

DDL Output (C Code)

```
/* SCHEMA PRODUCED DATE - TIME :10/13/1995 13:23:16 */
#pragma section a
/* Definition A created on 10/13/1995 at 13:23 */
#pragma fieldalign shared8 __a
typedef struct __a
{
   char                              b;
   char                              c;
   char                              d;
   char                              filler_0;
} a_def;
#pragma section e
/* Definition E created on 10/13/1995 at 13:23 */
typedef char                                e_def;
#pragma section f
/* Definition F created on 10/13/1995 at 13:23 */
#pragma fieldalign shared8 __f
typedef struct __f
{
   short                             g;
   struct
   {
      e_def                             i;
      char                              filler_0;
      a_def                             j;
   } h;
   char                              k;
   char                              filler_1;
   short                             l;
} f_def;
```

DDL

The [NO]DDL command:

- Opens [closes] a DDL schema file
- Starts [stops] writing subsequent DDL object definitions to the open DDL schema file

```
{ DDL [ ddl-source-file [ ! ] ] }
{ NODDL                         }
```

Default: NODDL

DDL

closes any open DDL source code file, opens *ddl-source-file*, translates subsequent DDL objects defined by statements or specified in OUTPUT statements to DDL source code statements, and writes the DDL source code statements to *ddl-source-file*.

ddl-source-file

is the name of the DDL source code file to be created, if necessary, and opened. The file must be one of:

- EDIT file
- Unstructured file
- Sequential device (such as a terminal, spooler, or process)

If the file exists but is not one of these types, the DDL compiler issues an error message and does not open the file.

If *ddl-source-file* is an EDIT file, and it exceeds 99,999 lines, the DDL compiler issues FILE ERROR - filename - Edit file line number too large (537) on page A-17.

Default: home terminal

!

purges the contents of *ddl-source-file* before opening it, if it exists. If *ddl-source-file* does not exist, the exclamation point has no effect.

Without the exclamation point, the DDL compiler appends the new DDL source code statements to the end of *ddl-source-file*.

NODDL

closes any open DDL source code file and stops translating DDL object definitions to DDL source code statements.

The compiler can translate DDL objects specified in an OUTPUT statement only if the dictionary containing these structures is open.

Only one DDL source code file can be open at a time. If you use the DDL command when you already have a source code file open, the DDL compiler closes the current source code file before opening the new source code file.

If the DDL source code file already exists and the exclamation point is omitted, the DDL compiler appends the DDL object definitions to the end of the file's original contents. The DDL compiler does not replace any existing structures in the DDL source code file.

The specified DDL source code file must be an EDIT file, an unstructured file, or a sequential device such as a terminal, a spooler, or a process. If the file exists but is not one of these types, the DDL compiler issues an error message and does not open the file.

Each DDL object translated to DDL source is written to the source code file in a separate section that has the same name as the DDL structure it contains. You can suppress the individual section headings with the SETSECTION command.

For dictionaries created from the DDL compiler, the DDL source is almost identical to the original schema at the time the dictionary was created, and can be used to rebuild a dictionary.

△ **Caution.** Do not attempt to rebuild a dictionary installed by the Pathmaker product from DDL source code; Pathmaker application design information will be lost.

If the file \DALLAS.$DATA.SALES.DDLSRC already exists, the exclamation point in the following line of code directs the DDL compiler to purge the contents of the file before opening it. If the file does not exist, the DDL compiler creates a new file with the specified name.

```
?DDL \dallas.$data.sales.ddlsrc !
```

In Example 9-20 on page 9-44, the compiler writes all the definitions and records from the open dictionary to the DDL source code file, DDLSRC, first purging any data in that file. Assume that the dictionary:

- Does not contain Pathmaker information.
- Contains one record, CUSTOMER, and all definitions necessary to build that record.

The DDL source code file can be used to reconstruct the dictionary on another subvolume:

Example 9-20. DDL Command (page 1 of 2)

DDL Input

```
29> DDL
!?DICT $data.sales      Open existing dictionary.
!?DDL ddlsrc !          Clear and open DDL source code file
!OUTPUT *.              Write all definitions and records to DDL source code file.
```

Example 9-20. DDL Command (page 2 of 2)

DDLSRC

```
?Section NAME
Definition NAME.
  02 LAST-NAME                          Pic "X(12)"
                                        UPSHIFT.
  02 FIRST-NAME                         Pic "X(8)"
                                        UPSHIFT.
  02 MIDINIT                            Pic "X(2)"
                                        UPSHIFT.
  End

?Section ADDR
Definition ADDR.
  02 ADDRESS                            Pic "X(22)".
  02 CITY                               Pic "X(14)".
  02 STATE                              Pic "X(2)".
  02 ZIP-CODE                           Pic "9(5)".
  End

?Section CUSTNUM
Definition CUSTNUM                      Pic "X(4)".

?Section CUSTOMER
Record CUSTOMER.
  File is "CUSTOMER" Key-sequenced.
  02 CUSTNUM                            Type *.
  02 CUSTNAME                           Type NAME.
  02 CUSTADDR                           Type ADDR.
  Key is CUSTNUM Duplicates not allowed.
  Key "CN" is CUSTNAME.
  End
```

Reconstructing the Dictionary on Another Subvolume

```
>30
!?DICT $data.backup !  Open and clear dictionary.
!?SOURCE ddlsrc        Write source to dictionary.
!EXIT
```

DEFLIST

The DEFLIST command includes in (or excludes from) the compiler listing a description of each definition that is referenced by a DEFINITION or RECORD statement.

```
[NO]DEFLIST
```

Default: NODEFLIST

DEFLIST

includes in the compiler listing the level number, name, size, and byte offset of definitions referenced by a DEFINITION or RECORD statement.

NODEFLIST

excludes from the the compiler listing the level number, name, size, and byte offset of definitions referenced by a DEFINITION or RECORD statement.

DEFLIST can be useful when a referenced definition is included in generated source code files.

The DEFLIST description appears in either of these formats:

level-number field-name (*offset*:*length*) [*min*:*max*]

level-number field-name (*offset*:*length*)

depending on whether a minimum and maximum (or total) number of occurrences have been defined.

Variable	Value
level-number	Level number assigned to the field in the referring structure.
field-name	Name of the included field or group.
offset	Starting byte position of the field or group within the referring structure.
length	Length of the field in bytes.
min	Minimum number of occurrences for OCCURS DEPENDING ON or the total number for OCCURS.
max	Maximum number of occurrences for OCCURS DEPENDING ON or the total number for OCCURS.

For users of SPI, DEFLIST can help you understand ZSPI-DDL-PARM-ERR.Z-OFFSET, the definition that provides the byte offset within a structure that is in error. DEFLIST shows where the error is.

In Example 9-21 on page 9-46, assume that the record CUSTOMER indirectly refers to three definitions–CUSTNUM, NAME, and ADDR–through the definition CUSTINFO. (These definitions are in the sample database schema in Appendix B, Sample Schemas.)

Example 9-21. DEFLIST Command (page 1 of 2)

Definition in Dictionary

```
DEF variable-table.
  02 table-size   TYPE BINARY 16.
  02 data-table   TYPE BINARY 32
                  OCCURS 1 TO 100 TIMES DEPENDING ON table-size.
END
```

Example 9-21. DEFLIST Command (page 2 of 2)

DEFLIST Command Output

```
!?DEFLIST
!?OUTPUT DEF variable-table
```

`Loading Definition VARIABLE-TABLE`	Table starts at byte 0 with a maximum length of 402 bytes
`Including 01 VARIABLE-TABLE (0:42)`	Element starts at byte 0 and has a length of 2 bytes
`Including 02 TABLE-SIZE (2:4)[1:100]`	Element starts at byte 2, has a length of 4 bytes, and occurs 1 to 100 times
`!?DEFLIST` `!OUTPUT RECORD customer`	Include descriptions of referenced definitions in the listing
`Loading Record CUSTOMER`	
`Including: 01 CUSTINFO (0:69)`	69 bytes starting at 0
`Including: 02 CUSTNUM (0:4)`	4 bytes starting at 0
`Including: 03 LAST-NAME (4:12)`	12 bytes starting at 4
`Including: 03 FIRST-NAME (16:8)`	8 bytes starting at 16
`Including: 03 MIDINIT (24:2)`	2 bytes starting at 24
`Including: 02 ADDR (26:43)`	43 bytes starting at 26
`Including: 03 ADDRESS (26:22)`	22 bytes starting at 26
`Including: 03 CITY (48:14)`	2 bytes starting at 62
`Including: 03 STATE (62:2)`	5 bytes starting at 64
`Including: 03 ZIP-CODE (64:5)`	

DICT

The DICT command:

- Opens [closes] a dictionary
- Starts [stops] writing object definitions to the dictionary

```
{ DICT [ dict-subvol-name ] [ ! ] }
{ NODICT                          }
```

Default: NODICT

`DICT`

closes any open dictionary, opens a dictionary on *`dict-subvol-name`* (creating it if it does not exist), and writes subsequent object definitions to that dictionary.

Note. The DDL compiler creates an audited dictionary only if the subvolume is audited. A dictionary on a nonaudited subvolume is also nonaudited.

dict-subvol-name

is the name of the dictionary subvolume, which has this form:

[*node-name*.][$*volume-name*.][*subvolume-name*]

Syntax Element	Default
dict-subvol-name	Current system, volume, and subvolume
node-name	Current system
volume-name	Current volume
subvolume-name	Current subvolume

!

purges existing dictionary files in *dict-subvol-name* and creates new dictionary files there. The new dictionary files have the same extent sizes, MAXEXTENTS value, security, and ownership as the purged dictionary files had.

Note. If you do not have purge access to the original dictionary files, the DDL compiler does not execute the command DICT !.

If *dict-subvol-name* has no dictionary files, the exclamation point has no effect.

NODICT

closes any open dictionary.

The dictionary consists of 14 files with predefined file names. For this reason, any given subvolume can contain only one dictionary.

If the specified subvolume does not exist, the DDL compiler creates and opens the dictionary on the new subvolume. If the subvolume exists, but does not contain a dictionary, the DDL compiler creates and opens a dictionary on the specified subvolume.

If a dictionary already exists on the specified subvolume, the DDL compiler opens the dictionary for update access. More than one user can open the dictionary for concurrent update access.

If a dictionary already exists, you can either:

- Purge the dictionary and re-create it by specifying an exclamation point after the subvolume name.
- Add new DDL objects to the existing dictionary by omitting the exclamation point.

For a Pathmaker dictionary, DICT! deletes only DDL objects, not Pathmaker objects (services, servers, requesters, and screens); Pathmaker objects can be modified or deleted only within the Pathmaker environment. If the Pathmaker dictionary is an earlier product version than your dictionary, the DDL compiler does not delete any Pathmaker objects.

The file security of the dictionary files is the default file security of whoever compiles the DDL source that creates the dictionary.

Only one dictionary can be open at a time. If you use the DICT command when you already have a dictionary open, the DDL compiler closes the current dictionary before opening the specified dictionary.

After a DICT or DICTN command creates a dictionary, using DICT or DICTN to open the dictionary has no effect on whether the dictionary is audited or not. Either command can open the dictionary, but the audited state of the dictionary does not change.

If an existing dictionary that you open with the DICT command is nonaudited, the DDL compiler issues a warning message.

To open a dictionary on the volume $DATA and subvolume SALES, enter:

```
?DICT $data.sales
```

If there is no dictionary on $DATA.SALES, the DDL compiler creates and then opens the dictionary.

To execute the DDL compiler interactively and then clear and open the dictionary on the current subvolume, you can enter:

```
31>DDL
?DICT !
```

Alternatively, you can include the DICT command when you run the DDL compiler:

```
32>DDL DICT
```

You can create a dictionary either by using the DICT command or by running the Pathmaker application systems generator. When you add a Pathmaker project, the Pathmaker software creates a dictionary for you.

More than one user can write to a dictionary at the same time, whether the Pathmaker tool or the DDL compiler created the dictionary.

DICTN

The DICTN command:

- Creates and opens a nonaudited dictionary or opens an existing dictionary
- Writes subsequent object definitions to the open dictionary

△ **Caution.** The TMF cannot monitor nonaudited files. The integrity of a nonaudited dictionary on an audited disk might be jeopardized if corruptions of the disk occur.

```
DICTN [ dict-subvol-name ] [ ! ]
```

dict-subvol-name

is the name of the dictionary subvolume, which has this form:

[*node-name*.][$*volume-name*.][*subvolume-name*]

Syntax Element	Default
dict-subvol-name	Current system, volume, and subvolume
node-name	Current system
volume-name	Current volume
subvolume-name	Current subvolume

!

purges existing dictionary files in *dict-subvol-name* and creates new dictionary files there. The new dictionary files have the same extent sizes, MAXEXTENTS value, security, and ownership as the purged dictionary files had.

Note. If you do not have purge access to the original dictionary files, the DDL compiler does not execute the command DICT !.

If *dict-subvol-name* has no dictionary files, the exclamation point has no effect.

If the dictionary is audited or was created by an older DDL product version, the DDL compiler deletes the dictionary and re-creates it as a nonaudited dictionary, provided the dictionary is not part of a Pathmaker catalog. The DDL compiler issues a warning message if it cannot re-create the dictionary as a nonaudited dictionary.

When used on an audited dictionary created for a Pathmaker application, this command purges only DDL files; it does not purge Pathmaker objects. If the dictionary is part of an older product version of the Pathmaker catalog, the DDL compiler cannot purge any objects from the dictionary.

The dictionary consists of 14 files with predefined file names. For this reason, any given subvolume can contain only one dictionary.

If the specified subvolume does not exist, the DDL compiler creates and opens the dictionary on the new subvolume.

If the subvolume exists but does not contain a dictionary, the DDL compiler creates and opens a dictionary on the specified subvolume.

If a dictionary already exists on the specified subvolume, the DDL compiler opens the dictionary for update access. More than one user can open the dictionary for concurrent update access.

If a dictionary already exists, you can either:

- Purge the dictionary and re-create it by specifying an exclamation point after the subvolume name.
- Add new DDL objects to the dictionary by omitting the exclamation point.

For a Pathmaker dictionary, DICTN! deletes only the dictionary, not Pathmaker objects (services, servers, requesters, and screens); Pathmaker objects can be modified or deleted only within the Pathmaker environment.

The file security of the dictionary files is the default file security of whoever compiles the DDL source code.

Only one dictionary can be open at a time. If you use the DICTN command when you already have a dictionary open, the DDL compiler closes the current dictionary before opening the specified dictionary.

After a DICTN or DICT command creates a dictionary, using DICTN or DICT to open the dictionary has no effect on whether the dictionary is audited or not. Either command can open the dictionary, but the audited state of the dictionary does not change.

If an existing dictionary that you open with the DICTN command is audited, the DDL compiler issues a warning message.

If an audited dictionary exists on $DATA.SALES, this command causes the DDL compiler to delete the dictionary and create a nonaudited dictionary on the subvolume:

```
?DICTN $data.sales !
```

DICTR

The DICTR command opens an existing dictionary for read-only access.

```
DICTR [ dict-subvol-name ]
```

dict-subvol-name

is the name of the dictionary subvolume, which has this form:

```
[\node-name.][$volume-name.][subvolume-name]
```

Syntax Element	Default
dict-subvol-name	Current system, volume, and subvolume
node-name	Current system
volume-name	Current volume
subvolume-name	Current subvolume

If *dict-subvol-name* has no dictionary, the DDL compiler issues a warning message and continues.

If you use the DICTR command when you already have a dictionary open, the DDL compiler closes the current dictionary before opening the specified dictionary.

The DDL compiler ignores the NOSAVE command when a dictionary is opened with DICTR.

DO_PTAL_ON

The DO_PTAL_ON[OFF] command generates code that cannot [can] be compiled by older pTAL or TAL compilers that do not recognize FIELDALIGN clauses.

```
{ DO_PTAL_ON | DO_PTAL_OFF }
```

Default: DO_PTAL_ON

`DO_PTAL_ON`

includes a FIELDALIGN clause for each structure in pTAL or TAL output. Compilers that do not recognize the FIELDALIGN clause cannot compile the resulting code.

`DO_PTAL_OFF`

also includes a FIELDALIGN clause for each structure in pTAL or TAL output, but encloses each FIELDALIGN clause within IF PTAL compiler directives. Compilers that do not recognize the FIELDALIGN clause can compile the resulting code.

Example 9-22. DO_PTAL_ON and DO_PTAL_OFF Commands (page 1 of 2)

DDL Input

```
DEF FIRST
02 FLD1 PIC X.
02 SUB.
  03 FLD2 PIC X.
END.
```

DDL Output with DO_PTAL_ON

```
! SCHEMA PRODUCED DATE - TIME : 3/10/1995 15:26:30
?SECTION FIRST
?PAGE
! Definition FIRST created on 3/10/1995 at 15:26
STRUCT FIRST^DEF (*) FIELDALIGN (SHARED2);
   BEGIN
   STRING   FLD1;
   STRUCT   SUB;
      BEGIN
      STRING   FLD2;
      END;
   END;
```

Example 9-22. DO_PTAL_ON and DO_PTAL_OFF Commands (page 2 of 2)

DDL Output with DO_PTAL_OFF

```
! SCHEMA PRODUCED DATE - TIME : 3/10/1995 15:26:30
?SECTION FIRST
?PAGE
! Definition FIRST created on 3/10/1995 at 15:26
STRUCT FIRST^DEF (*)
?IF PTAL
FIELDALIGN (SHARED2)
?ENDIF PTAL
;
   BEGIN
   STRING   FLD1;
   STRUCT   SUB;
      BEGIN
      STRING   FLD2;
      END;
   END;
```

EDIT

The EDIT command:

- Suspends compilation
- Starts an EDIT process
- Opens the specified file, executes the specified commands, and closes the file
- Resumes compilation when the EDIT process stops

You can use EDIT only in an interactive DDL session.

```
EDIT [ edit-file-name [ ; edit-parameter ] ... ]
```

edit-file-name

is the name of an EDIT file.

Default: The most recent *edit-file-name* specified in the current DDL session, if any. If none, you are prompted for a file name.

edit-parameter

is an EDIT command.

Default: The most recent *edit-file-name* specified in the current DDL session, if any. If none, you are prompted for an EDIT command.

Issuing the EDIT command within a DDL session is like issuing the EDIT command from the command interpreter. The EDIT prompt is the same, and you can use all the same functions.

When you stop an EDIT process by issuing the EXIT command, control returns to the DDL compiler.

You must close any source code file before editing it. For instance, if you have opened a COBOL source code file, entered some text in this file, and then want to view it with the text editor, you must issue the NOCOBOL command before you issue the EDIT command.

When you specify *edit-file-name* in the EDIT command, the DDL compiler passes that name to the current EDIT process and also stores the name. If you omit *edit-file-name* from the next EDIT command in the same session, the DDL compiler passes the stored name to the new EDIT process.

When you specify *edit-parameter* in the EDIT command, the DDL compiler passes that parameter to the EDIT process. The DDL compiler also stores the parameter.

If you omit *edit-file-name* from the next EDIT command, the DDL compiler passes any parameter saved from the last EDIT command to the new EDIT process.

If you specify *edit-file-name* in the next EDIT command, the DDL compiler discards any previously stored parameter.

Example 9-23. EDIT Command (page 1 of 2)

```
33>DDL DICT

!?DDL ddlfil                                  Open DDLFIL.

!RECORD sum.                                  Add a record.

...

!?NODDL                                       Close DDLFIL.

!?EDIT ddlfil; LIST ALL                       Start an EDIT process, list DDLFIL.

CURRENT FILE IS $DATA.PARTS.DDLFIL
   1 Record SUM.
   2 File is $DATA.SALES.SUM Unstructured
   3 Def is SUM-DEF.
   4 End

* FIX 3                                       Fix a record.

   3 Def is SUM-DEF.
         iR
   3 Def is RSUM-DEF.
         cr
* EXIT

!?SOURCE ddlfil                               Add the record to the dictionary.

!?EDIT                                        Use the previous file and parameter.

CURRENT FILE IS $DATA.PARTS.DDLFIL
   1 Record SUM.
   2 File is $DATA.SALES.SUM Unstructured
   3 Def is RSUM-DEF.
   4 End
```

Example 9-23. EDIT Command (page 2 of 2)

`* EXIT`	Stop the EDIT process.
`!EXIT`	Exit DDL.

ERRORS

The ERRORS command specifies the number of errors allowed before compilation stops.

```
ERRORS [ max-errors ]
```

Default: Compilation continues until the end of the source code file regardless of the number of errors

max-errors

is a number from 1 through 32,767 that specifies the maximum number of compilation errors allowed before the DDL compiler stops compiling the source code file.

Default: 1

When compilation stops because the specified number of errors is reached, the DDL compiler closes the open dictionary and any open files, issues session statistics, and stops.

The specified maximum number of errors applies only to errors that occur after the appearance of the ERRORS command. For example, if two errors occur before an ERRORS 3 command appears, the fifth error to occur (the third error after the command appeared) stops compilation.

This command directs the DDL compiler to stop compiling when it encounters the third compilation error.

```
?ERRORS 3
```

If the DDL compiler encounters a third compilation error, the DDL compiler issues the error message for the third error followed by the fatal error message:

```
Too Many Errors - Compilation Terminating.
```

Example 9-24. ERRORS Command

`?SECTION start` `...`	Compile regardless of errors

`?SECTION rest-of-schema`
`?ERRORS 1`
`...`
	Stop compiling source if any error is encountered

EXPANDC

The [NO]EXPANDC command generates a C referenced type definition inline [as a structure name].

```
[NO]EXPANDC
```

Default: NOEXPANDC

EXPANDC

generates a referenced type definition inline.

NOEXPANDC

generates a referenced type definition as a structure name.

A referenced definition type is a type of a line item and is a definition defined prior to the line item that references it.

In C applications, a structure being referenced by a line item is not always in the same module. Further, the structure being referenced can refer to yet another structure that might be in another module. Without inline expansion, a dependency chain or modules must be developed to ensure proper resolution of references.

The EXPANDC command does not apply to type ENUM, because the C compiler requires each enumerator to be unique. For ENUM types, the DDL compiler outputs a referenced type definition as a structure name. The DDL compiler generates a C enumerator for each level-89 clause in a type ENUM definition.

Example 9-25. EXPANDC Command (page 1 of 2)

	C Output	
DDL Definition	**With NOEXPANDC**	**With EXPANDC**
`def a pic x (10).`	`typedef char a_def[10];` `#pragma fieldalign shared2 __b`	Same as NOEXPANDC

`def b.`
` 2 b1 type binary.`
` 2 b2 pic x(10).`
` 2 b3 occurs 5 times.`
`  3 b31 type binary.`
`  3 b32 pic x(10).`
`end`
	`typedef struct __b`
	`{`
	` short b1;`
	` char b2[10];`
	` struct`
	` {`
	`  short b31;`
	`  char b32[10];`
	` } b3[5];`
	`} b_def;`
		Same as NOEXPANDC

`def c type a.`	`typedef a_def c_def;`	`typedef char c_def[10];` `#pragma fieldalign shared2 __d`

Example 9-25. EXPANDC Command (page 2 of 2)

DDL Definition	C Output	
	With NOEXPANDC	**With EXPANDC**

`def d type b.`	`typedef b_def d_def;`	`typedef struct __d`
		` short b1;`
		` char  b2[10];`
		` struct`
		` {`
		`  short b31;`
		`  char  b32[10];`
		` } b3[5];`
		`} d_def;`

`def e.`
` 2 e1 type c.`
` 2 e2 type d`
`  Occurs 15 times.`
`end.`
	`#pragma fieldalign shared2 __e`
	`typedef struct __e`
	`{`
	` c_def  e1;`
	` d_def  e2[15];`
	`} e_def;`
		`#pragma fieldalign`
		`shared2 __e`
		`typedef struct __e`
		`{`
		` char e1[10];`
		` struct`
		` {`
		`  short b1;`
		`  char b2[10];`
		`  struct`
		`  {`
		`   short b31;`
		`   char  b32[10];`
		`  } b3[5];`
		` } e2[15];`
		`} e_def;`

`def f.`
` 2 f1 pic x(100).`
` 2 f2 redefines f1.`
`  3 f3 type b.`
`3 f4 pic x(10).`
`end.`
	`#pragma fieldalign shared2 __f`
	`typedef struct __f`
	`{`
	` union`
	`  {`
	`  char f1[100];`
	`  struct`
	`  {`
	`  b_def f3;`
	`  char f4[10];`
	`  } f2;`
	` } u_f1;`
	`} f_def;`
		`#pragma fieldalign`
		`shared2 __f`
		`typedef struct __f`
		`{`
		`  union`
		` {`
		`  char f1[100];`
		`  struct`
		`  {`
		`  struct`
		`  {`
		`  short b1;`
		`  char b2[10];`
		`  struct`
		`  {`
		`  short`
		`b31;`
		`  char`
		`b32[10];`
		`  } b3[5];`
		`  } f3;`
		`  char f4[10];`
		`  } f2;`
		` } u_f1;`
		`} f_def;`

FIELDALIGN_SHARED8

The FIELDALIGN_SHARED8 command stores data structures in the dictionary with SHARED8 alignment.

```
FIELDALIGN_SHARED8
```

Use the FIELDALIGN_SHARED8 command to generate TAL (pTAL) or C source code that will produce optimal performance on a RISC processor.

The FIELDALIGN_SHARED8 command causes the DDL compiler to generate explicit filler fields:

- To align an item according to its width
- At the end of a structure to make its length a multiple of its alignment
- To prevent bit fields less than 16 bits from straddling a 2-byte boundary

Example 9-26. FIELDALIGN_SHARED8 Command (page 1 of 2)

DDL Input

```
?FIELDALIGN_SHARED8
   def a.
     02 b type character 1.
     02 c type character 1.
     02 d type character 1.
   end.

   def e type character 1.

   def f.
     02 g type binary 16.
     02 h.
        03 i type e.
        03 j type a.
     02 k type character 1.
     02 l type binary 16.
   end.
```

```
/* SCHEMA PRODUCED DATE - TIME :10/13/1995 13:23:16 */
#pragma section a
/* Definition A created on 10/13/1995 at 13:23 */
#pragma fieldalign shared8 __a
typedef struct __a
{
   char                              b;
   char                              c;
   char                              d;
   char                              filler_0;
} a_def;
```

Example 9-26. FIELDALIGN_SHARED8 Command (page 2 of 2)

DDL Output (C Code)

```
#pragma section e
/* Definition E created on 10/13/1995 at 13:23 */
typedef char                              e_def;
#pragma section f
/* Definition F created on 10/13/1995 at 13:23 */
#pragma fieldalign shared8 __f
typedef struct __f
{
   short                              g;
   struct
   {
      e_def                              i;
      char                               filler_0;
      a_def                              j;
   } h;
   char                               k;
   char                               filler_1;
   short                              l;
} f_def;
```

FILLER

The FILLER command specifies the algorithm for generating filler bytes for source code.

```
FILLER { 1 | 0 }
```

Default: FILLER 1

`1`

specifies filler algorithm 1, which is recommended for new dictionaries.

`0`

specifies filler algorithm 0, which is provided for compatibility with dictionaries created by versions of DDL prior to the B00 software product version.

The DDL compiler compiles source code in several phases. In each phase DDL evaluates records and definitions to see if filler bytes are necessary to make sure the next field or group starts on a word boundary.

Where filler bytes are necessary, the DDL compiler inserts FILLER fields according to the specified algorithm. Because the compiler uses the filler algorithm during each phase of compilation, the compiler might insert FILLER fields during one phase of compilation and remove the same FILLER fields during the next phase. The DDL compiler continues inserting and removing FILLER fields according to the specified filler algorithm until the source code is generated.

The DDL compiler removes only filler bytes generated by the DDL compiler; it never removes user-specified filler bytes.

When the CFIELDALIGN_MATCHED2 command is set, the DDL compiler uses a modified, extended FILLER 1. In this case, the DDL compiler ignores any FILLER 0 specification.

If FILLER 0 is specified, the DDL compiler generates filler bytes according to this algorithm:

- If a field or group described with a REDEFINES clause does not start on a word boundary, the DDL compiler inserts 1 byte of filler before the field or group being redefined, even if the redefined field is the first element in a group.
- If a single-item field or a group not described with a REDEFINES clause does not start on a word boundary, the DDL compiler inserts one byte of filler before the field or group.
- If the first element in a group not described with a REDEFINES clause does not start on a word boundary, the DDL compiler inserts one byte of filler before the group.
- If a group described with an OCCURS clause has both an odd number of bytes and an element that does not start on a word boundary, the DDL compiler inserts 1 byte of filler after the last element in the group. The filler bytes have the same level number as the first element in the group. (The DDL compiler can remove this byte of filler in a subsequent compilation phase.)
- If the first element of a group not described with a REDEFINES clause is a byte of filler generated by the DDL compiler and the group does not start on a word boundary, the DDL compiler inserts another filler byte before the group. In a subsequent compilation phase, the DDL compiler removes the filler byte from within the group.

FILLER 1 works exactly like FILLER 0 except that FILLER 1 has an additional rule that keeps user-defined TYPE definitions intact wherever they are used. The rest of the algorithm for FILLER 1 is this:

- For a group defined by a TYPE clause, the DDL compiler determines whether the group the clause refers to starts on a word boundary.
- If the group referenced starts on a word boundary, the DDL compiler does not insert any filler bytes for the referring group.
- If the group referenced does not start on a word boundary and is not described with a REDEFINES clause, the DDL compiler inserts a filler byte before the referring group.
- If the group referenced does not start on a word boundary but is described with a REDEFINES clause, the DDL compiler inserts a filler byte before the group being redefined.

Example 9-27. FILLER Command

DDL Input (Definition Statements)

```
DEF test1.
02  a               PIC XX.
02  b               PIC S9(4)       COMP.
END

DEF case1.
02  c               PIC X.
02  test1          TYPE *.
END

DEF case2.
02  test1          TYPE *.
02  c               PIC X.
END
```

DDL Output (COBOL Code) with FILLER 1

```
01 CASE1.
  02 C              PIC X.
  02 FILLER         PIC X(1).
  02 TEST1.
     03 A           PIC XX.
     03 B           PIC S9(4)       COMP.

01 CASE2.
  02 TEST1.
    03 A            PIC XX.
    03 B            PIC S9(4)       COMP.
  02 C              PIC X.
```

DDL Output (COBOL Code) with FILLER 0

```
01 CASE1.
  02 C              PIC X.
  02 TEST1.
    03 A            PIC XX.
    03 FILLER       PIC X(1).
    03 B            PIC S9(4)       COMP.

01 CASE2.
  02 TEST1.
    03 A            PIC XX.
    03 B            PIC S9(4)       COMP.
  02 C              PIC X.
```

In Example 9-27 on page 9-61, when FILLER 1 is specified, the structure of TEST1 is the same in both CASE1 and CASE2. A COBOL program containing these data structures can successfully execute the COBOL statement:

```
MOVE TEST1 OF CASE1 to TEST1 OF CASE2.
```

When FILLER 0 is specified, the structure of TEST1 in CASE1 differs from the structure of TEST1 in CASE2. A COBOL program containing these data structures can not successfully execute the preceding MOVE statement.

△ **Caution.** Mixing FILLER 1 and FILLER 0 can cause the DDL compiler to generate unusable code. Using the preceding CASE1 as an example, if you add the definition to a dictionary while FILLER 0 is in effect and later output the definition with FILLER 1 (the default) in effect, the DDL compiler generates COBOL source code with *two* added fillers (one preceding TEST1 and the other within it, as in both of the CASE1 examples), causing the computational item to begin in the middle of a word. This is incorrect for either FILLER option. Results are similar for languages other than COBOL.

FORCHECK

The [NO]FORCHECK command performs [suppresses] FORTRAN syntax checks on subsequent DDL object definitions without generating code.

```
[NO]FORCHECK
```

Default: FORCHECK if a FORTRAN source code file is open, otherwise NOFORCHECK

`FORCHECK`

performs the FORTRAN syntax checks as though FORTRAN source code were being produced.

`NOFORCHECK`

suppresses FORTRAN syntax checks.

If a FORTRAN source code file is open, the compiler performs the FORTRAN checks whether or not FORCHECK is set.

You can stop FORTRAN syntax checking with a NOFORCHECK command; you can restart checking with a subsequent FORCHECK.

The DDL compiler does not make all the lengthy syntax tests that the FORTRAN compiler makes. The DDL compiler tests the DDL statements to ensure that they follow the rules specified by FORTRAN:

- An elementary field must not be larger than 255 bytes.
- An element must not be described as TYPE CHARACTER 8; this data type is not supported in FORTRAN.

Example 9-28. FORCHECK Command

```
?FORCHECK
RECORD long.
FILE IS "$data.sales.long" KEY-SEQUENCED.
  02 lfield PIC X(256).
  02 sfield PIC X KEYTAG 0.
END
Record LONG size is 257 bytes.
*** WARNING *** FORTRAN OUTPUT DIAGNOSTICS.
*** ERROR *** Fortran element with size greater than 255 - LFIELD
?NOFORCHECK
```

When FORCHECK is in effect, the DDL compiler issues the following message for each DEFINITION or RECORD statement that passes the syntax check:

```
FORTRAN CHECK completed for name
```

FORTRAN

The [NO]FORTRAN command:

- Opens [closes] a FORTRAN source code file
- Starts [stops] writing translated DDL object definitions to the FORTRAN source code file

```
{ FORTRAN [ fortran-source-file [ ! ] ] }
{ NOFORTRAN                            }
```

Default: NOFORTRAN

FORTRAN

closes any open FORTRAN source code file, opens *fortran-source-file*, translates subsequent DDL objects defined by statements or specified in OUTPUT statements to FORTRAN source code statements, and writes the FORTRAN source code statements to *fortran-source-file*.

fortran-source-file

is the name of the FORTRAN source code file to be created, if necessary, and opened. The file must be one of:

- EDIT file
- Unstructured file
- Sequential device (such as a terminal, spooler, or process)

If the file exists but is not one of these types, the DDL compiler issues an error message and does not open the file.

If *fortran-source-file* is an EDIT file, and it exceeds 99,999 lines, the DDL compiler issues FILE ERROR - filename - Edit file line number too large (537) on page A-17.

Default: home terminal

!

purges the contents of *fortran-source-file* before opening it, if it exists. If *fortran-source-file* does not exist, the exclamation point has no effect.

Without the exclamation point, the DDL compiler appends the new FORTRAN source code statements to the end of *fortran-source-file*.

NOFORTRAN

closes any open FORTRAN source code file and stops translating DDL object definitions to FORTRAN source code statements.

For the data types that the DDL compiler generates for FORTRAN source code, see Table C-3 on page C-5.

The specified FORTRAN source code file must be an EDIT file, an unstructured file, or a sequential device such as a terminal, a spooler, or a process. If the file exists but is not one of these types, the DDL compiler issues an error message and does not open the file.

Only one FORTRAN source code file can be open at a time. If you use the FORTRAN command when you already have a FORTRAN source code file open, the DDL compiler closes the current source code file before opening the new source code file.

The only DDL objects that can be translated to FORTRAN source code are definitions and records.

The compiler can translate definitions and records specified in an OUTPUT statement only if the dictionary containing these objects is open.

If the FORTRAN source code file already exists and the exclamation point is omitted, the DDL compiler appends the DDL objects to the end of the file's original contents. The DDL compiler does not replace any existing structures.

Each DDL object translated to FORTRAN source code is written to the FORTRAN source code file in a separate section that has the same name as the DDL object it contains. You can suppress the individual section headings with the SETSECTION command.

The DDL compiler translates a DDL group definition or record to a FORTRAN record structure preceded by a comment line that identifies the record structure as a DDL definition.

Unless the command FORTRANUNDERSCORE on page 9-66 is in effect, the DDL compiler discards any hyphens in a DDL name before writing the name to FORTRAN source code.

Note. The DDL compiler translates objects named A-B and AB to data structures that have the same name in FORTRAN (unless FORTRANUNDERSCORE has been specified).

FORTRAN does not support unsigned numbers. If you specify an unsigned number, the DDL compiler translates it to a FORTRAN signed integer.

FORTRAN does not accept FILLER fields greater than 255 single-byte characters. the DDL compiler can add filler characters to ensure that structures start on word boundaries; if such padding generates more than 255 filler characters, the DDL compiler breaks them into smaller fields before writing the FORTRAN source code.

The DDL compiler performs all of the syntax checks listed under the FORCHECK command before writing source output. If the compiler finds a syntax error, it does not write the source output for the DDL object with the error; it does write source output for a DDL object if only a warning is issued.

The DDL compiler ignores the RENAMES clause when generating FORTRAN source output.

In Example 9-29 on page 9-65, the DDL compiler retrieves the definition of the record CUSTOMER from the open dictionary, translates it to FORTRAN source code, and writes it to the file \DALLAS.$DATA.SALES.FORSRC. If this file already exists, the DDL compiler appends the entry for CUSTOMER to the file. For the definition of the CUSTOMER record, see the sample database schema in Appendix B, Sample Schemas. FORTRAN does not recognize the UPSHIFT clause, but is included as a comment.

Example 9-29. FORTRAN Command (page 1 of 2)

DDL Input

```
?DICT
?FORTRAN \dallas.$data.sales.forsrc
OUTPUT RECORD customer.
```

Example 9-29. FORTRAN Command (page 2 of 2)

DDL Output (FORTRAN Code)

```
?SECTION CUSTOMER
?PAGE
C Definition CUSTOMER created on 06/11/1987 at 12:55
      RECORD CUSTOMER
         CHARACTER*4 CUSTNUM
         RECORD CUSTNAME
            CHARACTER*12 LASTNAME
C           Upshift
            CHARACTER*8 FIRSTNAME
C           Upshift
            CHARACTER*2 MIDINIT
C           Upshift
         END RECORD
         RECORD ADDR
            CHARACTER*22 ADDRESS
            CHARACTER*14 CITY
            CHARACTER*2 STATE
            CHARACTER*5 ZIP
         END RECORD
      END RECORD
```

FORTRANUNDERSCORE

The [NO]FORTRANUNDERSCORE command replaces with underscores [deletes] hyphens in DDL names for FORTRAN output.

```
[NO]FORTRANUNDERSCORE
```

Default: NOFORTRANUNDERSCORE

`FORTRANUNDERSCORE`

replaces each hyphen (-) with an underscore (_) in DDL names for FORTRAN output.

`NOFORTRANUNDERSCORE`

deletes hyphens from DDL names for FORTRAN output.

Versions of the FORTRAN compiler from the C10 and later software product versions allow underscores in source code names.

FUP

The [NO]FUP command:

- Opens [closes] a FUP source code file
- Starts [stops] writing translated DDL object definitions to the FUP source code file

```
{ FUP [ fup-source-file [ ! ] ] }
{ NOFUP                         }
```

Default: NOFUP

FUP

closes any open FUP source code file, opens *fup-source-file*, translates subsequent DDL objects defined by statements or specified in OUTPUT statements to FUP source code statements, and writes the FUP source code statements to *fup-source-file*.

fup-source-file

is the name of the FUP source code file to be created, if necessary, and opened. The file must be one of:

- EDIT file
- Unstructured file
- Sequential device (such as a terminal, spooler, or process)

If the file exists but is not one of these types, the DDL compiler issues an error message and does not open the file.

Default: home terminal

!

purges the contents of *fup-source-file* before opening it, if it exists. If *fup-source-file* does not exist, the exclamation point has no effect.

Without the exclamation point, the DDL compiler appends the new FUP source code statements to the end of *fup-source-file*.

NOFUP

closes any open FUP source code file and stops translating DDL object definitions to FUP source code statements.

The specified FUP source code file must be an EDIT file, an unstructured file, or a sequential device such as a terminal, a spooler, or a process. If the file exists but is not one of these types, the DDL compiler issues an error message and does not open the file.

Only one FUP source code file can be open at a time. If you use the FUP command when you already have a FUP source code file open, the DDL compiler closes the current source code file before opening the new source code file.

If the FUP source code file source code file already exists and the exclamation point is omitted, the DDL compiler appends the new FUP file-creation commands to the end of the file. The DDL compiler does not replace any existing commands in the file.

You can change any file-creation command after it is written to the FUP source code file by closing the file and then editing it using the EDIT program. You might need to edit either of these attributes:

Attribute	Reason for Change
File names	The file names generated for ALTFILE and CREATE commands might be unacceptable.
Alternate-key files	The DDL compiler generates one file for all keys that are not unique and one file for each unique alternate key. You can change this mapping of keys to files.

If you specify a file attribute with a value equal to the default in FUP, the DDL compiler will not specify the attribute in the generated FUP source code.

DDL names alternate key files by appending a number, starting with 0, to the data file name. If necessary, the DDL compiler truncates the file name so that the composite name does not exceed eight ASCII characters. Thus, if a file named LONGFILE has 11 unique alternate keys, the DDL compiler generates 11 alternate key files named LONGFIL0, LONGFIL1, and so forth through LONGFIL0.

Example 9-30 on page 9-68 shows an interactive session in which FUP source code is generated from a record in the dictionary.

Example 9-30. FUP Command (page 1 of 2)

DDL Input

`34> DDL DICT`	Run DDL and open dictionary.
`!?FUP fupsrc !`	Open FUPSRC; if file exists, purge existing data.
`!OUTPUT RECORD customer.`	Get record from dictionary, write FUP file-creation commands to FUPSRC.
`?NOFUP`	Close FUPSRC.
`!EXIT`	Exit DDL.

Example 9-30. FUP Command (page 2 of 2)

DDL Output (FUPSRC Content)

```
< SECTION CUSTOMER
RESET
   SET ALTKEY ( "cn", KEYOFF 4, KEYLEN 22, FILE 0 )
   SET NO ALTCREATE
   SET ALTFILE ( 0, $data.sales.custome0 )
   SET TYPE K
   SET KEYLEN 4
   SET REC 69
   SET BLOCK 4096
   SET IBLOCK 4096
   SET AUDIT
   SET MAXEXTENTS 100
   SET EXT( 4, 32 )
CREATE $data.sales.customer
   RESET
   SET TYPE K
   SET KEYLEN 28
   SET REC 28
   SET BLOCK 4096
   SET IBLOCK 4096
   SET EXT( 4, 32 )
   SET AUDIT
   SET MAXEXTENTS 100
CREATE $data.sales.custome0
```

Command to Create CUSTOMER file and its Alternate Key File CUSTOME0

```
35> FUP/IN fupsrc/
```

HELP

The HELP command briefly describes a specified command or all commands.

```
HELP [ command ]
```

command

is the name, or the beginning of the name, of a DDL command.

If you specify the first one or more characters of a command, the DDL compiler returns information about the first command (in alphabetic order) that matches the string of characters.

Default: all DDL commands

Example 9-31. HELP Command With Full Command Name

```
36> DDL
!?HELP SAVE
   SAVE - Don't purge dictionary when it is closed
!
```

Example 9-32. HELP Command With Partial Command Name

```
37> DDL
!?HELP C
   COBOL - Open COBOL source output on specified file
```

LINECOUNT

The LINECOUNT command specifies the number of lines for each page for all source code files.

```
LINECOUNT number
```

Default: LINECOUNT 56

number

is the number of lines per page on a report or listing. If *number* is outside the range from 1 through 56, the LINECOUNT command has no effect.

The LINECOUNT command is meaningful only when the listing or report destination is a line printer.

Example 9-33 on page 9-71 sets the number of lines per page for the compiler listing pages to 60 lines.

Example 9-33. LINECOUNT Command

```
37> DDL/IN myschema,OUT $S.#printer/LINECOUNT 60
```

Example 9-34 on page 9-71 sets the number of lines per page for a schema report to 24.

Example 9-34. LINECOUNT Command

```
38> DDL
!?REPORT $S.#printer
!?LINECOUNT 24
!?SOURCE myschema
!EXIT
```

LIST

The LIST command includes [excludes] subsequent DDL source lines in [from] the compiler listing.

```
[NO]LIST
```

Default: LIST

`LIST`

includes subsequent DDL source lines in the compiler listing.

`NOLIST`

excludes subsequent DDL source lines from the compiler listing.

You can specify the DDL listing destination either:

- With the OUT run option of the RUN DDL Command on page 3-1
- With the command OUT on page 9-82

The NOLIST command does not suppress the listing of error and warning messages. Messages are listed regardless of the LIST command setting. If NOLIST is in effect and an error is encountered, the line containing the error is listed.

The NOLIST command does not suppress production comments. Production comments describe such things as the total length of records and definitions and also describe actions taken by the compiler such as adding a record to the dictionary.

Example 9-35. LIST and NOLIST Commands

```
*beginning of source code file      List source lines by default.
...
?NOLIST                             Starting with this command, stop listing source
...                                 lines.
?LIST                               Resume listing source lines.
...
*end of source code file
```

NCLCONSTANT

The [NO]NCLCONSTANT command:

- Opens [closes] an NCL source code file
- Starts [stops] writing translated DDL constant definitions to the NCL source code file

```
{ NCLCONSTANT [ NCL-source-file [ ! ] ] }
{ NONCLCONSTANT                         }
```

Default: NONCLCONSTANT

NCLCONSTANT

closes any open NCL source code file, opens *NCL-source-file*, translates subsequent DDL constants defined by statements or specified in OUTPUT statements to NCL source code statements, and writes the NCL source code statements to *NCL-source-file*.

NCL-source-file

is the name of the NCL source code file to be created, if necessary, and opened. The file must be one of:

- EDIT file
- Unstructured file
- Sequential device (such as a terminal, spooler, or process)

If the file exists but is not one of these types, the DDL compiler issues an error message and does not open the file.

Default: home terminal

!

purges the contents of *NCL-source-file* before opening it, if it exists. If *NCL-source-file* does not exist, the exclamation point has no effect.

Without the exclamation point, the DDL compiler appends the new NCL source code statements to the end of *NCL-source-file*.

NONCLCONSTANT

closes any open NCL source code file and stops translating DDL constant definitions to NCL source code statements.

When NCLCONSTANT is in effect, The DDL compiler translates only DDL Constant objects. DDL record, definition, token type, token code, and token map objects are not translated. The DDL compiler issues a warning message when attempting to output a nonconstant in NCL.

A DDL constant name can have up to 30 ASCII characters. As a result, the maximum number of characters for an NCL constant name generated by the DDL compiler is 30.

The value of a DDL string constant can be from 1 to 130 ASCII characters, not including the beginning and ending quotation; therefore, the NCL value for a string generated by the DDL compiler can have up to 130 ASCII characters. The legal range of values for numeric constants depends on the TYPE clause specified in the statement CONSTANT on page 4-1.

The DDL compiler replaces any hyphen (-) in a DDL Constant name with an underscore (_) before writing the name to the NCL source code file.

All NCL constant names generated by the DDL compiler appear in uppercase characters.

The specified NCL source code file must be an EDIT file. If the source code file exists but is not an EDIT file, the DDL compiler issues an error message, does not open the file, and does not output any NCL source.

Only one NCL source code file can be open at a time. If you specify the NCLCONSTANT command when you already have an NCL source code file open, the DDL compiler closes the current source code file before opening the new source code file.

If the NCL source code file already exists and you omit the exclamation point, the DDL compiler appends the Constant objects to the end of the original contents of the file. The DDL compiler does not replace any existing objects.

The DDL compiler can translate DDL Constant objects specified in an OUTPUT statement only if the dictionary containing these objects is open.

Example 9-36. NCLCONSTANT Command

DDL Input

```
?NCLCONSTANT  $vol.subvol.myncl !
constant val-1     value 1.
constant VAL-2     value 2.
constant val-abc   value "abc".
constant Val-3     value 3.

! for the following definition, DDL generates a warning
! message and does not translate def-1 to NCL
definition def-1.  02 a pic x(10). end.
constant val-4     value 4.

?NONCLCONSTANT
constant val-5     value 5.
constant val-6     value 6.

?NCLCONSTANT  $vol.subvol.myncl
constant val-7     value 7.
constant val-8     value 8.
constant large-val value 32768 type binary 32.
```

DDL Output ($vol.subvol.myncl Content)

```
/* SCHEMA PRODUCED DATE - TIME :12/01/1992 10:43:29 */
/* Constant VAL-1 created on 12/01/1992 at 10:54 */
%%define VAL_1 1
/* Constant VAL-2 created on 12/01/1992 at 10:54 */
%%define VAL_2 2
/* Constant VAL-ABC created on 12/01/1992 at 10:54 */
%%define val_abc "abc"
/* Constant VAL-3 created on 12/01/1992 at 10:54 */
%%define val_3 3
/* Constant VAL-4 created on 12/01/1992 at 10:54 */
%%define val_4 4
/* Constant VAL-7 created on 12/01/1992 at 10:54 */
%%define val_7 7
/* Constant VAL-8 created on 12/01/1992 at 10:54 */
%%define val_8 8
/* Constant LARGE-VAL created on 12/01/1992 at 10:54 */
%%define large_val 32768
```

NEWFUP_FILEFORMAT

The NEWFUP_FILEFORMAT command specifies file format 2 for all FUP source code files and all FUP alternate key files.

```
{ NEWFUP_FILEFORMAT | OLDFUP_FILEFORMAT | NOFILEFORMAT }
```

Default: NOFILEFORMAT on page 9-77

The format specification of a file for a record will not be stored in the DDL dictionary. If the user compiles the records with a particular format command (using the OLDFUP_FILEFORMAT, NEWFUP_FILEFORMAT, or NOFILEFORMAT command) and stores the records in the dictionary, then the user must use the same commands while requesting FUP output of those records.

The format specification for both alternate key and main file is the same (file format 2 in this case).

The DDL compiler does not allow you to define record length more than the maximum allowed length for the particular type of file.

For format 2 files, the maximum allowed record length (assuming the block size as 4096 bytes) is.

File type	Record size (format 2 file)
Unstructured	4096 Bytes
Entry-sequenced	4048 Bytes
Relative	4048 Bytes
Key-Sequenced	4040 Bytes

If you attempt to define a record size greater than the above specified record sizes the DDL compiler issues an error message.

In Example 9-37 on page 9-75, the DDL compiler generates a statement in FUP source code files to create a format 2 file.

Example 9-37. NEWFUP_FILEFORMAT Command (page 1 of 3)

```
20> DDL
?DICT
?NEWFUP_FILEFORMAT
DEF EMP.
02  EMP-NAME   PIC X(20).
02  EMP-ID     PIC 9(4) COMP.
02  EMP-SALARY PIC 9(6)V9(2).
END.
```

Example 9-37. NEWFUP_FILEFORMAT Command (page 2 of 3)

```
?FUP
RECORD EMPL.
FILE     IS "EMPLOYEE".
DEF      IS  EMP.
KEY      IS  EMPL.EMP-ID.
KEY "MN" IS  EMPL.EMP-NAME.
END.

?DICT
Audited dictionary created on subvol $ADE101.MANUAL.
Dictionary opened on subvol $ADE101.MANUAL for update access.
?NEWFUP_FILEFORMAT
DEF EMP.
02  EMP-NAME PIC X(20).
02  EMP-ID   PIC 9(4) COMP.
02  EMP-SALARY PIC 9(6)V9(2).
END.
Definition EMP size is 30 bytes.
Definition EMP added to dictionary.

!?FUP
< SCHEMA PRODUCED DATE - TIME : 3/02/2000 17:59:53
 Output source for FUP is opened on $ZTNT.#PTVWAT5
!RECORD EMPL.
!FILE IS EMPLOYEE.
!DEF IS EMP.
!KEY IS EMPL.EMP-ID.
!KEY "MN" IS EMPL.EMP-NAME.
!END.
Record EMPL size is 30 bytes.
Record EMPL added to dictionary.

< SECTION EMPL
< Record EMPL created on 03/02/2000 at 18:00
RESET
SET FORMAT 2
SET ALTKEY ( "MN", KEYOFF 0, KEYLEN 20, FILE 0 )
SET NO ALTCREATE
SET ALTFILE ( 0, EMPLOYE0 )
SET TYPE K
SET KEYOFF 20
SET KEYLEN 2
SET REC 30
SET BLOCK 4096
SET IBLOCK 4096
SET EXT( 4, 32 )
SET MAXEXTENTS 100
CREATE EMPLOYEE
```

Example 9-37. NEWFUP_FILEFORMAT Command (page 3 of 3)

```
RESET
SET FORMAT 2
SET TYPE K
SET KEYLEN 24
SET REC 24
SET BLOCK 4096
SET IBLOCK 4096
SET EXT( 4, 32 )
SET MAXEXTENTS 100
CREATE EMPLOYE0
FUP output produced for EMPL.
```

NOFILEFORMAT

The NOFILEFORMAT command specifies no file format for all FUP source code files and all FUP alternate key files.

```
{ NOFILEFORMAT | NEWFUP_FILEFORMAT | OLDFUP_FILEFORMAT }
```

Default: NOFILEFORMAT

No format is specified for both alternate key and main files.

The NOFILEFORMAT command allows the user to generate FUP output without any format specification.

The DDL compiler does not allow you to define a record length that is more than the maximum allowed length for the particular type of file.

For files with no format specification, the maximum allowed record length (assuming the block size as 4096 bytes) is.

File type	Record size (<2GB file)
Unstructured	4096 Bytes
Key-Sequenced	4062 Bytes
Entry-sequenced	4072 Bytes
Relative	4072 Bytes

If you attempt to define a record size that is greater than the above specified record sizes the DDL compiler issues an error message.

Example 9-38. NOFILEFORMAT Command (page 1 of 2)

```
20> DDL
!?DICT
!?NOFILEFORMAT
!DEF EMP.
!02  EMP-NAME   PIC X(20).
!02  EMP-ID     PIC 9(4) COMP.
!02  EMP-SALARY PIC 9(6)V9(2).
!END.

!?FUP
!RECORD EMPL.
!FILE     IS "EMPLOYEE".
!DEF      IS  EMP.
!KEY      IS  EMPL.EMP-ID.
!KEY "MN" IS  EMPL.EMP-NAME.
!END.

!?DICT
Dictionary opened on subvol $ADE101.MANUAL for update access.
!DEF EMP.
!02  EMP-NAME PIC X(20).
!02  EMP-ID   PIC 9(4) COMP.
!02  EMP-SALARY PIC 9(6)V9(2).
!END.
Definition EMP size is 30 bytes.
Definition EMP added to dictionary.

!?FUP
< SCHEMA PRODUCED DATE - TIME : 3/02/2000 - 18:52:42
Output source for FUP is opened on $ZTNT.#PTVWAT5
!?NOFILEFORMAT
!RECORD EMPL.
!FILE IS "EMPLOYEE".
!DEF  IS  EMP.
!KEY  IS  EMPL.EMP-ID.
!KEY "MN" IS EMPL.EMP-NAME.
!END.
Record EMPL size is 30 bytes.
Record EMPL added to dictionary.
```

Example 9-38. NOFILEFORMAT Command (page 2 of 2)

```
< SECTION EMPL
< Record EMPL created on 03/02/2000 at 19:10
RESET
SET ALTKEY ( "MN", KEYOFF 0, KEYLEN 20, FILE 0 )
SET NO ALTCREATE
SET ALTFILE ( 0, EMPLOYE0 )
SET TYPE K
SET KEYOFF 20
SET KEYLEN 2
SET REC 30
SET BLOCK 4096
SET IBLOCK 4096
SET EXT( 4, 32 )
SET MAXEXTENTS 100
CREATE EMPLOYEE

RESET
SET TYPE K
SET KEYLEN 24
SET REC 24
SET BLOCK 4096
SET IBLOCK 4096
SET EXT( 4, 32 )
SET MAXEXTENTS 100
CREATE EMPLOYE0
FUP output produced for EMPL.
```

OLDFUP_FILEFORMAT

The OLDFUP_FILEFORMAT command specifies file format 1 for all FUP source code files and all FUP alternate key files.

```
{ OLDFUP_FILEFORMAT | NEWFUP_FILEFORMAT | NOFILEFORMAT }
```

Default: NOFILEFORMAT on page 9-77

The format specification of a file for a record is not stored in the dictionary. If the user compiles the records with a particular format command (using the OLDFUP_FILEFORMAT, NEWFUP_FILEFORMAT, or NOFILEFORMAT command) and stores the records in the dictionary, the user must use the same commands while requesting FUP output of those records.

The format specification for both the alternate key and the main file is the same (format 1 in this case).

The DDL compiler does not allow the user to define a record length greater than the maximum allowed length. For format 1 files, the maximum allowed record length (assuming the block size is 4096 bytes) is.

File type	Record size (format 1 file)
Unstructured	4096 bytes
Entry-sequenced	4072 bytes
Relative	4072 bytes
Key-Sequenced	4062 bytes

If you attempt to define a record size greater than the above specified record sizes the DDL compiler issues an error message.

In Example 9-39 on page 9-80, the DDL compiler generates a statement in FUP source code files to create a format 1 file.

Example 9-39. OLDFUP_FILEFORMAT Command (page 1 of 2)

```
20> DDL
?DICT
?OLDFUP_FILEFORMAT
DEF EMP.
02  EMP-NAME   PIC X(20).
02  EMP-ID     PIC 9(4) COMP.
02  EMP-SALARY PIC 9(6)V9(2).
END.

?FUP
RECORD EMPL.
FILE     IS "EMPLOYEE".
DEF      IS  EMP.
KEY      IS  EMPL.EMP-ID.
KEY "MN" IS  EMPL.EMP-NAME.
END.

?DICT!
Audited dictionary created on subvol $ADE101.MANUAL.
Dictionary opened on subvol $ADE101.MANUAL for update access.
DEF EMP.
02  EMP-NAME PIC X(10).
02  EMP-ID   PIC 9(6) COMP.
02  EMP-SALARY  PIC 9(7)V9(2).
END.
Definition EMP size is 23 bytes.
Definition EMP added to dictionary.
```

Example 9-39. OLDFUP_FILEFORMAT Command (page 2 of 2)

```
?FUP
< SCHEMA PRODUCED DATE - TIME : 3/01/2000 - 21:26:19
Output source for FUP is opened on $ZTNT.#PTVWAMU
?OLDFUP_FILEFORMAT
RECORD EMPL.
FILE  IS  "EMPLOYEE".
DEF   IS   EMP.
KEY   IS   EMPL.EMP-ID.
KEY "MN" IS EMPL.EMP-NAME.
END.
Record EMPL size is 23 bytes.
Record EMPL added to dictionary.

< SECTION EMPL
< Record EMPL created on 03/01/2000 at 21:26
RESET
SET FORMAT 1
SET ALTKEY ( "MN", KEYOFF 0, KEYLEN 10, FILE 0 )
SET NO ALTCREATE
SET ALTFILE ( 0, EMPLOYE0 )
SET TYPE K
SET KEYOFF 10
SET KEYLEN 4
SET REC 23
SET BLOCK 4096
SET IBLOCK 4096
SET EXT( 4, 32 )
SET MAXEXTENTS 100
CREATE EMPLOYEE

RESET
SET FORMAT 1
SET TYPE K
SET KEYLEN 16
SET REC 16
SET BLOCK 4096
SET IBLOCK 4096
SET EXT( 4, 32 )
SET MAXEXTENTS 100
CREATE EMPLOYE0
```

OUT

The OUT command specifies the destination for compiler output (source lines, warnings, and error messages).

```
OUT [ listing-destination ]
```

listing-destination

is a file name or output device.

Default: destination specified in the OUT run option of the RUN DDL Command on page 3-1

The OUT command can be used anywhere within a DDL source code file. Different portions of the listing can be written to different destinations.

If you use the OUT command in an interactive session to change the output device to a device other than your terminal, the session ceases to be interactive. As a result, you cannot use the EDIT command until a subsequent OUT command changes the output device back to your terminal.

If the listing destination you specify is an existing file, the DDL compiler appends the listing to the end of the existing file.

Example 9-40. OUT Command

`*beginning of source code file` `...`	List source lines on listing destination from the RUN DDL command.
`?OUT $S.#printer` `...`	List source lines and error messages on $S.#printer and list error messages on the listing destination from the RUN DDL command.

`?OUT`
`...`
`*end of source code file`
	Stop listing on $S.#printer and return to the RUN DDL command listing destination.

OUTPUT_SENSITIVE

The [NO]OUTPUT_SENSITIVE command generates case-sensitive [case-insensitive] output.

```
[NO]OUTPUT_SENSITIVE
```

Default: NOOUTPUT_SENSITIVE

`OUTPUT_SENSITIVE`

generates all source code files in case-sensitive form; that is, lowercase will remain lowercase and uppercase will remain uppercase.

`NOOUTPUT_SENSITIVE`

generates all source code files in a case-insensitive form. Overrides the OUTPUT_SENSITIVE command if that command is in effect.

The OUTPUT_SENSITIVE command allows the user to define all definitions, records, and constants in case-sensitive format. All lowercase remains lowercase and all uppercase remains uppercase.

In order to get case-sensitive output for a particular definition, record, or constant, the OUTPUT_SENSITIVE command must be used before adding that definition, record, or constant to the dictionary.

If a definition, record, or constant is defined using the OUTPUT_SENSITIVE (or NOOUTPUT_SENSITIVE) command, then the user must use the same definition while requesting output for that definition, record, or constant.

Example 9-41. OUTPUT_SENSITIVE Command (page 1 of 3)

```
> DDL
! ?DICT
! ?C
! ?TAL
! ?NOOUTPUT_SENSITIVE
! DEF   kiSHOy.
! 02    cuTNAME    PIC  X(10).
! 02    cdT-ID     PIC  9(6) .
! END.
```

Example 9-41. OUTPUT_SENSITIVE Command (page 2 of 3)

```
!?DICT
Audited dictionary created on subvol $ADE101.BUG.
Dictionary opened on subvol $ADE101.BUG for update access.
!?C
/* SCHEMA PRODUCED DATE - TIME : 3/06/2000 - 13:19:47 */
Output source for C is opened on $ZTN0.#PTS3Z89
!?TAL
! SCHEMA PRODUCED DATE - TIME : 3/06/2000 - 13:19:55
Output source for TAL is opened on $ZTN0.#PTS3Z89
!?NOOUTPUT_SENSITIVE
!DEF  kiSHOy.
!02   cuTNAME  PIC  X(10).
!02   cdT-ID   PIC  9(6) .
!END.
Definition KISHOY size is 16 bytes.
Definition KISHOY added to dictionary.

#pragma section kishoy
/* Definition KISHOY created on 03/06/2000 at 13:20 */
#pragma fieldalign shared2 __kishoy
typedef  struct __kishoy
{
char                              cutname[10];
char                              cdt_id[6];
} kishoy_def;
#define kishoy_def_Size 0

C output produced for KISHOY.
?SECTION KISHOY
?PAGE
! Definition KISHOY created on 03/06/2000 at 13:20
STRUCT       KISHOY^DEF (*) FIELDALIGN (SHARED2);
BEGIN
STRUCT       CUTNAME;
BEGIN STRING BYTE [1:10]; END;
STRUCT       CDT^ID;
BEGIN STRING BYTE [1:6]; END;
END;
```

Example 9-41. OUTPUT_SENSITIVE Command (page 3 of 3)

```
TAL output produced for KISHOY.
!?OUTPUT_SENSITIVE
!DEF  emp.
!02   emp-NAME  PIC X(10).
!02   emp-ID    PIC 9(6)   COMP.
!END.
Definition emp size is 14 bytes.
Definition emp added to dictionary.
#pragma section emp
/* Definition emp created on 03/06/2000 at 13:29 */
#pragma fieldalign shared2 __emp
typedef struct __emp
{
char                              emp_NAME[10];
unsigned long                     emp_ID;
} emp_def;
#define emp_def_Size 0

C output produced for emp.
?SECTION emp
?PAGE
! Definition emp created on 03/06/2000 at 13:29
STRUCT       emp^DEF (*) FIELDALIGN (SHARED2);
BEGIN
STRUCT       emp^NAME;
BEGIN STRING BYTE [1:10]; END;
INT(32)      emp^ID;
END;
TAL output produced for emp.
```

PAGE

The PAGE command writes the next line of the compiler listing at the top of the next page and (optionally) specifies a page title.

```
PAGE [ "listing-title" ]
```

listing-title

is an ASCII character string of at most 132 characters.

The PAGE command can be placed anywhere in the DDL source listing.

When the DDL compiler encounters a PAGE command, it stops listing on the current page, issues a page-ejection character to the listing destination, and resumes listing at the top of the next page.

If a title is specified, the DDL compiler lists that title at the top of every subsequent page until it encounters another PAGE command with a different listing title.

Example 9-42. PAGE Command

```
?PAGE "DEFINITIONS"
```

Each subsequent listing page has the title DEFINITIONS until the DDL compiler encounters another PAGE command with a new title.

PASCAL (D-Series Systems Only)

The [NO]PASCAL command:

- Opens [closes] a Pascal source code file
- Starts [stops] writing translated DDL object definitions to the Pascal source code file

```
{ PASCAL [ pascal-source-file { ! ] ] }
{ NOPASCAL                          }
```

Default: NOPASCAL

PASCAL

closes any open Pascal source code file, opens *pascal-source-file*, translates subsequent DDL objects defined by statements or specified in OUTPUT statements to Pascal source code statements, and writes the Pascal source code statements to *pascal-source-file*.

pascal-source-file

is the name of the Pascal source code file to be created, if necessary, and opened. The file must be one of:

- EDIT file
- Unstructured file
- Sequential device (such as a terminal, spooler, or process)

If the file exists but is not one of these types, the DDL compiler issues an error message and does not open the file.

If *pascal-source-file* is an EDIT file, and it exceeds 99,999 lines, the DDL compiler issues FILE ERROR - filename - Edit file line number too large (537) on page A-17.

Default: home terminal

!

purges the contents of *pascal-source-file* before opening it, if it exists. If *pascal-source-file* does not exist, the exclamation point has no effect.

Without the exclamation point, the DDL compiler appends the new Pascal source code statements to the end of *pascal-source-file*.

NOPASCAL

closes any open Pascal source code file and stops translating DDL object definitions to Pascal source code statements.

The specified Pascal source code file must be an EDIT file, an unstructured file, or a sequential device such as a terminal, a spooler, or a process. If the file exists but is not one of these types, the DDL compiler issues an error message and does not open the file.

Only one Pascal source code file can be open at a time. If you use the Pascal command when you already have a Pascal source code file open, the DDL compiler closes the current source code file before opening the new source code file.

If the Pascal source code file already exists and the exclamation point is omitted, the DDL compiler appends the DDL objects to the end of the file's original contents. The DDL compiler does not replace any existing objects.

The compiler can translate DDL objects specified in an OUTPUT statement only if the dictionary containing these objects is open.

Each DDL object translated to Pascal source code is written to the Pascal source code file in a separate section that has the same name as the DDL object it contains. You can suppress the generation of individual section headings with the SETSECTION command.

The DDL compiler replaces any hyphen in a DDL name with an underscore (_) before writing the name to the Pascal source code file.

The DDL compiler replaces any field reference dot character (.) in a primary or alternate-key name by an underscore before writing the name to the Pascal source code file.

Before writing a name to a Pascal source code file, the DDL compiler:

- Appends _DEF to every group definition name and record name (but not to any field definition name)
- Appends _KEY to every primary-key name and alternate-key name

As a result, the maximum length for the name of a DDL group definition, record, or key that is to be written to Pascal is 27 ASCII characters, not the standard DDL name length of 30 characters.

For the data types that the DDL compiler generates for Pascal source code, see Table C-4 on page C-7.

The DDL compiler performs all of the syntax checks listed under the PASCALCHECK command before writing the Pascal source output. If the DDL compiler finds a syntax error, it does not write the source output for the object with the error; it does write source output for an object if only a warning is issued.

When generating Pascal source code, the DDL compiler ignores these clauses:

- DISPLAY
- HEADING
- HELP
- MUST BE
- NULL
- OCCURS DEPENDING ON
- TACL
- UPSHIFT
- VALUE
- 66 RENAMES
- 88 condition-name

In Example 9-43 on page 9-88, the DDL compiler retrieves the record CUSTOMER from the open dictionary, translates it to Pascal source code, and appends the source code to the open Pascal file. For the DDL definition of the CUSTOMER record, see the sample database schema in Appendix B, Sample Schemas.

Example 9-43. PASCAL Command

DDL Input

```
39> DDL
!?DICT
!?Pascal $data.sales.passrc
!OUTPUT RECORD customer.
!EXIT
```

Example 9-43. PASCAL Command

DDL Output (CUSTOMER Record in PASSRC)

```
?Section CUSTOMER
{ Definition for CUSTOMER Record }
{ Contains customer information for each customer }
TYPE CUSTOMER_DEF = RECORD
{FILE IS "$data.sales.customer" KEY-SEQUENCED.}
   CUSTNUM                          : CUSTNUM_DEF;
   CUSTNAME                         : NAME_DEF;
   ADDR                             : ADDR_DEF;
END;
CONST CUSTOMER_CUSTNUM_KEY = 0;
CONST CUSTOMER_CUSTNAME_KEY = 25454;
```

PASCALBOUND (D-Series Systems Only)

The PASCALBOUND command sets the lower bound for Pascal arrays.

```
PASCALBOUND { 0 | 1 }
```

Default: PASCALBOUND 1

`0`

assigns any subsequent Pascal arrays a lower bound of 0.

`1`

assigns any subsequent Pascal arrays a lower bound of 1.

The DDL compiler stores the value of the lower bound in the dictionary with the field or group definition.

You can use the PASCALBOUND command as often as you need to set different bounds for different arrays.

The DDL compiler uses the value in the PASCALBOUND command when writing an element to the dictionary. After an element is in the dictionary, changing the PASCALBOUND value has no effect on the Pascal output for that element. To change the PASCALBOUND value for an entered element, you must replace the element in the dictionary.

Pascal arrays are declared for fields and groups described with an OCCURS clause, for fields described as TYPE CHARACTER, and for all fields described with an alphanumeric picture.

If you specify PASCALBOUND 0, the array bounds are:

`[0:`*`number`*` - 1]`

In the array bounds, *number* is the number of occurrences of a field described with an OCCURS clause, or the number of characters in a field described with an alphanumeric PICTURE or a TYPE CHARACTER clause.

If you specify PASCALBOUND 1, the array bounds are:

[1:*number*]

Example 9-44. PASCALBOUND Command

```
?PASCAL                     Open Pascal source code file.
...                         PASCALBOUND is 1 by default.

?PASCALBOUND 0              Change PASCALBOUND to 0.
DEF test0 PIC X(10).

?PASCALBOUND 1              Return to default setting
DEF test1 PIC X(10).
```

PASCALCHECK (D-Series Systems Only)

The [NO]PASCALCHECK command performs [suppresses] Pascal syntax checks on subsequent data descriptions without generating code.

```
[NO]PASCALCHECK
```

Default: PASCALCHECK if a Pascal source code file is open, otherwise NOPASCALCHECK

PASCALCHECK

performs Pascal syntax checks as though Pascal source code were being produced.

NOPASCALCHECK

suppresses Pascal syntax checks.

You can stop Pascal syntax checking by specifying NOPASCALCHECK; you can restart checking with a subsequent PASCALCHECK.

The DDL compiler does not perform the lengthy testing performed by the Pascal compiler. The DDL compiler tests the DDL statements to ensure that they follow these Pascal rules:

- A name cannot be longer than 31 ASCII characters. A name might become longer because the DDL compiler appends _DEF or _KEY to the end of the name of a definition, record, or key.
- Pascal reserved words cannot be DDL names.
- A Pascal named substructure that contains word data must be word aligned.

Example 9-45. PASCALCHECK Command

```
?PASCALCHECK                                          Start syntax checking.
DEF TRANSPORT.
  02 CASE PIC X(10).
  02 ORIGIN PIC X(10).
  02 DESTIN PIC X(10).
  02 LABEL PIC X(10).
  02 PACKED PIC X(10).
END.

Definition TRANSPORT size is 50 bytes.
Definition TRANSPORT added to dictionary.
  *** WARNING *** PASCAL OUTPUT DIAGNOSTICS:
  *** ERROR *** Reserved word - CASE
  *** ERROR *** Reserved word - LABEL
  *** ERROR *** Reserved word - PACKED
  *** WARNING *** Errors detected - no output for TRANSPORT

?NOPASCALCHECK                                        Stop syntax checking.
```

When PASCALCHECK is in effect, DDL issues the following message for each DDL object that passes the syntax check:

```
PASCAL CHECK completed for name
```

In this message, *name* is the name of the object checked by PASCALCHECK.

PASCALNAMEDVARIANT (D-Series Only)

The [NO]PASCALNAMEDVARIANT command generates the REDEFINES clause in the last element as a named [anonymous] variant record in Pascal output.

```
[NO]PASCALNAMEDVARIANT
```

Default: NOPASCALNAMEDVARIANT

PASCALNAMEDVARIANT

generates the REDEFINES clause in the last element as a named variant record in Pascal output.

NOPASCALNAMEDVARIANT

generates the REDEFINES clause in the last element as an anonymous variant record in Pascal output.

REPORT

The [NO]REPORT command:

- Opens [closes] a report file
- Starts [stops] writing a schema report to the report file

```
{ REPORT [ report-destination [ ! ] ] }
{ NOREPORT                            }
```

Default: NOREPORT

`REPORT`

closes any open report file, opens *report-destination*, and writes a schema report to that file.

report-destination

is the name of the file or output device to which the report is sent.

Default: home terminal

`!`

purges the contents of *report-destination* before opening it, if it exists. If *report-destination* does not exist, the exclamation point has no effect.

Without the exclamation point, the DDL compiler appends the new schema report to the end of *report-destination*.

`NOREPORT`

closes the report file.

The REPORT command produces a schema report when the DDL compiler compiles the schema; REPORT can be placed in a RUN DDL command or in the schema itself.

The report on each DDL object is written to a separate report page.

For each field in the schema, the report provides this information:

- Level number
- Name
- Offset in bytes from the start of a group or record
- Data type
- Size

If the field is defined by reference, the report also provides the source definition.

If the report destination you specify is an existing file, the DDL compiler appends the report to the contents of the file.

To produce a schema report on the output device $S.#PRINTER, you can enter the noninteractive command:

```
40> DDL/IN myschema/DICT, REPORT $S.#printer
```

You can generate the same report interactively:

```
41> DDL DICT
!?REPORT $S.#printer
!?SOURCE myschema
!EXIT
```

If any object is defined by reference to another object, the name of the referenced object appears under the heading “Source Definition.”

You do not need the schema to produce a report. You can have the DDL compiler generate the information from the open dictionary, as in Example 9-46 on page 9-93.

Example 9-46. REPORT Command

DDL Input

```
42> DDL
!?DICT                          Open the dictionary.
!?REPORT rptsrc !               Open the report file.
!OUTPUT DEF name.               Send the definition to RPTSRC.
!EXIT
```

DDL Output (Report in RPTSRC)

```
Dictionary Subvol: $BOOKS1.DDL

Definition NAME created.

    Num   LV Element Name          Offset Data Type & Size
Source Definition

    001    01 NAME                     0     Group 22
    002     02 LAST-NAME               0     Character 12
    003     02 FIRST-NAME             12     Character 8
    004     02 MIDINIT                20     Character 2

        Definition size is 22 bytes.
```

RESET

The RESET command stops compiling the current statement and returns to the state before compilation of that statement began.

```
RESET
```

Use RESET only in interactive sessions. (It functions in the noninteractive mode, but is more useful in the interactive mode.)

Use RESET whenever an error or series of errors makes it difficult to continue compilation.

Example 9-47. RESET Command

```
43> DDL
!?DICT
!DEF aa.
!  02 bb PIC X(4).
!  03 cc PIC 9(6).                          Wrong level number

*** ERROR *** Invalid lexical level

!?RESET                                     Reset DDL parser and continue
!DEF aa.
!  02 bb PIC X(4).
!  02 cc PIC 9(6).
!  02 dd.
!  03 dl PIC X(12).
!  03 ff PIC XX.
END
```

SAVE

The [NO]SAVE command saves [purges] the open dictionary when the dictionary is closed.

```
[NO]SAVE
```

Default: SAVE

SAVE

saves the open dictionary when the dictionary is closed.

NOSAVE

purges the contents of the open dictionary when the dictionary is closed unless the dictionary either:

- Was opened for read-only access with the command DICTR on page 9-51
- Is part of a Pathmaker project

The open dictionary closes in any of these situations:

- The NODICT command executes (see DICT on page 9-47)
- The DICT command opens another dictionary
- Compilation stops

If an existing dictionary is opened for update and NOSAVE is in effect when the dictionary is closed, the contents of the dictionary are purged. The NOSAVE command is ignored if the dictionary is part of a Pathmaker project.

If the DDL compiler encounters an error while processing a statement that describes a DDL object, it does not add that object to the dictionary. If the dictionary is saved (either by default or because of an explicit SAVE command), it does not contain all the DDL objects specified in the schema.

You can use the NOSAVE, ERRORS, and SAVE commands to ensure that a dictionary is saved only if the entire schema is compiled without errors in one of two ways:

- Put a NOSAVE command and an ERRORS 1 command at the beginning of the schema.
- Put a SAVE command at the end of the schema.

If the DDL compiler encounters an error, compilation stops while NOSAVE is in effect, and the dictionary is not saved. The dictionary is saved only if compilation completes with no errors; thus, the dictionary either contains all the requested objects or it is purged.

In Example 9-48 on page 9-96, ERRORS 1 directs the DDL compiler to cease processing the schema when it encounters the first error. While NOSAVE is in effect, it directs the compiler to purge the dictionary when compilation stops. The SAVE command is executed only if compilation reaches the SAVE line with no errors.

Example 9-48. SAVE Command

```
44> DDL
!?ERRORS 1                          First line of DDL schema
!?NOSAVE
!?DICT $data.sales
...                                 Body of DDL schema
!?SAVE                              Last line of DDL schema
!EXIT
```

SECTION

The SECTION command names a section of a DDL schema (without affecting the section headings in host-language source code files).

```
SECTION section-name
```

section-name

is the name of a section.

A section is defined as all the source lines following a SECTION command, up to and including the last line before the next SECTION command or the end of the DDL schema.

You can divide a DDL schema into any number of sections.

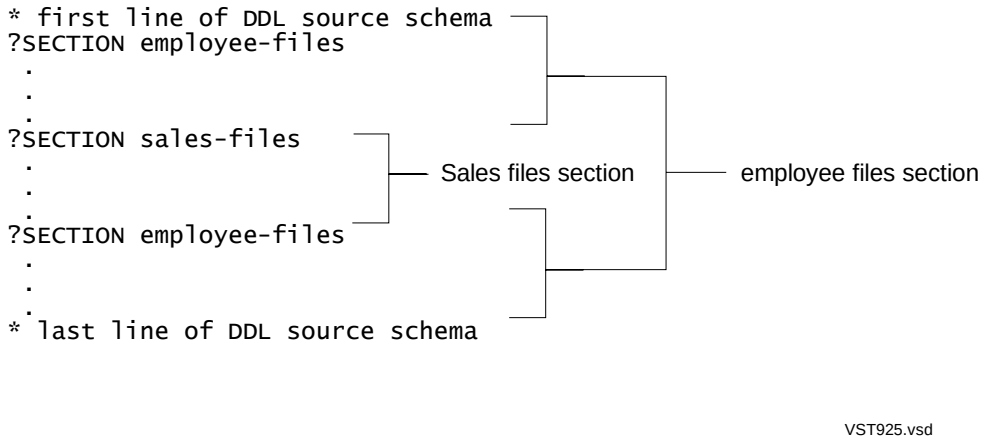
You can use the SOURCE command to include selected sections of a DDL schema file.

When specified in a SOURCE command, sections with the same name are grouped together during compilation.

The SECTION command only names sections in a schema. You can use the SETSECTION command to specify or suppress section names in host-language source-code output.

The source code file in Figure 9-1 on page 9-97 has two sections: SALES-FILES and EMPLOYEE-FILES. The EMPLOYEE-FILES section is made up of two portions of the schema separated by the SALES-FILES section.

Figure 9-1. SECTION Command



SETLOCALENAME

The SETLOCALENAME command specifies the language, territory, and character set for output of text items.

```
SETLOCALENAME [ locale-name ]
```

locale-name

specifies a language, territory, and character set for a text item.

Default: default system locale

When a programming language file is generated, the value associated with the specified locale will be output for each text item.

If multiple SETLOCALENAME commands are issued, the last one issued is in effect.

The SETLOCALENAME command can be set anytime. Different locales can be used when generating a programming language source program.

If there is not a text item with a locale name that matches the one given in the SETLOCALENAME command, an error occurs.

If there is more than one literal specified with the same locale name for a text item, an error occurs. The literal with the duplicate locale name is ignored.

Example 9-49 on page 9-98 shows the use of the SETLOCALENAME command to generate output for text items in French.

Example 9-49. SETLOCALENAME Command

DDL Input

```
? DICT !
? COBOL COBSRC !
? SETLOCALENAME no_NO.ISO8859-1
CONSTANT custnum-heading VALUE "Finnish" LN"fi_FI.ISO8859-1"
                               "Norwegian" LN"no_NO.ISO8859-1"
                               "Danish" LN"da_DK.ISO8859-1".
```

DDL Output (COBOL Code)

```
* SCHEMA PRODUCED DATE - TIME : 11/16/1994 16:17:21
?SECTION CUSTNUM-HEADING TANDEM
* Constant CUSTNUM-HEADING created on 11/16/1994 at 16:17
   01 CUSTNUM-HEADING PIC X(7), VALUE IS "Norwegian".
```

SETSECTION

The SETSECTION command determines SECTION headings for all open source code files except TACL source code files.

```
SETSECTION [ section-name ]
```

Default: SETSECTION without *section-name*

`SETSECTION` *section-name*

immediately generates a SECTION heading with the name *section-name* on all open source code files except TACL source code files, and generates no other SECTION headings.

Remains in effect until another SETSECTION command appears or the DDL session ends.

`SETSECTION`

generates a separate SECTION heading, with the object name as the section name, for each subsequent DDL object in the open source code file except TACL source code files.

Remains in effect until a SETSECTION *section-name* command appears or the DDL session ends.

If SETSECTION is not specified or if SETSECTION is specified without *section-name*, the DDL compiler precedes each DDL object in the open source code files with a SECTION heading that uses the DDL object name as the section name.

The SETSECTION command affects only host-language source code files (except TACL source code files) and DDL source code files opened by the DDL command.

If you give a SETSECTION command with a name, and then open a host-language source code file, no SECTION commands are written to the file. The SECTION command for the given name is not written because the file was not open at the time. Also, the SETSECTION command inhibits any SECTION commands for individual objects.

SETSECTION does not specify sections in the DDL schema being compiled. You use the SECTION command to specify section names in a schema in order to selectively compile source sections with the SOURCE command.

Example 9-50 on page 9-99 generates two source-code sections: one for constants and one for definitions.

Example 9-50. SETSECTION Command

DDL Input

```
?SETSECTION constants
CONSTANT custnum-heading   VALUE "Customer/Number".
CONSTANT mdy-date-display  VALUE "mm/dd/yy".
CONSTANT phone-display     VALUE "M(999) 999-9999".
...
   ?SETSECTION defs
DEF deliv-date     PIC 9(6)  DISPLAY mdy-date-display.
DEF custnum        PIC 9(4)  HEADING custnum-heading.
DEF custphone      PIC 9(10) DISPLAY phone-display.
...
```

DDL Output (COBOL Code)

```
?Section CONSTANTS,Tandem
   01 CUSTNUM-HEADING  PIC X(15), VALUE IS "Customer/Number".
   01 MDY-DATE-DISPLAY PIC X(11), VALUE IS "M99/99/99".
   01 PHONE DISPLAY    PIC X(17), VALUE IS "M(999) 999-9999".
   ...
   ?Section DEFS,Tandem
   01 CUSTNUM          PIC 9(4).
   01 CUSTPHONE        PIC 9(10).
   01 DELIV-DATE       PIC 9(6).
   ...
```

SOURCE

The SOURCE command compiles all or part a specified DDL schema.

```
SOURCE source-name [ ( section-name [ , section-name ] ...) ]
```

source-name

is the name of the file that contains the schema file to be compiled.

section-name

is the name of a section within the schema file. (Section names are specified with the command SECTION on page 9-96.)

If you specify one or more sections, the DDL compiler compiles only the specified sections.

If you do not specify sections, the DDL compiler compiles the entire schema.

A schema file is an EDIT file that contains DDL statements and commands; it can be either a file created with the EDIT program or a DDL source code file created with the DDL command.

If you specify more than one section, the sections are compiled in the order they occur in the source code file.

A single SOURCE command can extend over more than one input line. The first line begins with SOURCE, and each subsequent line begins with a question mark.

schema files can be nested; that is, schema A can contain a SOURCE command specifying schema B, and schema B can contain a SOURCE command specifying schema C.

If the DDL compiler is compiling a schema file and it encounters a SOURCE command in that file, the DDL compiler:

1. Suspends compilation of the current file.
2. Opens the file specified in the SOURCE command and compiles either the entire file or the specified sections.
3. Includes the compiled file (or sections) in the current file at the point where it encounters the SOURCE command.
4. Resumes compiling the current file.

In Example 9-51 on page 9-100, the DDL compiler opens a new DDL source code file called NEWSRC. The DDL compiler first compiles all of FILE1 and writes it to NEWSRC; then compiles SECT-1 and SECT-3 of FILE2 and appends them to the contents of NEWSRC; and lastly, compiles SECT-1 and SECT-5 of FILE3 and appends them to the end of NEWSRC.

Example 9-51. SOURCE Command

```
45> DDL
!?DICT
!?DDL newsrc
!?SOURCE file1
!?SOURCE file2 (sect-1, sect-3)
!?SOURCE file3 (sect-1,!          Continuation line
                sect-5)
!EXIT
```

SPACING

The SPACING command specifies the number of blank lines to insert between lines of a printed report.

```
SPACING { 0 | 1 | 2 }
```

Default: SPACING 0

`{ 0 | 1 | 2 }`

is the number of blank lines to insert between lines of a printed report.

The SPACING command controls spacing only on a printed report; it does not affect spacing in a report file or on a terminal display.

You can use the SPACING command as often as you want to within a DDL schema or in a DDL session.

In Example 9-52 on page 9-101, the SPACING command double spaces between lines of a DDL schema report printed on a line printer.

Example 9-52. SPACING Command

```
46> DDL DICT
!?REPORT $S.#printer     Select a printer for the report

!?SPACING 1              Specify double spacing
!OUTPUT DEF name.        Select a definition to print
!EXIT
```

TACL

The [NO]TACL command:

- Opens [closes] a TACL source code file
- Starts [stops] writing translated DDL object definitions to the TACL source code file

```
{ TACL [ tacl-source-file [ ! ] ]
{ NOTACL                        }
```

Default: NOTACL

`TACL`

closes any open TACL source code file, opens *`tacl-source-file`*, translates subsequent DDL objects defined by statements or specified in OUTPUT statements to TACL source code statements, and writes the TACL source code statements to *`tacl-source-file`*.

tacl-source-file

is the name of the TACL source code file to be created, if necessary, and opened. The file must be one of:

- EDIT file
- Unstructured file
- Sequential device (such as a terminal, spooler, or process)

If the file exists but is not one of these types, the DDL compiler issues an error message and does not open the file.

Default: home terminal

!

purges the contents of *tacl-source-file* before opening it, if it exists. If *tacl-source-file* does not exist, the exclamation point has no effect.

Without the exclamation point, the DDL compiler appends the new TACL source code statements to the end of *tacl-source-file*.

NOTACL

closes any open TACL source code file and stops translating DDL object definitions to TACL source code statements.

The specified TACL source code file must be an EDIT file, an unstructured file, or a sequential device such as a terminal, a spooler, or a process. If the file exists but is not one of these types, the DDL compiler issues an error message and does not open the file.

Only one TACL source code file can be open at a time. If you use the TACL command when you already have a TACL source code file open, the DDL compiler closes the current source code file before opening the new source code file.

If the TACL source code file already exists and the exclamation point is omitted, the DDL compiler appends the DDL objects to the end of the file's original contents. The DDL compiler does not replace any existing structures.

The compiler can translate DDL objects specified in an OUTPUT statement only if the dictionary containing these objects is open.

Each DDL object translated to TACL source code is written to the TACL source code file in a separate section that has the same name as the DDL object it contains. You cannot suppress the generation of individual section headings in TACL source code with the SETSECTION command.

Any DDL object defined in a DDL CONSTANT, DEFINITION, RECORD, TOKEN-CODE, TOKEN-MAP, or TOKEN-TYPE statement can be translated to TACL data structures.

The DDL compiler replaces any hyphen in a DDL object name with a circumflex (^) before writing the name to the TACL source code file.

The DDL compiler translates each DDL object (except constants) to one or more TACL structures; the DDL compiler translates constants to TACL text variables.

For the data types that the DDL compiler generates for TACL source code, see Table C-5 on page C-9.

When translating a definition or record, the DDL compiler generates a TACL structure corresponding to the data type of each field or group of fields in the definition or record, unless the field or group is defined with a TACL clause.

If a field definition or a field or group description includes a TACL clause, the DDL compiler generates a TACL structure with the high-level TACL data type specified in the TACL clause.

The DDL compiler performs these checks before generating TACL source code:

- Checks whether a definition or record contains a data type not supported by TACL; if so, the DDL compiler issues a warning.
- Checks whether a definition, record, or token map generates a TACL structure with more than 5,000 bytes; if so, the DDL compiler issues an error message and does not generate the object.
- Checks whether a CONSTANT generates a TACL text variable with a value greater than 130 ASCII characters, including any tildes (~) emitted by the DDL compiler; if so, the DDL compiler issues an error message and does not generate the object.

In Example 9-53 on page 9-104, the DDL compiler retrieves the record CUSTOMER from the open dictionary, translates it to TACL source code, and writes the source code to the file \DALLAS.$DATA.SALES.TACLSRC. If this file already exists, the DDL compiler appends the entry for CUSTOMER to the file. For the definition of the CUSTOMER record, see the sample database schema in Appendix B, Sample Schemas.

Example 9-53. TACL Command

DDL Input

```
?DICT
?TACL \dallas.$data.sales.taclsrc
OUTPUT RECORD customer.
```

DDL Output (TACL Code)

```
?Section CUSTOMER Struct
Begin
STRUCT   CUSTNUM;
   BEGIN CHAR BYTE(0:3); END;
STRUCT   CUSTNAME;
   Begin
   STRUCT   LAST^NAME;
      BEGIN CHAR BYTE(0:11); END;
   STRUCT   FIRST^NAME;
      BEGIN CHAR BYTE(0:7); END;
   STRUCT   MIDINIT;
      BEGIN CHAR BYTE(0:1); END;
   End;
STRUCT   ADDR;
   Begin
   STRUCT   ADDRESS;
      BEGIN CHAR BYTE(0:21); END;
   STRUCT   CITY;
      BEGIN CHAR BYTE(0:13); END;
   STRUCT   STATE;
      BEGIN CHAR BYTE(0:1); END;
   STRUCT   ZIP;
      BEGIN CHAR BYTE(0:4); END;
   End;
End;
```

TACLGEN

The TACLGEN command specifies a TACL source code generation product version.

```
TACLGEN 0
```

Default: TACLGEN 0

`0`

specifies the current product version of TACL.

Because the DDL compiler generates only one product version of TACL code at the current time, the TACLGEN command does not affect output.

TAL

The [NO]TAL command:

- Opens [closes] a pTAL or TAL source code file
- Starts [stops] writing translated DDL object definitions to the pTAL or TAL source code file

```
{ TAL [ tal-source-file [ ! ] ] }
{ NOTAL                       }
```

Default: NOTAL

`TAL`

closes any open pTAL or TAL source code file, opens *`tal-source-file`*, translates subsequent DDL objects defined by statements or specified in OUTPUT statements to pTAL or TAL source code statements, and writes the pTAL or TAL source code statements to *`tal-source-file`*.

`tal-source-file`

is the name of the pTAL or TAL source code file to be created, if necessary, and opened. The file must be one of:

- EDIT file
- Unstructured file
- Sequential device (such as a terminal, spooler, or process)

If the file exists but is not one of these types, the DDL compiler issues an error message and does not open the file.

If *`tal-source-file`* is an EDIT file, and it exceeds 99,999 lines, the DDL compiler issues FILE ERROR - filename - Edit file line number too large (537) on page A-17.

Default: home terminal

`!`

purges the contents of *`tal-source-file`* before opening it, if it exists. If *`tal-source-file`* does not exist, the exclamation point has no effect.

Without the exclamation point, the DDL compiler appends the new pTAL or TAL source code statements to the end of *`tal-source-file`*.

`NOTAL`

closes any open pTAL or TAL source code file and stops translating DDL object definitions to TACL source code statements.

The specified pTAL or TAL source code file must be an EDIT file, an unstructured file, or a sequential device such as a terminal, a spooler, or a process. If the file exists but is not one of these types, the DDL compiler issues an error message and does not open the file.

Only one pTAL or TAL source code file can be open at a time. If you use the TAL command when you already have a pTAL or TAL source code file open, the DDL compiler closes the current source code file before opening the new source code file.

If the pTAL or TAL source code file already exists and the exclamation point is omitted, the DDL compiler appends the DDL objects to the end of the file's original contents. The DDL compiler does not replace any existing objects.

The compiler can translate DDL objects specified in an OUTPUT statement only if the dictionary containing these objects is open.

Each DDL object translated to pTAL or TAL source code is written to the pTAL or TAL source code file in a separate section that has the same name as the DDL object it contains. You can suppress the generation of individual section headings with the SETSECTION command.

The DDL compiler translates complex objects, such as group definitions and group records, to pTAL or TAL STRUCT template declarations. The DDL compiler translates simple objects, such as field definitions and records containing only one field and no groups, to simple pTAL or TAL variables or to pTAL or TAL STRUCT declarations, rather than to STRUCT template declarations. (But see TALALLOCATE on page 9-108.)

Unless the command TALUNDERSCORE on page 9-111 is in effect, the DDL compiler replaces any hyphen in a DDL name with a circumflex (^) before writing the name to the pTAL or TAL source code file.

Before writing a name to a pTAL or TAL source code file, the DDL compiler:

- Appends ^DEF to every group definition name and record name (but not to any field definition name)
- Appends ^WLN to every SPI TOKEN-MAP name
- Appends ^KEY to every primary-key name and alternate-key name

As a result, the maximum length for the name of a DDL group definition, record, token map, or key that is to be written to pTAL or TAL is 27 ASCII characters, not the standard DDL name length of 30 characters.

The pTAL or TAL source code for a definition or record compiled with the CFIELDALIGN_MATCHED2 command contains the fillers added by the DDL compiler as specified by the alignment algorithm in effect when the definition or record was compiled.

For the data types that the DDL compiler generates for pTAL or TAL source code, see Table C-6 on page C-11.

The DDL compiler performs all of the syntax checks listed under the TALCHECK command before writing the pTAL or TAL source output. If the DDL compiler finds a syntax error, it does not write the source output for the object with the error; it does write source output for an object if only a warning is issued.

The lower bound for pTAL or TAL arrays can be set with the TALBOUND command.

In Example 9-54 on page 9-107, the DDL compiler opens the dictionary, opens and then clears the file \DALLAS.$DATA.SALES.TALSRC, retrieves the record CUSTOMER from the open dictionary, translates it to pTAL or TAL source code, and then writes it to the open pTAL or TAL source code file. For the definition of the CUSTOMER record, see the sample database schema in Appendix B, Sample Schemas.

Example 9-54. TAL Command

```
47> DDL
!?DICT
!?TAL \dallas.$data.sales.talsrc !
!OUTPUT RECORD customer.
?SECTION CUSTOMER
?PAGE
STRUCT CUSTOMER^DEF (*);
  BEGIN
  STRUCT CUSTNUM;
    BEGIN STRING BYTE [1:4]; END;
  STRUCT CUSTNAME;
  BEGIN
    STRUCT LAST^NAME;
      BEGIN STRING BYTE [1:12]; END;
    !Upshift
    STRUCT FIRST^NAME;
      BEGIN STRING BYTE [1:8]; END;
    !Upshift
    STRUCT MIDINIT;
      BEGIN STRING BYTE [1:2]; END;
    !Upshift
  END;
STRUCT ADDR;
  BEGIN
  STRUCT ADDRESS;
    BEGIN STRING BYTE [1:22]; END;
  STRUCT CITY;
    BEGIN STRING BYTE [1:14]; END;
  STRUCT STATE;
    BEGIN STRING BYTE [1:2]; END;
  STRUCT ZIP-CODE;
    BEGIN STRING BYTE [1:5]; END;
  END;
END;
LITERAL CUSTOMER^CUSTNUM^KEY = %000000;
LITERAL CUSTOMER^CUSTNAME^KEY = %061556;  !"cn"
```

TALALLOCATE

The [NO]TALALLOCATE command causes [suppresses] memory allocation in pTAL or TAL for single-field definitions when the TAL command is in effect.

```
[NO]TALALLOCATE
```

Default: TALALLOCATE

TALALLOCATE

allocates memory for single-field definitions when the TAL command is in effect.

NOTALALLOCATE

suppresses memory allocation when the TAL command is in effect, causing the DDL compiler to translate single-field definitions to pTAL or TAL DEFINEs or STRUCT templates.

Example 9-55. TALALLOCATE Command

DDL Type	pTAL or TAL Type

?TALALLOCATE
DEF status TYPE ENUM BEGIN.
  89 no-error.
  89 read-error.
  89 write-error VALUE 6.
END.
	LITERAL NO^ERROR = 0,
	  READ^ERROR = 1,
	  WRITE^ERROR = 6;
	INT STATUS;

DEF letter Pic "X".	STRING LETTER;
DEF number Pic "9(5)".	STRUCT NUMBER; BEGIN STRING BYTE [1:5]; END;

?NOTALALLOCATE
DEF status TYPE ENUM BEGIN.
  89 no-error.
  89 read-error.
  89 write-error VALUE 6.
END.
	LITERAL NO^ERROR = 0,
	  READ^ERROR = 1,
	  WRITE^ERROR = 6;
	DEFINE STATUS = INT #;

DEF letter Pic "X".	DEFINE LETTER = STRING #;
DEF number Pic "9(5)".	STRUCT NUMBER (*); BEGIN STRING BYTE [1:5]; END;

TALBOUND

The TALBOUND command sets the lower bound for pTAL or TAL arrays.

```
TALBOUND { 0 | 1 }
```

Default: TALBOUND 1

`0`

assigns any subsequent pTAL or TAL arrays a lower bound of 0.

`1`

assigns any subsequent pTAL or TAL arrays a lower bound of 1.

The DDL compiler stores the value of the lower bound in the dictionary with the field or group definition.

You can use the TALBOUND command as often as needed to set different bounds for different arrays.

The DDL compiler uses the value specified by the TALBOUND command when it writes an element to the dictionary. After an element is in the dictionary, changing the TALBOUND value has no effect on the pTAL or TAL output for that element. To change the TALBOUND value for an entered element, you must replace the element in the dictionary.

pTAL or TAL arrays are declared for fields and groups described with an OCCURS clause, for fields described as TYPE CHARACTER, and for all fields described with an alphanumeric PICTURE.

If you specify TALBOUND 0, the array bounds are:

`[0:`*`number`*` - 1]`

In the array bounds, *`number`* is the number of occurrences of a field described with an OCCURS clause, or the number of characters in a field described with an alphanumeric PICTURE or a TYPE CHARACTER clause. If you specify TALBOUND 1, the array bounds are:

`[1:`*`number`*` ]`

Example 9-56. TALBOUND Command

DDL Input

```
?TAL                                    Open TAL source code file
...                                     TALBOUND is 1 by default

?TALBOUND 0                             Change TALBOUND to 0
DEF test0 PIC X(10).

?TALBOUND 1                             Return to default setting
DEF test1 PIC X(10).
```

DDL Output (pTAL or TAL Code)

```
?SECTION TEST0
STRUCT TEST0

BEGIN STRING BYTE [0:9]; END;           TALBOUND 0 in source

?SECTION TEST1
STRUCT TEST1

BEGIN STRING BYTE [0:10]; END;          TALBOUND 1 in source
```

TALCHECK

The [NO]TALCHECK command performs [suppresses] pTAL or TAL syntax checking on subsequent data descriptions without generating code.

```
[NO]TALCHECK
```

Default: TALCHECK if a TAL or pTAL source code file is open, otherwise NOTAL CHECK

`TALCHECK`

performs pTAL or TAL syntax checks as though pTAL or TAL source code were being produced.

`NOTALCHECK`

suppresses pTAL or TAL syntax checks.

If a pTAL or TAL source code file is open, the compiler performs checks whether or not TALCHECK is set.

You can stop pTAL or TAL syntax checking by specifying NOTALCHECK; you can restart checking with a subsequent TALCHECK.

The DDL compiler does not perform the lengthy testing performed by the pTAL or TAL compiler. The DDL compiler tests the DDL statements to ensure that they follow the rules specified by pTAL or TAL:

- pTAL or TAL reserved words cannot be DDL names.
- A constant value must not be greater than its defined limit.
- A name cannot be longer than 31 ASCII characters, including suffixes (such as ^DEF, ^WLN, or ^KEY).
- A REDEFINES clause cannot be at the level directly following that of a definition or record.

Example 9-57. TALCHECK Command

```
?TALCHECK                                              Start syntax checking
RECORD location.
FILE IS "$data.sales.location" Key-sequenced.
  02 resident PIC X(15).
  02 loc PIC X(3).
  02 code PIC 999.
  KEY IS resident.
END
Record LOCATION size is 21 bytes.
*** WARNING *** TAL OUTPUT DIAGNOSTICS:
*** ERROR *** Reserved word - RESIDENT
*** ERROR *** Reserved word - CODE

?NOTALCHECK                                            Stop syntax checking
```

When TALCHECK is in effect, the DDL compiler issues the following message for each DDL object that passes the syntax check:

```
TAL CHECK completed for name
```

In the message, *name* is the name of the object checked by TALCHECK.

TALUNDERSCORE

The [NO]TALUNDERSCORE command replaces hyphens with underscores [circumflexes] in DDL names for pTAL or TAL output.

```
[NO]TALUNDERSCORE
```

Default: NOTALUNDERSCORE

TALUNDERSCORE

replaces each hyphen (-) with an underscore (_) in DDL names for pTAL or TAL output.

NOTALUNDERSCORE

replaces each hyphen (-) with a circumflex (^) in DDL names for pTAL or TAL output.

Example 9-58. TALUNDERSCORE Command

DDL Input

```
CONSTANT This-Is-A-Literal VALUE is 99.
```

DDL Output (pTAL or TAL Code) with TALUNDERSCORE

```
LITERAL This_Is_A_Literal = 99;
```

DDL Output (pTAL or TAL Code) with NOTALUNDERSCORE

```
LITERAL This^Is^A^Literal = 99;
```

TEDIT

The TEDIT command:

- Suspends compilation
- Starts a PS Text Edit (TEDIT) process
- Opens the specified file, executes the specified commands, and closes the file
- Resumes compilation when the TEDIT process stops

You can use TEDIT only in an interactive DDL session.

```
TEDIT [ edit-file-name [ ; edit-parameter ] ... ]
```

edit-file-name

is the name of an EDIT file.

Default: The most recent *edit-file-name* specified in the current DDL session, if any. If none, you are prompted for a file name.

edit-parameter

is a PS Text Edit command.

Default: The most recent *edit-file-name* specified in the current DDL session, if any. If none, you are prompted for a PS Text Edit command.

Issuing the TEDIT command within a DDL session is like issuing the TEDIT command from the command interpreter; the PS Text Edit session is the same, and you can use all the same functions.

When you stop a PS Text Edit process by issuing the EXIT command, control returns to the DDL compiler.

You must close any source code file before editing it. For instance, if you have opened a COBOL source code file and entered some text in it, and then you want to view the source code file with the text editor, you must issue the NOCOBOL command before you issue the TEDIT command.

When you specify *edit-file-name* in the TEDIT command, the DDL compiler passes that name to the current PS Text Edit process and also stores the name. If you omit *edit-file-name* from the next TEDIT command in the same session, the DDL compiler passes the stored name to the new PS Text Edit process.

When you specify *edit-parameter* in the TEDIT command, the DDL compiler passes that parameter to the PS Text Edit process. The DDL compiler also stores the parameter.

If you omit *edit-file-name* from the next TEDIT command, the DDL compiler passes any parameter saved from the last TEDIT or EDIT command to the new PS Text Edit process.

If you specify *edit-file-name* in the next TEDIT command, the DDL compiler discards any previously stored parameter.

Example 9-59. TEDIT Command

```
48> DDL DICT
!?DDL ddlfil                    Open DDLFIL
!RECORD sum.                    Add a record
...
!?NODDL                         Close DDLFIL
!?TEDIT ddlfil                  Start a PS TEXT EDIT process
...
!?SOURCE ddlfil                 Add the record to the dictionary
!?TEDIT                         Use the previous file
...
```

TIMESTAMP

The [NO]TIMESTAMP command includes [excludes] data and time comments in [from] source code listings.

```
[NO]TIMESTAMP
```

Default: TIMESTAMP

`TIMESTAMP`

includes date and time comments in source code listings.

NOTIMESTAMP

excludes date and time comments from source code listings.

The DDL compiler produces a number of starred timestamp comments on the listings of DDL or host-language source code. A comment at the beginning of the listing tells the date and time the schema was produced; individual comment lines preceding each section tell when each DDL object was created.

You can suppress these timestamp comments with NOTIMESTAMP and then include them with a subsequent TIMESTAMP.

Example 9-60 on page 9-114 shows selective listing or suppression of the timestamp comments in a COBOL source code file.

Example 9-60. TIMESTAMP Command

DDL Input

```
?COBOL cobsrc
DEF aa   PIC X(8).
?NOTIMESTAMP
DEF bb   PIC 9(5).
    .
    .
    .
?TIMESTAMP
RECORD rec1.  FILE IS $data.sales.rec1  KEY-SEQUENCED.
  02 aa  TYPE *.
  02 bb  TYPE *.
END
```

DDL Output (COBOL Code)

```
*SCHEMA PRODUCED DATE - TIME : 4/30/1991 12:29:35   Timestamp
?SECTION AA, TANDEM

*Definition AA created on 4/30/1991 12:29           Timestamp
01 AA PIC X(8).
?SECTION BB, TANDEM

01 BB PIC 9(5).                                     No timestamp
...
?SECTION REC1, TANDEM

*Definition REC1 created on 4/30/1991 12:29         Timestamp
01 REC1.
02 AA PIC X(8).
02 BB PIC 9(5).
```

VALUES

The [NO]VALUES command includes [excludes] initial values from DEFINITION and RECORD statements in [from] DDL or COBOL source code.

```
[NO]VALUES
```

Default: VALUES

VALUES

includes initial values from DEFINITION and RECORD statements in COBOL or DDL source code.

NOVALUES

excludes initial values from DEFINITION and RECORD statements from DDL or COBOL source code.

The VALUES command does not affect VALUE clauses associated with level-88 items or with CONSTANT, TOKEN-CODE, TOKEN-MAP, or TOKEN-TYPE statements.

When the VALUES command is specified and the DDL compiler is generating source code for FORTRAN, the compiler translates any initial values to comments.

The VALUES command is useful for definitions used in the Linkage sections of COBOL or SCREEN COBOL, where COBOL initial values are not allowed.

Example 9-61 on page 9-115 suppresses initial values for the definition NEW-NAME.

Example 9-61. VALUES and NOVALUES Commands

DDL Input

```
?NOTIMESTAMP                        Suppress timestamp comments
?COBOL cobsrc

DEF new-numb PIC 9(12)              By default, include initial values in COBSRC
  VALUE IS ZEROS.

?NOVALUES                           Suppress initial values
DEF new-name PIC X(18)
  VALUE IS "JONES".

?VALUES                             Include initial values again
```

DDL Output (COBOL Code)

```
?SECTION NEW-NUMB,TANDEM
  01 NEW-NUMB                 PIC 9(12)
                              VALUE ZEROS.
?SECTION NEW-NAME,TANDEM
  01 NEW-NAME                 PIC X(18).
```

WARN

The [NO]WARN command includes [excludes] warnings in [from] the compiler listing.

```
[NO]WARN
```

Default: WARN

`WARN`

includes warnings in the compiler listing.

`NOWARN`

excludes warnings from the compiler listing.

Example 9-62. WARN and NOWARN Commands

```
*start of source code file  List warning messages by default
...
?NOWARN                     Suppress warning messages for subsequent
...                         statements
?WARN                       List warning messages for subsequent statements
...
*end of source code file
```

WARNINGS

The WARNINGS command specifies the number of warnings allowed before compilation stops.

```
WARNINGS [ max-warnings ]
```

Default: Compilation continues until the end of the source code file regardless of the number of warnings

max-warnings

is a number from 1 through 32,767 that specifies the maximum number of warnings allowed before the DDL compiler stops compiling the source code file.

Default: 1

When compilation stops because the specified number of warnings is exceeded, the DDL compiler closes the open dictionary and any open files, issues session statistics, and stops.

The specified maximum number of warnings applies only to warnings that occur after the appearance of the WARNINGS command. For example, if two warnings occur before a WARNINGS 5 command appears, the seventh warning to occur (the fifth warning after the command appeared) stops compilation.

If the NOWARN command is in effect, you cannot use the WARNINGS command.

The WARNINGS command does not count the number of occurrences of conditions that result in a warning, but instead counts the number of messages issued that begin with *** WARNING ***. Some warning conditions can generate more than one such message. For example:

```
*** WARNING *** COBOL OUTPUT DIAGNOSTICS:
*** WARNING *** Unsupported data type in element A
*** WARNING *** Unsupported data type in element B
```

If WARNINGS 3 was specified, compilation stops after this condition occurs.

This WARNINGS command directs the DDL compiler to stop compiling when it encounters the third compilation warning:

```
?WARNINGS 3
```

If a third compilation warning is encountered, the DDL compiler issues the warning message for the third warning, followed by the fatal error message:

```
Too Many Warnings - Compilation Terminating.
```

Example 9-63. WARNINGS Command

```
?SECTION start           Compile regardless of warnings
...
?SECTION rest-of-schema
?WARNINGS 1              Stop compiling if any warning is encountered
...
```

10 Dictionary Maintenance

This section briefly describes these dictionary maintenance procedures:

Note. Changing a dictionary does not change any database described by the dictionary, nor do any changes to a database affect the dictionary.

Generating a schema From a Dictionary

Some reasons to generate a new schema from the dictionary are:

- You can use the new schema as a backup for the dictionary.
- You made so many changes to definitions and records in the dictionary that the original schema is out of date.
- You lost the original schema.

Note. Do not attempt to back up a dictionary that is part of a Pathmaker catalog using the following procedure. Pathmaker dictionaries contain application design information that is not generated in DDL schemas.

To generate a schema from a dictionary:

1. Use the DICT, DICTN, or DICTR command to open the dictionary.
2. Use the DDL command to open a DDL source code file to contain the new schema (this DDL source code file will be the new schema file).
3. Use an OUTPUT statement to generate DDL source statements from the dictionary objects and write them to the new schema file.

Example 10-1. Generating a schema From a Dictionary

`39> DDL`	Run the DDL compiler interactively.
`!?DICT $data.sales`	Open the dictionary.
`!?DDL $data.sales.schembak !`	Open a DDL source file.
`!?OUTPUT *.`	Generate DDL source statements from the dictionary objects and write them to the open DDL source file (which is now the new schema file).
`!EXIT`	End the interactive session.

Adding Dictionary Objects

Adding a new dictionary object is usually easier than modifying an existing object because a new object cannot be referenced by an existing object.

If statements describing new objects refer to previously defined objects in the dictionary, the dictionary must be open.

When you add a new object, you must be careful that it does not have the same name as an existing object of the same object type. If it does have the same name, the DDL compiler replaces the existing object with the new object. If you try to add a new object that has the same name but is of a different type (for example, a new constant with the same name as an existing record), the DDL compiler issues an error message and does not replace the existing object.

You can specify the new object in an interactive DDL session, but errors are not easy to correct in a session. So if the object requires more than a few lines of code, it is generally easier and safer to add a new object noninteractively. To add the object, you specify the appropriate statements in a file, open the dictionary, and compile the file.

To add new objects to the dictionary, do this:

1. Specify the new objects.
2. Use EDIT to create a source file that contains the statements that define the new objects.
3. Open the dictionary. You can include the DICT or DICTN command to open the dictionary in the source file, or you can specify it later when you compile the source file.
4. Compile the source file. You can do this by running the DDL compiler interactively and using a SOURCE command to specify the source file, or you can use the RUN DDL command and specify the source file as the IN parameter.

Example 10-2. Adding a New Record to a Dictionary

Add the record:

```
40> EDIT newsrc !; ADD                                  Specify a new record
   1 RECORD dependents.
   2 FILE IS $data.sales.empdep KEY-SEQUENCED
   3 AUDIT.
   4 O2 dep-key.
   5 04 empnum TYPE *.
   6 04 depnum PIC X(4).
   7 02 depname TYPE name.
   8 02 age PIC 9(2).
   9 02 sex PIC X(2).
   10 88 female VALUE "01".
   11 88 male VALUE "02".
   12 KEY IS dep-key.
   13 END
   14 //
*EXIT
```

View the record:

```
41> DDL DICT $data.sales
!?SOURCE newsrc
    1       RECORD dependents.
    2         FILE IS $data.sales.empdep  KEY-SEQUENCED
    3                                     AUDIT.
    4           02 dep-key.
    5              04  empnum          TYPE *.
    6              04  depnum          PIC X(4).
    7           02  depname            TYPE name.
    8           02  age                PIC 9(2).
    9           02  sex                PIC X(2).
   10               88  female         VALUE "01".
   11               88  male           VALUE "02".
   12         KEY IS dep-key.
   13       END
            Record DEPENDENTS size is 34 bytes.
            Record DEPENDENTS added to dictionary.
!EXIT
```

In Example 10-2 on page 10-3, the record DEPENDENTS contains two fields (EMPNUM and DEPNAME) that refer to existing objects in the dictionary.

Deleting Dictionary Objects

Deleting a dictionary object is comparatively easy if the object is not referenced by other objects in the dictionary. Objects that are never referenced by other objects are:

- Records
- SPI token codes
- SPI token maps

When an object is referenced by other objects, you must first delete the referring objects. Objects that can be referenced by other objects are:

- Constants
- Definitions
- SPI token types

Topics:

Deleting Unreferenced Objects

When you delete a record, an SPI token code, or an SPI token map, you need not be concerned that the deletion affects other objects in the dictionary. These objects are never referenced by other objects. Other objects that can be referenced might also be unreferenced. You can use the SHOW USE OF command to determine whether the object you want to delete is referenced by other objects.

To delete an object that is not referenced by any other object, use a DELETE statement that specifies the object to be deleted. The exact procedure depends on whether you make the deletion interactively or compile a source file containing the DELETE statement. In either case, you must first open the dictionary from which you are deleting the object.

To delete the object from the dictionary, do this:

1. Open the dictionary. The dictionary must be open before the DDL compiler executes the DELETE statement.
2. Specify the DELETE statement or statements. You can specify the statement in an interactive DDL session.
3. Compile the DELETE statement. If you enter the statement in an interactive session, the DDL compiler compiles the statement as you enter it. If the statement is in a source file, you can specify the file as the IN parameter of a RUN DDL command or you can run the DDL compiler and use the SOURCE command to specify the source file.

Example 10-3 on page 10-5 builds a source file containing the code to open the dictionary and delete one record. When the DDL compiler compiles the source file, it opens the dictionary and deletes the record DEPENDENT-INFO from the open dictionary.

Example 10-3. Deleting an Unreferenced Object From a Dictionary

```
42> EDIT delsrc; add
   1 ?DICT $data.sales

   2 DELETE RECORD dependent-info.  Remember the period
   3 //
*EXIT

43> DDL /IN delsrc/                 Send listing to your terminal
```

Deleting Referenced Objects

When you delete a constant, a definition, or a SPI token type that is referenced by any other objects, you must first delete all objects that refer directly or indirectly to the object you want to delete. You can use the SHOW USE OF statement to determine whether the object is referenced and by which other objects.

If the object you want to delete is never referenced by another object, use the technique in Deleting Unreferenced Objects on page 10-4. If the object you want to delete is referenced, you must first delete the referring objects in an exact sequence.

In Example 10-4 on page 10-5, to delete the constant A, you must first delete definition B because it refers to A; however, to delete B, you must first delete definition C because it refers to B. Thus, the sequence of deletions is to delete C, then B, then A. This ensures that you do not attempt to delete an object referenced by another object.

Example 10-4. Objects That Reference Other Objects

```
CONSTANT a   VALUE IS 1.
DEF      b   TYPE BINARY  VALUE IS a.
DEF      c   TYPE b.
```

When an object is referenced by many objects, it is a tedious process to delete all the objects that refer to it directly and indirectly and to delete them in the correct sequence. The statement OUTPUT UPDATE on page 8-7 helps you with this task by performing the following functions:

- It locates all constants, definitions, records, token codes, token types, and token maps that refer to the object to be deleted.
- It writes the DELETE statements to delete the referring objects in the correct sequence in the first section of an open DDL source file.
- It redefines the specified object, in the second section of an open DDL source file.

- It writes the statements to rebuild the objects that referenced the specified object, in a section for each referring object.

After executing OUTPUT UPDATE, you can use the SOURCE command to execute the DELETE statements in the DDL source file section written by OUTPUT UPDATE. After the objects that refer to an object are deleted, you can delete the referenced object.

Assume you are running the DDL compiler interactively. To delete a referenced constant, definition, or SPI token type from the dictionary, do this:

1. Open the dictionary containing the object to be deleted.
2. Open a new DDL source file for the output from OUTPUT UPDATE.
3. Use the OUTPUT UPDATE statement to write the DDL source file containing the DELETE statements for objects that refer to the specified object.
4. Examine the DDL source file to get the section name containing the DELETE statements; to do this:
    a. Close the DDL source file with a NODDL command. If you omit this step, you will get a FILE IN USE message when you try to edit this file.
    b. Use the EDIT command to examine the DDL source file. Make a note of the name of the section that contains the DELETE statements produced by OUTPUT UPDATE; then exit from the editor.
5. Delete all the referring objects from the dictionary. Use the SOURCE command to submit the DDL source file section containing the DELETE statements for these objects to the DDL compiler. This step executes the DELETE statements, effectively deleting the objects from the dictionary.
6. Delete the object. Use a DELETE statement to delete the object from the dictionary.

Example 10-5 on page 10-7 shows the DDL statements and commands needed to delete the referenced object AGE from the dictionary.

Example 10-5. Deleting a Referenced Object From a Dictionary (page 1 of 2)

The referenced object, age, in the dictionary:

```
DEF age           PIC 99.

DEF employ.
  02 empnum       PIC 9(4).
  02 empname      PIC X(18).
  02 age          TYPE *.
END

RECORD employee. FILE IS ASSIGNED.
  02 employ       TYPE *.
  02 region       PIC 9(4).
  02 branchnum    PIC 9(4).
END
```

Removing age from the dictionary:

```
44> DDL DICT $data.sales
```
Run the DDL compiler and open dictionary

```
Dictionary opened on subvol $DATA.SALES for update access

!?DDL delfile
```
Open new DDL source file

```
Output source for DDL opened on $DATA.SALES.DELFILE

!OUTPUT UPDATE age.
```
Write update statements to DELFILE

```
Searching for objects affected by AGE
Loading Definition AGE
DDL source output produced for AGE.
Loading Definition EMPLOY
DDL source output produced for EMPLOY.
Loading Record EMPLOYEE
DDL source output produced for EMPLOYEE.

!?NODDL delfile
```
Close DDL file DELFILE

```
!?EDIT delfile; L 1/10
```
List DELFILE

```
   1 ?Section AGE-DELETES
   2 Delete Record EMPLOYEE
   3 Delete Definition EMPLOY
   4
```
Get name of section with DELETE statements

```
   5 ?Section AGE
   6 ...
   7

   8 ?Section EMPLOY
   9 ...
   10
*EXIT
```
Section to define AGE followed by sections to redefine objects that refer to AGE

```
!?SOURCE delfile (age-deletes)
```
Submit section to DDL

```
   1 ?Section AGE-DELETES
   2 Delete Record EMPLOYEE
```
DDL compiler executes AGE-DELETES

Example 10-5. Deleting a Referenced Object From a Dictionary (page 2 of 2)

```
     Record EMPLOYEE deleted from dictionary.
   3 Delete Definition EMPLOY
     Definition EMPLOY deleted from dictionary.
   4

!DELETE DEF age.                          Delete AGE definition

Definition AGE deleted from dictionary.

!EXIT                                     Exit from DDL compiler

Objects:     Added Replaced Deleted
Definitions  0     0        2
Records      0     0        1
Dictionary on subvol \SYS1.$DATA.SALES is closed.
Errors detected: 0
Warnings detected: 0
```

Modifying Dictionary Objects

Modifying an object stored in a dictionary is similar to deleting an object. If the object is never referenced by other objects, the modification is comparatively simple. If the object is referenced by other objects, then you must first delete and then redefine the referring objects.

Objects that can be referenced by other objects are:

- Constants
- Definitions
- SPI token types

Objects that are never referenced by other objects are:

- Records
- SPI token codes
- SPI token maps

Topics:

- Modifying Unreferenced Objects on page 10-9
- Modifying Referenced Objects on page 10-10

Modifying Unreferenced Objects

Records, SPI token codes, and SPI token maps are never referenced by other objects. Other types of objects can be referenced. You can use the SHOW USE OF command to determine whether the object you want to modify is referenced by other objects.

To modify an object not referenced by other objects, build a source file that contains the definition of the changed object, then compile this source file into the dictionary. You can, of course, change the original schema directly and recompile the dictionary, but this causes unnecessary processing if your dictionary is large.

To modify the object, do this:

1. Open the dictionary. Use a DICT or DICTN command to open the dictionary containing the object to be modified.
2. Modify the object. To avoid recompiling the entire schema, write the object definition from the dictionary to a DDL source file using an OUTPUT statement, close the DDL source file, and then edit the object definition in the DDL source file.
3. Compile the modified object into the open dictionary. Run the DDL compiler with the DDL source file as the input file, or compile the source file interactively with the SOURCE command.
4. Modify your original schema if you plan to ever use it to rebuild the dictionary.

Suppose you want to add a new alternate key field, ORDERDATE, to the record ORDERS defined in the sample database schema in Appendix B, Sample Schemas. The new key field is already defined in the definition ORDERINFO. To specify the key as an alternate key in ORDER-REC, use the OUTPUT statement to write the record definition from the open dictionary to a DDL source file. Add the new key specifier to the record definition, and then compile the record definition back into the dictionary with a SOURCE command, as in Example 10-6 on page 10-9.

Example 10-6. Modifying an Unreferenced Object (page 1 of 2)

`45> DDL DICT $data.sales`	Run DDL and open dictionary
`!?DDL newsrc !`	Open and clear source file

`!OUTPUT RECORD orders.`
`Loading Record ORDERS`
`DDL source output produced for ORDERS`
	Write record definition to source file

`!?NODDL`	Close source file

`!?EDIT newsrc; LA`
`   3 ?Section ORDERS`
`   6 Record ORDERS`
	List and edit record ORDERS

Example 10-6. Modifying an Unreferenced Object (page 2 of 2)

```
7 File is "$data.sales.orders" Key-sequenced
8                              Audit
9 Definition is ORDERINFO.
10
11 Key is ORDERNUM Duplicates not allowed.
12 Key "sn" is SALESPERSON.
13 Key "cn" is CUSTNUM.
14 End
*A 13
13 Key "cn" is CUSTNUM.

13.1 KEY "od" is ORDERDATE.          Add new alternate key
13.2 //
*EXIT

!?SOURCE newsrc                      Compile modified record
!EXIT
```

Alternatively, assume that ORDERDATE is not defined in ORDERINFO. In this case, write the definition of ORDERINFO together with all the definitions and records that refer to it to the DDL source file, as in Example 10-7 on page 10-10.

Example 10-7. Modifying an Unreferenced Object

```
46> DDL DICT
!?DDL newsrc !

!OUTPUT UPDATE orderinfo.    ORDER-REC is included in NEWSRC
!?NODDL, EDIT newsrc

...                          Add ORDERDATE to ORDERINFO and add key field
                             to ORDER-REC
*EXIT
!?SOURCE newsrc
```

Modifying Referenced Objects

Constants, definitions, and SPI token maps can be referenced by other objects. You can use the SHOW USE OF command to determine whether the object you want to modify is referenced by any other objects.

Before you can modify a referenced object, you must first delete any objects that refer to that object. After you modify the object, you must redefine the deleted objects. The deletion and the redefinition must be done in exact sequence.

In Example 10-8 on page 10-11, to modify the constant A, you must first delete definition C and then delete definition B in that order. After modifying constant A, you must first redefine definition B and then redefine definition C in the reverse order. This sequence ensures that you do not try to delete a referenced object or add an object that refers to a nonexistent object.

Example 10-8. Objects That Reference Other Objects

```
CONSTANT a   VALUE IS 1.
DEF      b   TYPE BINARY  VALUE IS a.
DEF      c   TYPE b.
```

When an object is referenced by many other objects, deleting and then redefining all the referring objects is a time-consuming and error-prone process. The statement OUTPUT UPDATE on page 8-7 helps you by performing these functions:

- Locating all the objects that refer to a specified object.
- Writing the DELETE statements to delete the referring objects to an open DDL source file. OUTPUT UPDATE deletes the objects in sequence so that an object is never deleted before an object that refers to it.
- Writing the statements to define the object to be modified, followed by the statements to redefine each deleted object to the open DDL source file. OUTPUT UPDATE redefines objects in sequence so that referenced objects are defined before any objects that refer to them.

After executing OUTPUT UPDATE, you can edit the statement that defines the object you are changing. When the statement is changed to your satisfaction, compile the DDL source file into the open dictionary. The source file contains the code to delete all referring objects and then rebuild them. If you decide to make no changes, the source file contains the code to return your dictionary to its initial state.

You can edit and compile the source file in an interactive session, or you can perform these functions noninteractively. Generally, you run the DDL compiler interactively to make minor modifications.

Assume you are in an interactive DDL session. To modify a referenced object, do this:

1. Open the dictionary containing the object you want to modify.
2. Open a DDL source file to contain the statements generated by OUTPUT UPDATE. If the file already exists, make sure it is empty before OUTPUT UPDATE writes to it.
3. Use OUTPUT UPDATE to write the statements that delete the referring objects, that define the object to be modified, and that redefine the deleted objects to the open DDL source file.
4. Use the NODDL command to close the DDL source file
5. Edit the object definition in the DDL source file. You can do the editing interactively or you can exit from the DDL compiler. If you remain interactive, you enter the editor with the DDL EDIT command; if you exit from the DDL compiler, you run the editor from the command interpreter.

6. Compile the DDL source file. If you are still in an interactive session, use the SOURCE command to submit the source file to the compiler. If you exited from the DDL compiler after Step 4, run the DDL compiler from the command interpreter specifying the DDL source file as the input file.

Suppose postal zip codes must be changed from five digits to nine digits. The sample database schema in Appendix B, Sample Schemas, includes a definition of the object ZIP-CD, which is referenced by three definitions (SUPPINFO, CUSTINFO, and ADDR) and by two records (SUPPLIER and CUSTOMER). Example 10-9 on page 10-12 shows the statements and commands you can use to modify ZIP-CD.

Example 10-9. Modifying a Reference Object (page 1 of 2)

```
47> DDL DICT $data.sales                Run the DDL compiler and open dictionary

Dictionary opened on subvol $DATA.SALES for update access

!?DDL modfile !                         Open and clear DDL source MODFILE

Output source for DDL is opened on $DATA.SALES.MODFILE

!OUTPUT UPDATE zip-cd.                  Write update statements to MODFILE

Searching for objects affected by ZIP-CD
Loading Definition ZIP-CD
DDL source output produced for ZIP-CD.
Loading Definition ADDR
DDL source output produced for ADDR.
Loading Definition CUSTINFO
DDL source output produced for CUSTINFO.
Loading Definition SUPPINFO
DDL source output produced for SUPPINFO.
Loading Definition CUSTOMER
DDL source output produced for CUSTOMER.
Loading Definition SUPPLIER
DDL source output produced for SUPPLIER.

!?NODDL                                 Close DDL source file

!?EDIT modfile; xvs f                   Edit MODFILE

?Section ZIP-CD-DELETES                 Statements to delete referring objects
Delete Record SUPPLIER.
...
Delete Definition ADDR.

?Section ZIP-CD                         Definition to be modified

Definition ZIP-CD PIC "9(5)".           Change to 9(9)

?Section ADDR                           Statements to redefine deleted objects
...
?Section SUPPLIER
Record SUPPLIER.
...
End
```

Example 10-9. Modifying a Reference Object (page 2 of 2)

`*EXIT`	Exit editor
`!?SOURCE modfile` `!EXIT`	Compile changes into dictionary and exit DDL

Making Major Modifications

If you have made many changes to a dictionary interactively through the DDL compiler, HP recommends that you recompile the entire dictionary.

Rather than using the original schema, which might not reflect all changes to the dictionary, create a new schema from the dictionary. You can generate a new schema exactly as in Generating a schema From a Dictionary on page 10-1.

Note. do not use the procedure for generating a new schema to modify a dictionary that is part of a Pathmaker catalog. Pathmaker dictionaries contain application design information that is not in generated DDL schemas.

After you have a schema that accurately reflects the current dictionary, you can edit the schema and then recompile it to build a new dictionary. If you made the changes directly to the dictionary schema, you can compile the schema to build a new dictionary.

In either case, be sure that all objects referenced by other objects are added first: DDL cannot compile a referring object if the object it refers to is not already in the dictionary.

To recompile a dictionary:

1. Generate a new schema from the current dictionary using the procedure in Generating a schema From a Dictionary on page 10-1. For example:

   ```
   48> VOLUME $data.sales
   49> DDL DICT
   !?DDL newsrc !
   !OUTPUT *.
   !EXIT
   ```

2. Edit the schema, making the necessary deletions, modifications, and additions.

3. Compile the schema into a new dictionary.

   If you do not need the old dictionary, you can clear it at this time and write the new dictionary objects back into the cleared dictionary files. For example:

   ```
   50> DDL /IN newsrc/DICT $data.sales !
   ```

   If you want to keep the old dictionary while you test the new dictionary, you can create the new dictionary on a different subvolume. For example:

   ```
   51> DDL/IN newsrc/DICT $data.newsales
   ```

Changing Dictionary Security

Dictionary files are created with the default file-creation security of the user who created them. If you are the owner of the files, you can change the security applied to the dictionary files by the DDL compiler with the FUP SECURE command.

To change file security:

1. Use the FUP INFO command to determine the current security of the dictionary files.
2. Use the FUP SECURE command to specify the security you want.

Suppose the dictionary on $DATA.SALES was created with a user default security of “AAAA,” where the dictionary files can be read, written to, and purged by any other user. Example 10-10 on page 10-14 shows how to change this security so that any user can read or execute the dictionary files, but only the owner can write to or purge them.

Example 10-10. Changing Dictionary Security

```
52> VOLUME $data.sales                    Go to dictionary volume and subvolume

53> FILES                                 List dictionary files and change their security

$DATA.SALES

DICTALT DICTCDF DICTDDF DICTKDF DICTMAP DICTOBL DICTODF DICTOTF
DICTOUF DICTOUK DICTRDF DICTTKN DICTTYP DICTVER

54> FUP SECURE (DICTALT,DICTCDF,DICTDDF,DICTKDF,DICTMAP), "AOAO"
55> FUP SECURE (DICTOBL,DICTODF,DICTOTF,DICTOUF,DICTOUK), "AOAO"
56> FUP SECURE (DICTRDF,DICTTKN,DICTTYP,DICTVER), "AOAO"
```

For a description of the FUP SECURE command, see the *File Utility Program (FUP) Reference Manual*.

Moving a Dictionary

You can move a dictionary from one subvolume to another subvolume with a combination of FUP commands. If you are creating a backup dictionary on the new subvolume, keep the original dictionary. If you want only one copy of the dictionary, purge the original dictionary after the move.

Note. Do not attempt to move a dictionary that is part of a Pathmaker application catalog using this procedure. Refer to Pathmaker documentation for instructions about how to move a dictionary that is part of a Pathmaker application catalog.

The procedure for moving a nonaudited dictionary differs from the procedure for moving a dictionary audited by the Transaction Monitoring Facility (TMF) subsystem.

Topics:

Moving a Nonaudited Dictionary

To move a nonaudited dictionary:

1. Duplicate the dictionary files on another subvolume using FUP DUP commands.
2. Change the subvolume name of the alternate key file in the file label of each dictionary file that has alternate keys using FUP ALTER commands. (You can determine which files use alternate keys by looking for the symbol *A* in the TYPE column of a FUP INFO display; then use FUP INFO, DETAIL on those files to determine the alternate key name.)
3. Optionally, you can purge the dictionary from the old subvolume.

You can enter these commands interactively, or you can build a file containing these commands and then execute the file.

Example 10-11 on page 10-15 moves a dictionary from $DATA.SALES to $MKT.SALESDIC.

Example 10-11. Moving a Nonaudited Dictionary

```
VOLUME $data.sales                          Default volume and subvolume

FUP DUP (DICTALT,DICTCDF,DICTDDF,DICTKDF,DICTMAP), $mkt.salesdic.*
FUP DUP (DICTOBL,DICTODF,DICTOTF,DICTOUF,DICTOUK), $mkt.salesdic.*
FUP DUP (DICTRDF,DICTTKN,DICTTYP,DICTVER), $mkt.salesdic.*

VOLUME $mkt.salesdic                        New default volume and subvolume

FUP ALTER DICTKDF, ALTFILE (O,DICTALT)      Change alternate-key subvolume names in
FUP ALTER DICTOBL, ALTFILE (O,DICTALT)      the file labels of all files with alternate keys
FUP ALTER DICTODF, ALTFILE (O,DICTALT)
FUP ALTER DICTOUF, ALTFILE (O,DICTALT)
FUP ALTER DICTFDF, ALTFILE (O,DICTALT)
```

Suppose that the commands of Example 10-11 on page 10-15 are in the file $DATA.SALES.DICMOVE. You can execute the commands by entering this command:

```
57> OBEY $data.sales.dicmove
```

If you no longer need the original dictionary on $DATA.SALES, you can purge the dictionary files as in Purging a Dictionary on page 10-18.

Moving an Audited Dictionary

Moving an audited dictionary requires more steps than moving a nonaudited dictionary because of these actions of the FUP utility on audited files:

- When you FUP DUP an audited file, FUP automatically disables auditing on the file but does not disable auditing on or change pointers to any associated alternate key files. As a result, duplicated files that use alternate keys point to audited alternate key files on the original subvolume. For this reason, you must use FUP ALTER to disable auditing before using FUP DUP to duplicate files, then use FUP ALTER to reenable auditing after duplicating the files.
- When you use FUP ALTER to disable auditing on a file that uses alternate keys, FUP also disables auditing on the associated alternate key file. As a result, any other files that use the same alternate key will be associated with a nonaudited alternate key file. But you cannot use FUP ALTER to disable auditing on a file that has a nonaudited alternate key file, so you must reenable auditing on the alternate key file in order to disable auditing on a file using that alternate key file.
- Conversely, when you use FUP ALTER to enable auditing on a file, it automatically enables auditing on any alternate key file used by the file. But you cannot use FUP ALTER to enable auditing on a file that uses an audited alternate key file, so you must disable auditing on any alternate key file before enabling auditing on a file that uses that alternate key file.

To move an audited dictionary:

1. Disable auditing using FUP ALTER commands. Where necessary, reenable auditing on alternate key files before disabling auditing on files that use the alternate key files.
2. Duplicate the dictionary files on another subvolume using FUP DUP commands.
3. Change the subvolume name of the alternate key file in the file label of each dictionary file that has alternate keys using FUP ALTER commands.
4. ENABLE auditing on the new subvolume using FUP ALTER commands. Where necessary, disable auditing on alternate key files before enabling auditing on files that use the alternate key files.
5. Do not audit the DICTDDF file.
6. Optionally, purge the dictionary from the old subvolume.
7. If you keep the original dictionary, you might want to re-enable auditing (see Step 4).

While you can enter these commands interactively, it is best to create a FUP file-creation source file containing these commands and execute that file.

Example 10-12 on page 10-17 moves an audited dictionary from $MKT.SALESDIC to $DATA.SALES.

Example 10-12. Moving an Audited Dictionary

```
VOLUME $mkt.salesdic                          Default volume and subvolume

FUP ALTER DICTCDF, NO AUDIT

FUP ALTER DICTKDF, NO AUDIT                   Also disables auditing on DICTALT

FUP ALTER DICTMAP, NO AUDIT

FUP ALTER DICTALT, AUDIT                      Re-enable auditing on DICTALT before
                                              disabling auditing on DICTOBL, DICTODF,
                                              and other files that use DICTALT

FUP ALTER DICTOBL, NO AUDIT
FUP ALTER DICTALT, AUDIT
FUP ALTER DICTODF, NO AUDIT
FUP ALTER DICTOTF, NO AUDIT

FUP ALTER DICTOUF, NO AUDIT                   Also disables auditing on DICTOUK, but no
                                              other files use DICTOUK as an alternate-
                                              key file

FUP ALTER DICTRDF, NO AUDIT
FUP ALTER DICTTKN, NO AUDIT
FUP ALTER DICTTYP, NO AUDIT
FUP ALTER DICTVER, NO AUDIT

FUP DUP (DICTALT,DICTCDF,DICTDDF,DICTKDF,DICTMAP), $data.sales.*
FUP DUP (DICTOBL,DICTODF,DICTOTF,DICTOUF,DICTOUK), $data.sales.*
FUP DUP (DICTRDF,DICTTKN,DICTTYP,DICTVER), $data.sales.*

VOLUME $data.sales                            New default volume and subvolume

FUP ALTER DICTKDF, ALTFILE (O,DICTALT)        Change alternate-key subvolume names in
FUP ALTER DICTOBL, ALTFILE (O,DICTALT)        the file labels of all files with alternate keys
FUP ALTER DICTODF, ALTFILE (O,DICTALT)
FUP ALTER DICTOUF, ALTFILE (O,DICTALT)
FUP ALTER DICTFDF, ALTFILE (O,DICTALT)

FUP ALTER DICTCDF, AUDIT

FUP ALTER DICTKDF, AUDIT                      Also enables auditing on DICTALT

FUP ALTER DICTMAP, AUDIT

FUP ALTER DICTALT, NO AUDIT                   Disable auditing on DICTALT before
                                              enabling auditing on DICTOBL, DICTODF,
                                              and other files that use DICTALT

FUP ALTER DICTOBL, AUDIT
FUP ALTER DICTALT, NO AUDIT
FUP ALTER DICTODF, AUDIT
FUP ALTER DICTOUF, AUDIT
FUP ALTER DICTFDF, AUDIT
FUP ALTER DICTALT, NO AUDIT
FUP ALTER DICTRDF, AUDIT
FUP ALTER DICTTKN, AUDIT
FUP ALTER DICTTYP, AUDIT
FUP ALTER DICTVER, AUDIT
```

Suppose that the commands of Example 10-12 on page 10-17 are in the file $MKT.SALESDIC.DICMOVE. You can execute the commands by entering this command:

```
58> OBEY $mkt.salesdic.dicmove
```

If you no longer need the original dictionary on $MKT.SALESDIC, you can purge the dictionary files as in Purging a Dictionary on page 10-18. If you keep the original dictionary, you might want to re-audit the dictionary files.

Purging a Dictionary

You can purge a dictionary by purging each dictionary file individually or, if the dictionary is open, by entering the NOSAVE command following the DICT or DICTN command.

To purge dictionary files individually, you must know the file names. You can see Appendix D, Dictionary Database Structure, for the file names, or you can position yourself on the subvolume that contains the dictionary and use a FILES command to list the dictionary files. Dictionary file names always begin with DICT.

Note. Do not purge a dictionary that is part of a Pathmaker catalog. Refer to Pathmaker documentation for instructions about how to purge a dictionary that is part of a Pathmaker application catalog.

Example 10-13 on page 10-18 lists and then purges the dictionary files on $DATA.SALES.:

Example 10-13. Listing and Purging Dictionary Files

```
59> VOLUME $data.sales
60> PURGE DICTALT,DICTCDF,DICTDDF,DICTKDF,DICTMAP,DICTOBL,DICTODF
61> PURGE DICTOTF,DICTOUF,DICTOUK,DICTRDF,DICTTKN,DICTTYP,DICTVER
```

You can purge the open dictionary with the NOSAVE command. If NOSAVE is in effect when you exit from the DDL compiler, when another dictionary is opened, or when the dictionary is closed using the NODICT command, the DDL compiler purges the open dictionary. NOSAVE is ignored if the dictionary is part of a Pathmaker catalog.

Example 10-14. Purging Dictionary Files With the NOSAVE Command

`62> DDL`
`!?DICT $data.parts`
`...`
	Open dictionary

`!?NOSAVE`	Ignored for Pathmaker dictionaries
`!?DICT $data.parts`	Purge the open dictionary and open a new one

Increasing Dictionary File Size

The DDL compiler creates the dictionary files with primary and secondary extent sizes.

Table 10-1. Dictionary File Extent Sizes

Dictionary File	Primary Extent Size	Secondary Extent Size
DICTALT	4 pages	32 pages
DICTCDF	4 pages	32 pages
DICTDDF	4 pages	32 pages
DICTKDF	4 pages	32 pages
DICTMAP	4 pages	32 pages
DICTOBL	4 pages	32 pages
DICTODF	4 pages	32 pages
DICTOTF	4 pages	32 pages
DICTOUF	4 pages	32 pages
DICTOUK	4 pages	32 pages
DICTRDF	4 pages	32 pages
DICTTKN	4 pages	32 pages
DICTTYP	4 pages	32 pages
DICTVER	4 pages	32 pages

With these size limits, it is possible that one or more of the dictionary files can be filled to capacity. If a dictionary file runs out of space, a FILE ERROR 45 (file is full) results.

You can increase dictionary file size by using the FUP ALTER MAXEXTENTS command to increase the maximum number of file extents.

To increase the maximum number of file extents, do the following:

1. Start an interactive FUP session.
2. Use ALTER to increase the value of MAXEXTENTS.
3. Use FUP INFO file-name,DETAIL to display and verify your changes.
4. Exit from the interactive FUP session.

If you are altering only one file, you can use a single FUP command to change MAXEXTENTS.

Example 10-15. Increasing a Dictionary's File Size

```
63> FUP ALTER DICTOTF, MAXEXTENTS 200
64> FUP INFO DICTOTF, DETAIL
$DATA.SALES.DICTOTF
   TYPE K
   CODE 203
   ...
   MAXEXTENTS 200                          Verify change
   ...
```

Rebuilding a Dictionary

The procedure for rebuilding a nonaudited dictionary differs from the procedure for rebuilding a dictionary audited by the Transaction Monitoring Facility (TMF) subsystem.

Topics:

Rebuilding a Nonaudited Dictionary

Occasionally, a nonaudited dictionary can become corrupt and you must rebuild it. A corrupt dictionary is one in which an entry in the dictionary files is missing or contains the wrong value. For example, a dictionary is corrupt if there is no DICTRDF record for a record in the dictionary. A dictionary is badly corrupted and cannot be rebuilt if one of the dictionary files is deleted.

One way to rebuild the dictionary is to generate a schema following the procedure in Generating a schema From a Dictionary on page 10-1. This procedure rebuilds information about DDL objects and can be used for dictionaries created from the DDL compiler, but not for dictionaries that are part of a Pathmaker catalog.

To rebuild a dictionary created from the DDL compiler, do the following:

1. Start an interactive DDL session.
2. Open the corrupted dictionary; open an EDIT file for DDL output.
3. Use OUTPUT * to generate a schema for all the definitions and records in the dictionary.
4. Close the EDIT file and the corrupted dictionary.
5. Open a new dictionary on another subvolume and source in the generated schema.
6. Exit from the DDL compiler.

Example 10-16 on page 10-21 rebuilds a dictionary with a schema generated from a corrupted dictionary.

Example 10-16. Rebuilding a Nonaudited Dictionary

```
65> DDL
!?DICT $data.sales
!?DDL $data.newsales.ddlsrc
!OUTPUT *
!?NODDL
!?DICT $data.newsales
!?SOURCE $data.newsales.ddlsrc
!EXIT
```

In some cases, you cannot generate a schema from a corrupted dictionary. To protect your dictionary from such an occurrence, keep a fairly current backup schema of any important dictionary.

Rebuilding an Audited Dictionary

Audited dictionaries rarely need to be rebuilt. The TMF subsystem protects your dictionary from becoming corrupt by packaging changes into transactions, or units of recovery. A transaction either modifies the dictionary, or it fails. If a transaction fails, the TMF subsystem undoes the changes and restores the dictionary to its initial state. You need to rebuild an audited dictionary only if the TMF system failure occurs.

In Example 10-17 on page 10-21:

- The dictionary was created on the subvolume $DATA.SALES.
- All dictionary files are audited except DICTDDF.

Example 10-17. Determining If a Dictionary is Audited

```
66> FUP INFO $data.sales.*

                CODE      EOF  LAST MODIF   OWNER RWEP  TYPE   REC BLOCK
$DATA.SALES
 DICTALT         201A   12288       17:06   8,47  CUCU  K       38  4096
 DICTCDF         207A   12288       17:06   8,47  CUCU  K       11  4096
 DICTDDF         200       30       17:06   8,47  CUCU
 DICTKDF         206A   12288       17:06   8,47  CUCU  KA      94  4096
 DICTMAP         209A   12288       17:06   8,47  CUCU  K       22  4096
 DICTOBL         204A   36864       17:06   8,47  CUCU  KA     194  4096
 DICTODF         202A   16384       17:06   8,47  CUCU  KA      86  4096
 DICTOTF         203A   12288       17:06   8,47  CUCU  K      145  4096
 DICTOUF         208A   16384       17:06   8,47  CUCU  KA      65  4096
 DICTOUK         208A   16384       17:06   8,47  CUCU  K       98  4096
 DICTRDF         205A   12288       17:06   8,47  CUCU  KA      89  4096
 DICTTKN         209A   12288       17:06   8,47  CUCU  K        6  4096
 DICTTYP         209A   12288       17:06   8,47  CUCU  K       24  4096
 DICTVER         209A   12288       17:06   8,47  CUCU  K       19  4096
```

If you have a system failure and must rebuild the audited files, follow the procedures in the *TMF Management Programming Manual*.

Converting a Dictionary

As of the D-series software product version, the DDL compiler supports these dictionaries:

DDL Compiler Product Version	Dictionary Product Version
C20 and C30	5
D00, D10, and D20	6
D30 and later	7
H01	8

To have full use of a dictionary created with DDL software prior to product version D00 from a D-series product version of the DDL compiler, convert the dictionary to product version 6 or 7.

To have full use of a dictionary created with DDL software prior to product version H01 from a D-series or G-series product version of the DDL compiler, convert the dictionary to product version 8.

Note. Do not attempt to convert a dictionary that is part of a Pathmaker application catalog using this procedure. Instead, refer to Pathmaker documentation for the appropriate process.

To convert a dictionary, perform the following steps:

1. Generate a schema from the existing dictionary. Use the DDL OUTPUT * statement to generate a schema in a DDL source file (see Generating a schema From a Dictionary on page 10-1).
2. Check the new DDL source before continuing.
3. Close the DDL source code file.
4. Clear the existing dictionary. Use the DICT ! command to clear the dictionary files of their objects.
5. Recreate the dictionary. Use the DDL SOURCE command to create a new dictionary from the DDL source file. Any dictionary on the subvolume used for the conversion will be overwritten.

Example 10-18 on page 10-23 converts a product version 4 or 5 dictionary on subvolume $DATA.SALES to a new dictionary on the same subvolume.

Example 10-18. Converting a Dictionary From One Product Version to Another

```
67> VOLUME $data.sales
68> DDL
```

`!?DICT`	Open the dictionary on $DATA.SALES
`!?DDL ddlsrc !`	Open the DDL source file DDLSRC
`!OUTPUT *.`	Generate dictionary schema on DDLSRC
`!?NODDL`	Close DDLSRC and check DDLSRC before continuing
`!?DICT !`	Clear the dictionary on $DATA.SALES
`!?SOURCE ddlsrc`	Generate new dictionary objects from the schema on DDLSRC

If you run Enform Plus reports using $SYSTEM.SYSTEM.DDQUERYS against a converted dictionary, change the dictionary description embedded in the dictionary on $SYSTEM.DDL and install the product version of Enform Plus that corresponds to this product version of the DDL compiler. After you upgrade the dictionary on $SYSTEM.DDL to product version 7 or 8 (on a G-series or H-series system, respectively) and install the D-series product version of Enform Plus, you can still use the product version 7 or 8 dictionary to report against dictionaries that have not yet been converted as long as you do not use D-series, G-series, or H-series dictionary features in the reports.

Alternately, you can keep a dictionary with the earlier dictionary description in one subvolume and the current dictionary description in another subvolume. Use the earlier dictionary for Enform Plus reports for unconverted files and the D-series, G-series, or H-series dictionary for Enform Plus reports on D-series, G-series, or H-series files.

To change the dictionary description, do this:

1. Move to the subvolume $SYSTEM.DDL.
2. Purge the existing dictionary files on that subvolume.
3. Run the DDL compiler to compile the dictionary schema $SYSTEM.SYSTEM.DDLSCHEMA and print a listing of the compiled schema.

Example 10-19. Changing a Dictionary Description

```
69> VOLUME $system.ddl
70> PURGE
dictalt,dictddf,dictodf,dictkdf,dictrdf,dictotf,dictobl
71> PURGE
dictcdf,dictmap,dictouf,dictouk,dicttkn,dicttyp,dictver
72> DDL /IN $system.system.ddschema, OUT $s.#printer, NOWAIT/
```

A DDL Messages

This appendix lists all of the DDL error and warning messages in alphabetic order. For each message, it gives the cause, effect, and recovery procedure.

During DDL processing, you might receive a message from a sequential I/O procedure. Sequential I/O error messages, numbered from 500 to 600, are not documented in this manual. For information about the corrective action to take when you get such an error, see the *Guardian Procedure Errors and Messages Manual*.

Table A-1. DDL Message Types

Message Type	Indicates ...
WARNING	An error or ambiguity that does not prevent compilation of a DDL record or definition, but that might cause results other than those desired. The ERRORS command does not count warnings as errors.
ERROR	An error that affects the dictionary or source output from the dictionary. Generally, when such an error occurs, the DDL compiler continues compilation but does not add the object in error to the dictionary or to any open source file. The ERRORS command counts errors.
FATAL ERROR	An error from which the DDL compiler cannot recover. The DDL compiler stops compiling when it detects a fatal error.

An alphabetic list of DDL error and warning messages follows. The messages are alphabetized on the first word following the ***ERROR***, ***FATAL ERROR***, or ***WARNING*** prefix.

```
***ERROR*** A noncomputational item was specified in group-
group-name
```

Cause. A group described with a USAGE IS COMP clause contains a field with a data type that cannot be computational.

Effect. The DDL compiler rejects the object.

Recovery. Change the data type of the field, or remove the USAGE IS COMP clause from the group definition or description.

```
A non PACKED-DECIMAL item was specified in group - group_name
```

Cause. The group identified by *group_name* is described with a USAGE IS COMP-3, COMPUTATIONAL-3, or PACKED-DECIMAL clause but contains a field with a data type that is not PACKED-DECIMAL.

Effect. The DDL compiler rejects the object. For example:

```
!DEF EMP8.
!02  FLD8 PIC 9(5).
!END.
 Definition EMP8 size is 5 bytes.
 Definition EMP8 added to dictionary.
!DEF EMP9.
!02 FLD9 TYPE EMP8 COMP-3.
!END.
*** ERROR *** A non PACKED-DECIMAL item was specified in group -  FLD9
*** WARNING *** Errors detected - no output produced for EMP9
!
```

Recovery. Change the data type of the field, or remove the USAGE IS COMP-3, COMPUTATIONAL-3, or PACKED-DECIMAL clause from the group definition.

```
***FATAL ERROR*** Address to be freed not in address list
```

Cause. This is an internal compiler error–no user error is implied.

Effect. The DDL compiler closes the dictionary and all source code files and stops processing current source file.

Recovery. Report the error to your service provider.

```
***ERROR*** ALL must not precede a numeric literal
```

Cause. The figurative constant ALL precedes a numeric literal in a VALUE or MUST BE clause.

Effect. The DDL compiler rejects the object.

Recovery. Either remove the figurative constant ALL or replace the numeric literal with a character literal, a national literal, or another figurative constant. Then recompile the object.

ERROR Ambiguous reference- *object-name*

Cause. A referenced field does not have sufficient qualification to distinguish it from another field of the same name.

Effect. The DDL compiler rejects the object.

Recovery. Qualify the referenced field, or rename one of the fields so that no ambiguity exists, and recompile the object.

```
***ERROR*** Attribute already specified- attribute
```

Cause. A definition attribute clause is specified more than once for the same field, or a file creation attribute is specified more than once for the same record.

Effect. The DDL compiler rejects the object.

Recovery. Remove the repeated clause or file-creation attribute and recompile the object.

```
***ERROR*** Attribute cannot be specified for bit fields-
attribute-name
```

Cause. The definition or description for a field of type BIT includes an OCCURS or REDEFINES clause.

Effect. The DDL compiler rejects the object.

Recovery. Remove any OCCURS or REDEFINES clause from the bit field definition or description, or change the type of the field, and recompile the object.

```
***ERROR*** Attribute cannot be specified for object type-
attribute-name
```

Cause. An attribute specified for a field in a RECORD statement can be specified only in a DEFINITION statement. For example, the SPI-NULL clause cannot be in a RECORD statement but can be in a DEFINITION statement.

Effect. The DDL compiler rejects the object.

Recovery. Remove the attribute from the RECORD statement and recompile the object.

```
***ERROR*** AUTIDCOMPRESS specified without AUDIT
```

Cause. You specified the AUDITCOMPRESS attribute for a record but did not specify AUDIT.

Effect. The DDL compiler rejects the object.

Recovery. Remove AUDITCOMPRESS or add AUDIT and recompile the object.

```
***ERROR*** Bit field cannot be used as key- element
```

Cause. You specified a bit field as a key.

Effect. The DDL compiler rejects the object.

Recovery. Specify a field of a type other than bit as the key.

```
***ERROR*** BLOCK must be 512, 1024, 2048, or 4096 bytes
```

Cause. A block length other than 512, 1,024, 2,048, or 4,096 bytes was specified.

Effect. The DDL compiler rejects the object.

Recovery. Specify a valid block length and recompile the object.

```
***ERROR*** BLOCK specified for an UNSTRUCTURED file
```

Cause. A block length was specified for an unstructured file.

Effect. The DDL compiler rejects the object.

Recovery. Remove the block-length specification or change the file type to key-sequenced, entry-sequenced, or relative.

```
***FATAL ERROR*** Buffer stack too close to data stack
```

Cause. The source files are nested too deeply. The DDL compiler allows approximately 20 levels of nesting.

Effect. The DDL compiler closes the dictionary and source code files and stops processing.

Recovery. Reduce the number of nesting levels of source files.

```
***ERROR*** BUFFERSIZE must be 512, 1024, 2048, or 4096 bytes
```

Cause. An invalid value was specified for BUFFERSIZE.

Effect. Effect. The DDL compiler rejects the record.

Recovery. Recovery. Change the BUFFERSIZE value to 512, 1,024, 2,048, or 4,096 and recompile.

```
***ERROR*** BUFFERSIZE specified for a structured file
```

Cause. The record definition for a structured file has a BUFFERSIZE clause; BUFFERSIZE applies only to unstructured files.

Effect. The DDL compiler rejects the object.

Recovery. Change the file type to unstructured or change BUFFERSIZE to BLOCK and recompile.

```
***ERROR*** C DEF or RECORD or union tag name too long
```

Cause. The name of a definition or record exceeds the limit of 31 ASCII characters that C allows for these names.

Effect. The DDL compiler does not write the definition or record to the C source file.

Recovery. Shorten the name and recompile the definition or record.

```
***WARNING*** C OUTPUT DIAGNOSTICS:
```

Cause. You requested C output, but the object does not conform to C rules. A message follows that describes the C error.

Effect. The DDL compiler does not write the object to the C source file.

Recovery. Correct the error and recompile.

```
***ERROR*** Cannot replace- object already defined
```

Cause. The specified definition is referenced by another definition or record. The DDL compiler cannot replace this definition without corrupting definitions or records that refer to this definition.

Effect. The DDL compiler rejects the object.

Recovery. Use OUTPUT UPDATE to rebuild objects that refer to the corrected definition.

```
***WARNING*** CIFNDEF or CIFDEF is not ended by CENDIF in C
output.
```

Cause. At least one `#ifdef` or `#ifndef` statement in C output was not closed.

Effect. The DDL compiler issues this warning message. For example:

```
!?DICT
Audited dictionary created on subvol $ADE101.ALPHA
Dictionary opened on subvol $ADE101.ALPHA for update access.
!?C
/*SCHEMA PRODUCED DATE - TIME : 7/21/2000 - 19:45:15 */
Output source for C is opened on $ZTN1.#PTPJHU8
!?CIFDEF EMP
#ifdef EMP
!CONSTANT EMP1 VALUE "JYOTI".
Constant EMP1 defined.
Constant EMP1 added to dictionary.
#pragma section emp1
/* Constant EMP1 created on 07/21/2000 at 19:45 */
#define EMP1 "JYOTI"
C output produced for EMP1.
?!NOC
Output source for C is closed.
***WARNING*** CIFNDEF or CIFDEF is not ended by CENDIF for C output
!
```

Recovery. No recovery is necessary. Just ensure that the required number of CENDIF statements were used.

```
***WARNING*** COBOL base is not a legal positive int- value
not changed
```

Cause. A COBOL base in a COBLEVEL command is not a positive integer from 1 through 49.

Effect. The DDL compiler issues a warning; base level is set to 1.

Recovery. Correct the error and recompile.

```
***ERROR*** COBOL maximum occurs nesting exceeded- nth nested
element
```

Cause. An object to be written to a COBOL source file has more than 7 levels of nested OCCURS clauses.

Effect. The DDL compiler does not write the object to the COBOL source file.

Recovery. Reduce the levels of nested OCCURS clauses and recompile the object.

```
***WARNING*** COBOL85 OUTPUT DIAGNOSTICS:
```

Cause. You requested COBOL output by default or with a SETCOBOL85 command, but the object does not conform to COBOL rules.

Effect. A message follows describing the COBOL error. The DDL compiler does not write the object to the COBOL source file.

Recovery. If you want COBOL output, correct the object definition the conform to COBOL rules and recompile.

```
***WARNING*** CODE withing range reserved by TANDEM, 100-999
```

Cause. A file code in a record definition is an integer from 100 through 999, the range reserved for use by HP.

Effect. The DDL compiler continues compiling the statement.

Recovery. Change the file code to an integer from 0 through 99 or from 1,000 through 65,535 and recompile the statement.

```
***WARNING*** COLUMNS must be between 12 and 132- value not
changed
```

Cause. The COLUMNS command specified fewer than 12 or more than 132 columns.

Effect. The DDL compiler does not change the number of significant columns in an input line.

Recovery. Reissue the COLUMNS command with a value from 12 to 132.

```
***WARNING*** Command not supported for old dictionary
versions
```

Cause. You issued a command or statement that attempts to update a dictionary created by a product version of the DDL compiler prior to the C00 software product version. For example, you entered an OUTPUT UPDATE statement for a dictionary created prior to C00.

Effect. The DDL compiler does not execute the statement.

Recovery. Convert dictionary to the current product version and reenter command.

```
***ERROR*** Command not supported for specified object type
```

Cause. You have entered a command that does not apply to the particular object type; for example, OUTPUT UPDATE specifies RECORD, or SHOW USE OF specifies TOKEN-CODE, as the object type.

Effect. The DDL compiler does not execute the command.

Recovery. Use a different command for the particular object type.

```
***WARNING*** Comment lines within element definition cannort
be saved
```

Cause. A field description that you attempted to compile contains one or more dictionary comments. A field description begins with a level number and ends with the next period (.).

Effect. The DDL compiler does not enter the comment or comments in the dictionary.

Recovery. Specify the comment or comments before the entire field description and recompile the object.

```
***ERROR*** COMP item found within VALUE
```

Cause. You have defined a computational item within a group defined with a VALUE clause. For example, the following definition is invalid:

```
Def a.
    02 b value zeros.
    03 c pic 9 comp.
End
```

An initial value at the group level must be alphanumeric.

Effect. The DDL compiler rejects the object.

Recovery. Change the data type of the computational item, or remove the VALUE clause from the group.

```
***ERROR*** COMP item must be binary or of the form PIC
[S]9(n) [V9(n)]
```

Cause. The data type of an item described with a USAGE IS COMP clause is not a computational data type.

Effect. The DDL compiler rejects the object.

Recovery. Correct the error and recompile. For more information about computational items, see USAGE on page 6-70.

```
***ERROR*** COMP specified with reference item which is not
COMP- element_name
```

Cause. The data type of the referenced item identified by *element_name* is described with a USAGE IS COMP clause but is not a computational data type.

Effect. The DDL compiler rejects the object.

Recovery. Correct the error and recompile. For more information about computational items, see USAGE on page 6-70.

```
***ERROR*** COMP-3 data item must be of the form PIC
[S]9(n)[V9(n)]
```

Cause. The data type of an item described with a USAGE IS COMP-3, COMPUTATIONAL-3, or PACKED-DECIMAL clause is not PACKED-DECIMAL.

Effect. The DDL compiler rejects the object. For example:

```
!def emp pic x PACKED-DECIMAL.
                           ^
*** ERROR *** COMP-3 data item must be of the form PIC [S]9(n)[V9(n)]
Last diagnostic on page 1
*** WARNING *** Errors detected - no output produced for EMP
!
```

Recovery. Correct the error and recompile.

```
***ERROR*** COMP-3 specified with reference item which is not
COMP-3 - element_name
```

Cause. The data type of the item identified as *element_name* is described with a USAGE IS COMP-3, COMPUTATIONAL-3, or PACKED-DECIMAL clause but is not a PACKED-DECIMAL item.

Effect. The DDL compiler rejects the object. For example:

```
!def emp1 pic 9(5).
 Definition EMP1 size is 5 bytes.
 Definition EMP1 added to dictionary.
!def emp2.
!02  fld2 type emp1 comp-3.
*** ERROR *** COMP-3 specified with reference item which is not COMP-3 - FLD2
!
```

Recovery. Correct the error and recompile.

```
***ERROR*** COMPRESS, DCOMPRESS, or ICOMPRESS on a non KEY-
SEQUENCED file
```

Cause. The record definition of an entry-sequenced, relative, or unstructured file contains a COMPRESS, DCOMPRESS, or ICOMPRESS clause. These clauses apply only to key-sequenced files.

Effect. The DDL compiler rejects the object.

Recovery. Change the file structure to key sequenced, or remove the COMPRESS, DCOMPRESS, or ICOMPRESS clause.

```
***ERROR*** CONSTANT data type is incompatible with
referenced value
```

Cause. A CONSTANT statement has a value that is incompatible with its data type. For example, the value is too large for the size indicated by the data type, or is alphabetic when the data type is numeric, or is a signed value when the data type is unsigned.

Effect. The DDL compiler rejects the constant.

Recovery. Change either the data type or the specified value of the constant.

```
***ERROR*** CONSTANT in Pascal exceeds DDL's 130-byte limit-
constant-name
```

Cause. A CONSTANT statement has a value that is greater than the 130 ASCII character limit set for the DDL compiler.

Effect. The DDL compiler does not generate Pascal code for the constant or for any object that refers to the constant.

Recovery. Shorten the constant value and regenerate the Pascal constant or object that refers to the constant.

```
***ERROR*** CONSTANT's representation exceeds TAL's 128-byte
limit- constant-name
```

Cause. A CONSTANT statement has a value greater than 128 ASCII characters, and TAL source code is requested.

Effect. The DDL compiler does not generate TAL code for the constant or for any object that refers to the constant.

Recovery. Shorten the constant value and regenerate the TAL constant or object that refers to the constant.

```
***ERROR*** CONSTANT's TACL representation exceeds DDL's
130-byte limit- constant-name
```

Cause. A CONSTANT statement has either a value greater than 130 ASCII characters or a value that was made greater by the DDL compiler emitting a tilde (~) preceding the special TACL characters [ ] { } | ==, and TACL code is requested.

Effect. The DDL compiler does not generate TACL code for the constant or for any object that refers to the constant.

Recovery. Shorten the constant value and regenerate the TACL constant or object that refers to the constant.

```
***ERROR*** DATETIME or INTERVAL item found within group with
VALUE clause
```

Cause. You specified an SQL DATETIME or SQL INTERVAL item within a group that contains a VALUE clause.

Effect. The DDL compiler rejects the group.

Recovery. Remove the VALUE clause or the SQL DATETIME or SQL INTERVAL item from the group, and then recompile.

```
***ERROR*** DCOMPRESS made record one byte too long for block
```

Cause. The record definition of a key-sequenced file contains a DCOMPRESS clause that makes the block size of the record 1 byte longer than specified in the BLOCK clause.

Effect. The DDL compiler rejects the object.

Recovery. Specify a smaller record size, or remove the DCOMPRESS clause.

```
***ERROR*** DCOMPRESS specified but primary key has nonzero
offset
```

Cause. The record definition of a key-sequenced file contains a DCOMPRESS clause, but the specified primary key does not have offset 0.

Effect. The DDL compiler rejects the object.

Recovery. Specify a primary key that has offset 0, or remove the DCOMPRESS clause.

```
***FATAL ERROR*** DDL cannot run on this version of GUARDIAN
```

Cause. You are attempting to run the DDL compiler on an unsupported product version of the operating system.

Effect. The DDL process does not start.

Recovery. Consult your system manager.

```
***FATAL ERROR*** DDL internal error
```

Cause. An internal error has occurred. No user error is implied.

Effect. The DDL compiler closes the dictionary and source code files and stops processing.

Recovery. Consult your system manager.

```
***FATAL ERROR*** DDL microcode not installed on this cpu
```

Cause. The DDL slap bit is not set.

Effect. The DDL compiler stops processing.

Recovery. Consult your system manager.

```
***WARNING*** DDL OUTPUT DIAGNOSTICS:
```

Cause. You requested DDL output, but the object does not conform to DDL rules.

Effect. A message follows describing the DDL error. The DDL compiler does not write the object to the DDL source file.

Recovery. Correct the error and recompile.

```
***ERROR*** DEF or RECORD exceeds C 32767-byte limit-
object-name
```

Cause. A definition or record generates a C structure that is greater than 32,767 bytes.

Effect. The DDL compiler does not generate C output for the specified definition or record.

Recovery. Shorten the definition or record and regenerate the C source code.

```
***ERROR*** DEF or RECORD exceeds Pascal 32766-byte limit-
object-name
```

Cause. A definition or record generates a Pascal type definition that is greater than 32,766 bytes.

Effect. The DDL compiler does not generate Pascal output for the specified definition or record.

Recovery. Shorten the definition or record and regenerate the Pascal source code.

```
***ERROR*** DEF or RECORD exceeds Pascal 32766-byte limit-
object-name
```

Cause. A definition or record generates a TACL structure that is greater than 5,000 bytes. The entire TACL structure, not just individual fields, must be less than or equal to 5,000 bytes.

Effect. The DDL compiler does not generate TACL output for the specified definition or record.

Recovery. Shorten the definition or record and regenerate the TACL source code.

```
***ERROR*** DEFINITION not found
```

Cause. A definition named in a TOKEN-MAP or a TOKEN-TYPE statement cannot be found in the dictionary.

Effect. The DDL compiler rejects the token map or token type.

Recovery. Correct the definition name or add the referenced definition to the dictionary, then recompile the token map or token type.

```
***ERROR*** Definition or record name already used
```

Cause. You have specified a definition name or record name that has already been used for another object.

Effect. The DDL compiler rejects the duplicate object.

Recovery. Change the name of the definition or record and recompile the statement.

```
***ERROR*** Definition type reference is recursive
```

Cause. The field currently being defined tried to refer to itself.

Effect. The DDL compiler rejects the object.

Recovery. Take out the reference to the field and recompile the statement.

```
***ERROR*** DEPENDING ON element not within OCCURS range
```

Cause. The DEPENDING ON element has a VALUE clause in which the specified value is not within the range specified by the OCCURS clause.

Effect. The DDL compiler rejects the object.

Recovery. Specify a valid value and recompile.

```
***FATAL ERROR*** Dict has been moved and key files were not
FUP ALTERed
```

Cause. You moved the dictionary from another subvolume, but did not alter the key files to reflect this.

Effect. The DDL compiler closes the dictionary and source code files and stops processing.

Recovery. Follow the procedure Moving a Dictionary on page 10-14.

```
***ERROR*** Dictionary conversion failed
```

Cause. The DDL compiler was unable to convert the dictionary. To determine the cause, see the preceding error message in your output listing.

Effect. The dictionary is not converted.

Recovery. Correct the error and rebuild the dictionary.

```
***FATAL ERROR*** DICTIONARY IS CORRUPT- purge and restart
```

Cause. Data stored in the dictionary is in an inconsistent state, or some of the dictionary files are missing.

Effect. The DDL compiler closes the dictionary and source code files and stops processing.

Recovery. Purge the dictionary files manually and rebuild the dictionary from a saved source.

```
***ERROR*** Dictionary is only partially purged in subvol-
subvolume-name
```

Cause. Some of the dictionary files have been purged, but not all of them. The remaining files might not have been purged because they were in use, or there was a security violation on the files.

Effect. You cannot use the dictionary with part of the files missing.

Recovery. Determine the status of the individual files and purge them manually when possible.

```
***WARNING*** Dictionary not found
```

Cause. You entered a DICTR command, but no dictionary exists on the specified volume and subvolume.

Effect. The DDL compiler does not open the specified dictionary.

Recovery. Reissue the DICTR command specifying the correct volume and subvolume.

```
***WARNING*** Dictionary opened- cannot reopen while defining
an object
```

Cause. You placed a DICT command within a group definition or a RECORD statement, and dictionary is already open.

Effect. The DICT command is ignored, and the open dictionary remains open.

Recovery. Place the DICT command between statements and retry.

```
***WARNING*** Dictionary version is current, no conversion is
done
```

Cause. You attempted to convert a current dictionary.

Effect. The DDL compiler closes the dictionary and stops processing.

Recovery. No recovery is necessary.

```
***ERROR*** Disk file exists but is not an EDIT file
```

Cause. A COBOL, DDL, FORTRAN, FUP, TACL, TAL, C, or Pascal command specified a file that is not in EDIT format.

Effect. The DDL compiler does not produce the requested source code.

Recovery. Reissue the COBOL, DDL, FORTRAN, FUP, TACL, TAL, C, or Pascal command, specifying an EDIT file.

```
***ERROR*** Duplicate text item for locale locale-name
```

Cause. A literal with the same locale name has already been associated with the text item.

Effect. The DDL compiler rejects the object.

Recovery. Ensure that each locale name for a text item is unique.

```
***ERROR*** Duplicate VALUE on Level 89 item- field-name
```

Cause. An enumeration clause for a field specifies the same value as another enumeration clause for the field.

Effect. The DDL compiler rejects the object.

Recovery. Specify a different enumeration value and recompile.

```
***WARNING*** Editors only work from DDL when in interactive
mode
```

Cause. You specified an EDIT command in a DDL source file or session in which an OUT command, or the OUT run-option, has specified the source code file to be a file other than an interactive terminal. You can use EDIT only in an interactive session: a session in which the input/output file is an interactive terminal.

Effect. The DDL compiler issues a warning and ignores the EDIT command.

Recovery. Remove the EDIT command and recompile if necessary.

```
***WARNING*** EDIT did not recieve the startup message- File
error file-error
```

Cause. The EDIT or T4/30/10EDIT process did not receive a startup message because of the file error identified by *file-error*.

Effect. The DDL compiler cannot start the EDIT process.

Recovery. Reissue the EDIT command. If the problem persists, see your system manager

```
***WARNING*** EDIT file could not be opened- File error
file-error
```

Cause. The EDIT process cannot be started because of the file error identified by *file-error*.

Effect. The DDL compiler cannot start the EDIT process.

Recovery. Reissue the EDIT command. If the problem persists, see your system manager.

```
FILE ERROR - filename - Edit file line number too large (537)
```

Cause. The source output file, *filename*, is an EDIT file and the source output exceeded 99,999 lines.

Effect. The source output file is incomplete.

Recovery.

1. Purge the incomplete source output file.
2. Use these FUP commands to create a file for source output:

   ```
   SET TYPE E
   SET EXT (large-number,large-number)
   SET MAXEXTENTS large-number
   CREATE filename
   ```

3. Use the file that you created in Step 2 as the source output file in one of these source output commands:
   - C on page 9-8
   - COBOL on page 9-26
   - DDL on page 9-42
   - FORTRAN on page 9-63
   - PASCAL (D-Series Systems Only) on page 9-86
   - TAL on page 9-105

```
***ERROR*** Edit picture inconsistent with data Type
```

Cause. The edit picture specified in an EDIT-PIC clause is not valid for the data type of the field being defined.

Effect. The DDL compiler rejects the object.

Recovery. Specify a valid edit picture in the clause and resubmit the statement to the DDL compiler.

```
***WARNING*** EDIT process could not be created- File error
file-error
```

Cause. A file error indicated by *file-error* occurred during creation of the EDIT process. No user error is implied.

Effect. The DDL compiler cannot start the EDIT process.

Recovery. Reissue the EDIT command. If the problem persists, consult your system manager.

```
***WARNING*** EDIT process could not be created- Newprocess
error newprocess-error
```

Cause. A NEWPROCESS error occurred during creation of the EDIT process. No user error is implied.

Effect. The DDL compiler cannot start the EDIT process.

Recovery. Reissue the EDIT command. If problem persists, consult your system manager.

```
***WARNING*** EDIT stopped or abnormally ended during
execution
```

Cause. Usually, this is a system error.

Effect. The DDL compiler stops the EDIT process.

Recovery. Consult your system manager.

```
***ERROR*** Element contains BINARY 64 UNSIGNED data type -
element_name
***ERROR*** BINARY 64 UNSIGNED is not supported in
language_name
```

Cause. The DDL compiler was asked to generate output for the source language *language_name* (which is neither C nor TAL) and the DDL item contains the BINARY 64 UNSIGNED field identified by *element_name*.

Effect. The DDL compiler issues error messages and does not generate output for the requested language. For example:

```
!?Cobol
! def def1 type binary 64 unsigned.
*** WARNING *** COBOL 85 OUTPUT DIAGNOSTICS:
*** ERROR *** Element contains BINARY 64 UNSIGNED data type – DEF1
*** ERROR *** BINARY 64 UNSIGNED data type is not supported in COBOL
```

Recovery. Recovery is not possible. Remove the BINARY 64 UNSIGNED data item from the definition or record.

```
***ERROR*** Element contains PACKED-DECIMAL data type -
element_name
```

Cause. The DDL compiler was asked to generate output for a source language other than COBOL and the DDL item contains the PACKED-DECIMAL field identified by *element_name*.

Effect. The DDL compiler issues error messages and does not generate output for the requested language. For example:

```
!?tal
! SCHEMA PRODUCED DATE - TIME : 8/01/2000 - 15:05:22
  Output source for TAL is opened on $ZTN1.#PTPJHYV
!def emp pic 9999 PACKED-DECIMAL.
  Definition EMP size is 3 bytes.
  Definition EMP added to dictionary.
  *** WARNING *** TAL OUTPUT DIAGNOSTICS:
  *** ERROR *** Element contains PACKED-DECIMAL data type - EMP
  *** ERROR *** PACKED-DECIMAL data type is not supported in  TAL
  *** ERROR *** Errors detected - no output produced for EMP
```

Recovery. Recovery is not possible. Remove the COMP-3, COMPUTATIONAL-3, or PACKED-DECIMAL data item from the definition or record.

```
***ERROR*** Element being redefined not found in this group
```

Cause. The element being redefined is not an element in the same group as the redefining element.

Effect. The DDL compiler rejects the object.

Recovery. Put the element in the same group and recompile.

```
***ERROR*** Element being redefined redefines another element
```

Cause. An element has a REDEFINES clause redefining an element that also has a REDEFINES clause, and the first element refers to a different field than the second element

Effect. The DDL compiler rejects the object.

Recovery. Change the REDEFINES clause in the first element to refer to the same field as does the REDEFINES clause in the second element.

```
***ERROR*** Element name already used in or qualifies this
group- element-name
```

Cause. An element at the same lexical level as this element, and within the same group, has the same name.

Effect. The DDL compiler rejects the object.

Recovery. Change one of the names and recompile.

```
***ERROR*** Element/group size does not match the size of the
TACL type- name
```

Cause. A TACL clause is specified for an item whose length does not agree with the TACL data type. For more information about TACL data type lengths, see TACL on page 6-44.

Effect. The DDL compiler rejects the object.

Recovery. Change the length of the item, or remove the TACL clause, and recompile.

```
***FATAL ERROR*** Encountered an unsupported version of the
dictionary
```

Cause. The DDL compiler encountered a product version of a dictionary that the current product version of the DDL compiler cannot access.

Effect. The DDL compiler stops processing.

Recovery. Consult your system manager.

```
***WARNING*** ENFORM reserved word- word
```

Cause. You used an Enform Plus reserved word as a record, group, or field name.

Effect. The DDL compiler continues processing the statement.

Recovery. Change the name and recompile for Enform Plus access. If Enform Plus is not to be used, recompilation is not necessary.

```
***ERROR*** ENUM values out of range for bit field-
field-name
```

Cause. The enumeration definition specified for a bit field has a value or values that do not fit in the bit field.

Effect. The DDL compiler rejects the object that includes the bit field.

Recovery. Do one of the following:

- Change the values in the enumeration definition to fit the specified number of bits.
- Specify enough bits to contain the largest value in the enumeration definition.
- Specify an enumeration definition whose values fit in the bit field.
- Omit the ENUM clause from the type specification; then recompile the object that contains the bit field.

```
***FATAL ERROR*** Error aborting a transaction
```

Cause. The DDL compiler encountered an error while trying to abort a transaction in an audited dictionary. Transactions are aborted when a change to the dictionary is begun but cannot be completed.

Effect. The DDL compiler closes the dictionary and source code files and stops processing.

Recovery. Your dictionary might be corrupt; consult your system manager.

```
***FATAL ERROR*** Error beginning a transaction
```

Cause. The DDL compiler encountered an error while trying to begin a transaction in an audited dictionary. A transaction begins when the dictionary files must be updated.

Effect. The DDL compiler closes the dictionary and source code files and stops processing.

Recovery. If the problem persists, consult your system manager.

```
***FATAL ERROR*** Error ending a transaction
```

Cause. The DDL compiler encountered a file error while trying to end a transaction in an audited dictionary. A transaction ends when all related files are updated.

Effect. The DDL compiler closes the dictionary and source code files and stops processing.

Recovery. If the problem persists, consult your system manager.

```
***WARNING*** Errors detected- no output produced for
object-name
```

Cause. The DDL compiler detected one or more errors while processing the statement for *object-name*.

Effect. The DDL compiler rejects the object.

Recovery. Correct the error or errors and recompile.

```
***WARNING*** ERRORS is not a legal positive integer- value
not changed
```

Cause. The ERRORS command specifies an integer that is not in the range 1 through 32767.

Effect. The DDL compiler ignores the ERRORS command.

Recovery. Specify a valid value in the ERRORS command and recompile.

```
***WARNING*** Expecting continuation of source command
```

Cause. A SOURCE command contains multiple input lines, but the second and following input lines do not start with ?.

Effect. The DDL compiler might process subsequent input lines incorrectly.

Recovery. Put ? at start of each SOURCE input line and recompile.

```
***ERROR*** EXT is not a multiple of BLOCK
```

Cause. A value for EXT is not a multiple of the block size; the default block size is 4096 bytes.

Effect. The DDL compiler rejects the object.

Recovery. Change the EXT value to a multiple of BLOCK. For example, if BLOCK = 4096, 4 is a valid value for EXT, but 3 is not.

```
***ERROR*** EXT is not a multiple of BUFFERSIZE
```

Cause. A value for EXT is not a multiple of the buffer size; the default buffer size is 4096 bytes.

Effect. The DDL compiler rejects the object.

Recovery. Change the EXT value to a multiple of BUFFERSIZE. For example, if BUFFERSIZE = 4096, 4 is a valid value for EXT, but 3 is not.

```
***ERROR*** EXT must be a positive integer
```

Cause. A value for EXT is equal to or less than 0; the extent size must be a positive integer.

Effect. The DDL compiler rejects the object.

Recovery. Correct the extent size and recompile.

```
***ERROR*** External clause must be on object name level
```

Cause. You specified an EXTERNAL clause for a DEFINITION statement and the clause was not on the object name level.

Effect. The DDL compiler does not execute the DEFINITION statement.

Recovery. Specify the EXTERNAL clause on the object name level and recompile.

```
***WARNING*** Extra level of reference introduced in C's
union- object-name
```

Cause. The DDL compiler generated a C union because the DDL compiler encountered a REDEFINES clause. The C structure containing such a union has one more item level than the corresponding DDL structure containing the REDEFINES clause.

Effect. The DDL compiler still generates source code for C.

Recovery. No recovery is necessary.

```
***WARNING*** Extra level of reference introduced in Pascal's
variant- object-name
```

Cause. The DDL compiler generated an anonymous Pascal record because the DDL compiler encountered a REDEFINES clause. The record is anonymous because it has a name but no type. The DDL compiler generated the record name by prefixing a V_ to the name of the first structure being redefined.

Effect. The DDL compiler still generates source code for Pascal.

Recovery. No recovery is necessary.

```
***ERROR*** Field has variable OCCURS-
Line.LineItem.LocalName
```

Cause. The field inside the DEFINITION used in the TOKEN-MAP statement has an OCCURS DEPENDING ON clause.

Effect. The DDL compiler rejects the object.

Recovery. Remove the OCCURS DEPENDING ON clause for the field and recompile.

```
***WARNING*** File name ignored when opening dictionary
```

Cause. A fully qualified file name was specified as a dictionary subvolume in a DICT, DICTN, or DICTR command.

Effect. The DDL compiler ignores the file name and opens the dictionary on the specified subvolume.

Recovery. No recovery is necessary.

```
***WARNING*** File name not specified
```

Cause. The DDL compiler looked for a file name, but did not find it because of an incorrect command.

Effect. The DDL compiler skips the command.

Recovery. Correct the command and recompile.

```
***WARNING*** File name specified for primary key is ignored
```

Cause. The key assignment clause for a primary key includes a file name. You can specify a file name in the KEY IS clause only for an alternate key.

Effect. The DDL compiler uses the file name specified in the file creation part of the record statement and ignores the file name specified in the key assignment clause.

Recovery. No recovery is necessary because a primary key does not require a separate file.

```
***ERROR*** Filler cannot have a DISPLAY clause
```

Cause. A FILLER field is described with a DISPLAY clause.

Effect. The DDL compiler rejects the object.

Recovery. Remove the DISPLAY clause and recompile.

```
***ERROR*** Filler cannot have a HEADING clause
```

Cause. A FILLER field is described with a HEADING clause.

Effect. The DDL compiler rejects the object.

Recovery. Remove the HEADING clause and recompile.

```
***ERROR*** Filler cannot have a HELP clause
```

Cause. A FILLER field is described with a HELP clause.

Effect. The DDL compiler rejects the object.

Recovery. Remove the HELP clause and recompile.

```
***ERROR*** Filler cannot have a KEYTAG clause
```

Cause. A FILLER field is described with a KEYTAG clause.

Effect. The DDL compiler rejects the record.

Recovery. Remove the KEYTAG clause and recompile.

```
***ERROR*** Filler cannot have a MUST BE clause
```

Cause. A FILLER field is described with a MUST BE clause.

Effect. The DDL compiler rejects the object.

Recovery. Remove the MUST BE clause and recompile.

```
***ERROR*** Filler cannot have a NULL clause
```

Cause. A FILLER field is described with a NULL clause.

Effect. The DDL compiler rejects the object.

Recovery. Remove the NULL clause and recompile.

```
***ERROR*** Filler cannot have a REDEFINES clause
```

Cause. A FILLER field is described with a REDEFINES clause.

Effect. The DDL compiler rejects the object.

Recovery. Remove the REDEFINES clause and recompile.

```
***ERROR*** Filler cannot have a TACL clause
```

Cause. A FILLER field is described with a TACL clause.

Effect. The DDL compiler rejects the object.

Recovery. Remove the TACL clause and recompile.

```
***ERROR*** Filler cannot have an EXTERNAL clause
```

Cause. A FILLER field is described with an EXTERNAL clause.

Effect. The DDL compiler rejects the object.

Recovery. Remove the EXTERNAL clause and recompile.

```
***ERROR*** Filler cannot have an UPSHIFT clause
```

Cause. A FILLER field is described with an UPSHIFT clause.

Effect. The DDL compiler rejects the object.

Recovery. Remove the UPSHIFT clause and recompile.

```
***ERROR*** Filler cannot have a USER clause
```

Cause. A FILLER field is described with one or more USER clauses.

Effect. The DDL compiler rejects the object.

Recovery. Remove the USER clauses and recompile.

```
***ERROR*** Filler must have a PICTURE or TYPE clause
```

Cause. A FILLER field has no PICTURE or TYPE clause to specify its length.

Effect. The DDL compiler rejects the object.

Recovery. Add a PICTURE or TYPE clause to the field description and recompile.

```
***ERROR*** FORTRAN element with size greater than 255 bytes-
element-name
```

Cause. An elementary field is larger than 255 bytes, and output to a FORTRAN source file is requested.

Effect. The DDL compiler does not write object containing field larger than 255 bytes to the FORTRAN source file.

Recovery. Describe the field as two or more smaller fields and recompile.

```
***WARNING*** FORTRAN OUTPUT DIAGNOSTICS:
```

Cause. You requested FORTRAN output, but the object does not conform to FORTRAN syntax rules.

Effect. A message follows describing the FORTRAN error. The DDL compiler does not write the object to the FORTRAN source file.

Recovery. Correct the error and recompile if you want FORTRAN output.

```
***WARNING*** FUP OUTPUT DIAGNOSTICS:
```

Cause. You requested FUP output, but the object does not conform to FUP rules.

Effect. A message follows describing the FUP error. The DDL compiler does not write the object to the FUP file-creation source file.

Recovery. Correct the error and recompile.

```
***WARNING*** FUPBLOCKSIZE must be from 1 to 4096- value not
changed
```

Cause. A RECORD statement specified a block size less than 1 or greater than 4,096.

Effect. The DDL compiler uses the default block size, 4,096.

Recovery. Specify a block size from 1 to 4,096 and recompile the record.

```
***ERROR*** Group item exceeds Pascal's nesting limit for
records- group-name
```

Cause. The DDL compiler encountered a group item that exceeds the 30-level nesting limit for Pascal.

Effect. The DDL compiler rejects the object.

Recovery. Reduce the number of nesting levels and recompile.

```
***ERROR*** Group items cannot have a JUSTIFIED clause
```

Cause. You specified a JUSTIFIED clause for a group.

Effect. The DDL compiler rejects the object.

Recovery. Remove the JUSTIFIED clause from the group definition or description, add a JUSTIFIED clause to the description of each field in the group, and recompile.

```
***ERROR*** Group items cannot have a MUST BE clause
```

Cause. You specified a MUST BE clause for a group.

Effect. The DDL compiler rejects the object.

Recovery. Remove the MUST BE clause from the group definition or description, add a MUST BE clause to the description of each field in the group, and recompile.

```
***ERROR*** Group level initialization VALUE must be
alphanumeric
```

Cause. You specified a numeric value with a VALUE clause for a group; group values must be alphanumeric

Effect. The DDL compiler rejects the object.

Recovery. Specify an alphanumeric value and recompile.

```
***ERROR*** Group with initial VALUE contains MUST BE-
group-name
```

Cause. You specified a VALUE clause for a group that contains a field described with a MUST BE clause. If a group has an initial value, none of its field descriptions can include a MUST BE clause.

Effect. The DDL compiler rejects the object.

Recovery. Remove the MUST BE clause and recompile.

```
***WARNING*** HELP line exceeds 77 characters
```

Cause. A single line of help text exceeds 77 characters.

Effect. The DDL compiler enters the object definition in the dictionary.

Recovery. If the help text must fit on Pathmaker screens, break the text into shorter lines and recompile. A single HELP clause can have many lines of help text, but each line must be no more than 77 characters long.

```
***WARNING*** Identifier name changed, might cause duplicate-
object-name
```

Cause. A The DDL compiler name containing a hyphen (-) might duplicate a FORTRAN name.

Effect. When generating FORTRAN source code, the DDL compiler might use another identifier with the same FORTRAN name but a different DDL name.

Recovery. Avoid using a hyphen in a name for a FORTRAN object.

```
***ERROR*** Identifier too long
```

Cause. A definition, record, or element name has more than 30 ASCII characters.

Effect. The DDL compiler rejects the object.

Recovery. Shorten the name and recompile.

```
***ERROR*** Improper type of Constant for this usage
```

Cause. A constant is used as a value, but the constant data type is not consistent with the data type of the object receiving the value.

Effect. The DDL compiler rejects the object.

Recovery. Check the data type and use a constant whose value is a number for a numeric type or a string for an alphanumeric type, then recompile the object.

```
***ERROR*** Inconsistent VERSION within byte- bit-field-name
```

Cause. The product version specified in the TOKEN-MAP statement does not match bit fields stored in the same byte.

Effect. The DDL compiler rejects the TOKEN-MAP object.

Recovery. Specify the same product version for bit fields that share the same byte. If a bit field extends across two bytes, specify the same product version for bit fields that share the same word.

```
***WARNING*** Increment is not a legal positive int- value
not changed
```

Cause. A COBOL level-number increment in a COBLEVEL command is equal to or less than 0.

Effect. The DDL compiler does not change the increment.

Recovery. Correct the error and recompile if you want to.

```
***ERROR*** INDEX must be a 1 or 2 word single item and
computational
```

Cause. A field whose definition or description includes a USAGE IS INDEX clause is not 2 or 4 bytes, is not a single field, or is not a computational item.

Effect. The DDL compiler rejects the field or the object that includes the field.

Recovery. Remove the USAGE IS INDEX clause, or change the field definition to meet the requirements for using this clause, and recompile the object.

```
***WARNING*** INDEX must be 1 word for COBOL 74 and 2 words
for COBOL85- index-name
```

Cause. A field described with the USAGE IS INDEX clause is the wrong size for COBOL output.

Effect. The DDL compiler does not produce the requested output for the object.

Recovery. Change the size of the index field, specify an index field of the correct size, or request the output appropriate for the field size.

```
***ERROR*** INDEXED BY is invalid without OCCURS clause-
object-name
```

Cause. A definition or record description includes an INDEXED BY attribute without an OCCURS or OCCURS DEPENDING ON clause.

Effect. The DDL compiler rejects the object.

Recovery. Add an OCCURS or OCCURS DEPENDING ON clause, or remove the INDEXED BY attribute, and recompile the object.

```
***ERROR*** Initial VALUE exceeds size of group name
```

Cause. An initial value for a group exceeds the combined size of the fields within the group.

Effect. The DDL compiler rejects the object.

Recovery. Change the initial value or the combined field size and recompile.

```
***ERROR*** Initial VALUE violates MUST BE constraint
```

Cause. An initial value for a field is outside the range specified for that field in a MUST BE clause.

Effect. The DDL compiler rejects the object.

Recovery. Change the MUST BE range or the VALUE clause and recompile.

```
***ERROR*** Initial VALUE's conflict in group- name
```

Cause. A group with a VALUE clause contains a field that also has a VALUE clause.

Effect. The DDL compiler rejects the object.

Recovery. Remove the VALUE clause from the field description and recompile.

```
***WARNING*** Input line exceeds 132 characters; truncation
occurred
```

Cause. An input line contains more than 132 ASCII characters.

Effect. The DDL compiler truncates the line to 132 characters.

Recovery. Break the line into several shorter lines and reenter them.

```
***ERROR*** Integer conversion error- value
```

Cause. A numeric value is greater than 32,767 or less than -32,768.

Effect. The DDL compiler rejects the object.

Recovery. Change the value and recompile.

```
***WARNING*** Integer type is generated for bit field-
field-name
```

Cause. You requested language output other than Pascal for a bit map declared as a single field. To ensure that bit maps outside group structures are compatible between languages, field definitions for bit fields are generated as 16-bit integer items.

Effect. The DDL compiler takes no action beyond the warning message.

Recovery. No recovery is necessary.

```
***ERROR*** Invalid character
```

Cause. You used an invalid special character.

Effect. The DDL compiler rejects the object.

Recovery. Remove the invalid character and recompile.

```
***WARNING*** Invalid compiler command- incorrect-command
```

Cause. The indicated command is invalid.

Effect. The DDL compiler ignores the command.

Recovery. Correct the command and recompile if necessary.

```
***ERROR*** Invalid display format string
```

Cause. The display format in a DISPLAY clause is incorrect. For display format rules, see the description of the AS modifier of the LIST command in the *Enform Plus Reference Manual*.

Effect. The DDL compiler rejects the object.

Recovery. Correct the display format and recompile.

```
***ERROR*** Invalid EDIT picture
```

Cause. The edit picture you specified in an EDIT-PIC clause does not follow the COBOL rules for edit pictures.

Effect. The DDL compiler rejects the object.

Recovery. Specify a valid edit picture and recompile.

```
***WARNING*** Invalid file name
```

Cause. A command specifies an invalid file name.

Effect. The DDL compiler ignores the command.

Recovery. Correct the file name and recompile if necessary.

```
***ERROR*** Invalid file name
```

Cause. The FILE IS clause of a RECORD statement specifies an invalid file name

Effect. The DDL compiler rejects the object.

Recovery. Correct the file name and recompile.

```
***ERROR*** Invalid identifier format
```

Cause. A hyphen (-) is the last character of a name identifying a record, definition, group, or field.

Effect. The DDL compiler rejects the object.

Recovery. Correct the error and recompile.

```
***ERROR*** Invalid lexical level
```

Cause. A specified level number is less than 02 or greater than 49, or an elementary field with level *n* is followed by an elementary field or a group with level *n* +1 or greater.

Effect. The DDL compiler rejects the object.

Recovery. Correct the level number and recompile.

```
***ERROR*** Invalid locale name
```

Cause. The locale name is invalid.

Effect. The DDL compiler rejects the text item.

Recovery. Use a valid locale name.

```
***ERROR*** Invalid number
```

Cause. You entered an invalid number; for example, %8.

Effect. The DDL compiler rejects the object containing the invalid number.

Recovery. Specify the number correctly and recompile.

```
***ERROR*** Invalid OCCURS value
```

Cause. In an OCCURS *max* TIMES clause, the value *max* is less than or equal to 1.

Effect. The DDL compiler rejects the object.

Recovery. Correct *max* and recompile.

```
***ERROR*** Invalid PICTURE string
```

Cause. PICTURE string does not conform to required syntax.

Effect. The DDL compiler rejects the object.

Recovery. Correct the PICTURE string and recompile.

```
***ERROR*** Invalid range specified
```

Cause. The first value in a specified range is greater than the second value.

Effect. The DDL compiler rejects the object containing invalid range.

Recovery. Correct the range and recompile.

```
***WARNING*** Invalid section name
```

Cause. A section name in a SOURCE command is invalid or is not present when expected, or a comma is missing between section names in a SOURCE command.

Effect. The DDL compiler issues a warning and ignores the SOURCE command.

Recovery. Correct the error and recompile.

```
***ERROR*** Invalid size for element type
```

Cause. A TYPE clause specifies a size that is invalid for the particular data type.

Effect. The DDL compiler rejects the object.

Recovery. Correct the error and recompile.

```
***ERROR*** Invalid SSID format
```

Cause. The subsystem ID you specified in an SSID clause is not in the correct format for a subsystem ID.

Effect. The DDL compiler rejects the object.

Recovery. Specify the subsystem ID correctly and recompile.

```
***ERROR*** Invalid syntax- ^ under symbol where error
encountered
```

Cause. A statement violates DDL syntax rules. Specifying a DDL keyword as the constant name in a CONSTANT statement can cause this error.

Effect. The DDL compiler rejects the object.

Recovery. Modify the statement to conform to DDL syntax rules and recompile.

```
***FATAL ERROR*** Invalid text type for comment
```

Cause. The code in the TEXT-TYPE field of the dictionary file DICTOTF is supposed to identify a comment, but the code is invalid for a comment.

Effect. The DDL compiler cannot use the dictionary.

Recovery. This error cannot be recovered using the DDL compiler alone. Report the error to your service provider.

```
***FATAL ERROR*** Invalid text type in dictionary
```

Cause. A code in the TEXT-TYPE field of the dictionary file DICTOTF is invalid for a text type.

Effect. The DDL compiler cannot use the dictionary.

Recovery. This error cannot be recovered using the DDL compiler alone. Report the error to your service provider.

```
***ERROR*** Invalid value spedified for MAXEXTENTS
```

Cause. You specified a MAXEXTENTS value that is outside the valid range. As many as 978 maximum extents can be specified, but the actual upper limit depends on the number of alternate keys. For more information about maximum extents, see the *Enscribe Programmer's Guide*.

Effect. The DDL compiler rejects the object.

Recovery. Specify the MAXEXTENTS value correctly and recompile.

```
***ERROR*** Invalid version number format
```

Cause. You specified a product version in a VERSION constant or in the VERSION clause of a TOKEN-MAP statement that is not of the form *ann*, in which *a* is a letter and *nn* is a two-digit number.

Effect. The DDL compiler rejects the object.

Recovery. Specify the product version correctly and recompile.

```
***ERROR*** It is not possible to REDEFINE a level 66 item
```

Cause. A REDEFINES clause refers to a level 66 item.

Effect. The DDL compiler rejects the object.

Recovery. Correct or remove the REDEFINES clause and recompile.

```
***ERROR*** Item with MUST BE found on or within REDEFINES
item- field-name
```

Cause. You specified a MUST BE clause for a redefining field.

Effect. The DDL compiler rejects the object.

Recovery. Remove the MUST BE or REDEFINES clause and recompile.

```
***ERROR*** Item with UPSHIFT found on nonalphabetic data
item- field-name
```

Cause. You specified an UPSHIFT clause for a field that does not have an alphabetic data type; UPSHIFT is allowed only for fields described by PIC A, PIC X, TYPE CHARACTER, TYPE *, or TYPE *def-name*, in which *def-name* or * is a definition of an alphabetic or alphanumeric type field.

Effect. The DDL compiler rejects the object.

Recovery. Change the data type of the field or remove the UPSHIFT clause.

```
***ERROR*** Item with UPSHIFT found on or within REDEFINES
item- object-name
```

Cause. You specified an UPSHIFT clause within a redefinition.

Effect. The DDL compiler rejects the object.

Recovery. Remove the UPSHIFT clause from the redefining group or field and recompile.

```
***FATAL ERROR*** I/O Error accessing $System.System.UserID
```

Cause. The DDL compiler encountered an error while attempting to access the USERID file on $SYSTEM.SYSTEM. The DDL compiler must access this file to record the creator user ID and modifier user ID in the dictionary.

Effect. The DDL compiler closes the dictionary and source code files and stops processing.

Recovery. If the problem persists, consult your system manager.

```
***ERROR*** JUSTIFIED must be on alphabetic or alphanumeric
item
```

Cause. The JUSTIFIED clause can appear only in an alphabetic or alphanumeric elementary item.

Effect. The DDL compiler rejects the object.

Recovery. Remove the JUSTIFIED clause and recompile.

```
***ERROR*** Key attribute already spedified- key-attribute
```

Cause. You have specified the indicated key attribute on an alternate key specification.

Effect. The DDL compiler rejects the object.

Recovery. Specify a different key attribute.

```
***ERROR*** Key element invalid- Dictionary is corrupt
```

Cause. A dictionary was updated incorrectly.

Effect. The DDL compiler rejects the object.

Recovery. Rebuild the dictionary.

```
***ERROR*** Key file name already used- file-name
```

Cause. You specified a file name for an alternate key assignment in a RECORD statement that is not unique. The file name is already specified for another alternate key, and either or both keys are specified as unique.

Effect. The DDL compiler rejects the object.

Recovery. Specify a different file name for the alternate key.

```
***ERROR*** Keys specified for unstructured file
```

Cause. You specified a key field for an unstructured file; unstructured files cannot have key fields.

Effect. The DDL compiler rejects the record.

Recovery. Remove the key specification and recompile.

```
***KEYTAG string must not exceed 2 bytes
```

Cause. You specified a KEYTAG string with more than 2 characters.

Effect. The DDL compiler rejects the record.

Recovery. Specify a KEYTAG string no longer than 2 characters and recompile.

```
***ERROR*** KEYTAG used twice- keytag value
```

Cause. The same KEYTAG value occurs more than once in a RECORD statement, or the value is equivalent in both numeric and ASCII form.

Effect. The DDL compiler rejects the record.

Recovery. Correct the error and recompile.

```
***ERROR*** KEYTAG with repeating group or element
```

Cause. A field or group that is a key field has an OCCURS clause.

Effect. The DDL compiler rejects the record.

Recovery. Remove the OCCURS clause or the key specification for the field and recompile.

```
***WARNING*** Language check redundant: output already being
produced
```

Cause. A language checking command (CCHECK, COBCHECK, FORCHECK, PASCALCHECK, or TALCHECK) follows a command (C, COBOL, FORTRAN, pTAL, Pascal, or TAL) that requests source output.

Effect. The DDL compiler ignores the command.

Recovery. No recovery is necessary.

```
***ERROR*** LAst element is not elementary
```

Cause. A group element is the last element in a RECORD or DEFINITION statement; every group must contain at least one elementary field.

Effect. The DDL compiler rejects the object.

Recovery. Correct the error and recompile.

```
***ERROR*** LEvel 88 not allowed for bit items
```

Cause. One or more level 88 clauses follow the definition or description of a bit field. The DDL compiler does not allow level 88 clauses for bit maps because COBOL does not support such structures, and level 88 items are meaningful only for COBOL.

Effect. The DDL compiler rejects the object.

Recovery. To avoid the error message, remove the level 88 clause or clauses following the bit field and recompile the object.

```
***ERROR*** Level 88 or level 89 must follow elementary items
only
```

Cause. A level 88 element directly follows a group element or precedes all elements.

Effect. The DDL compiler rejects the object.

Recovery. Put the level 88 element after a field description and recompile.

```
***ERROR*** LEvel 88 value inconsistent with data type
```

Cause. A level 88 value is incompatible with the type of the field that the level 88 clause describes.

Effect. The DDL compiler rejects the object.

Recovery. Correct the error and recompile.

```
***ERROR*** Level 89 clause must follow an item with TYPE
ENUM
```

Cause. A field definition or description whose type is not ENUM contains a level 89 clause.

Effect. The DDL compiler rejects the object.

Recovery. Remove any level 89 clauses from the field definition or description, or change the type to ENUM, and recompile.

```
***WARNING*** LINECOUNT is not a legalk positive integer-
value not changed
```

Cause. The number in a LINECOUNT *number* command is not a positive integer.

Effect. The line count is unchanged.

Recovery. Correct *number* and recompile if necessary.

```
***WARNING*** Literal too long, commenting out literal for
key key-value
```

Cause. A literal name you used as a key in a record definition written to a pTAL or TAL source file exceeds the pTAL or TAL limit on name size.

Effect. The DDL compiler changes the literal name to a comment.

Recovery. Shorten the key name and recompile.

```
***WARNING*** Literal too long for key key value
```

Cause. A TALCHECK command found that a literal name used as a key in a record definition to be written to a pTAL or TAL source file exceeds the pTAL or TAL limit on name size.

Effect. The DDL compiler does not do anything.

Recovery. If you want the DDL compiler to write the record definition to a pTAL or TAL source file without changing the literal name to a comment, shorten the key name and recompile.

```
***ERROR*** Logical type mixing not supported by FORTRAN-
object-name
```

Cause. An object to be written to a FORTRAN source file contains both LOGICAL 2 and LOGICAL 4 data types.

Effect. The DDL compiler does not write the object containing different LOGICAL data types to the FORTRAN source file.

Recovery. Correct the error and recompile; rebuild the dictionary if necessary.

```
***WARNING*** Matched2 alignment not supported in Pascal
```

Cause. You requested Pascal source output for a record or definition that was created with matched2 alignment.

Effect. The DDL compiler does not generate the Pascal source.

Recovery. Compile the definition without the CFIELDALIGN_MATCHED2 command.

```
***ERROR*** Missing section name
```

Cause. A schema contains a SECTION command with no section name, and a SOURCE command requests a section in that schema.

Effect. The DDL compiler ignores the SECTION command.

Recovery. Specify a name in the SECTION command and recompile if you want to.

```
***ERROR*** Missing subfields
```

Cause. A group description at lexical level *n* is followed by a group or field description at level *n* or less; for example:

```
02 A.
02 B PIC X.
```

Effect. The DDL compiler rejects the object.

Recovery. Correct the level numbers and recompile.

```
***ERROR*** More than one initial VALUE specified
```

Cause. You have entered a VALUE clause that contains more than one initial value.

Effect. The DDL compiler rejects the object.

Recovery. Remove all but one initial value and recompile.

```
***ERROR*** More than one sequence clause specified-
record-name
```

Cause. The RECORD statement contains more than one SEQUENCE IS clause.

Effect. The DDL compiler rejects the object.

Recovery. Remove all but one SEQUENCE IS clause and recompile.

```
***FATAL ERROR*** More than one version specified- field-name
```

Cause. A field in a token map has more than one product version because the field belongs to a group that has a product version, and the field itself has a product version.

Effect. The DDL compiler closes the dictionary and source code files and stops processing.

Recovery. Remove the product version specification from the field and recompile.

```
***ERROR*** More than 32 I18N definitions
```

Cause. More than 32 I18N definitions were associated with a text item.

Effect. The DDL compiler rejects the text item.

Recovery. Have a maximum of 32 I18N definitions associated with a text item.

```
***WARNING*** Multiple key file names specified for non-
unique keys- file-name
```

Cause. When you specified alternate keys that were not unique, you specified different file names on different keys.

Effect. The DDL compiler continues processing, using only the first file name encountered.

Recovery. Specify file names correctly as required by your situation.

```
***WARNING*** Multiple keys with same offset
```

Cause. More than one key is defined at the same offset in the record, and COBOL does not accept a file definition in which two keys have the same offset.

Effect. If COBOL output is requested, the DDL compiler issues a COBOL error message and suppresses COBOL output.

Recovery. Remove all but one key at the same offset and recompile.

```
***ERROR*** Multiple primary keys
```

Cause. More than one key is identified as a primary key; a key-sequenced file has exactly one primary key.

Effect. The DDL compiler rejects the record.

Recovery. Remove all but one key and recompile.

```
***ERROR*** Must Be not valid on a non-referencing
ENUMeration
```

Cause. A MUST BE clause is specified for a field of type ENUM, and the field does not refer to another field of type ENUM.

Effect. The DDL compiler rejects the object.

Recovery. Remove the MUST BE clause, or make the ENUM field refer to another ENUM field, and recompile.

```
***ERROR*** Name is embedded in a group of the same name-
field-name
```

Cause. A field name has the same name as a group, record, or definition that contains the field, and COBOL output was requested.

Effect. The DDL compiler suppresses COBOL output.

Recovery. Change the field name so that it differs from the names that qualify it and recompile.

```
***WARNING*** No CIFNDEF or CIFDEF is used for this CENDIF,
no output produced for CENDIF.
```

Cause. A CENDIF command was used that did not match with any CIFDEF or CIFNDEF commands used before.

Effect. The DDL compiler ignores the statement and generates a warning message. For example:

```
!?dict
Dictionary opened on subvol $ADE101.ALPHA for update access.
!?C
/*SCHEMA PRODUCED DATE - TIME : 7/21/2000 - 19: 52:07 */
Output sourcefor C is opened on $ZTN1.#PTPJHU8
!?cendif
***WARNING*** No CIFNDEF or CIFDEF is used for this CENDIF, no
output produced for CENDIF.
!
```

Recovery. No recovery is necessary.

```
***WARNING*** No DDL output file; no UPDATE output produced
```

Cause. A DDL command was not entered to open a DDL source file before an OUTPUT UPDATE statement was issued.

Effect. The DDL compiler does not generate DDL source update code.

Recovery. Specify the DDL command before issuing the OUTPUT UPDATE statement and recompile.

```
***ERROR*** No definition for object-name
```

Cause. No record or definition called *object-name* is in the dictionary.

Effect. The DDL compiler rejects the object.

Recovery. Correct the error and recompile.

```
***WARNING*** No dictionary is open, NOSAVE command ignored
```

Cause. You entered a NOSAVE command, but no dictionary is open.

Effect. The DDL compiler ignores the NOSAVE command.

Recovery. Open a dictionary using the DICT command and reissue the NOSAVE command.

```
***ERROR*** No file name- file is assigned or temporary
```

Cause. FUP output is being generated, and a file is specified in DDL as assigned or temporary. FUP output can be generated only for permanent files. Assigned and temporary files can be used only in C, COBOL, FORTRAN, pTAL, Pascal, or TAL programs.

Effect. The DDL compiler does not produce FUP output.

Recovery. If you want FUP output, remove the TEMPORARY or ASSIGNED specification and recompile; otherwise, close the FUP source code file.

```
***ERROR*** No JUSTIFIED clause allowed within a group with
VALUE clause
```

Cause. You specified a JUSTIFIED clause for an elementary item that is subordinate to a group item with a VALUE clause.

Effect. The DDL compiler rejects the object.

Recovery. Remove the JUSTIFIED clause or change the group VALUE clause and recompile.

```
***ERROR*** No level 89 clauses specified for item with TYPE
ENUM- field-name
```

Cause. The definition or description of a field whose type is ENUM does not include any level 89 clauses.

Effect. The DDL compiler rejects the object.

Recovery. Add one or more level 89 clauses to the field definition or description and recompile. (In a single-field definition, BEGIN must precede the first period, and END must follow the last clause.)

```
***ERROR*** NO ODDUNSTR specified for a structured file
```

Cause. NO ODDUNSTR describes a key-sequenced, entry-sequenced, or relative file; NO ODDUNSTR applies only to unstructured files.

Effect. The DDL compiler rejects the object.

Recovery. Change the file type to unstructured, or remove the NO ODDUNSTR attribute, and recompile.

```
***ERROR*** No primary key for key-sequenced file
```

Cause. A FILE clause specified a key-sequenced file, but no primary key was specified in a KEYTAG or KEY IS clause.

Effect. The DDL compiler rejects the record.

Recovery. Specify a primary key, or change the file type, and recompile.

```
***FATAL ERROR*** No records in object build list for
referenced def
```

Cause. The definition in the dictionary is corrupt, and the DDL compiler cannot make the reference.

Effect. The DDL compiler closes the dictionary and source code files and stops processing.

Recovery. Rebuild the dictionary.

```
***ERROR*** No REDERINES clause allowed within an object with
EXTERNAL
```

Cause. You attempted to specify an EXTERNAL clause and one or more line items in the definition or record have a REDEFINES clause.

Effect. The DDL compiler rejects the record.

Recovery. Remove the EXTERNAL clause or the REDEFINES clause and recompile.

```
***ERROR*** No VALUE clause allowed within an object with
EXTERNAL
```

Cause. You attempted to specify an EXTERNAL clause and one or more line items in the definition or record have a VALUE clause.

Effect. The DDL compiler rejects the record.

Recovery. Remove the EXTERNAL clause or the VALUE clause and recompile.

```
***ERROR*** Nonalphanumeric key element- element-name
```

Cause. You specified a numeric field as a key field when COBOL output is requested; COBOL does not allow numeric keys.

Effect. The DDL compiler suppresses COBOL output.

Recovery. Redefine the key field as alphanumeric, or specify a different alphanumeric field as the key, and recompile.

```
***ERROR*** Nonexistent record (File error)
```

Cause. The source file in the SOURCE command is not found in the mentioned subvolume.

Effect. The DDL compiler cannot start reading in the file.

Recovery. Add the file to the correct location and recompile the command.

```
***WARNING*** NOSAVE is not allowed on a PATHMAKER dictionary
```

Cause. A NOSAVE command is issued when a Pathmaker dictionary is open. NOSAVE cannot be used for a Pathmaker dictionary.

Effect. The DDL compiler ignores the NOSAVE command.

Recovery. If you want to run a test compilation using NOSAVE, you can create a test dictionary on a subvolume unconnected with the Pathmaker project.

```
***ERROR*** NOVALUE cannot be specified unless referencing a
DEF name
```

Cause. A NOVALUE clause describes a field that has a PICTURE or TYPE *data-type* clause; NOVALUE can be used only in definitions that refer to previous definitions with TYPE * or TYPE *name* clauses.

Effect. The DDL compiler rejects the object.

Recovery. Remove the NOVALUE clause and recompile.

```
***WARNING*** NULL on referencing item ignored, NULL
inherited field-name
```

Cause. A NULL clause describes a field that is defined by reference to an existing definition, and the referenced definition already has a NULL clause with the same null value.

Effect. The DDL compiler issues a warning message, and the null value of the referenced definition is inherited by the referring definition. The DDL compiler does not produce output for inherited attributes.

Recovery. Remove the NULL clause from the DDL schema, or change the null value so that it differs from the referenced definition, and recompile.

```
***ERROR*** NULL value cannot fit in one byte
```

Cause. A NULL clause character string is longer than 1 byte, or a NULL clause number is greater than 255 or less than 0.

Effect. The DDL compiler rejects the object.

Recovery. Shorten the character string or correct the number and recompile.

```
***ERROR*** Number exceeds COBOL max of 18 digits-
element-name
```

Cause. A numeric picture size is greater than 18 digits when COBOL output is requested.

Effect. The DDL compiler does not write the object definition to the COBOL source file.

Recovery. Reduce the numeric picture size and recompile; rebuild the dictionary if the object in error is referenced by another object.

```
***ERROR*** Object element invalid- Dictionary is corrupt
```

Cause. A dictionary was updated incorrectly.

Effect. The DDL compiler rejects the object.

Recovery. Rebuild the dictionary.

```
***ERROR*** Object is used by some other object(s)-
object-name
```

Cause. An object you attempted to create or delete is referenced by another object.

Effect. The DDL compiler does not create or delete the object.

Recovery. Delete the referring object before creating or deleting the specified object. You can use the SHOW USE OF statement to determine which objects use the specified object and the OUTPUT UPDATE statement to perform the deletion.

```
***ERROR*** Object name already exists in dictionary
```

Cause. The object you attempted to create has the same name as an object that is already in the open dictionary.

Effect. The DDL compiler rejects the object.

Recovery. Give the object a different name and recompile.

```
***ERROR*** Object not in dictionary- object-name
```

Cause. An object specified in a statement or command or referenced by another object is not in the open dictionary.

Effect. The DDL compiler does not execute the statement or command, or the DDL compiler rejects the referring object.

Recovery. Define the missing object and reissue the command or statement, or recompile the referring object.

```
***FATAL ERROR*** Object number exceeded maximum value in DDF
```

Cause. The *next-obj* number in DICTDDF exceeds the unsigned 32-bit range. The dictionary is full and no more objects can be added.

Effect. The DDL compiler closes the dictionary and source code files and stops processing.

Recovery. Delete some objects from the dictionary and recompile.

```
***WARNING*** Object type not specified- DEF assumed
```

Cause. An OUTPUT UPDATE or SHOW USE OF statement does not include a keyword to specify an object type; the DDL compiler assumes the object is a definition.

Effect. The DDL compiler attempts to execute the statement for a definition.

Recovery. If the object is not a definition, specify the object type and resubmit the statement.

```
***WARNING*** Object type not supported in FORTRAN-
object-name
```

Cause. You requested FORTRAN output for a constant, token code, token map, or token type.

Effect. The DDL compiler does not generate FORTRAN code for the object.

Recovery. You can generate source code for these objects in C, COBOL, Pascal, TACL, or TAL.

```
***ERROR*** OCCURS DEPENDING element not found
```

Cause. The field you specified in the DEPENDING ON phrase of an OCCURS clause is not defined.

Effect. The DDL compiler rejects the object.

Recovery. Define the field referenced in the DEPENDING ON phrase and recompile.

```
***ERROR*** OCCURS DEPENDING element not integer numeric
```

Cause. The field specified in the DEPENDING ON phrase of an OCCURS clause is not a numeric data type.

Effect. The DDL compiler rejects the object.

Recovery. Correct the error and recompile.

```
***ERROR*** OCCURS DEPENDING ON cannot be within an OCCURS
```

Cause. An OCCURS DEPENDING ON clause is nested within an OCCURS clause or another OCCURS DEPENDING ON clause.

Effect. The DDL compiler rejects the object.

Recovery. Correct the error and recompile.

```
***ERROR*** OCCURS DEPENDING ON found with or within
REDEFINES- name
```

Cause. A definition that contains a REDEFINES clause also contains an OCCURS DEPENDING ON clause when COBOL output is requested.

Effect. The DDL compiler suppresses COBOL output.

Recovery. Remove the REDEFINES or OCCURS DEPENDING ON clause and recompile.

```
***ERROR*** OCCURS DEPENDING ON is not last element or group
```

Cause. A field or group follows a field or group described with OCCURS DEPENDING ON.

Effect. The DDL compiler rejects the object.

Recovery. Reorder the definition so that the field or group described with OCCURS DEPENDING ON is the last field or group in the data structure.

```
***ERROR*** OCCURS on first element
```

Cause. An OCCURS clause is at the definition or record level; OCCURS can be specified only at level number 02 or greater.

Effect. The DDL compiler rejects the object.

Recovery. Correct the error and recompile.

```
***ERROR*** Octal numbers cannot be used with BINARY 64
UNSIGNED data type
```

Cause. There is an octal number in the value clause of a data item that is defined with BINARY 64 UNSIGNED data type. DDL does not allow octal values to be specified in the value clause of a data item defined with BINARY 64 UNSIGNED data item.

Effect. The DDL compiler rejects the object.

Recovery. Correct the number and recompile.

```
***ERROR*** Octal numbers can't contain decimal points
```

Cause. A number in octal format contains a decimal point; for example, %6.5 is not a valid octal number.

Effect. The DDL compiler rejects the object.

Recovery. Correct the number and recompile.

```
***ERROR*** Old dictionary is only partially purged in
subvol- subvolume-name
```

Cause. The security of some dictionary files prevented the DDL compiler from deleting the files from the dictionary.

Effect. The dictionary is not purged.

Recovery. Change the file security and manually purge the remaining dictionary files.

```
***ERROR*** Only one TYPE clause per element allowed
```

Cause. A field definition or description has more than one PICTURE or TYPE clause.

Effect. The DDL compiler rejects the object.

Recovery. Remove all but one PICTURE or TYPE clause and recompile.

```
***ERROR***  PACKED-DECIMAL data type is not supported in
language_name
```

Cause. The DDL compiler was asked to generate output for a source language other than COBOL and the DDL item contains a PACKED-DECIMAL field.

Effect. The DDL compiler issues error messages and does not generate output for the language identified as *language_name*. For example:

```
!?tal
! SCHEMA PRODUCED DATE - TIME : 8/01/2000 - 15:05:22
  Output source for TAL is opened on $ZTN1.#PTPJHYV
!def emp pic 9999 PACKED-DECIMAL.
  Definition EMP size is 3 bytes.
  Definition EMP added to dictionary.
  *** WARNING *** TAL OUTPUT DIAGNOSTICS:
  *** ERROR *** Element contains PACKED-DECIMAL data type - EMP
  *** ERROR *** PACKED-DECIMAL data type is not supported in  TAL
  *** ERROR *** Errors detected - no output produced for EMP
```

Effect. Recovery is not possible. Remove the COMP-3, COMPUTATIONAL-3, or PACKED-DECIMAL data item from the definition or record. Such data items are not supported in C, FORTRAN, pTAL, PASCAL, TAL, or TACL.

```
***ERROR*** Pascal DEF or RECORD or variant record name too
long- object-name
```

Effect. The name of a definition or record exceeds the Pascal limit of 31 ASCII characters for these names.

Effect. The DDL compiler does not write the definition or record to the Pascal source file.

Recovery. Shorten the name and recompile the definition or record.

```
***WARNING*** PASCAL OUTPUT DIAGNOSTICS:
```

Cause. You requested Pascal output, but the object does not conform to Pascal syntax rules.

Effect. A message follows describing the Pascal error. The DDL compiler does not write the object to the Pascal source file.

Recovery. Correct the error and recompile if you want Pascal output.

```
***WARNING*** PATHMAKER subvol check failed, assuming
PATHMAKER subvol- subvolume-name
```

Cause. A file error occurred when the DDL compiler attempted to determine if the current subvolume is a Pathmaker subvolume.

Effect. The DDL compiler assumes that the current subvolume is a Pathmaker subvolume.

Recovery. If the problem persists, consult your system manager.

```
***ERROR*** PICTURE clause contains more than 18 nines -
element_name
```

Cause. The picture clause of the PACKED-DECIMAL data item identified by *element_name* contains more than the maximum of 18 nines.

Effect. The DDL compiler rejects the object. For example:

```
!def emp.
!02 fld1 pic 9(19) comp-3.
                       ^
*** ERROR *** PICTURE clause contains more than 18 nines - FLD1
```

Recovery. Correct the error and recompile.

```
***ERROR*** PICTURE string exceeds COBOL max of 30
characters- object-name
```

Cause. A PICTURE string exceeds 30 ASCII characters when COBOL output is requested.

Effect. The DDL compiler does not write the object to the COBOL source file.

Recovery. Shorten the PICTURE string and recompile; rebuild the dictionary if the object in error is referenced by another object.

```
***ERROR*** Primary key must be unique
```

Cause. A primary-key field is defined with DUPLICATES ALLOWED; primary keys must be unique.

Effect. The DDL compiler rejects the object.

Recovery. Remove DUPLICATES ALLOWED and recompile.

```
***ERROR*** Primary key specified but file is not key-
sequenced- file-name
```

Cause. A primary key is declared in a KEYTAG or KEY IS clause, but the FILE IS clause declares the file as other than key-sequenced.

Effect. The DDL compiler rejects the object.

Recovery. Remove the primary key specification, or specify a different file type, and recompile.

```
***ERROR*** qualified-name cannot be a Level 66 item
```

Cause. A level 66 item is referenced, but a qualified name was expected (for example, the reference was made from another level 66 RENAMES clause or from an OCCURS DEPENDING ON clause).

Effect. The DDL compiler rejects the object.

Recovery. Correct the error and recompile.

```
***ERROR*** RECORD is too big for BLOCK
```

Cause. The record entered is too large for the block size; at least one record and a header must fit in a block.

Effect. The DDL compiler rejects the object.

Recovery. Change block size or record sizes and recompile.

```
***ERROR*** Record locked- Please try again later
```

Cause. The DDL compiler tried to access a dictionary object when the object was locked. An object is locked when another user is updating the object.

Effect. The DDL compiler does not process the object.

Recovery. Wait a few minutes and try again.

```
***ERROR*** Record or definition too large
```

Cause. A definition or record is larger than 32767 bytes.

Effect. The DDL compiler rejects the object.

Recovery. Correct the error and recompile.

```
***ERROR*** Record size too big for file type
```

Cause. The total length of a record is greater than the maximum size allowed for the special file type:

For Format 1 files:

File Type	Maximum Length
Unstructured	4,096 bytes
Entry-Sequenced	4,072 bytes
Relative	4,072 bytes
Key-Sequenced	4,062 bytes

For Format 2 files:

File Type	Maximum Length
Unstructured	4,096 bytes
Entry-sequenced	4,048 bytes
Relative	4,048 bytes
Key-sequenced	4,040 bytes

Effect. The DDL compiler rejects the record.

Recovery. Correct the error and recompile.

```
***ERROR*** Redefined element has OCCURS clause- element-name
```

Cause. A field or group that has an OCCURS clause is redefined by another field when COBOL output is requested.

Effect. The DDL compiler suppresses COBOL output.

Recovery. Correct the error and recompile; rebuild the dictionary if necessary.

```
***ERROR*** Redefined element not immediately preceding
```

Cause. A redefined field or group does not immediately precede the redefining field or group.

Effect. The DDL compiler rejects the object.

Recovery. Move the redefining field or group to follow the field or group it redefines, then recompile.

```
***ERROR*** REDEFINES not allowed on or with bit fields-
element-name
```

Cause. A REDEFINES clause follows the description of a bit field or is with a bit field.

Effect. The DDL compiler rejects the object.

Recovery. Remove the REDEFINES clause or the bit field and recompile the object.

```
***ERROR*** REDEFINES too large
```

Cause. A field or group is larger than the field or group it redefines.

Effect. The DDL compiler rejects the object.

Recovery. Reorder and change the descriptions so that the smaller field or group redefines the larger one and recompile.

```
***ERROR*** REDEFINES too small, unable to pad with FILLER -
element-name
```

Cause. A field or group is smaller than the field or group it redefines. The DDL compiler tries to pad it with filler, but is unable to pad as the filler that is required by the OCCURS count does not produce an integral result. COBOL output is then requested.

Effect. The DDL compiler suppresses COBOL output.

Recovery. Make the redefining field or group the same size as the field or group it redefines and recompile; rebuild the dictionary if necessary.

```
***ERROR*** Reference invalid- dictionary is not open
```

Cause. A reference is made to a record or definition, and the dictionary is not open.

Effect. The DDL compiler does not generate output.

Recovery. Open the dictionary and recompile.

```
***ERROR*** Referenced constant may not be internationalized
```

Cause. A locale name is defined by a reference to a previously defined constant. The referenced constant has locale information associated with it.

Effect. The DDL compiler rejects the text item.

Recovery. Only define a locale by a literal or by a previously defined constant assigned a value without an associated locale.

```
***ERROR*** Referenced Def has incompatible alignment
```

Cause. An attempt was made to compile a record or definition with matched2 alignment, that referenced a definition previously compiled without matched2 alignment.

Effect. The DDL compiler does not add the new record or definition to the dictionary.

Recovery. Recompile the referenced definition with matched2 alignment or recompile the referenced definition without matched2 alignment. Matched2 alignment is specified with the CFIELDALIGN_MATCHED2 command.

```
***ERROR*** Referenced element is not defined
```

Cause. A qualified name in this statement is not declared within the record that qualifies it.

Effect. The DDL compiler rejects the object.

Recovery. Declare the referenced element earlier and recompile.

```
***ERROR*** Referenced object is not type ENUM- object-type
```

Cause. The ENUM clause in the type specification for a bit field refers to a definition that is not of type ENUM.

Effect. The DDL compiler rejects the bit field definition or the group containing the bit field description.

Recovery. Change the ENUM clause to refer to a definition of type ENUM, or omit the ENUM clause, and recompile the object.

```
***ERROR*** Renamed element nested too deeply for TAL
```

Cause. A renamed field is not at the outermost level, and pTAL or TAL output is requested.

Effect. The DDL compiler does not produce pTAL or TAL output.

Recovery. Rebuild the object and recompile.

```
***ERROR*** RENAMES element has OCCURS or is within OCCURS
```

Cause. The starting or ending element of the renamed element is embedded in an OCCURS clause.

Effect. The DDL compiler rejects the object.

Recovery. Correct the error and recompile.

```
***ERROR*** RENAMES elements overlap improperly
```

Cause. In a phrase such as “A RENAMES B THRU C,” either field C starts before field B, or field B ends after field C.

Effect. The DDL compiler rejects the object.

Recovery. Correct the error and recompile.

```
***ERROR*** Reserved word- reserved-word
```

Cause. The specified element name is a reserved word in C, COBOL, pTAL, Pascal, or TAL; and C, COBOL, Pascal, or TAL output is requested.

Effect. The DDL compiler suppresses C, COBOL, pTAL, Pascal, or TAL output for the object.

Recovery. Change the name and recompile; rebuild the dictionary if necessary.

```
***ERROR*** Reserved word for CONSTANT name- constant-name
```

Cause. The indicated constant name is a DDL reserved word.

Effect. The DDL compiler rejects the constant.

Recovery. Change the constant name to a name that is not a DDL reserved word and recompile. For a list of DDL reserved words, see Keywords on page 2-6.

```
***ERROR*** Scale factor too large for data type
```

Cause. The specified number of decimal places exceeds the precision of the specified data type.

Effect. The DDL compiler rejects the object.

Recovery. Correct the error and recompile.

```
***ERROR*** Section name not found in source file-
section-name
```

Cause. A SOURCE command specifies a section name in a source file, but the section name is not in the source file.

Effect. The DDL compiler takes no action.

Recovery. Correct the error and recompile.

```
***ERROR*** SETCOBOL cannot be specified when COBOL output
file is open
```

Cause. You entered a SETCOBOL74 or SETCOBOL85 command when a COBOL source code file is open.

Effect. The DDL compiler takes no action.

Recovery. Close the open COBOL source code file and reenter the command.

```
***WARNING*** Short REDEFINES encountered: FILLER emitted,
results in incompatible source structure - item-name
```

Cause. A redefining field or group is shorter than the field or group it redefines, and COBOL output is requested.

Effect. The DDL compiler generates filler to make the redefining field or group the same size as the redefined field or group.

Recovery. No recovery is necessary.

```
***WARNING*** Spacing must be 0,1,2, or 3- value not changed
```

Cause. A SPACING command specifies a value other than 0, 1, 2, or 3.

Effect. The value of SPACING is unchanged.

Recovery. This error affects only the report listing. No recovery is necessary unless you want a report with a different spacing. Use the OUTPUT statement with a correct SPACING command to produce the report you want.

```
***ERROR*** SPI-NULL conflict in group- group-name
```

Cause. An SPI-NULL clause is associated with a field in a group, but the group has an SPI-NULL clause. Fields within a group inherit the group's SPI-NULL clause.

Effect. The DDL compiler rejects the definition containing the SPI-NULL clause.

Recovery. Either remove the SPI-NULL clause from the field within the group, or remove the SPI-NULL clause from the group definition or description and specify an SPI-NULL clause for the field.

```
***ERROR*** SPI-NULL value cannot fit in one byte
```

Cause. An SPI-NULL clause specifies a value that cannot fit in 1 byte; that is, the number is not within the range 0 through 255.

Effect. The DDL compiler rejects the object.

Recovery. Specify an SPI null value from 0 through 255 and recompile.

```
***ERROR*** SPI-NULL value on a bit field must be 255-
field-name
```

Cause. The value is not 255 in the SPI-NULL clause of the type specification for a bit field.

Effect. The DDL compiler rejects the bit field definition or the group definition containing the bit field description.

Recovery. Change the value in the SPI-NULL clause to 255, or omit the SPI-NULL clause, and recompile the object.

```
***ERROR*** Structure alignment in C is incompatible with
DDL- element-name
```

Cause. A DDL definition cannot be translated to C because word alignment is not maintained. A group data item that does not begin with a word-aligned object follows an item that ends on an odd-byte boundary, or a group data item ends on an odd-byte boundary and is not followed by a word-aligned object. This condition can occur only if the C_MATCH_HISTORIC_TAL command is not in effect and the definition was compiled without matched2 alignment.

Effect. The DDL compiler does not write the definition to the C source file.

Recovery. Change the DDL definition so that all character or FILLER items contain an even number of characters, use the C_MATCH_HISTORIC_TAL command, or recompile the definition with the CFIELDALIGN_MATCHED2 command set.

```
***ERROR*** Structure alignment in Pascal is incompatible
with DDL- element-name
```

Cause. A DDL definition cannot be translated to Pascal because word alignment is not maintained. A named group data item that does not begin with a word-aligned object follows an item that ends on an odd-byte boundary, or a named group data item ends on an odd-byte boundary and is not followed by a word-aligned object.

Effect. The DDL compiler does not write the definition to the Pascal source file.

Recovery. Change the DDL definition so that all character or FILLER items contain an even number of characters. Recompile the definition.

```
***ERROR*** Structure alignment in TAL is incompatible with
DDL- element-name
```

Cause. An odd-length string definition contains an OCCURS clause that cannot be translated to pTAL or TAL. When generating pTAL or TAL source for a string, the DDL compiler usually emits a struct for the string field. Because this particular string is an odd length, the DDL compiler would have to add a filler to word-align the struct.

Effect. The DDL compiler does not write the definition to the pTAL or TAL source file.

Recovery. Change the DDL definition so make the definition even length. Recompile the definition.

```
***FATAL ERROR*** Symbol table is full
```

Cause. A record or definition is too large to fit in the symbol table.

Effect. The DDL compiler closes the dictionary and source code files and stops processing.

Recovery. Reduce the complexity of the data structure by defining the object in multiple DEFINITION or RECORD statements, and recompile the object.

```
***WARNING*** TACL OUTPUT DIAGNOSTICS:
```

Cause. You requested TACL output for an object, but the object does not conform to TACL rules.

Effect. A message follows describing the TACL error. The DDL compiler does not write the object to the TACL source file.

Recovery. Correct the error and recompile.

```
***ERROR*** TAL DEF or TOKEN MAP name too long
```

Cause. The name of a definition or token map exceeds the limit of 31 ASCII characters pTAL or TAL allows for these names.

Effect. The DDL compiler does not write the definition or token map to the pTAL or TAL source file.

Recovery. Shorten the name, recompile the definition or token map, and request output to the pTAL or TAL source file.

```
***ERROR*** TAL name literal too long- commenting out
```

Cause. The key name constructed by the DDL compiler exceeds the maximum allowed length of a pTAL or TAL name. The DDL compiler constructs the name from the names of each element in the name hierarchy and adds ^KEY to the end.

Effect. The DDL compiler changes the keytag literal to a pTAL or TAL comment.

Recovery. Change the names of elements in the record or reduce the number of levels that make up this key.

```
***WARNING*** TAL OUTPUT DIAGNOSTICS:
```

Cause. You requested pTAL or TAL output for an object, but the object does not conform to pTAL or TAL rules.

Effect. A message follows describing the pTAL or TAL error. The DDL compiler does not write the object to the pTAL or TAL source file.

Recovery. Correct the error and recompile.

```
***WARNING*** TALBOUND ot PASCALBOUND must be 0 or 1- value
not changed
```

Cause. A TALBOUND or PASCALBOUND command has a value other than 0 or 1.

Effect. The TALBOUND or PASCALBOUND value does not change.

Recovery. Correct the error and recompile if necessary. TALBOUND or PASCALBOUND affects only pTAL or TAL source output.

```
***FATAL ERROR*** Text ID number exceeded maximum value in
DDF
```

Cause. The *next-text-id* number in DICTDDF exceeded the unsigned 32-bit range. The object text file in the dictionary is full and no more objects can be added.

Effect. The DDL compiler closes the dictionary and source code files and stops processing.

Recovery. Delete objects from the dictionary and recompile.

```
***ERROR*** THIS DICTIONARY CANNOT BE UPDATED
```

Cause. You do not have write access to the dictionary.

Effect. The DDL compiler restricts dictionary access to read-only operations.

Recovery. Consult your system manager to obtain write access to the dictionary.

```
***WARNING*** This dictionary IS audited
```

Cause. You used a DICTN command to open an audited dictionary.

Effect. The DDL compiler opens the specified dictionary anyway.

Recovery. No recovery is necessary.

```
***WARNING*** This dictionary is NOT audited
```

Cause. You used a DICT or DICTR command to open a nonaudited dictionary.

Effect. The DDL compiler opens the specified dictionary anyway.

Recovery. No recovery is necessary.

```
***ERROR*** TOKEN-CODE VALUE multiply defined
```

Cause. The VALUE IS clause in a TOKEN-CODE statement is already specified.

Effect. The DDL compiler rejects the token code.

Recovery. Remove one of the VALUE IS clauses and recompile.

```
***ERROR*** TOKEN-CODE VALUE not defined
```

Cause. No VALUE clause is specified in a TOKEN-CODE statement.

Effect. The DDL compiler rejects the token code.

Recovery. Specify a VALUE clause.

```
***ERROR*** TOKEN-CODE TOKEN-TYPE not specified
```

Cause. No TOKEN-TYPE clause is specified in a TOKEN-CODE statement.

Effect. The DDL compiler rejects the token code.

Recovery. Specify a TOKEN-TYPE clause.

```
***ERROR*** TOKEN-MAP DEFINITION multiply defined
```

Cause. The DEF IS clause in a TOKEN-MAP statement is already specified.

Effect. The DDL compiler rejects the token map.

Recovery. Remove one of the DEF IS clauses and recompile.

```
***ERROR*** TOKEN-MAP DEFINITION not specified
```

Cause. No DEF clause is specified in a TOKEN-MAP statement.

Effect. The DDL compiler rejects the token map.

Recovery. Specify a DEFINITION in the open dictionary with a DEF clause.

```
***ERROR*** TOKEN-MAP exceeds C 32767-byte limit-
token-map-name
```

Cause. A TOKEN-MAP statement generates a C structure that is greater than 32,767 bytes. The entire C structure, not just individual fields, must be less than or equal to 32,767 bytes.

Effect. The DDL compiler does not generate C output for the specified token map.

Recovery. Shorten the definition referenced in the TOKEN-MAP statement and regenerate the C source code.

```
***ERROR*** TOKEN-MAP exceeds Pascal 32766-byte limit-
token-map-name
```

Cause. A TOKEN-MAP statement generates a Pascal structure that is greater than 32,766 bytes. The entire Pascal structure, not just individual fields, must be less than or equal to 32,766 bytes.

Effect. The DDL compiler does not generate Pascal output for the specified token map.

Recovery. Shorten the definition referenced in the TOKEN-MAP statement and regenerate the Pascal source code.

```
***ERROR*** TOKEN-MAP exceeds TACL 5000-byte limit-
token-map-name
```

Cause. A TOKEN-MAP statement generates a TACL structure that is greater than 5,000 bytes. The entire TACL structure, not just individual fields, must be less than or equal to 5,000 bytes.

Effect. The DDL compiler does not generate TACL output for the specified token map.

Recovery. Shorten the definition referenced in the TOKEN-MAP statement and regenerate the TACL source code.

```
***ERROR*** TOKEN-MAP VALUE multiply defined
```

Cause. The VALUE clause in a TOKEN-MAP statement is already specified.

Effect. The DDL compiler rejects the token map.

Recovery. Remove one of the VALUE IS clauses and recompile.

```
***ERROR*** TOKEN-MAP VALUE not defined
```

Cause. No VALUE clause is specified in a TOKEN-MAP statement.

Effect. The DDL compiler rejects the token map.

Recovery. Specify a VALUE clause.

```
***ERROR*** TOKEN-MAP VERSION not specified for
Line.LineItem.LocalName
```

Cause. VERSION is not specified for the field inside the DEFINITION used in the TOKEN-MAP statement.

Effect. The DDL compiler rejects the token map.

Recovery. Specify a VERSION for the field.

```
***ERROR*** TOKEN-TYPE can occur 1 to 254 times
```

Cause. The OCCURS specification inside the DEF clause in a TOKEN-TYPE statement contains a number outside the acceptable range.

Effect. The DDL compiler rejects the token type.

Recovery. Specify a correct number of occurrences.

```
***ERROR*** TOKEN-TYPE DEFINITION exceeds 254 bytes
```

Cause. The definition referenced in a TOKEN-TYPE statement is longer than 254 bytes. The total length of the definition is derived from the sum of the length of individual fields in the definition, optionally repeated by an OCCURS *n* TIMES clause.

Effect. The DDL compiler rejects the token type.

Recovery. Shorten the referenced definition, make it a variable-length type, or use the TOKEN-MAP statement instead; then recompile.

```
***ERROR*** TOKEN-TYPE DEFINITION Length * OCCURS exceeds 254
bytes
```

Cause. The length of the definition referenced in a TOKEN-TYPE statement multiplied by the OCCURS value in the statement is longer than 254 bytes.

Effect. The DDL compiler rejects the token type.

Recovery. Shorten the referenced definition, make it a variable-length type, make the OCCURS value smaller, or use the TOKEN-MAP statement instead; then recompile.

```
***ERROR*** TOKEN-TYPE DEFINITION multiply defined
```

Cause. The DEF IS clause in a TOKEN-TYPE statement is already specified.

Effect. The DDL compiler rejects the token type.

Recovery. Remove one of the DEF IS clauses and recompile.

```
***ERROR*** TOKEN-TYPE multiply defined
```

Cause. A token type is already specified in the TOKEN-CODE statement.

Effect. The DDL compiler rejects the token code.

Recovery. Delete one of the token type specifications and recompile.

```
***ERROR*** TOKEN-TYPE not found
```

Cause. The token type specified in the TOKEN-CODE statement is not in the open dictionary.

Effect. The DDL compiler rejects the token code.

Recovery. Use the name of an existing token type and recompile.

```
***ERROR*** TOKEN-TYPE OCCURS multiply defined
```

Cause. An OCCURS clause is already specified in the TOKEN-TYPE statement.

Effect. The DDL compiler rejects the token type.

Recovery. Delete one of the OCCURS clauses and recompile.

```
***ERROR*** TOKEN-TYPE VALUE multiply defined
```

Cause. A VALUE clause is already specified in the TOKEN-TYPE statement.

Effect. The DDL compiler rejects the token type.

Recovery. Delete one of the VALUE clauses and recompile.

```
***ERROR*** TOKEN-TYPE VALUE not defined
```

Cause. No VALUE clause is specified in the TOKEN-TYPE statement

Effect. The DDL compiler rejects the token type.

Recovery. Specify a VALUE clause.

```
***ERROR*** TOKEN-TYPE VALUE must be between 0 and 255
```

Cause. The VALUE clause in a TOKEN-TYPE statement contains or represents a token data type number that is outside the acceptable range.

Effect. The DDL compiler rejects the token type.

Recovery. Specify a correct number in the VALUE clause.

```
***FATAL ERROR*** Too many elements for symbol table
```

Cause. A definition or record contains more than 2,000 elements, the symbol table maximum limit.

Effect. The DDL compiler rejects the object, closes the dictionary, and stops processing.

Recovery. Shorten the definition or record and recompile.

```
***FATAL ERROR*** Too many errors- complilation terminating
```

Cause. The number of errors specified in the ERRORS command has been reached.

Effect. The DDL compiler closes the dictionary and source code files and stops processing.

Recovery. Correct the errors and recompile.

```
***ERROR*** Too many names- statement has been ignored
```

Cause. An OUTPUT, OUTPUT UPDATE, DELETE, or SHOW USE OF statement has too many definitions or records.

Effect. The DDL compiler ignores the statement.

Recovery. Break the list of definitions or records into two or more statements.

```
***ERROR*** Too many text (common) lines
```

Cause. There are more than 65,777 lines of comment text in a single record or definition or in an element of a record or definition.

Effect. The DDL compiler enters no more comment lines in the dictionary.

Recovery. Reduce the number of comment lines to 65,777 or fewer.

```
***ERROR*** Too many values in MUST BE or VALUES clause
```

Cause. A MUST BE or level 88 clause has too many values.

Effect. The DDL compiler rejects the object.

Recovery. Split the level 88 clause into two level 88 clauses with shorter values lists. If possible, use ranges instead of listing values; for example, replace “1,2,3,4,5” with “1 through 5.”

```
***ERROR*** Too many warnings- compilation terminating
```

Cause. The number of warnings specified in the WARNINGS command has been reached.

Effect. The DDL compiler closes the dictionary and source code files and stops processing.

Recovery. Correct the errors and recompile.

```
***ERROR*** Unable to delete dictionary in subvol-
subvolume-name
```

Cause. Most likely, there is a security violation on the dictionary files that you are trying to delete.

Effect. The dictionary you were trying to delete still exists.

Recovery. Determine the status of the files and proceed accordingly.

```
***ERROR*** Unable to match locale- locale-name
```

Cause. A text item did not have a literal with a locale name the same as specified either with the SETLOCALENAME command, if set, or with the system default locale, if the SETLOCALENAME was not set.

Effect. The programming source language statement will not be emitted.

Recovery. Either use SETLOCALENAME to set locale or change a locale name associated with the text item so there is a match between the locale and a locale name.

```
***FATAL ERROR*** Unable to relinquish lock
```

Cause. The DDL compiler encountered an error while trying to unlock a locked record or file.

Effect. The DDL compiler closes the dictionary and source code files and stops processing.

Recovery. Retry operation; if problem persists, consult your system manager.

```
***ERROR*** Underscore not valid in Identifier - <identifier
name>
```

Cause. COBOL, FORTRAN, Pascal, FUP or NCL output requested a definition, record, constant or token statement which contains an underscore as a part of its identifier.

Effect. The DDL compiler issues this message and does not generate output. For example:

```
!?DICT
 Dictionary opened on subvol $ADE101.ALPHA for update access.
!?COBOL
*SCHEMA PRODUCED DATE - TIME : 7/21/2000 - 19:42:49
Output source for COBOL is opened on $ZTN1.#PTPJHU8
!DEF EMPLOYEE
!02 EMP_NAME PIC X(20).
!END
 Definition EMPLOYEE size is 20 bytes.
 Definition EMPLOYEE added to dictionary.
***WARNING*** COBOL 85 OUTPUT DIAGNOSTICS:
***ERROR*** Underscore not valid in Identifier - EMP_NAME
***WARNING*** Errors detected - no output produced for EMPLOYEE.
```

Recovery. Replace underscore with a valid character.

```
***ERROR*** Unexpected DDL exception
```

Cause. The DDL compiler encountered an unexpected error.

Effect. The DDL compiler issues this message followed by a fatal error message.

Recovery. If the problem persists, consult your system manager.

```
***ERROR*** Union Alignment in C is incompatible with DDL-
element
```

Cause. The size of the union field is not a multiple of the alignment of the widest field in the union. This occurs if the redefines variable in the DDL definition is an elementary item and the size of the variable is not a multiple of the alignment of the redefining variables

Effect. The DDL compiler does not write the definition to the C source file.

Recovery. Change the size of the redefines variable so that it is a multiple of the alignment of the redefining variables, or add on level of indirection in the definition and make the variable a group item.

```
***ERROR*** Unrecognized data type in element- definition
```

Cause. The DDL compiler has attempted to access the indicated definition in an existing dictionary, but does not recognize the type associated with that definition. The most likely cause is that the product version of the DDL compiler is older than that of the dictionary, which contains a data type not supported by the older DDL compiler.

Effect. The DDL compiler rejects the definition.

Recovery. Use a product version of the DDL compiler that is recent enough to recognize all data types in the dictionary.

```
***ERROR*** Unrecognized data type in structure- definition
```

Cause. The DDL compiler has attempted to access the indicated definition in an existing dictionary, while trying to output C or Pascal code for a definition that references the indicated definition, but does not recognize the type associated with the dictionary definition. The most likely cause is that the product version of the DDL compiler is older than that of the dictionary, which contains a data type not supported by the older DDL compiler.

Effect. The DDL compiler rejects the definition.

Recovery. Use a product version of the DDL compiler that is recent enough to recognize all data types in the dictionary.

```
***ERROR*** Unsigned integer conversion eror- object-name
```

Cause. The value of an unsigned integer has an invalid form: either a negative number or a decimal or octal value.

Effect. The DDL compiler rejects the object.

Recovery. Correct the error and recompile.

```
***ERROR*** Unspecified value
```

Cause. You specified a VALUE clause that does not contain a value.

Effect. The DDL compiler rejects the object.

Recovery. Correct the VALUE clause and recompile the object.

```
***WARNING*** UNSUPPORTED data type in element
```

Cause. The data type of the element is not supported in the requested source language. For valid data types in the requested source language, see Appendix C, DDL Data Translation.

Effect. The DDL compiler still generates output for the requested source language to a data type with the same size.

Recovery. Change the data type of the element and recompile.

```
***WARNING*** Unsupported data type in structure
```

Cause. The data type of an element in the referenced group definition is not supported in the requested source language. For valid data types in the requested source language, see Appendix C, DDL Data Translation.

Effect. The DDL compiler still generates output for the requested source language to a data type with the same size.

Recovery. Change the data type of the element in the referenced group definition and recompile.

```
***WARNING*** Unsupported data type in word starting at
element- element-name
```

Cause. The BIT data type is not supported in the requested source language. Only C, pTAL, TAL, and Pascal support the BIT data type.

Effect. The DDL compiler groups the bit fields that reside in the same word and generates the output to the integer data type.

Recovery. Change the data type and recompile.

```
***ERROR*** Unterminated SOURCE command on last line not
processed
```

Cause. A SOURCE command does not stop with a file name or a closing parenthesis following a list of section names, or a comma is missing after a section name.

Effect. The DDL compiler ignores the SOURCE command.

Recovery. Correct the error and recompile if necessary.

```
***ERROR*** Unterminated string
```

Cause. A single or double quotation mark does not have a corresponding closing quotation mark on the same input line.

Effect. The DDL compiler rejects the object; syntax errors can result.

Recovery. Correct the error and recompile.

```
***ERROR*** Update conflict, retry your operation
```

Cause. You are attempting to update a record that another user is updating.

Effect. The DDL compiler does not complete the update.

Recovery. Retry the operation.

```
***ERROR*** UPSHIFT and initial VALUE conflict- value
```

Cause. Both a VALUE and a MUST BE clause describe the same field, but the initial value is not upshifted or cannot be upshifted.

Effect. The DDL compiler rejects the object.

Recovery. Correct the error and recompile.

```
***ERROR*** UPSHIFT and MUST BE conflict- value
```

Cause. Both an UPSHIFT and a MUST BE clause describe the same field, but the MUST BE range is not upshifted or cannot be upshifted.

Effect. The DDL compiler rejects the object.

Recovery. Correct the error and recompile.

```
***WARNING*** UPSHIFT on referencing item ignored, UPSHIFT
inherited field-name
```

Cause. An UPSHIFT clause describes a field that is defined by reference to an existing definition, and the referenced definition already has an UPSHIFT clause. UPSHIFT cannot be overridden in a referring definition.

Effect. The DDL compiler continues processing.

Recovery. Remove the UPSHIFT clause from the referring definition.

```
***WARNING*** Valid FILLER parameters are 0 and 1. FILLER
unchanged
```

Cause. The FILLER command has a parameter other than 0 or 1.

Effect. The DDL compiler ignores command.

Recovery. Correct the command and reenter it.

```
***ERROR*** Valid TACLGEN parameter is 0, TACLGEN unchanged
```

Cause. A value other than zero was specified for the TACLGEN command.

Effect. The DDL compiler does not execute the command.

Recovery. Set the value of the TACLGEN parameter to zero.

```
***ERROR*** VALUE conflicts with the COBOL data type for item
```

Cause. A VALUE clause specifies a value that is outside the range of values allowed for the COBOL data type generated from the data item.

Effect. The DDL compiler does not generate COBOL output for the data item.

Recovery. Correct the value and recompile.

```
***ERROR*** VALUE contains too many digits for PIC data-pic
```

Cause. A binary item has a larger value than the maximum value of the generated COBOL picture, and COBOL output is requested.

Effect. The DDL compiler rejects the object.

Recovery. Correct the error and recompile.

```
***ERROR*** VALUE invalid or inconsistent with data type-
value
```

Cause. The value specified in a VALUE clause cannot be mapped to the declared data type.

Effect. The DDL compiler rejects the object.

Recovery. Correct the error and recompile.

```
***ERROR*** VALUE item found with or within an OCCURS name
```

Cause. A field definition or description contains both OCCURS and VALUE clauses. A field described with an OCCURS clause cannot have an initial value.

Effect. The DDL compiler rejects the object.

Recovery. Correct the error and recompile.

```
***ERROR*** VALUE item found with or within REDEFINES name
```

Cause. A field definition or description contains both REDEFINES and VALUE clauses. A field described with a REDEFINES clause cannot have an initial value.

Effect. The DDL compiler rejects the object.

Recovery. Correct the error and recompile.

```
***ERROR*** Version FOR name not found
```

Cause. The field or group name following the keyword FOR in a TOKEN-MAP VERSION clause cannot be found in the referenced definition.

Effect. The DDL compiler rejects the token map.

Recovery. Check the referenced definition. Either correct the name in the VERSION clause, or correct the field or group name in the referenced definition; then recompile.

```
***ERROR*** Version FOR name not unique
```

Cause. A field or group name in a definition referenced by a TOKEN-MAP VERSION clause is not unique within the definition.

Effect. The DDL compiler rejects the token map.

Recovery. Check the referenced definition, and qualify the group or field name that follows FOR in the VERSION clause to make the name unique; then recompile.

```
***ERROR*** Version THRU element doesn't follow FOR element
```

Cause. The field or group specified after THRU does not follow the field or group specified after FOR in a TOKEN-MAP VERSION clause.

Effect. The DDL compiler rejects the token map.

Recovery. Check the definition referenced in the TOKEN-MAP statement, and correct the VERSION clause to specify a THRU element that follows a FOR item in the referenced definition; then recompile.

```
***ERROR*** Version THRU name not found
```

Cause. A field or group name following the keyword THRU in a TOKEN-MAP VERSION clause cannot be found in the referenced definition.

Effect. The DDL compiler rejects the token map.

Recovery. Check the referenced definition, and correct the VERSION clause to specify a field or group name in the referenced definition; then recompile.

```
***ERROR*** Version THRU name not unique
```

Cause. A field or group name in a definition referenced by a TOKEN-MAP VERSION clause is not unique within the definition.

Effect. The DDL compiler rejects the token map.

Recovery. Check the referenced definition, and qualify the group or field name that follows THRU in the VERSION clause to make the name unique; then recompile.

```
***ERROR*** WARNINGS parameter is invalid
```

Cause. You specified a value for the WARNINGS command that is not in the range 1 through 32767.

Effect. The DDL compiler ignores the WARNINGS command.

Recovery. Specify a valid value in the WARNINGS command and recompile.

```
***ERROR*** You do not have authority to alter the dictionary
```

Cause. You attempted to update a dictionary to which you do not have write access.

Effect. The DDL compiler rejects the request.

Recovery. Consult your system manager to obtain write access to the dictionary.

B Sample Schemas

Sample Database Schema

The sample database schema defines a database consisting of nine files. The database files are defined in RECORD statements that refer to previous DEFINITION statements for their record structures. Many of the individual and group fields within each record also refer to previous DEFINITION statements. All DISPLAY and HEADING values in DEFINITION statements are defined in previous CONSTANT statements.

When the DDL compiler compiles this schema, the DDL compiler builds a dictionary and generates two source code files:

Source Code File	Description
FUPSRC	A FUP file-creation source file that contains the FUP commands to create each file described in the schema. The database is created only when FUP is executed with this file as input.
COBSRC	A COBOL source file that contains COBOL data descriptions for all the fields, groups, and records in the schema.

Topics:

Host-Language Source Code

A COBOL program that accesses the database can use the COBOL source code in COBSRC as its Data division. C, FORTRAN, Pascal (on D-series systems), pTAL, TACL, and TAL source code can also be generated for this schema. For C and TACL, the DDL compiler issues warning messages when generating the source code. For C, two data items (PRICE and PARTCOST) are described with PICTURE clauses that are not supported by C. For TACL, three data items (INVENTORY, PRICE, and PARTCOST) are described with unsupported PICTURE clauses. The DDL compiler generates compatible C and TACL data types for these items.

For Pascal, pTAL, and TAL, there are data items that contain unsupported data types, but these languages do not issue warning messages. For Pascal, PRICE and PARTCOST are described with PICTURE clauses that are unsupported. For pTAL or TAL, INVENTORY, PRICE, and PARTCOST are described with unsupported PICTURE clauses.

For FORTRAN, the DDL compiler issues warning messages for all constants; constants are not supported by FORTRAN. The constants that describe display, heading, and help values are not used by the source code, so this causes no problems; the generated source code will execute successfully.

Database Schema Listing

Figure B-1. Database Schema Listing (page 1 of 5)

```
!*************************************************************
! COMPILER COMMANDS:
!*************************************************************

? ERRORS 1

? DICT !
? COMMENTS

? FUP FUPSRC !
? COBOL COBSRC85 !

!*************************************************************
! CONSTANT DEFINITIONS
!*************************************************************
CONSTANT custnum-heading      VALUE "Customer/Number"LN en_US.ISO8859-1,
                                    "Cliente/Numero"LN.es_ES.ISO8859-1
                                    "Client/Numero"LN fr_FR.ISO8859-1.

CONSTANT suppnum-heading      VALUE "Supplier/Number"LN en_US.ISO8859-1,
                                    "Proveedor/Numero",LN.es_ES.ISO8859-1
                                    "Fournisseur/Numero"LN fr_FR.ISO8859-1.

CONSTANT partnum-heading      VALUE "Part/Number"LN en_US.ISO8859-1,
                                    "Repuesto/Numero"LN.es_ES.ISO8859-1,
                                    "Piece/Numero"LN fr_FR.ISO8859-1.

CONSTANT ordernum-heading     VALUE "Order/Number"LN en_US.ISO8859-1,
                                    "Orden/Numero",LN.es_ES.ISO8859-1
                                    "Commande/Numero"LN fr_FR.ISO8859-1.

CONSTANT empnum-heading       VALUE "Employee/Number"LN en_US.ISO8859-1
                                    "Empleado/Numero"LN.es_ES.ISO8859-1,
                                    "Employe/Numero"LN fr_FR.ISO8859-1.

CONSTANT regnum-heading       VALUE "Region/Number"LN en_US.ISO8859-1,
                                    "Region/Numero"LN.es_ES.ISO8859-1,
                                    "Region/Numero"LN fr_FR.ISO8859-1.

CONSTANT branchnum-heading    VALUE "Branch/Number"LN en_US.ISO8859-1,
                                    "Sucursal/Numero"LN.es_ES.ISO8859-1,
                                    "Bureau/Numero"LN fr_FR.ISO8859-1.

CONSTANT manager-heading      VALUE "Manager"LN en_US.ISO8859-1,
                                    "Gerente",LN.es_ES.ISO8859-1
                                    "Chef De Service"LN fr_FR.ISO8859-1.

CONSTANT salesperson-heading  VALUE "Salesperson"LN en_US.ISO8859-1,
                                    "Vendedor"LN.es_ES.ISO8859-1,
                                    "Vendeur"LN fr_FR.ISO8859-1.

CONSTANT mdy-date-display     VALUE "M<99/99/99>".

CONSTANT part-cost-display    VALUE "M<ZZZ,ZZ9,99>".
```

Figure B-1. Database Schema Listing (page 2 of 5)

```
!****************************************************************
! FIELD DEFINITIONS
!****************************************************************
DEFINITION custnum              PIC 9(4)
                                  HEADING custnum-heading.

DEFINITION suppnum              PIC 9(4)
                                  HEADING suppnum-heading.

DEFINITION partnum              PIC 9(4)
                                  HEADING partnum-heading.

DEFINITION ordernum             PIC 9(4)
                                  HEADING ordernum-heading.

DEFINITION empnum               PIC 9(4)
                                  HEADING empnum-heading.

DEFINITION regnum               PIC 9(2)
                                  HEADING regnum-heading
                                  MUST BE 1 THRU 99.

DEFINITION branchnum            PIC 9(2)
                                  HEADING branchnum-heading
                                  MUST BE 1 THRU 99.

DEFINITION zip-cd               PIC 9(5).

!****************************************************************
! GROUP DEFINITIONS
!****************************************************************
DEFINITION name.
  02 last-name                  PIC X(12)
                                  UPSHIFT.
  02 first-name                 PIC X(8)
                                  UPSHIFT.
  02 midinit                    PIC X(2)
                                  UPSHIFT.
END

DEFINITION addr.
  02 address                    PIC X(22).
  02 city                       PIC X(14).
  02 state                      PIC X(2)
                                  HELP "Enter 2-character code".
  02 zip                        TYPE zip-cd.
END

DEFINITION mdy-date.            DISPLAY mdy-date-display
                                  HELP "Enter date as:"
                                  "mm/dd/yy".
  02 month                      PIC 99
                                  MUST BE 1 THRU 12.
  02 day-of-month               PIC 99
                                  MUST BE 1 THRU 31.
  02 year                       PIC 99.
END

!****************************************************************
! RECORD DEFINITIONS
!****************************************************************
* Definition for CUSTOMER Record
DEFINITION custinfo.
  02 custnum                    TYPE *.
  02 custname                   TYPE name.
  02 addr                       TYPE *.
END
```

Figure B-1. Database Schema Listing (page 3 of 5)

```
* Definition for ORDERS Record
DEFINITION orderinfo.
  02 ordernum                 TYPE *.
  02 orderdate                TYPE mdy-date.
  02 deldate                  TYPE mdy-date.
  02 salesperson              TYPE empnum
                                HEADING salesperson-heading.
  02 custnum                  TYPE *.
END

* Definition for PARTS Record
DEFINITION partsinfo.
  02 partnum                  TYPE *.
  02 partname                 TYPE name.
  02 inventory                PIC 9(3)S.
  02 location                 PIC X(3).
  02 price                    PIC 9(6)V9(2).
END

* Definition for SUPPLIER Record
DEFINITION suppinfo.
  02 suppnum                  TYPE *.
  02 suppname                 TYPE name.
  02 addr                     TYPE *.
END

* Definition for REGION Record
DEFINITION reginfo.
  02 regnum                   TYPE *.
  02 regname                  PIC X(12).
  02 location                 PIC X(14).
  02 manager                  TYPE empnum
                                HEADING manager-heading.
END

* Definition for BRANCH Record
DEF branchinfo.
  02 primkey.
    03 regnum                 TYPE *.
    03 branchnum              TYPE *.
  02 branchname               PIC X(14).
  02 manager                  TYPE empnum
                              HEADING manager-heading.
END

* Definition for EMPLOYEE Record
DEFINITION empinfo.
  02 empnum                   TYPE *.
  02 empname                  TYPE name.
  02 dept.
    03 regnum                 TYPE *.
    03 branchnum              TYPE *.
  02 job                      PIC X(12).
  02 age                      PIC 9(2).
  02 salary                   PIC 9(6).
  02 vacation                 PIC 9(2).
END
```

Figure B-1. Database Schema Listing (page 4 of 5)

```
!*************************************************************
! FILE DEFINITIONS
!*************************************************************
* Contains customer information for each customer
RECORD customer.
  FILE IS "$data.sales.customer" KEY-SEQUENCED
                                 AUDIT
                                 MAXEXTENTS 100.
  DEFINITION IS custinfo.
  KEY IS customer.custnum DUPLICATES NOT ALLOWED.
  KEY "cn" IS customer.custname.
END

* Contains order information for each order
RECORD orders.
  FILE IS "$data.sales.orders" KEY-SEQUENCED
                               AUDIT
                               MAXEXTENTS 100.
  DEFINITION IS orderinfo.
  KEY IS orders.ordernum DUPLICATES NOT ALLOWED.
  KEY "sn" IS orders.salesperson.
  KEY "cn" IS orders.custnum.
END

* Contains each order line for each order
RECORD odetail.
  FILE IS "$data.sales.odetail" KEY-SEQUENCED
                                AUDIT
                                MAXEXTENTS 100.
  02 primkey.
    03 ordernum                 TYPE *.
    03 partnum                  TYPE *.
  02 quantity                   PIC 9(3).
  KEY IS primkey DUPLICATES NOT ALLOWED.
END

* Contains information on each part
RECORD parts.
  FILE IS "$data.sales.parts" KEY-SEQUENCED
                              AUDIT
                              MAXEXTENTS 100.
  DEFINITION IS partsinfo.
  KEY IS parts.partnum DUPLICATES NOT ALLOWED.
  KEY "pn" IS parts.partname.
END

* Contains a record of each part ordered from each supplier
RECORD fromsup.
  FILE IS "$data.sales.fromsup" KEY-SEQUENCED
                                AUDIT
                                MAXEXTENTS 100.
  02 primkey.
    03 partnum                  TYPE *.
    03 suppnum                  TYPE *.
  02 partcost                   PIC 9(6)V9(2)
                                DISPLAY part-cost-display.
  KEY IS primkey DUPLICATES NOT ALLOWED.
END
```

Figure B-1. Database Schema Listing (page 5 of 5)

```
* Contains information about each supplier of parts
RECORD supplier.
  FILE IS "$data.sales.supplier" KEY-SEQUENCED
                                 AUDIT
                                 MAXEXTENTS 100.
  DEFINITION IS suppinfo.
  KEY IS supplier.suppnum DUPLICATES NOT ALLOWED.
  KEY "su" IS supplier.suppname.
END

* Contains information about company's regional offices
RECORD region.
  FILE IS "$data.sales.region" KEY-SEQUENCED
                               AUDIT
                               MAXEXTENTS 100.
  DEFINITION IS reginfo.
  KEY IS region.regnum DUPLICATES NOT ALLOWED.
  KEY "rn" IS region.regname.
END

* Contains information about company's branch offices
RECORD branch.
  FILE IS "$data.sales.branch" KEY-SEQUENCED
                               AUDIT
                               MAXEXTENTS 100.
  DEFINITION IS branchinfo.
  KEY IS branch.primkey DUPLICATES NOT ALLOWED.
END

* Contains information about each employee
RECORD employee.
  FILE IS "$data.sales.employee" KEY-SEQUENCED
                                 AUDIT
                                 MAXEXTENTS 100.
  DEFINITION IS empinfo.
  KEY IS employee.empnum DUPLICATES NOT ALLOWED.
  KEY "en" IS employee.empname.
  KEY "dp" IS employee.dept.
END

!*************************************************************
! END OF SCHEMA1  DATABASE DESCRIPTION
!*************************************************************
```

Sample SPI Schema

The sample SPI schema contains the DDL commands to build a dictionary containing the token definitions and other information needed by a subsystem that sends and receives SPI messages. If you do not plan to use SPI messages to communicate among processes in a Distributed Systems Management (DSM) environment, you need not refer to this schema.

The sample SPI schema uses standard SPI definitions wherever applicable and nonstandard definitions only where needed by the subsystem. The standard SPI definitions are in the file ZSPIDEF.ZSPIDDL on the disk volume selected for your system. The first step in creating the dictionary is to compile this entire file into your dictionary using the DDL SOURCE command.

You do not generate COBOL, pTAL, TAL, or TACL source code for the standard SPI definitions; HP supplies the COBOL, pTAL, TAL, or TACL source code in these files:

Language	File
COBOL	ZSPIDEF.ZSPICOB
pTAL or TAL	ZSPIDEF.ZSPITAL
TACL	ZSPIDEF.ZSPITACL

For this example, the DDL definitions that are not part of the standard SPI definition file are contained in the file ASSNDDL. When you write a subsystem, the file name of the file containing the subsystem's DDL definitions must have this format:

*ssss*DDL

In the format, *ssss* is a code to identify the subsystem. In Figure B-1 on page B-2, the subsystem code happens to be ASSN.

The file ASSNDDL contains the source code to define four simple tokens and one extensible structured token. It contains all the DDL statements needed to define the token types, token codes, and the token map. It also contains the DDL statements to define the subsystem ID and the SPI message buffer.

When the DDL compiler compiles the source code in file ASSNDDL, it adds the definitions in this file to the dictionary and generates three source code files:

File	Description
ASSNCOB	A COBOL source file that contains COBOL data descriptions for the DDL statements in ASSNDDL.
ASSNTAL	A pTAL or TAL source file that contains pTAL or TAL data definitions for the DDL statements in ASSNDDL.
ASSNTACL	A TACL source file that contains TACL data definitions for the DDL statements in ASSNDDL.

The DDL compiler does not generate FORTRAN source code for SPI objects.

Topics:

DDL Commands to Create an SPI Schema

Example B-1 on page B-8 creates a dictionary from the DDL source file containing standard SPI definitions (ZSPIDEF.ZSPIDDL) and from the file ASSNDDL containing subsystem-specific definitions.

Example B-1. Creating an SPI Schema

```
?ERRORS 1
?DICT !
?COMMENTS
?SOURCE ZSIPDEF.ZSPIDDL                    On current default volume
?SETSECTION                                Assure default DDL sectioning
?COBOL ASSNCOB !                           COBOL source for subsystem
?TAL ASSNTAL !                             TAL source for subsystem
?TACL ASSNTACL !                           TACL source for subsystem
?SOURCE ASSNDDL (ASSN-DEFS)                Definitions specific to subsystem
?SETSECTION CONSTANTS
?SOURCE ASSNDDL (ASSN-TOKEN-INFO)          Token-related definitions
?SETSECTION
?SOURCE ASSNDDL (ASSN-BUFFER)              Buffer structure for subsystem
```

The SETSECTION commands in Example B-1 on page B-8 divide the host-language source files into sections as recommended in the *Distributed Name Service (DNS) Management Programming Manual*.

Selected ZSPIDDL Statements

The DDL statements from the file ZSPIDEF.ZSPIDDL in Example B-2 on page B-9 are either used by the DDL statements in the file ASSNDDL or used in examples in this manual. They are included here for documentation only. Do not copy these statements individually—use the command SOURCE ZSPIDEF.ZSPIDDL to compile the entire set of standard DDL statements into your dictionary as shown in the preceding set of DDL commands.

Note. Certain ZSPIDDL definitions cause the DDL compiler to issue warning messages when it generates host-language source code. For example, the definition of ZSPI-DDL-BYTE causes the DDL compiler to issue a warning when it generates COBOL source code. Because COBOL does not recognize the BINARY 8 data type, the DDL compiler translates this definition to PIC X(1). For data type translations that cause the DDL compiler to issue warnings, see Appendix C, DDL Data Translation.

Example B-2. ZSPIDDL Statements

```
DEF  zspi-ddl-int         TYPE BINARY 16           SPI-NULL 0.
DEF  zspi-ddl-int2        TYPE BINARY 32           SPI-NULL 0.
DEF  zspi-ddl-uint        TYPE BINARY 16 UNSIGNED SPI-NULL 0.
DEF  zspi-ddl-enum        PIC S9(4) COMP           SPI-NULL 255
                                                   TACL enum.

DEF  zspi-ddl-boolean     TYPE zspi-ddl-int        SPI-NULL " ".
DEF  zspi-ddl-byte        TYPE BINARY 8 UNSIGNED   SPI-NULL 0.

DEF  zspi-ddl-char8.
   02 z-c                 PIC X(8)                 SPI-NULL " ".
   02 z-s REDEFINES z-c.
      03  z-i             TYPE BINARY 16           OCCURS 4 TIMES.
   02 z-b REDEFINES z-c   PIC X                    OCCURS 8 TIMES.
END

DEF  zspi-ddl-username    TACL username.
   02 z-groupname         TYPE zspi-ddl-char8.
   02 z-username          TYPE zspi-ddl-char8.
END

CONSTANT zspi-tdt-int             VALUE IS  2.
CONSTANT zspi-tdt-int2            VALUE IS  3.
CONSTANT zspi-tdt-map             VALUE IS  8.
CONSTANT zspi-tdt-boolean         VALUE IS 10.
CONSTANT zspi-tdt-enum            VALUE IS 11.
CONSTANT zspi-tdt-byte            VALUE IS 12.
CONSTANT zspi-tnm-command         VALUE IS -510.
CONSTANT zspi-tnm-retcode         VALUE IS    0.

TOKEN-TYPE  zspi-typ-enum         VALUE IS zspi-tdt-enum
                                  DEF IS zspi-ddl-enum.

TOKEN-TYPE  zspi-typ-map          VALUE IS zspi-tdt-map
                                  OCCURS VARYING.

TOKEN-CODE zspi-tkn-command       VALUE IS zspi-tnm-command
                                  TOKEN-TYPE IS zspi-typ-enum.

TOKEN-CODE zspi-tkn-retcode       VALUE IS zspi-tnm-retcode
                                  TOKEN-TYPE IS zspi-typ-enum.
```

ASSNDDL Statements

The DDL statements in Figure B-2 on page B-10 are in the sample DDL file ASSNDDL. They are the statements needed by the sample subsystem in addition to those provided by ZSPIDEF.ZSPIDDL.

Figure B-2. Sample DDL File ASSNDDL

```
? SECTION assn-defs

DEF assn-variable-token.
  02  table-size          TYPE zspi-ddl-int.
  02  data-table          TYPE zspi-ddl-int2  OCCURS 100 TIMES.
END

DEF assn-ddl-jobinfo.
  02  jnumber             TYPE zspi-ddl-int.
  02  priority            TYPE zspi-ddl-int.
  02  location            TYPE zspi-ddl-char8 SPI-NULL "X".
  02  jobclass-is-present TYPE zspi-ddl-boolean.
  02  jobclass            TYPE zspi-ddl-int.
  02  jobusername         TYPE zspi-ddl-username.
END

?SECTION assn-token-info

! Constants to define token numbers:
CONSTANT assn-tnm-my-status       VALUE IS 101.
CONSTANT assn-tnm-stat-reply      VALUE IS 102.
CONSTANT assn-tnm-jobinfo         VALUE IS 3.

! Constants for subsystem-ID:
CONSTANT assn-val-yourco          VALUE IS "YOUR-CO ".
CONSTANT assn-ssn-assn            VALUE IS 1.
CONSTANT assn-val-version         VALUE IS VERSION "D30".

! Constant for buffer length:
CONSTANT assn-val-buflen          VALUE IS 600.

! Token-type definitions:
TOKEN-TYPE assn-typ-variable-token VALUE IS zspi-tdt-int2
                                    OCCURS VARYING
                                    DEF IS assn-variable-token.

TOKEN-TYPE assn-typ-status        VALUE IS zspi-tdt-enum
                                  DEF   IS zspi-ddl-enum.

! Token-code definitions:
TOKEN-CODE assn-tkn-my-status     VALUE IS assn-tnm-my-status
                                  TOKEN-TYPE IS assn-typ-status.

TOKEN-CODE assn-tkn-stat-reply  VALUE IS assn-tnm-stat-reply
                                TOKEN-TYPE IS assn-typ-status.
```

Figure B-2. Sample DDL File ASSNDDL

```
! Token-map definition:
TOKEN-MAP assn-map-jobinfo      VALUE IS assn-tnm-jobinfo
                                DEF IS assn-ddl-jobinfo.
          VERSION "D00"         FOR jnumber THRU location.
          VERSION "D30"         FOR jobclass-is-present.
          NOVERSION             FOR jobclass.
          VERSION "D30"         FOR jobusername.
END

! Subsystem-ID definition:
DEF assn-val-ssid               TACL ssid.
  02 z-filler                   TYPE CHARACTER 8
                                VALUE IS assn-val-yourco.
  02 z-owner                    TYPE zspi-ddl-char8
                                REDEFINES z-filler.
  02 z-number                   TYPE zspi-ddl-int
                                VALUE IS assn-ssn-assn.
  02 z-version                  TYPE zspi-ddl-uint
                                VALUE IS assn-val-version.
END

? SECTION assn-buffer

! Buffer definition
DEF assn-ddl-msg-buffer.
 02 z-msgcode                  TYPE zspi-ddl-int.
 02 z-buflen                   TYPE zspi-ddl-uint.
 02 z-occurs                   TYPE zspi-ddl-uint.
 02 z-filler                   TYPE zspi-ddl-byte
                               OCCURS 0 TO assn-val-buflen TIMES
                               DEPENDING ON z-occurs.
END
```

C DDL Data Translation

This appendix explains how data defined in DDL is translated to each of the seven supported host languages.

The DDL compiler can translate any definition or record to data-declaration source code for host languages [C, COBOL, FORTRAN, Pascal (on D-series systems), TACL, TAL, and pTAL]. The only restriction on translation is that not all data types are supported in all languages, as indicated by the following:

- Whenever a declared data type is not supported in a particular language, the DDL compiler attempts to translate the data type to a declaration with a compatible data type. For example, DDL structures described with PICTURE X or PICTURE 9 clauses are translated to CHARACTER data type in FORTRAN or STRING BYTE data type in pTAL or TAL; a structure described as PICTURE S9(4) COMP is translated to an INT data type in pTAL, TAL, or TACL, or a NATIVE-2 data type in COBOL; a DDL TYPE BINARY 64 data type is translated to a `long long` data type in C or an INT64 data type in Pascal.
- When no compatible data type is available, the DDL compiler translates the data type to a character-string declaration. For example, a structure described as TYPE FLOAT, which is the REAL data type used by FORTRAN, pTAL or TAL, is translated to a PICTURE X(4) data type in COBOL.

These tables summarize how the DDL compiler translates its definitions to each host language:

Note. For information about how DDL translates SQL data types, see the *SQL/MP Reference Manual* and the *SQL/MX Reference Manual*.

Table C-1. Sample DDL/C Data Translation Table (page 1 of 3)

DDL Clause Type	DDL Clause Specification	C Data Type
PICTURE	PIC A(10)	`char [10]`
	PIC 9(10)	`char [10]`
	PIC X(10)	`char [10]`

* Field definition does not have bit length generated.

** H06.03 and later RVUs

Table C-1. Sample DDL/C Data Translation Table (page 2 of 3)

DDL Clause Type	DDL Clause Specification	C Data Type
PICTURE	PIC A(2)X(10)9(2)A(5)	`char [19]`
	PIC SV9(3)	`char [4]`
	PIC 9V9(2)	`char [4]`
	PIC T9V9	`char [3]`
	PIC 9(2)T	`char [3]`
	PIC N(10)	`char [20]`
	PIC 9(4) COMP	`unsigned short`
	PIC S9(4) COMP	`short`
	PIC 9(5) COMP	`unsigned long`
	PIC S9(5) COMP	`long`
	PIC 9(10) COMP	`unsigned long long**`
	PIC S9(10) COMP	`long long`
	PIC 9999V99 OCCURS 52 TIMES	`char [52][6]`
	PIC 9 OCCURS 0 TO 10 TIMES DEPENDING ON *item*	`char [10]`
TYPE	TYPE CHARACTER *len*	`char [`*len* `]`
	TYPE BINARY 8	`signed char`
	TYPE BINARY 8 UNSIGNED	`char`
	TYPE BINARY [16]	`short`
	TYPE BINARY [16] UNSIGNED	`unsigned short`
	TYPE BINARY 16,2	`short`
	TYPE BINARY 32	`long`
	TYPE BINARY 32 UNSIGNED	`unsigned long`
	TYPE BINARY 64	`long long`
	TYPE BINARY 64,-16	`long long`
	TYPE BINARY 64 UNSIGNED	`unsigned long long**`

* Field definition does not have bit length generated.

** H06.03 and later RVUs

Table C-1. Sample DDL/C Data Translation Table (page 3 of 3)

DDL Clause Type	DDL Clause Specification	C Data Type
TYPE	TYPE BIT *len* *	`short` or `unsigned short` *len*
	TYPE BIT *len* UNSIGNED	`unsigned short:`*len*
	TYPE FLOAT [32]	`float`
	TYPE FLOAT 64	`double`
	TYPE COMPLEX	`double` (inaccurate representation)
	TYPE LOGICAL 1	`char`
	TYPE LOGICAL[2]	`short`
	TYPE LOGICAL 4	`long`
	TYPE ENUM	`enum`
	TYPE CHARACTER 8 OCCURS 100 TIMES	`char[100][8]`
	TYPE BINARY 16 OCCURS 3 TIMES	`short[3]`

* Field definition does not have bit length generated.

** H06.03 and later RVUs

Table C-2. Sample DDL/COBOL Data Translation Table (page 1 of 2)

DDL Clause Type	DDL Clause Specification	COBOL Data Type
PICTURE	PIC A(10)	PIC A(10)
	PIC 9(10)	PIC 9(10)
	PIC X(10)	PIC X(10)
	PIC N(10)	PIC N(10)
	PIC A(2)X(10)9(3)	PIC A(2)X(10)9(3)
	PIC SV9(3)	PIC SV9(3) SIGN LEADING SEPARATE
	PIC 9V9(2)S	PIC S9V9(2) SIGN TRAILING SEPARATE
	PIC T9V9	PIC S99V9 SIGN LEADING
	PIC 99T	PIC S9(2)9 SIGN TRAILING
	PIC 9(4) COMP	PIC 9(4) COMP
	PIC S9(4) COMP	PIC S9(4) COMP
	PIC 9(5) COMP	PIC 9(5) COMP
	PIC S9(5) COMP	PIC S9(5) COMP
	PIC S9(10) COMP	PIC S9(10) COMP

* Groups bit fields that can fit in the same word and generates a filler in the word's type.

Table C-2. Sample DDL/COBOL Data Translation Table (page 2 of 2)

DDL Clause Type	DDL Clause Specification	COBOL Data Type
	PIC 9999V99 OCCURS 52 TIMES	PIC 9999V99 Occurs 52 TIMES
	PIC 9 OCCURS 0 TO 10 TIMES DEPENDING ON *item*	PIC 9 OCCURS 0 TO 10 TIMES DEPENDING ON *item*
TYPE	TYPE CHARACTER 8	PIC X(8)
	TYPE BINARY 8	PIC X(1) FILLER PIC X(1) (added by DDL)
	TYPE BINARY [16]	NATIVE-2
	TYPE BINARY [16] UNSIGNED	NATIVE-2
	TYPE BINARY 16,2	PIC S9(2)V9(2) COMP
	TYPE BINARY 32	NATIVE-4
	TYPE BINARY 32 UNSIGNED	NATIVE-4
	TYPE BINARY 64	NATIVE-8
	TYPE BINARY 64,-16	PIC S9(2)P(16) COMP
	TYPE BIT *len* *	NATIVE-2
	TYPE FLOAT [32]	PIC X(4)
TYPE	TYPE FLOAT 64	PIC X(8)
	TYPE COMPLEX	PIC X(8)
	TYPE LOGICAL [2]	PIC X(2)
	TYPE LOGICAL 4	PIC X(4)
	TYPE ENUM	NATIVE-2
	TYPE CHARACTER 8 OCCURS 100 TIMES	PIC X(8) Occurs 100 TIMES
	TYPE BINARY 16 OCCURS 3 TIMES	NATIVE-2 Occurs 3 TIMES

* Groups bit fields that can fit in the same word and generates a filler in the word's type.

Table C-3. Sample DDL/FORTRAN Data Translation Table (page 1 of 2)

DDL Clause Type	DDL Clause Specification	FORTRAN Data Type
PICTURE	PIC A(10)	CHARACTER*10
	PIC 9(10)	CHARACTER*10
	PIC X(10)	CHARACTER*10
	PIC N(10)	CHARACTER*20
	PIC A(2)X(10)9(3)	CHARACTER*15
	PIC SV9(3)	CHARACTER*4
	PIC 9V9(2)S	CHARACTER*4
	PIC T9V9	CHARACTER*3
	PIC 99T	CHARACTER*3
	PIC 9(4) COMP	INTEGER*2
	PIC S9(4) COMP	INTEGER*2
	PIC 9(5) COMP	INTEGER*4
	PIC S9(5) COMP	INTEGER*4
	PIC S9(10) COMP	INTEGER*8
	PIC 9(4)V9(2) OCCURS 52 TIMES	CHARACTER*6 (1:52)
	PIC 9 OCCURS 0 TO 10 TIMES DEPENDING ON *item*	CHARACTER*1 (1:10)
TYPE	TYPE CHARACTER 8	CHARACTER*8
	TYPE BINARY 8	CHARACTER*1 FILLER*1 (added by DDL)
	TYPE BINARY [16]	INTEGER*2
	TYPE BINARY [16] UNSIGNED	INTEGER*2
	TYPE BINARY 16,2	INTEGER*2
	TYPE BINARY 32	INTEGER*4
	TYPE BINARY 32 UNSIGNED	INTEGER*4
	TYPE BINARY 64	INTEGER*8
	TYPE BIT *len*[1]	FILLER*2
	TYPE FLOAT [32]	REAL
	TYPE FLOAT 64	DOUBLE PRECISION
	TYPE COMPLEX	COMPLEX

1. Groups bit fields that can fit in the same word and generates a filler in the word's type.

2. Only one of the possible forms of the clause; see DICTOBL (Object Build List) on page D-15, for the byte lengths of all forms of the clause.

Table C-3. Sample DDL/FORTRAN Data Translation Table (page 2 of 2)

DDL Clause Type	DDL Clause Specification	FORTRAN Data Type
	TYPE LOGICAL [2]	LOGICAL
	TYPE LOGICAL 4	LOGICAL*4
	TYPE ENUM	INTEGER*2
	TYPE CHARACTER 8 OCCURS 100 TIMES	CHARACTER*8 (1:100)
	TYPE BINARY 16 OCCURS 3 TIMES	INTEGER*2

1. Groups bit fields that can fit in the same word and generates a filler in the word's type.

2. Only one of the possible forms of the clause; see DICTOBL (Object Build List) on page D-15, for the byte lengths of all forms of the clause.

Table C-4. Sample DDL/Pascal Data Translation Table (page 1 of 2)

DDL Clause Type	DDL Clause Specification	Pascal Data Type
PICTURE	PIC A(10)	CHARACTER*10
	PIC 9(10)	CHARACTER*10
	PIC X(10)	CHARACTER*10
	PIC N(10)	CHARACTER*20
	PIC A(2)X(10)9(3)	CHARACTER*15
	PIC SV9(3)	CHARACTER*4
	PIC 9V9(2)S	CHARACTER*4
	PIC T9V9	CHARACTER*3
	PIC 99T	CHARACTER*3
	PIC 9(4) COMP	INTEGER*2
	PIC S9(4) COMP	INTEGER*2
	PIC 9(5) COMP	INTEGER*4
	PIC S9(5) COMP	INTEGER*4
	PIC S9(10) COMP	INTEGER*8
	PIC 9(4)V9(2) OCCURS 52 TIMES	CHARACTER*6 (1:52)
	PIC 9 OCCURS 0 TO 10 TIMES DEPENDING ON *item*	CHARACTER*1 (1:10)
TYPE	TYPE CHARACTER *len*	FSTRING(*len*)
	TYPE BINARY 8	BYTE
	TYPE BINARY [16]	INT16
	TYPE BINARY [16] UNSIGNED	CARDINAL
	TYPE BINARY 16,2	INT16
	TYPE BINARY 32	INT32
	TYPE BINARY 32 UNSIGNED	INT32
	TYPE BINARY 64	INT64
	TYPE BINARY 64,-16	INT64
	TYPE BIT *len*	INT(*len*)
	TYPE BIT *len* UNSIGNED	UNSIGNED(*len*)
	TYPE FLOAT [32]	REAL
	TYPE FLOAT 64	LONGREAL
	TYPE COMPLEX	INT64

* Only one of the possible forms of the clause; see DICTOBL (Object Build List) on page D-15, for the byte lengths of all forms of the clause..

Table C-4. Sample DDL/Pascal Data Translation Table (page 2 of 2)

DDL Clause Type	DDL Clause Specification	Pascal Data Type
TYPE	TYPE LOGICAL 1	BOOLEAN
	TYPE LOGICAL[2]	INT16
	TYPE LOGICAL 4	INT32
	TYPE ENUM	INT16
	TYPE CHARACTER 8 OCCURS 100 TIMES	Array[1..100] of FSTRING(8)
	TYPE BINARY 16 OCCURS 3 TIMES	Array[1..3] of INT16

* Only one of the possible forms of the clause; see DICTOBL (Object Build List) on page D-15, for the byte lengths of all forms of the clause..

Table C-5. Sample DDL/TACL Data Translation Table (page 1 of 2)

DDL Clause Type	DDL Clause Specification	TACL Data Type
PICTURE	PIC A(10)	CHAR BYTE(0:9)
	PIC 9(10)	CHAR BYTE(0:9)
	PIC X(10)	CHAR BYTE(0:9)
	PIC N(10)	CHAR BYTE(0:19)
	PIC A(2)X(10)9(2)A(5)	CHAR BYTE(0:18)
	PIC SV9(3)	CHAR BYTE(0:3)
	PIC 9V9(2)S	CHAR BYTE(0:3)
	PIC T9V9	CHAR BYTE(0:2)
	PIC 9(2)T	CHAR BYTE(0:2)
	PIC 9(4) COMP	UINT
	PIC S9(4) COMP	INT
	PIC 9(5) COMP	INT2
	PIC S9(5) COMP	INT2
	PIC S9(10) COMP	INT4
	PIC 9(4)V9(2) OCCURS 52 TIMES	STRUCT (0:51) CHAR BYTE(0:5)
	PIC 9 OCCURS 0 TO 10 TIMES DEPENDING ON *item*	CHAR (0:9)

TYPE	TYPE CHARACTER *len*	BEGIN
		CHAR BYTE(0:*len*)
		END;

	TYPE BINARY 8	BYTE
	TYPE BINARY [16]	INT
	TYPE BINARY [16] UNSIGNED	UINT
	TYPE BINARY 16,2	INT
	TYPE BINARY 32	INT2
	TYPE BINARY 32 UNSIGNED	INT2
	TYPE BINARY 64	INT4
	TYPE BINARY 64,-16	INT4
	TYPE BIT *len* *	FILLER 2
	TYPE FLOAT [32]	INT2
	TYPE FLOAT 64	INT4
	TYPE COMPLEX	INT4

* Groups bit fields that can fit in the same word and generates a filler in the word's type.

Table C-5. Sample DDL/TACL Data Translation Table (page 2 of 2)

DDL Clause Type	DDL Clause Specification	TACL Data Type
TYPE	TYPE LOGICAL 1	BYTE
	TYPE LOGICAL[2]	BOOL
	TYPE LOGICAL 4	INT2
	TYPE ENUM	ENUM
	TYPE CHARACTER 8 OCCURS 100 TIMES	STRUCT(0:99); BEGIN CHAR BYTE(0:9); END;
	TYPE BINARY 16 OCCURS 3 TIMES	INT(0:2);

* Groups bit fields that can fit in the same word and generates a filler in the word's type.

Table C-6. Sample DDL/pTAL and TAL Data Translation Table (page 1 of 2)

DDL Clause Type	DDL Clause Specification	pTAL or TAL Data Type
PICTURE	PIC A(10)	STRUCT BEGIN STRING BYTE [1:10]; END;
	PIC 9(10)	STRUCT BEGIN STRING BYTE [1:10]; END;
	PIC X(10)	STRUCT BEGIN STRING BYTE [1:10]; END;
	PIC N(10)	STRUCT BEGIN STRING BYTE [1:20]; END;
	PIC A(2)X(10)9(3)	STRUCT BEGIN STRING BYTE [1:15]; END;
	PIC SV9(3)	STRUCT BEGIN STRING BYTE [1:4]; END;
	PIC 9V9(2)S	STRUCT BEGIN STRING BYTE [1:4]; END;
	PIC T9V9	STRUCT BEGIN STRING BYTE [1:3]; END;
	PIC 9(2)T	STRUCT BEGIN STRING BYTE [1:3]; END;
	PIC 9(4) COMP	INT
	PIC S9(4) COMP	INT
	PIC 9(5) COMP	INT(32)
	PIC S9(5) COMP	INT(32)
	PIC 9(10) COMP	FIXED
	PIC S9(10) COMP	FIXED
	PIC 9(4)V9(2) OCCURS 52 TIMES	STRUCT [1:52]; BEGIN STRING BYTE [1:6]; END;

* Field definition generates INT.

** Only one of the possible forms of the clause; see DICTOBL (Object Build List) on page D-15, for the byte lengths of all forms of the clause.

Table C-6. Sample DDL/pTAL and TAL Data Translation Table (page 2 of 2)

DDL Clause Type	DDL Clause Specification	pTAL or TAL Data Type
PICTURE	PIC 9 OCCURS 0 TO 10 TIMES DEPENDING ON *item*	STRUCT (*) STRING [1:10]
TYPE	TYPE CHARACTER 8	STRUCT BEGIN STRING BYTE [1:8]; END;
	TYPE BINARY 8	STRING FILLER 1; (added by DDL)
	TYPE BINARY [16]	INT
	TYPE BINARY [16] UNSIGNED	INT
	TYPE BINARY 16,2	INT
	TYPE BINARY 32	INT(32)
	TYPE BINARY 32 UNSIGNED	INT(32)
	TYPE BINARY 64	FIXED
	TYPE BINARY 64,-16	FIXED(-16)
	TYPE BINARY 64 UNSIGNED	FIXED
	TYPE BIT *len* *	UNSIGNED(*len*)
	TYPE FLOAT [32]	REAL
	TYPE FLOAT 64	REAL(64)
	TYPE COMPLEX	FIXED
	TYPE LOGICAL 1	STRING
	TYPE LOGICAL [2]	INT
	TYPE LOGICAL 4	INT(32)
	TYPE ENUM	INT
	TYPE CHARACTER 8 OCCURS 100 TIMES	STRUCT [1:100]; BEGIN STRING BYTE [1:8]; END;
	TYPE BINARY 16 OCCURS 3 TIMES	INT [1:3];

* Field definition generates INT.

** Only one of the possible forms of the clause; see DICTOBL (Object Build List) on page D-15, for the byte lengths of all forms of the clause.

D Dictionary Database Structure

A dictionary is itself a DDL database consisting of 14 files. The DDL compiler supplies the names of the dictionary database files; these names must not be changed.

Topics in this appendix:

Note. Information in this appendix is not guaranteed to remain the same or to change in compatible ways from RVU to RVU.

Dictionary Components

A dictionary has three basic components:

Objects

An object can be:

- A single element:
  - Constant
  - Single-field definition
  - SPI token type
  - SPI token code
- A group of elements:
  - Record
  - Group definition
  - SPI token map

The DDL compiler can add these objects to and delete them from a dictionary, as well as perform other operations on them.

If a dictionary is part of a Pathmaker application catalog, the Pathmaker product can store additional objects in the dictionary. The Pathmaker product manages four types of objects that it can store in a dictionary:

- Servers
- Services
- Requesters
- Screens

The DDL compiler assigns each object a unique object number for identification. Object numbers are assigned in ascending order and are never reused. When an object is removed from the dictionary, all references to its object number are also removed.

Elements

Definitions and records can contain one or more elements. For example, a single-field definition contains a single element; a record or group definition contains an element for itself and additional elements for each field description within the record or group definition.

The DDL compiler assigns each element a unique element number for identification.

Example D-1. Objects

```
DEF partnum  PIC 999.              ! Object with one element

RECORD parts.                      ! Object with five elements
  02 partnum    TYPE *.            ! Element 1
  02 partname   PIC X(18).         ! Element 2
  02 inventory  PIC 999.           ! Element 3
  02 location   PIC XXX.           ! Element 4
  02 price      PIC 999999V99.     ! Element 5
END
```

Text Items

Text items can contain any text associated with an object. A text item can be one of the following five types:

Type	Description
Number	ASCII representation of a numeric literal in a VALUE or MUST BE clause
String	Alphanumeric string in a COMMENT, DISPLAY, PICTURE, VALUE, HEADING, HELP, or MUST BE clause
Keyword	Keyword in a VALUE or MUST BE clause
Enumeration	Name of a value in a level 89 enumeration clause
National	National string in a VALUE or MUST BE clause
International	Internationalized text items in an AS, HEADING, 88, or VALUE clause.

The dictionary stores all of the text items associated with each statement in a text file. Each text item is uniquely identified by a text ID number. A single text item can consist of a list of several lines of text; the list is ordered by element number.

Table D-1 on page D-3 shows the text items described in the following three objects:

```
CONSTANT custnum-heading  VALUE "Customer".

DEF initials    PIC XXX   VALUE ALL SPACES
                          HELP "Initials".

DEF quantity    PIC 999   MUST BE 1 THRU 999.
```

In Table D-1 on page D-3, each text item is assigned a unique text ID number. A single text item can contain more than one text type.

Table D-1. Text IDs Assigned to Text Items

Text ID	Element Number	Text Item	Type
1	0	“Customer”	S (String)
2	0	“XXX”	S (String)
3	0	“ALL”	K (Keyword)
3	1	“SPACES”	K (Keyword)
4	0	“Initials”	S (String)
5	0	“999”	S (String)
6	0	“1”	N (Number)
6	1	“THRU”	K (Keyword)
6	2	“999”	N (Number)

Dictionary Files

When the DDL compiler compiles a schema in a dictionary, it builds these 14 dictionary files:

- DICTALT (Alternate Key File) on page D-4
- DICTCDF (Constant Definition File) on page D-4
- DICTDDF (Dictionary Definition File) on page D-6
- DICTKDF (Key Definition File) on page D-8
- DICTMAP (Token Map File) on page D-13
- DICTOBL (Object Build List) on page D-15
- DICTODF (Object Definition File) on page D-37
- DICTOTF (Object Text File) on page D-41
- DICTOUF (Object Usage File) on page D-45

- DICTOUK (Object Usage Key File) on page D-47
- DICTRDF (Record Definition File) on page D-47
- DICTTKN (Token Code File) on page D-56
- DICTTYP (Token Type File) on page D-58
- DICTVER (Token Map Field Version File) on page D-61

All dictionary files are key-sequenced except DICTDDF, which is unstructured.

The dictionary is itself a database. HP supplies the DDL schema for the dictionary database in the following file:

```
$SYSTEM.SYSTEM.DDSCHEMA
```

In the following topics, the record definitions for the dictionary database files are fully expanded to show the field descriptions. The field names and structures are the same as those used in DDSCHEMA for the data dictionary. Some field descriptions are expanded from referenced definitions.

DICTALT (Alternate Key File)

DICTALT (Alternate Key File) contains keys for:

- DICTKDF (Key Definition File) on page D-8
- DICTOBL (Object Build List) on page D-15
- DICTODF (Object Definition File) on page D-37
- DICTRDF (Record Definition File) on page D-47

DICTCDF (Constant Definition File)

DICTCDF (Constant Definition File) is a key-sequenced file that contains one CDF record for each constant in the dictionary. The CDF record links the constant with the constant text in the DICTOTF (Object Text File).

DICTCDF is different on G-series and H-series systems—see:

- Figure D-1, DICTCDF (Constant Definition File)—G-Series, on page D-5
- Figure D-2, DICTCDF (Constant Definition File)—H-Series, on page D-5

Change bars in Figure D-2 on page D-5 show where it differs from Figure D-1 on page D-5.

Table D-2, DICTCDF (Constant Definition File) Fields, on page D-6, applies to both G-series and H-series systems.

Figure D-1. DICTCDF (Constant Definition File)—G-Series

```
Record CDF.
File is "DICTCDF"                       Key-sequenced
                                        Code 207
                                        Audit.

  02 OBJECT-NUMBER                      Pic "9(9)" COMP
                                        Null 0
                                        Display "[BZ]I10"
                                        Heading "Object/Number".

  02 TEXT-ID                            Pic "9(9)" COMP
                                        Null 0
                                        Display "[BZ]I10"
                                        Heading "Text Id/Number".

  02 CONSTANT-TYPE-STRING               Pic "XX".
    88 CONSTANT-STRING                  Value is "ST".
    88 CONSTANT-VERSION                 Value is "VR".
    88 CONSTANT-BINARY-16               Value is "2S".
    88 CONSTANT-BINARY-16-UNSIGNED      Value is "2U".
    88 CONSTANT-BINARY-32               Value is "4S".
    88 CONSTANT-BINARY-32-UNSIGNED      Value is "4U".
    88 CONSTANT-BINARY-64               Value is "8S".
    88 CONSTANT-NATIONAL-STRING         Value is "NS".

  02 CONSTANT-TYPE Redefines CONSTANT-TYPE-STRING Type Binary 16.

  02 CONSTANT-TYPE-EXPLICIT             Type Character 1
                                        MUST BE "Y", "N"
                                        UPSHIFT
                                        Display "A1"
                                        Heading "Type/Explicitly Given".

  Key is OBJECT-NUMBER Duplicates not allowed.

End
```

Figure D-2. DICTCDF (Constant Definition File)—H-Series (page 1 of 2)

```
Record CDF.
File is "DICTCDF"                       Key-sequenced
                                        Code 207
                                        Audit
                                        MaxExtents 500.

  02 OBJECT-NUMBER                      Type *
                                        Heading "Constant/Object #".

  02 TEXT-ID                            Type *
                                        Heading "Constant/Text-Id".

  02 CONSTANT-TYPE-STRING               Pic "XX".
    88 CONSTANT-STRING                  Value is "ST".
    88 CONSTANT-VERSION                 Value is "VR".
    88 CONSTANT-BINARY-16               Value is "2S".
    88 CONSTANT-BINARY-16-UNSIGNED      Value is "2U".
    88 CONSTANT-BINARY-32               Value is "4S".
    88 CONSTANT-BINARY-32-UNSIGNED      Value is "4U".
    88 CONSTANT-BINARY-64               Value is "8S".
    88 CONSTANT-NATIONAL-STRING         Value is "NS".

  02 CONSTANT-TYPE Redefines CONSTANT-TYPE-STRING Type Binary 16.
```

Figure D-2. DICTCDF (Constant Definition File)—H-Series (page 2 of 2)

```
  02 CONSTANT-TYPE-EXPLICIT          Type ASCII-SWITCH
                                     Heading "Type/Explicitly Given".

  Key is OBJECT-NUMBER Duplicates not allowed.

End
```

Table D-2. DICTCDF (Constant Definition File) Fields

Field	Description
OBJECT-NUMBER	The object number of this record from DICTODF.OBJECT. The record in DICTODF contains the constant name and its object-type code, “CD.”
TEXT-ID	The text ID assigned to the constant; it is used to link the constant record to the record for this constant in DICTOTF.
CONSTANT-TYPE-STRING	A two-character ASCII code that identifies the type of the constant.
CONSTANT-TYPE	A numeric code identifying the type of the constant.
CONSTANT-TYPE-EXPLICIT	Contains an ASCII character Y (yes) to indicate the constant type was entered explicitly or N (no) to indicate the constant type was inherited from another constant or by default.

DICTDDF (Dictionary Definition File)

DICTDDF (Dictionary Definition File) is an unstructured file that contains one DDF record with the next object number to be assigned, the next text ID number to be assigned, the DDL compiler product version information, and the creator’s user ID. The DDF record is updated every time the DDL compiler adds a new object to the dictionary. DICTDDF cannot be an audited file.

DICTDDF is different on G-series and H-series systems—see:

- Figure D-3, DICTDDF (Dictionary Definition File)—G-Series, on page D-7
- Figure D-4, DICTDDF (Dictionary Definition File)—H-Series, on page D-7

Change bars in Figure D-4 on page D-7 show where it differs from Figure D-3 on page D-7.

Table D-3, DICTDDF (Dictionary Definition File) Fields, on page D-8, applies to both G-series and H-series systems.

Figure D-3. DICTDDF (Dictionary Definition File)—G-Series

```
Record DDF.
  File is "DICTDDF"                  Unstructured
                                     Code 200
                                     Ext 2.

  02 NEXT-OBJ                        Pic "9(9)" COMP
                                     Null 0
                                     Display "[BZ]I10"
                                     Heading "Next/Object".

  02 NEXT-TEXT-ID                    Pic "9(9)" COMP
                                     Null 0
                                     Display "[BZ]I10"
                                     Heading "Next/Text Id".

  02 VERSION                         Type Binary 16 Unsigned
                                     MUST BE 6
                                     VALUE 6
                                     Heading "Dict/Version".

  02 CREATOR-USERID                  Heading "Creator User Id"
                                     TACL USERNAME.

     03 GROUP-NAME                   Type Character 8
                                     UPSHIFT
                                     Heading "Group".

     03 USER-NAME                    Type Character 8
                                     UPSHIFT
                                     Heading "User".

  02 NEXT-QUAL-ID                    Pic "9(9)" COMP
                                     Null 0
                                     Display "[BZ]I10"
                                     Heading "Next/Qual Id".

End
```

Figure D-4. DICTDDF (Dictionary Definition File)—H-Series

```
Record DDF.
  File is "DICTDDF"                  Unstructured
                                     Code 200
                                     Ext 2.

  02 NEXT-OBJ                        Type OBJECT-NUMBER
                                     Heading "Next/Object".

  02 NEXT-TEXT-ID                    Type TEXT-ID
                                     Heading "Next/Text Id".

  02 VERSION                         Type Binary 16 Unsigned
                                     MUST BE 8
                                     VALUE 8
                                     Heading "Dict/Version".

  02 CREATOR-USERID                  Type USERID-NAME
                                     Heading "Creator User Id".

  02 NEXT-QUAL-ID                    Type QUALIFIER-ID
                                     Heading "Next/Qual Id".

End
```

Table D-3. DICTDDF (Dictionary Definition File) Fields

Field	Description
NEXT-OBJ	Object number that the DDL compiler assigns to the next record or definition added to the dictionary, or that the Pathmaker product assigns to the next service, server, requester, or screen added to the dictionary.
NEXT-TEXT-ID	The text ID number that the DDL compiler or the Pathmaker product assigns to the next text item stored in DICTOTF (refer to DICTOTF fields for a description of the types of text items stored in the dictionary).
VERSION	A product version number that is incremented every time the internal structure of the dictionary changes. The product version number encoded in the DDL compiler is checked against this field whenever a dictionary is opened.
CREATOR-USERID	A group field that describes the user ID of the person who created this dictionary. CREATOR-USERID consists of the next two fields, GROUP-NAME and USER-NAME.
GROUP-NAME	The name of the group to which the user belongs.
USER-NAME	A name identifying the user within the group.
NEXT-QUAL-ID	A field that the Pathmaker product uses to obtain field qualifier IDs.

DICTKDF (Key Definition File)

DICTKDF (Key Definition File) is a key-sequenced file that contains one KDF record for each key assignment defined in the schema; that is, one record for each alternate and primary key (structured files) or one record for each SEQUENCE IS field. Each KDF record describes the key and provides a link back to the element in DICTOBL (Object Build List) that defines the key field.

DICTKDF is different on G-series and H-series systems—see:

- Figure D-5, DICTKDF (Key Definition File)—G-Series, on page D-9
- Figure D-6, DICTKDF (Key Definition File)—H-Series, on page D-10

Change bars in Figure D-6 on page D-10 show where it differs from Figure D-5 on page D-9.

These tables apply to both G-series and H-series systems:

- Table D-4, DICTKDF (Key Definition File) Fields, on page D-11
- Table D-5, KEY-CLASS Codes, on page D-12

Figure D-5. DICTKDF (Key Definition File)—G-Series (page 1 of 2)

```
Record KDF.
File is "DICTKDF"                       Key-sequenced
                                        Code 206
                                        Audit.

  02 IDENTIFIER.

     03 RECORD-NUMBER                    Pic "9(9)" COMP
                                        Null 0
                                        Display "[BZ]I10"
                                        Heading "Record/Number".
     03 ELEMENT                          Type Binary 16
                                        Display "I3"
                                        Heading "Key/Num".

  02 OBL-KEY.

     03 OBJECT                           Pic "9(9)" COMP
                                        Null 0
                                        Display "[BZ]I10"
                                        Heading "Object/Number".

     03 ELEMENT                          Type Binary 16
                                        Display "I3"
                                        Heading "Element/Number".

  02 KEYTAG-VALUE                       Type Binary 16
                                        VALUE 0
                                        Display "I5"
                                        Heading "Keytag/Value".

  02 KEYTAG-STRING Redefines KEYTAG-VALUE Type Character 2
                                        Display "A2"
                                        Heading "Keytag/Value".

  02 KEYTAG-OBJECT                      Type OBJECT-NUMBER
                                        VALUE 0
                                        Null 0
                                        Display "I5"
                                        Heading "Keytag/Object".

  02 FIELD.

     03 OFFSET                          Type Binary 16
                                        Display "I4"
                                        Heading "Offset".

     03 ELEMENT-SIZE                    Type Binary 16
                                        Display "I4"
                                        Heading "Size".

  02 NULL-VALUE                         Type Binary 16
                                        Display "I5"
                                        Heading "Null/Value".

  02 NULL-VALUE-SPECIFIED               Type ASCII-SWITCH
                                        VALUE "N"
                                        Heading "Null/Specified".

  02 KEY-CLASS                          Pic "XXX"
                                        VALUE "PRI"
                                        Heading "Key/Class".
```

Figure D-5. DICTKDF (Key Definition File)—G-Series (page 2 of 2)

```
  02 KEY-UNIQUE                        Type Character 1
                                       MUST BE "Y", "N"
                                       UPSHIFT
                                       Display "A1"
                                       VALUE "Y"
                                       Heading "Key/Uniq".

  02 KEY-UPDATE                        Type Character 1
                                       MUST BE "Y", "N"
                                       UPSHIFT
                                       Display "A1"
                                       VALUE "Y"
                                       Heading "Key/Update".

  02 KEY-FILE-NAME                     Type Character 34
                                       UPSHIFT
                                       Heading"Physical File Name".

  02 FILLER                            Type Character 30.

  Key is IDENTIFIER Duplicates not allowed.
  Key "OK" is OBL-KEY.

End
```

Figure D-6. DICTKDF (Key Definition File)—H-Series (page 1 of 2)

```
Record KDF.
File is "DICTKDF"                      Key-sequenced
                                       Code 206
                                       Audit
                                       MaxExtents 500.

  02 IDENTIFIER.

     03 RECORD-NUMBER                   Type OBJECT-NUMBER
                                       Heading "Record/Number".

     03 ELEMENT                         Type Binary 16
                                       Display "I3"
                                       Heading "Key/Num".

  02 OBL-KEY                           Type FIELD.

  02 KEYTAG-VALUE                      Type Binary 16
                                       VALUE 0
                                       Display "I5"
                                       Heading "Keytag/Value".

  02 KEYTAG-STRING Redefines KEYTAG-VALUE Type Character 2
                                       Display "A2"
                                       Heading "Keytag/Value".

  02 KEYTAG-OBJECT                     Type OBJECT-NUMBER
                                       VALUE 0
                                       Null 0
                                       Display "I5"
                                       Heading "Keytag/Object".
```

Figure D-6. DICTKDF (Key Definition File)—H-Series (page 2 of 2)

```
  02 FIELD.
     03 OFFSET                        Type Binary 16
                                      Display "I4"
                                      Heading "Offset".

     03 ELEMENT-SIZE                  Type Binary 16
                                      Display "I4"
                                      Heading "Size".

  02 NULL-VALUE                       Type *.

  02 NULL-VALUE-SPECIFIED             Type ASCII-SWITCH
                                      VALUE "N"
                                      Heading "Null/Specified".

  02 KEY-CLASS                        Pic "XXX"
                                      VALUE "PRI"
                                      Heading "Key/Class".

  02 KEY-UNIQUE                       Type ASCII-SWITCH
                                      VALUE "Y"
                                      Heading "Key/Uniq".

  02 KEY-UPDATE                       Type ASCII-SWITCH
                                      VALUE "Y"
                                      Heading "Key/Update".

  02 KEY-FILE-NAME                    Type FILE-NAME.

  02 FILLER                           Type Character 30.

  Key is IDENTIFIER Duplicates not allowed.
  Key "OK" is OBL-KEY File is "DICTALT".

End
```

Table D-4. DICTKDF (Key Definition File) Fields (page 1 of 2)

Field	Description
IDENTIFIER	The primary key of the KDF record, consisting of the next two fields, RECORD-NUMBER and ELEMENT.
RECORD-NUMBER	The object number of the record that has this DICTKDF element as a key; the same as OBJECT in DICTRDF.
ELEMENT	A sequentially assigned number to guarantee that IDENTIFIER is unique, starting with 0.
OBL-KEY	The primary key of the OBL record that describes this key field, consisting of the next two fields, OBJECT and ELEMENT.
OBJECT	The object number of the record containing this key field; the same as OBJECT in DICTOBL.
ELEMENT	The element number of this key field; the same as ELEMENT in DICTOBL.
KEYTAG-VALUE	The Enscribe key specifier of this key; a one-word integer representing the primary or alternate key number.

Table D-4. DICTKDF (Key Definition File) Fields (page 2 of 2)

Field	Description
KEYTAG-STRING	A 2-character string used when the Enscribe key specifier is declared as two ASCII characters. KEYTAG-STRING redefines KEYTAG-VALUE as a 2-byte string.
KEYTAG-OBJECT	If the keytag value is defined by a constant, contains the object number of the constant.
FIELD	A group containing the OFFSET and ELEMENT-SIZE values from the OBL record that describes this key field.
OFFSET	The offset of this key field within the record that contains it. This field is copied from OFFSET in DICTOBL; it is duplicated here for efficient access.
ELEMENT-SIZE	The size in bytes of the key field. This field is copied from SIZE in DICTOBL; it is duplicated here for efficient access.
NULL-VALUE	A value that indicates whether the field has been initialized. If NULL-VALUE contains the null value specified by the user, then the field has not been initialized. This field is currently used by the Enform Plus product when producing reports and by FUP when producing a FUP file-creation source file. An alternate key field filled with null values is not added to an alternate key file.
NULL-VALUE-SPECIFIED	Contains the ASCII character Y (yes) or N (no) to indicate whether the user specified a null value for this item. N is the default.
KEY-CLASS	Indicates the type of key this record defines. Codes are in Table D-5 on page D-12. PRI is the default.
KEY-UNIQUE	Contains the ASCII character Y (yes) or N (no) to indicate whether the key that defines this record is unique. Y is the default.
KEY-UPDATE	Contains the ASCII character Y (yes) or N (no) to indicate if the key might be updated. Y is the default.
KEY-FILE-NAME	Contains the actual Guardian file name to be used for the key. The name is stored in external form and might be a network name; for example, \NEWYORK.$MARKET.DATAFILE.FILE1.

Table D-5. KEY-CLASS Codes

Code	Meaning
PRI	Primary key (default)
ALT	Alternate Key
DSF	Descending sort order
ASF	Ascending sort order

DICTMAP (Token Map File)

DICTMAP (Token Map File) is a key-sequenced file that contains one record for each SPI token map. Each record contains detailed information about a token map, including its unique token number, and the object number of the definition that describes the extensible structured token associated with the token map. Additional information about token maps is contained in the Version File (DICTVER).

DICTMAP is different on G-series and H-series systems—see:

- Figure D-7, DICTMAP (Token Map File)—G-Series, on page D-13
- Figure D-8, DICTMAP (Token Map File)—H-Series, on page D-14

Change bars in Figure D-8 on page D-14 show where it differs from Figure D-7 on page D-13.

Table D-6, DICTMAP (Token Map File) Fields, on page D-14, applies to both G-series and H-series systems.

Figure D-7. DICTMAP (Token Map File)—G-Series

```
Record MAP.
  File is "DICTMAP"                    Key-sequenced
                                       Code 209
                                       Audit.

  02 OBJECT-NUMBER                     Pic "9(9)" COMP
                                       Null 0
                                       Display "[BZ]I10"
                                       Heading "Token Map/Object".

  02 TOKEN-NUMBER-VALUE                Type Binary 16
                                       Heading "Token Numb".

  02 TOKEN-NUMBER-CONSTANT             Pic "9(9)" COMP
                                       Null 0
                                       Display "[BZ]I10"
                                       Heading "Token Numb/Object".

  02 MAP-DEF                           Pic "9(9)" COMP
                                       Null 0
                                       Display "[BZ]I10"
                                       Heading "Map Def/Object".

  02 SSID-TEXT                         Pic "9(9)" COMP
                                       Null 0
                                       Display "[BZ]I10"
                                       Heading "SSID".

  02 HEADING-TEXT                      Pic "9(9)" COMP
                                       Null 0
                                       Display "[BZ]I10"
                                       Heading "Heading".

  Key is OBJECT-NUMBER Duplicates not allowed.

End
```

Figure D-8. DICTMAP (Token Map File)—H-Series

```
Record MAP.
  File is "DICTMAP"                    Key-sequenced
                                       Code 209
                                       Audit
                                       MaxExtents 500.

  02 OBJECT-NUMBER                     Type *
                                       Heading "Token Map/Object".

  02 TOKEN-NUMBER-VALUE                Type Binary 16
                                       Heading "Token Numb".

  02 TOKEN-NUMBER-CONSTANT             Type OBJECT-NUMBER
                                       Heading "Token Numb/Object".

  02 MAP-DEF                           Type OBJECT-NUMBER
                                       Heading "Map Def/Object".

  02 SSID-TEXT                         Type TEXT-ID
                                       Heading "SSID".

  02 HEADING-TEXT                      Type TEXT-ID
                                       Heading "Heading".

  Key is OBJECT-NUMBER Duplicates not allowed.

End
```

Table D-6. DICTMAP (Token Map File) Fields

Field	Description
OBJECT-NUMBER	Contains the object number of this token map from DICTODF.OBJECT. The record in DICTODF contains the token-map name and its object type, “TM.”
TOKEN-NUMBER-VALUE	Contains the token number of the token map taken from the VALUE clause of the TOKEN-MAP statement that defines the token map. Token numbers can be in the range -32,768 through 32,767. Any user-supplied token numbers must be in the range 1 through 9,998; the other token numbers are reserved by HP or are previously defined by SPI.
TOKEN-NUMBER-CONSTANT	Contains the object-number of the constant used to define the token number; if the token number was not specified using a constant, this field is set to 0.
MAP-DEF	Contains the object number of the definition (DEF) for the token map. The definition defines the data structure of the extensible structured token described by the token map.
SSID-TEXT	Contains the text ID of the OTF record that contains the subsystem ID value for the token map.
HEADING-TEXT	Contains the text ID of the OTF record that contains the heading value for the token map.

DICTOBL (Object Build List)

DICTOBL (Object Build List) is a key-sequenced file that contains one record for each element of each unique object in the dictionary. The primary key of the file is the OBJECT field from DICTODF plus a sequentially assigned element number.

An object can contain one or more elements.

Example D-2. Object With Multiple Elements

```
DEF example.                ! Element 0
    02 field-1 PIC X.       ! Element 1
    02 group-2              ! Element 2
       03 field-3 PIC X.    ! Element 1
END
```

If an object or an element within an object is defined by a TYPE * or TYPE *def-name* clause, elements of the referenced object are copied to each DICTOBL field for the referring object. The top-level SOURCE-DEF field in this file contains the object number of the referenced definition.

If a record's structure is defined by a DEFINITION IS *def-name* clause, DICTOBL has no entry for the referenced object. Instead, linkage is made through DICTRDF (Record Definition File). DICTRDF.DEF-NUMBER contains the object number of the referenced definition. For all other records, DICTRDF.DEF-NUMBER contains the object number of the record itself.

DICTOBL is different on G-series and H-series systems—see:

- Figure D-9, DICTOBL (Object Build List)—G-Series, on page D-16
- Figure D-10, DICTOBL (Object Build List)—H-Series, on page D-21

Change bars in Figure D-10 on page D-21 show where it differs from Figure D-9 on page D-16.

These tables apply to both G-series and H-series systems:

- Table D-7, DICTOBL (Object Build List) Fields, on page D-25
- Table D-8, VALUE-TEXT Codes, on page D-32
- Table D-9, TACL-TYPE Codes, on page D-32
- Table D-10, OBJ-CLASS Codes, on page D-33
- Table D-11, STRUCTURE Codes, on page D-33
- Table D-12, SQL DATETIME Element Sizes, on page D-35
- Table D-13, SQL INTERVAL Element Sizes, on page D-36

Figure D-9. DICTOBL (Object Build List)—G-Series (page 1 of 6)

```
Record OBL.
  File is "DICTOBL"                Key-sequenced
                                   MaxExtents 500
                                   Code 204
                                   Audit.

  02 IDENTIFIER.

    03 OBJECT                      Pic "9(9)" COMP
                                   Null 0
                                   Display "[BZ]I10"
                                   Heading "Object/Number".

    03 ELEMENT                     Type Binary 16
                                   Display "I3"
                                   Heading "Element/Number".

  02 LEVEL                         Type Binary 16
                                   Display "I2"
                                   Heading "LV".

  02 LOCAL-NAME                    Type Character 30
                                   Heading "Element Name".

  02 COMMENT-TEXT                  Pic "9(9)" COMP
                                   Null 0
                                   Display "[BZ]I10"
                                   Heading "Comment/Text ID".

  02 VALUE-TEXT                    Pic "9(9)" COMP
                                   Null 0
                                   Display "[BZ]I10"
                                   Heading "Value/Text ID".

  02 AS-TEXT                       Pic "9(9)" COMP
                                   Null 0
                                   Display "[BZ]I10"
                                   Heading "As/Text ID".

  02 HEADING-TEXT                  Pic "9(9)" COMP
                                   Null 0
                                   Display "[BZ]I10"
                                   Heading "Heading/Text ID".

  02 DISPLAY-TEXT                  Pic "9(9)" COMP
                                   Null 0
                                   Display "[BZ]I10"
                                   Heading "Display/Text ID".

  02 PICTURE-TEXT                  Pic "9(9)" COMP
                                   Null 0
                                   Display "[BZ]I10"
                                   Heading "Picture/Text ID".

  02 HELP-TEXT                     Pic "9(9)" COMP
                                   Null 0
                                   Display "[BZ]I10"
                                   Heading "Help/Text ID".

  02 MUST-BE-TEXT                  Pic "9(9)" COMP
                                   Null 0
                                   Display "[BZ]I10"
                                   Heading "Must Be/Text ID".
```

Figure D-9. DICTOBL (Object Build List)—G-Series (page 2 of 6)

```
02 EDIT-PIC-TEXT                Pic "9(9)" COMP
                                Null 0
                                Display "[BZ]I10"
                                Heading "Edit Pic/Text ID".

02 TACL-TYPE                    Type Character 2
                                UPSHIFT
                                Heading "TACL/Type"
                                MUST BE "CP", "DV", "EN", "FN",
                                        "F3", "PH", "SI", "SV",
                                        "TI", "TS", "UN", "  ".

02 SOURCE-DEF                   Pic "9(9)" COMP
                                Display "[BZ]I10"
                                Null 0
                                Heading "Source/Def".

02 ELEMENT-REDEFINED            Type Binary 16
                                Display "[BZ]I3"
                                Heading "Element/Redefined".

02 OBJ-CLASS                    Type Binary 16
                                Display "[ZA1'Grp',PA1'Elm']I3"
                                Heading "Grp/Elm".

02 STRUCTURE                    Type Binary 16
                                Display "I2"
                                Heading "Data/Type".

02 ELEMENT-SIZE                 Type Binary 16
                                Display "I4"
                                Heading "Size".

02 SCALE                        Type Binary 16
                                Display "[BZ]I2"
                                Heading "Scale".

02 OFFSET                       Type Binary 16
                                Display "I4"
                                Heading "Offset".

02 OCCURS-MIN                   Type Binary 16
                                VALUE 1
                                Display "I4"
                                Heading "Occurs/Min".

02 OCCURS-MAX                   Type Binary 16
                                VALUE 1
                                Display "I4"
                                Heading "Occurs/Max".

02 OCCURS-MIN-OBJECT            Pic "9(9)" COMP
                                Display "[BZ]I10"
                                Null 0
                                Heading "Occurs Min/Object #".

02 OCCURS-MAX-OBJECT            Pic "9(9)" COMP
                                Display "[BZ]I10"
                                Null 0
                                Heading "Occurs Max/Object #".

02 OCCURS-ELEMENT               Type Binary 16
                                Display "I4"
                                Heading "Occurs/Element".
```

Figure D-9. DICTOBL (Object Build List)—G-Series (page 3 of 6)

```
02 STARTING                     Type Binary 16
                                Display "I4"
                                Heading "Starting/Element".

02 ENDING                       Type Binary 16
                                Display "I4"
                                Heading "Ending/Element".

02 TALBOUND                     Type Binary 16
                                Heading "Talbound".

02 NULL-VALUE                   Type Binary 16
                                Display "I5"
                                Heading "Null/Value".

02 NULL-VALUE-OBJECT            Pic "9(9)" COMP
                                Display "[BZ]I10"
                                Null 0
                                Heading "Null Value/Object #".

02 SPI-NULL-VALUE               Type Binary 16
                                Display "I5"
                                Heading "SPI-Null/Value".

02 SPI-NULL-VALUE-OBJECT        Pic "9(9)" COMP
                                Display "[BZ]I10"
                                Null 0
                                Heading "SPI-Null/Object #".

02 NULL-VALUE-SPECIFIED         Type Character 1
                                MUST BE "Y", "N"
                                UPSHIFT
                                Display "A1"
                                VALUE "N"
                                Heading "Null/Specified".

02 SPI-NULL-VALUE-SPECIFIED     Type Character 1
                                MUST BE "Y", "N"
                                UPSHIFT
                                Display "A1"
                                VALUE "N"
                                Heading "SPI-Null/Specified".

02 UPSHIFT                      Type Character 1
                                MUST BE "Y", "N"
                                UPSHIFT
                                Display "A1"
                                VALUE "N"
                                Heading "Upshift".

02 USER-DEFINED-FILLER          Type Character 1
                                MUST BE "Y", "N"
                                UPSHIFT
                                Display "A1"
                                VALUE "N"
                                Heading "User Defined/Filler".

02 PADDED-FILLER                REDEFINES USER-DEFINED-FILLER
                                Type Character 1
                                MUST BE "Y", "N"
                                UPSHIFT
                                Display "A1"
                                VALUE "N"
                                Heading "Padded/Filler".
```

Figure D-9. DICTOBL (Object Build List)—G-Series (page 4 of 6)

```
02 GROUP-COMP                    Type Character 1
                                 MUST BE "Y", "N"
                                 UPSHIFT
                                 Display "A1"
                                 VALUE "N"
                                 Heading "Computational".

02 SOURCE-DEF-FLAG               Type Character 1
                                 MUST BE "Y", "N"
                                 UPSHIFT
                                 Display "A1"
                                 VALUE "N"
                                 Heading "Sourced/Item".

02 NOVALUE                       Type Character 1
                                 MUST BE "Y", "N"
                                 UPSHIFT
                                 Display "A1"
                                 VALUE "N"
                                 Heading "No Value".

02 TACL-INHERITED                Type Character 1
                                 MUST BE "Y", "N"
                                 UPSHIFT
                                 Display "A1"
                                 Heading "TACL Clause/Inherited".

02 NULL-INHERITED                Type Character 1
                                 MUST BE "Y", "N"
                                 UPSHIFT
                                 Display "A1"
                                 Heading "Null/Inherited".

02 SPI-NULL-INHERITED            Type Character 1
                                 MUST BE "Y", "N"
                                 UPSHIFT
                                 Display "A1"
                                 Heading "SPI-Null/Inherited".

02 UPSHIFT-INHERITED             Type Character 1
                                 MUST BE "Y", "N"
                                 UPSHIFT
                                 Display "A1"
                                 Heading "Upshift/Inherited".

02 USAGE-IS-INDEX                Type Character 1
                                 MUST BE "Y", "N"
                                 UPSHIFT
                                 Display "A1"
                                 VALUE "N"
                                 Heading "Index/Usage".

02 BIT-LENGTH                    Type Binary 32
                                 Display "I10"
                                 Heading "Bit Size".

02 FIELD-ALIGN                   Type Binary 16
                                 Display "I4"
                                 Heading "Field/Alignment".

02 BIT-OFFSET                    Type Binary 16
                                 Display "I4"
                                 Heading " Bit/Offset".
```

Figure D-9. DICTOBL (Object Build List)—G-Series (page 5 of 6)

```
02 ENUM-DEF                        Pic "9(9)" COMP
                                   Display "[BZ]I10"
                                   Null 0
                                   Heading "Enum/Def".

02 PASCALBOUND                     Type Binary 16
                                   Heading "Pascalbound".

02 INDEX-NAME                      Type Character 30
                                   Heading "Indexed By".

02 EXTERNAL-SPECIFIED              Type Character 1
                                   MUST BE "Y", "N"
                                   UPSHIFT
                                   Display "A1"
                                   VALUE "N"
                                   Heading "External".

02 JUSTIFY                         Type Character 1
                                   MUST BE "Y", "N"
                                   UPSHIFT
                                   Display "A1"
                                   VALUE "N"
                                   Heading "Justify".

02 JUSTIFY-INHERITED               Type Character 1
                                   MUST BE "Y", "N"
                                   VALUE "N"
                                   UPSHIFT
                                   Display "A1"
                                   Heading "Just-Right/Inherited".

02 SQLNULLABLE-SPECIFIED           Type Character 1
                                   MUST BE "Y", "N"
                                   VALUE "N"
                                   UPSHIFT
                                   Display "A1"
                                   Heading "SqlNull/Spec".

02 INTERVAL-LEADING-PRECISION      Type Binary 16
                                   Display "I4"
                                   Heading "Leading/Prec".

02 DATETIME-FRACTION-PRECISION     Type Binary 16
                                   Display "I4"
                                   Heading "Fraction/Prec".

02 SQLNULLABLE-FLAG                Type Character 1
                                   MUST BE "Y", "N"
                                   VALUE "N"
                                   UPSHIFT
                                   Display "A1"
                                   Heading "SqlNull/Flag".

02 GROUP-COMP3                     Type Character 1
                                   MUST BE "Y", "N"
                                   VALUE "N"
                                   UPSHIFT
                                   Display "A1"
                                   Heading "Computational-3".

02 FILLER                          Type Character 2.
```

Figure D-9. DICTOBL (Object Build List)—G-Series (page 6 of 6)

```
  02 SQLNULLABLE-FILLERS          Type Binary 16
                                  Display "I4"
                                  Heading "SqlNull/Filler"
                                  VALUE "0"
End
```

Figure D-10. DICTOBL (Object Build List)—H-Series (page 1 of 4)

```
Record OBL.
  File is "DICTOBL"               Key-sequenced
                                  Code 204
                                  Audit
                                  MaxExtents 500.

  02 IDENTIFIER                   Type FIELD.

  02 LEVEL                        Type Binary 16
                                  Display "I2"
                                  Heading "LV".

  02 LOCAL-NAME                   Type NAME
                                  Heading "Element Name".

  02 COMMENT-TEXT                 Type TEXT-ID
                                  Heading "Comment/Text ID".

  02 VALUE-TEXT                   Type TEXT-ID
                                  Heading "Value/Text ID".

  02 AS-TEXT                      Type TEXT-ID
                                  Heading "As/Text ID".

  02 HEADING-TEXT                 Type TEXT-ID
                                  Heading "Heading/Text ID".

  02 DISPLAY-TEXT                 Type TEXT-ID
                                  Heading "Display/Text ID".

  02 PICTURE-TEXT                 Type TEXT-ID
                                  Heading "Picture/Text ID".

  02 HELP-TEXT                    Type TEXT-ID
                                  Heading "Help/Text ID".

  02 MUST-BE-TEXT                 Type TEXT-ID
                                  Heading "Must Be/Text ID".

  02 EDIT-PIC-TEXT                Type TEXT-ID
                                  Heading "Edit Pic/Text ID".

  02 TACL-TYPE                    Type Character 2
                                  MUST BE "CP", "DV", "EN", "FN",
                                          "F3", "PH", "SI", "SV",
                                          "TI", "TS", "UN", "  ".
                                  UPSHIFT
                                  Heading "TACL/Type"

  02 SOURCE-DEF                   Type OBJECT-NUMBER
                                  Null 0
                                  Heading "Source/Def".
```

Figure D-10. DICTOBL (Object Build List)—H-Series (page 2 of 4)

```
  02 ELEMENT-REDEFINED            Type Binary 16
                                  Display "[BZ]I3"
                                  Heading "Element/Redefined".

  02 OBJ-CLASS                    Type Binary 16
                                  Display "[ZA1'Grp',PA1'Elm']I3"
                                  Heading "Grp/Elm".

  02 STRUCTURE                    Type Binary 16
                                  Display "I2"
                                  Heading "Data/Type".

  02 ELEMENT-SIZE                 Type Binary 16
                                  Display "I4"
                                  Heading "Size".

  02 SCALE                        Type Binary 16
                                  Display "[BZ]I2"
                                  Heading "Scale".

  02 OFFSET                       Type Binary 16
                                  Display "I4"
                                  Heading "Offset".

  02 OCCURS-MIN                   Type Binary 16
                                  VALUE 1
                                  Display "I4"
                                  Heading "Occurs/Min".

  02 OCCURS-MAX                   Type Binary 16
                                  VALUE 1
                                  Display "I4"
                                  Heading "Occurs/Max".

  02 OCCURS-MIN-OBJECT            Type OBJECT-NUMBER
                                  Null 0
                                  Heading "Occurs Min/Object #".

  02 OCCURS-MAX-OBJECT            Type OBJECT-NUMBER
                                  Display "[BZ]I10"
                                  Null 0
                                  Heading "Occurs Max/Object #".

  02 OCCURS-ELEMENT               Type Binary 16
                                  Display "I4"
                                  Heading "Occurs/Element".

  02 STARTING                     Type Binary 16
                                  Display "I4"
                                  Heading "Starting/Element".

  02 ENDING                       Type Binary 16
                                  Display "I4"
                                  Heading "Ending/Element".

  02 TALBOUND                     Type Binary 16
                                  Heading "Talbound".

  02 NULL-VALUE                   Type *.

  02 NULL-VALUE-OBJECT            Type OBJECT-NUMBER
                                  Null 0
                                  Heading "Null Value/Object #".
```

Figure D-10. DICTOBL (Object Build List)—H-Series (page 3 of 4)

```
02 SPI-NULL-VALUE                Type NULL-VALUE
                                 Heading "SPI-Null/Value".

02 SPI-NULL-VALUE-OBJECT         Type OBJECT-NUMBER
                                 Null 0
                                 Heading "SPI-Null/Object #".

02 NULL-VALUE-SPECIFIED          Type ASCII-SWITCH
                                 VALUE "N"
                                 Heading "Null/Specified".

02 SPI-NULL-VALUE-SPECIFIED      ASCII-SWITCH
                                 VALUE "N"
                                 Heading "SPI-Null/Specified".

02 UPSHIFT                       ASCII-SWITCH
                                 VALUE "N"
                                 Heading "Upshift".

02 USER-DEFINED-FILLER           Type ASCII-SWITCH
                                 VALUE "N"
                                 Heading "User Defined/Filler".

02 PADDED-FILLER                 REDEFINES USER-DEFINED-FILLER
                                 Type Character 1
                                 Heading "Padded/Filler".

02 GROUP-COMP                    Type ASCII-SWITCH
                                 VALUE "N"
                                 Heading "Computational".

02 SOURCE-DEF-FLAG               Type ASCII-SWITCH
                                 VALUE "N"
                                 Heading "Sourced/Item".

02 NOVALUE                       Type ASCII-SWITCH
                                 VALUE "N"
                                 Heading "No Value".

02 TACL-INHERITED                Type ASCII-SWITCH
                                 Heading "TACL Clause/Inherited".

02 NULL-INHERITED                Type ASCII-SWITCH
                                 Heading "Null/Inherited".

02 SPI-NULL-INHERITED            Type ASCII-SWITCH
                                 Heading "SPI-Null/Inherited".

02 UPSHIFT-INHERITED             Type ASCII-SWITCH
                                 Heading "Upshift/Inherited".

02 USAGE-IS-INDEX                Type ASCII-SWITCH
                                 VALUE "N"
                                 Heading "Index/Usage".

02 BIT-LENGTH                    Type Binary 32
                                 Display "I10"
                                 Heading "Bit Size".

02 FIELD-ALIGN                   Type Binary 16
                                 Display "I2"
                                 Heading "Field/Alignment".
```

Figure D-10. DICTOBL (Object Build List)—H-Series (page 4 of 4)

```
  02 BIT-OFFSET                     Type Binary 16
                                    Display "I4"
                                    Heading " Bit/Offset".

  02 ENUM-DEF                       Type OBJECT-NUMBER
                                    Null 0
                                    Heading "Enum/Def".

  02 PASCALBOUND                    Type Binary 16
                                    Heading "Pascalbound".

  02 INDEX-NAME                     Type NAME
                                    Heading "Indexed By".

  02 EXTERNAL-SPECIFIED             Type ASCII-SWITCH
                                    VALUE "N"
                                    Heading "External".

  02 JUSTIFY                        Type ASCII-SWITCH
                                    VALUE "N"
                                    Heading "Justify".

  02 JUSTIFY-INHERITED              Type ASCII-SWITCH
                                    VALUE "N"
                                    Heading "Just-Right/Inherited".

  02 SQLNULLABLE-SPECIFIED          Type ASCII-SWITCH
                                    VALUE "N"
                                    Heading "SqlNull/Spec".

  02 INTERVAL-LEADING-PRECISION     Type Binary 16
                                    VALUE 2
                                    Display "I4"
                                    Heading "Leading/Prec".

  02 DATETIME-FRACTION-PRECISION    Type Binary 16
                                    VALUE 0
                                    Display "I4"
                                    Heading "Fraction/Prec".

  02 SQLNULLABLE-FLAG               Type ASCII-SWITCH
                                    VALUE "N"
                                    Heading "SqlNull/Flag".

  02 GROUP-COMP3                    Type ASCII-SWITCH
                                    VALUE "N"
                                    Heading "Computational-3".

  02 FILLER                         Type Character 2.

  02 SQLNULLABLE-FILLERS            Type Binary 16
                                    VALUE 0
                                    Display "I4"
                                    Heading "SqlNull/Filler"

  Key is IDENTIFIER Duplicates not allowed.
  Key "SO" is SOURCE-DEF File is "DICTALT".

End
```

Table D-7. DICTOBL (Object Build List) Fields (page 1 of 8)

Field	Description
IDENTIFIER	A unique identifier for each object in the dictionary, consisting of the next two fields, OBJECT and ELEMENT.
OBJECT	The object number of the definition or record.
ELEMENT	The element number of the group or field within the object. Element numbers are assigned sequentially, starting with 0. Element number 0 describes the entire object.
LEVEL	The level number of this element relative to the level of the entire definition. The first element (the object name) has a level of 0. Subordinate groups and elementary items have higher level values, but none greater than 49. Although the level numbers in the schema can be incremented by values greater than one, the DDL compiler compresses all level values so that there is no skipping. This field also identifies Level 66 RENAMES and Level 88 clauses, which have reserved level numbers.
LOCAL-NAME	A field with 30 ASCII characters, containing the name of this element.
COMMENT-TEXT	The text ID of the OTF record that contains any comment for this element. If there is no comment text, this field is set to 0.
VALUE-TEXT	The text ID number of the OTF record that contains the value string for this element. Values and lists of values are represented in the OTF as sequences of records that have one of the text types in Table D-8 on page D-32. If there is no value text, this field is set to 0.
AS-TEXT	The text ID of the OTF record that contains the display text for level 89 items.
HEADING-TEXT	The text ID of the OTF record that contains the HEADING string for this element. The text type for this field is S (string). If the element has no HEADING clause, this field is set to 0.
DISPLAY-TEXT	The text ID of the OTF record that contains the DISPLAY string for this element. The text type for this field is S (string). If the element has no DISPLAY clause, this field is set to 0.

Table D-7. DICTOBL (Object Build List) Fields (page 2 of 8)

Field	Description
PICTURE-TEXT	The text ID of the OTF record that contains the PICTURE string for this element. The text type for this field is S (string). If the element was not defined with a PICTURE clause, this field is set to 0.
HELP-TEXT	The text ID of the OTF record that contains the help text for this element. The text type for this field is S (string). If the element was not defined with a HELP clause, this field is set to 0.
MUST-BE-TEXT	The text ID of the OTF record that contains the MUST BE string for this element. The MUST BE string consists of a value, a list of values, or ranges of values that can be entered in a field. If the element was not defined with a MUST BE clause, this field is set to 0. Text items can be one of three text types (K, N, S), as described under VALUE-TEXT. A single MUST BE string can be made up of text items of different types.
EDIT-PIC-TEXT	The text ID of the OTF record containing the edit picture value.
TACL-TYPE	Contains a 2-character ASCII code identifying the high-level TACL data type associated with the element. Valid codes for this field are in Table D-9 on page D-32. If this field is left blank, the item does not have a high-level TACL data type.

Table D-7. DICTOBL (Object Build List) Fields (page 3 of 8)

Field	Description
SOURCE-DEF	The object number of the referenced definition when this element is described by TYPE def-name or TYPE *. SOURCE-DEF is an alternate key to the OBL. The DDL compiler uses this key to determine whether a definition is referenced by any other record or definition and to find the name of the referenced definition. SOURCE-DEF is set to 0 for all elements not defined with a TYPE clause. If the referenced definition itself contains more than one element, these elements are copied to the current object's build list. SOURCE-DEF keeps only one level of reference, as shown in Example D-3 on page D-33. If the object number of DATE is 1, the SOURCE-DEF code for the element ORDER-DATE is 1, referring to the object DATE. If the object number of ORD-HEADER is 2, the SOURCE-DEF code for header is 2, referring to the element ORDER-DATE in the object ORD-HEADER; in this case, SOURCE-DEF does not indicate that ORDER-DATE in turn refers to DATE.
ELEMENT-REDEFINED	The element number of the group or field that this element redefines if this element redefines another element.
OBJ-CLASS	An indicator that describes this element as a group or elementary field; it can have one of the codes in Table D-10 on page D-33.
STRUCTURE	A field that identifies the storage structure of this element if it is an elementary field (OBJ-CLASS=1). The DDL compiler supports the STRUCTURE codes in Table D-11 on page D-33.

Table D-7. DICTOBL (Object Build List) Fields (page 4 of 8)

Field	Description
ELEMENT-SIZE	The number of bytes per occurrence of this element. The total size of the element is equal to ELEMENT-SIZE times OCCURS-MAX. For an SQL VARCHAR element, this field contains the actual length of the element. For SQL DATETIME elements, this field contains the byte length needed for the longest possible ANSI DATETIME string with a specific SQL DATETIME qualifier, as listed in Table D-12 on page D-35. For SQL INTERVAL elements, this field contains the value of the byte length required for the longest possible interval string with a specific interval qualifier, as listed in Table D-13 on page D-36.
SCALE	The scale factor in a numeric field; the scale is equal to the number of positions to the right of the implied decimal point.
OFFSET	The number of bytes from the first byte of the object to the first byte of this element; byte numbering begins with 0.
OCCURS-MIN	The minimum number of times LOCAL-NAME occurs; the default value is 1. If the element is described by an OCCURS *min* TO *max* TIMES DEPENDING ON clause, this field contains the value of min.
OCCURS-MAX	The number of times LOCAL-NAME occurs; the default value is 1. If the element is described by an OCCURS *max* TIMES clause, this field contains the value *max*.
OCCURS-MIN-OBJECT	If *min* is defined as a constant, contains the object number of the constant; otherwise, it is 0.
OCCURS-MAX-OBJECT	If *max* is defined by a constant, contains the object number of the constant; otherwise, it is 0.
OCCURS-ELEMENT	The element number of the field in an OCCURS DEPENDING ON field-name clause. This element must be defined as an integer. If there is no DEPENDING ON clause, the field is set to 0.
STARTING	The first element of the set of elements renamed by a level 66 RENAMES clause, where this starting element has the same offset as renaming element.

Table D-7. DICTOBL (Object Build List) Fields (page 5 of 8)

Field	Description
ENDING	The last element of the set of elements renamed by a level 66 RENAMES clause, where this ending element ends at the same position as the renaming element.
TALBOUND	Contains a binary value that specifies the lower limit of pTAL or TAL arrays. Valid values are 0 and 1. The default value for this field is 1.
NULL-VALUE	The ASCII value used by the DDL compiler when producing FUP source output for an alternate key.
NULL-VALUE-OBJECT	If the null value is defined by a constant, contains the object number of the constant; otherwise, it is 0.
SPI-NULL-VALUE	Contains a user-specified SPI null value used by SPI to process token maps.
SPI-NULL-VALUE-OBJECT	If the SPI null value is defined by a constant, contains the object number of the constant; otherwise, it is 0.
NULL-VALUE-SPECIFIED	Contains the ASCII character Y (yes) to indicate this item has a null value or N (no) to indicate it does not. N is the default.
SPI-NULL-VALUE-SPECIFIED	Contains the ASCII character Y (yes) to indicate a SPI null value was explicitly specified for this object or N (no) to indicate it was not. N is the default.
UPSHIFT	Contains the ASCII character Y (yes) to indicate this data item is to be upshifted or N (no) to indicate it is not to be upshifted. N is the default. Only an elementary item can be upshifted. The field must be declared as alphabetic or alphanumeric.
USER-DEFINED-FILLER	Contains the ASCII character Y (yes) to indicate this field is a user-defined FILLER field or N (no) to indicate it is not. N is the default.
PADDED-FILLER	Contains the ASCII character Y (yes) to indicate that an SQL VARCHAR element has an odd byte length and has an OCCURS clause associated with it; contains N (no) to indicate that the element does not. N is the default.
GROUP-COMP	For group items; contains the ASCII character Y (yes) to indicate the group is defined as computational or N (no) to indicate it is not. All elementary items within a group defined as computational are treated as though they were individually defined as computational. N is the default.

Table D-7. DICTOBL (Object Build List) Fields (page 6 of 8)

Field	Description
SOURCE-DEF-FLAG	Contains the ASCII character Y (yes) to indicate this item is defined with TYPE * or TYPE def-name or N (no) to indicate it is not. N is the default.
NOVALUE	Contains the ASCII character Y (yes) or N (no) to indicate whether this item has the NOVALUE attribute. NOVALUE suppresses any VALUE IS clause in a referenced definition. NOVALUE can be specified only for a field or group definition defined with a TYPE clause. N is the default.
TACL-INHERITED	Contains the ASCII character Y (yes) to indicate the TACL type was inherited from a definition or N (no) to indicate the type was explicitly specified.
NULL-INHERITED	Contains the ASCII character Y (yes) to indicate the null was inherited from a definition or N (no) to indicate it was explicitly specified.
SPI-NULL-INHERITED	Contains the ASCII character Y (yes) to indicate the SPI null was inherited from a definition or from the default, or N (no) to indicate the SPI null was explicitly specified.
UPSHIFT-INHERITED	Contains the ASCII character Y (yes) to indicate the upshift was inherited from a definition or from the default, or N (no) to indicate the upshift was explicitly specified.
USAGE-IS-INDEX	Contains the ASCII character Y (yes) or N (no) to indicate whether the item is to be used as an index. This field is set to Y if the definition or description of the item includes a USAGE IS INDEX clause. N is the default.
BIT-LENGTH	Contains the bit length of the current item. For an item that is not a bit map, the bit length is a multiple of 8. For a bit map item, the bit length is a value from 1 to 15.
FIELD-ALIGN	Contains the alignment method used when storing the item. C00CALIGN is the default.
BIT-OFFSET	Contains the bit offset from the (byte) offset that this elementary item is in. For an item that is not a bit map, the bit offset value is 0. For a bit map item, the bit offset value is from 0 to 15. (A group item that is a bit map or maps always starts on a word boundary.) The bit offset from the start of the structure for any item is the value of the BIT-OFFSET field plus 8 times the value of the OFFSET field in the DICTOBL file.

Table D-7. DICTOBL (Object Build List) Fields (page 7 of 8)

Field	Description
ENUM-DEF	Contains the object number of the enumeration definition specified in the ENUM clause of a bit map item. For an item that is not a bit map, this field contains the null value for OBJECT-NUMBER.
PASCALBOUND (D-series systems only)	Contains the value of the lower bound, 0 or 1, for Pascal arrays. 1 is the default.
INDEX-NAME	Contains the index name specified in the INDEXED BY attribute, padded with blanks.
EXTERNAL-SPECIFIED	Contains the ASCII character Y (yes) if the element is to be external. This attribute permits you to output the EXTERNAL clause in COBOL. Only elements of object name level can have this attribute. N (no) is the default.
JUSTIFY	Contains the ASCII character Y (yes) or N (no) depending on whether the element is to be right justified. N is the default.
JUSTIFY-INHERITED	Contains the ASCII character Y (yes) if the element is to be justified right because that attribute was inherited from a definition, or N (no) if right justification was specified on the line item by the appropriate clause. N is the default.
SQLNULLABLE-SPECIFIED	Contains the ASCII character Y (yes) if the line item has the [NOT]SQLNULLABLE clause specified, or N (no) if no such clause is specified. N is the default. If the value of this field is Y, the value of the SQLNULLABLE-FLAG field, described below, indicates whether SQLNULLABLE or NOT SQLNULLABLE is specified. (If the value of this field is N and the value of the SQLNULLABLE-FLAG field is Y, the indication is that the line item, although it has no explicit SQL-nullability specification, is nevertheless SQL-nullable because of a specification at the group level above.)
INTERVAL-LEADING-PRECISION	Contains the number of significant digits specified as the start-field-precision of the SQL INTERVAL line item. Only line items of data type SQL INTERVAL use this field. The valid range for this value is 1 through 18. If no *start-field-precision* is specified, the default value for this field is 2.

Table D-7. DICTOBL (Object Build List) Fields (page 8 of 8)

Field	Description
DATETIME-FRACTION-PRECISION	Contains the number of significant digits with which the fraction of a second is specified. This field stores the value of end-field-precision of FRACTION (one of the end-date-time qualifiers) in an SQL DATETIME or SQL INTERVAL line item. The valid range for this value is 1 through 6. If *end-field-precision* is not specified, the default value for this field is 6. If no end-date-time qualifier is specified with FRACTION, the default value for this field is zero.
SQLNULLABLE-FLAG	Contains the ASCII character Y (yes) if the line item is SQL-nullable, or N (no) if it is not. N is the default.
GROUP-COMP3	Contains the ASCII character Y (yes) if the line item is of type PACKED-DECIMAL or N (no) if it is not. N is the default.
SQLNULLABLE-FILLERS	Contains the number of fillers required to add after INDICATOR field to make the SQL-nullable well-aligned for SHARED8 alignment. This field is only used for SHARED8 alignment.

Table D-8. VALUE-TEXT Codes

Code	Meaning
K	Keyword
N	ASCII representation of a numeric literal
S	Alphanumeric string
E	Enumeration value name
J	National string

Table D-9. TACL-TYPE Codes (page 1 of 2)

Code	TACL Type
CP	CRTPID
DV	DEVICE
EN	ENUM
FN	FNAME
F3	FNAME32
PH	PHANDLE
SI	SSID
SV	SUBVOL

Table D-9. TACL-TYPE Codes (page 2 of 2)

Code	TACL Type
TI	TRANSID
TS	TSTAMP
UN	USERNAME

Example D-3. SOURCE-DEF Field

```
DEF date.
  02 year        PIC 99.
  02 month       PIC 99.
  02 day         PIC 99.
END

DEF ord-header.
  02 order-date TYPE date.
END

DEF order.
  02 header      TYPE ord-header.
END
```

Table D-10. OBJ-CLASS Codes

Code	Meaning
0	Group field
1	Elementary field

Table D-11. STRUCTURE Codes (page 1 of 3)

Code	Meaning
0	Alphanumeric string
1	Numeric string unsigned
2	Binary 16 signed
3	Binary 16 unsigned
4	Binary 32 signed
5	Binary 32 unsigned
6	Binary 64 signed
8	Float 32
9	Complex 32*2
10	Float 64
12	Numeric string trailing embedded sign
13	Numeric string trailing separate sign

Table D-11. STRUCTURE Codes (page 2 of 3)

Code	Meaning
14	Numeric string leading embedded sign
15	Numeric string leading separate sign
17	Logical*1
19	Logical*2
21	Logical*4
22	Binary 8 signed
23	Binary 8 unsigned
24	SQL VARCHAR
25	Enumeration
26	Bit signed
27	Bit unsigned
28	National string
32	SQL DATETIME YEAR
33	SQL DATETIME MONTH
34	SQL DATETIME YEAR TO MONTH
35	SQL DATETIME DAY
36	SQL DATETIME MONTH TO DAY
37	SQL DATETIME YEAR TO DAY
38	SQL DATETIME HOUR
39	SQL DATETIME DAY TO HOUR
40	SQL DATETIME MONTH TO HOUR
41	SQL DATETIME YEAR TO HOUR
42	SQL DATETIME MINUTE
43	SQL DATETIME HOUR TO MINUTE
44	SQL DATETIME DAY TO MINUTE
45	SQL DATETIME MONTH TO MINUTE
46	SQL DATETIME YEAR TO MINUTE
47	SQL DATETIME SECOND
48	SQL DATETIME MINUTE TO SECOND
49	SQL DATETIME HOUR TO SECOND
50	SQL DATETIME DAY TO SECOND
51	SQL DATETIME MONTH TO SECOND
52	SQL DATETIME YEAR TO SECOND
53	SQL DATETIME FRACTION

Table D-11. STRUCTURE Codes (page 3 of 3)

Code	Meaning
54	SQL DATETIME SECOND TO FRACTION
55	SQL DATETIME MINUTE TO FRACTION
56	SQL DATETIME HOUR TO FRACTION
57	SQL DATETIME DAY TO FRACTION
58	SQL DATETIME MONTH TO FRACTION
59	SQL DATETIME YEAR TO FRACTION
60	SQL INTERVAL YEAR
61	SQL INTERVAL MONTH
62	SQL INTERVAL YEAR TO MONTH
63	SQL INTERVAL DAY
64	SQL INTERVAL HOUR
65	SQL INTERVAL DAY TO HOUR
66	SQL INTERVAL MINUTE
67	SQL INTERVAL HOUR TO MINUTE
68	SQL INTERVAL DAY TO MINUTE
69	SQL INTERVAL SECOND
70	SQL INTERVAL MINUTE TO SECOND
71	SQL INTERVAL HOUR TO SECOND
72	SQL INTERVAL DAY TO SECOND
73	SQL INTERVAL FRACTION
74	SQL INTERVAL SECOND TO FRACTION
75	SQL INTERVAL MINUTE TO FRACTION
76	SQL INTERVAL HOUR TO FRACTION
77	SQL INTERVAL DAY TO FRACTION

Table D-12. SQL DATETIME Element Sizes (page 1 of 2)

Code	Meaning	Element Size
32	SQL DATETIME YEAR	4
33	SQL DATETIME MONTH	2
34	SQL DATETIME YEAR TO MONTH	7
35	SQL DATETIME DAY	2
36	SQL DATETIME MONTH TO DAY	5
37	SQL DATETIME YEAR TO DAY	10
38	SQL DATETIME HOUR	2

Table D-12. SQL DATETIME Element Sizes (page 2 of 2)

Code	Meaning	Element Size
39	SQL DATETIME DAY TO HOUR	5
40	SQL DATETIME MONTH TO HOUR	8
41	SQL DATETIME YEAR TO HOUR	13
42	SQL DATETIME MINUTE	2
43	SQL DATETIME HOUR TO MINUTE	5
44	SQL DATETIME DAY TO MINUTE	8
45	SQL DATETIME MONTH TO MINUTE	11
46	SQL DATETIME YEAR TO MINUTE	16
47	SQL DATETIME SECOND	2
48	SQL DATETIME MINUTE TO SECOND	5
49	SQL DATETIME HOUR TO SECOND	8
50	SQL DATETIME DAY TO SECOND	11
51	SQL DATETIME MONTH TO SECOND	14
52	SQL DATETIME YEAR TO SECOND	19
53	SQL DATETIME FRACTION	6
54	SQL DATETIME SECOND TO FRACTION	9
55	SQL DATETIME MINUTE TO FRACTION	12
56	SQL DATETIME HOUR TO FRACTION	15
57	SQL DATETIME DAY TO FRACTION	18
58	SQL DATETIME MONTH TO FRACTION	21
59	SQL DATETIME YEAR TO FRACTION	26

Table D-13. SQL INTERVAL Element Sizes (page 1 of 2)

Code	Meaning	Element
60	SQL INTERVAL YEAR	3
61	SQL INTERVAL MONTH	3
62	SQL INTERVAL YEAR TO MONTH	6
63	SQL INTERVAL DAY	3
64	SQL INTERVAL HOUR	3
65	SQL INTERVAL DAY TO HOUR	6
66	SQL INTERVAL MINUTE	3
67	SQL INTERVAL HOUR TO MINUTE	6
68	SQL INTERVAL DAY TO MINUTE	9
69	SQL INTERVAL SECOND	3

Table D-13. SQL INTERVAL Element Sizes (page 2 of 2)

Code	Meaning	Element
70	SQL INTERVAL MINUTE TO SECOND	6
71	SQL INTERVAL HOUR TO SECOND	9
72	SQL INTERVAL DAY TO SECOND	12
73	SQL INTERVAL FRACTION	7
74	SQL INTERVAL SECOND TO FRACTION	10
75	SQL INTERVAL MINUTE TO FRACTION	13
76	SQL INTERVAL HOUR TO FRACTION	16
77	SQL INTERVAL DAY TO FRACTION	19

DICTODF (Object Definition File)

DICTODF (Object Definition File) is a key-sequenced file that contains one record for each object in the dictionary.

DICTODF is an important entry point into the dictionary. Given an object name and object type, DICTODF provides the object number. Given an object number, DICTODF provides the object type and name.

DICTODF is different on G-series and H-series systems—see:

- Figure D-11, DICTODF (Object Definition File)—G-Series, on page D-37
- Figure D-12, DICTODF (Object Definition File)—H-Series, on page D-39

Change bars in Figure D-12 on page D-39 show where it differs from Figure D-11 on page D-37.

These tables apply to both G-series and H-series systems:

- Table D-14, DICTODF (Object Definition File) Fields, on page D-39
- Table D-15, OBJ-TYPE Values, on page D-40

Figure D-11. DICTODF (Object Definition File)—G-Series (page 1 of 2)

```
Record ODF.
  File is "DICTODF"                    Key-sequenced
                                       Code 202
                                       Audit.

  02 OBJECT                            Pic "9(9)" COMP
                                       Null 0
                                       Display "[BZ]I10"
                                       Heading "Object/Number".
```

Figure D-11. DICTODF (Object Definition File)—G-Series (page 2 of 2)

```
  02 IDENTIFIER                    Null " "
                                   Heading "Object Identifier".

    03 OBJ-TYPE                    Type Character 2
                                   MUST BE "ID","RD","CD","TT",
                                                "TC","TM","SR","SV",
                                                "RQ","TB"
                                   UPSHIFT
                                   Heading "Obj/Type".

    03 NAME                        Type Character 30
                                   Heading "Object Name".

  02 VERSION                       Type Binary 16 Unsigned
                                   Display "I3"
                                   Heading "Cur/Ver".

  02 DATE-CREATED                  Type Character 6
                                   Heading "Date-Time/Created".

  02 CREATOR-USERID                Heading "Created By"
                                   TACL USERNAME.

    03 GROUP-NAME                  Type Character 8
                                   UPSHIFT
                                   Heading "Group".

    03 USER-NAME                   Type Character 8
                                   UPSHIFT
                                   Heading "User".

  02 DATE-MODIFIED                 Type Character 6
                                   Heading "Date-Time/
                                   Last Modified".

  02 MODIFIER-USERID               Heading "Modified By"
                                   TACL USERNAME.

    03 GROUP-NAME                  Type Character 8
                                   UPSHIFT
                                   Heading "Group".

    03 USER-NAME                   Type Character 8
                                   UPSHIFT
                                   Heading "User".

  02 COMMENT-TEXT                  Pic "9(9)" COMP
                                   Null 0
                                   Display "[BZ]I10"
                                   Heading "Comments/Text Id".

  Key is OBJECT Duplicates not allowed.
  Key "ID" is IDENTIFIER Duplicates not allowed.

End
```

Figure D-12. DICTODF (Object Definition File)—H-Series

```
Record ODF.
  File is "DICTODF"                    Key-sequenced
                                       Code 202
                                       Audit
                                       MaxExtents 500.

  02 OBJECT                            Type OBJECT-NUMBER.

  02 IDENTIFIER                        Type OBJECT-IDENTIFIER.

  02 VERSION                           Type Binary 16 Unsigned
                                       Display "I3"
                                       Heading "Cur/Ver".

  02 DATE-CREATED                      Type INTERNAL-TIMESTAMP
                                       Heading "Date-Time/Created".

  02 CREATOR-USERID                    Type USERID-NAME
                                       Heading "Created By".

  02 DATE-MODIFIED                     Type INTERNAL-TIMESTAMP
                                       Heading "Date-Time/
                                       Last Modified".

  02 MODIFIER-USERID                   Type USERID-NAME
                                       Heading "Modified By".

  02 COMMENT-TEXT                      Type TEXT-ID
                                       Heading "Comments/Text Id".

  Key is OBJECT Duplicates not allowed.
  Key "ID" is IDENTIFIER File is "DICALT" Duplicates not allowed.

End
```

Table D-14. DICTODF (Object Definition File) Fields (page 1 of 2)

Field	Description
OBJECT	The object number, a system-assigned number that uniquely identifies each object within the dictionary. Each object corresponds to exactly one identifier.
IDENTIFIER	A unique identifier of each object in the dictionary, consisting of the next two fields, OBJ-TYPE and NAME.
OBJ-TYPE	A 2-byte field that has one of the values in Table D-15 on page D-40.
NAME	A 30-byte field containing the object's name, which must be unique within the type. Object names must begin with an alphabetic character. A dash (-) can be used as a word separator within a name. Records and definitions cannot have the same name.

Table D-14. DICTODF (Object Definition File) Fields (page 2 of 2)

Field	Description
VERSION	An integer value that is incremented every time the object is updated in this dictionary. Because the DDL compiler does not allow partial updating of an object, VERSION reflects how many times the object has been compiled since the dictionary was created.
DATE-CREATED	A 6-byte timestamp taken from the system when the object is added to the dictionary.
CREATOR-USERID	A group field that describes the user ID of the person who created this dictionary. CREATOR-USERID consists of the next two fields, GROUP-NAME and USER-NAME.
GROUP-NAME	The name of the group to which the user belongs.
USER-NAME	A name identifying the user within the group.
DATE-MODIFIED	A 6-byte timestamp that is updated every time the object is modified. DATE-MODIFIED is initially set to the same value as DATE-CREATED.
MODIFIER-USERID	A group field that describes the user ID of the last person to modify this dictionary. MODIFIER-USERID consists of the next two fields, GROUP-NAME and USER-NAME.
GROUP-NAME	The name of the group to which the user belongs.
USER-NAME	A name identifying the user within the group.
COMMENT-TEXT	A 32-bit number that identifies the comment text associated with the object. If no comment precedes the object, or the COMMENTS command is not set when the object is added to the dictionary, this field is set to 0; otherwise, the field contains the partial key to the comment text stored in DICTOTF. For a DEFINITION object, the comment text associated with the object is identified by the COMMENT-TEXT field in the DICTOBL file, not in the DICTODF file.

Table D-15. OBJ-TYPE Values (page 1 of 2)

Value	Object Type	Description
CD	CONSTANT	Constant
ID	DEFINITION	Field definition or group or field description
RD	RECORD	Record
TT	TOKEN TYPE	SPI token type
TC	TOKEN CODE	SPI token code
TM	TOKEN MAP	SPI token map

* This object type is defined and used in Pathmaker applications.

Table D-15. OBJ-TYPE Values (page 2 of 2)

Value	Object Type	Description
SR	SERVER*	Application program that performs one or more services
SV	SERVICE*	Unit of work performed by a server
RQ	REQUESTER*	Equivalent to a SCREEN COBOL program
SC	SCREEN*	Equivalent to the SCREEN SECTION of a SCREEN COBOL program.

* This object type is defined and used in Pathmaker applications.

DICTOTF (Object Text File)

DICTOTF (Object Text File) is a key-sequenced file that contains all of the text items associated with a schema. Each text block is assigned a unique text ID that links objects and elements to their associated text items in DICTOTF.

DICTOTF is different on G-series and H-series systems—see:

- Figure D-13, DICTOTF (Object Text File)—G-Series, on page D-41
- Figure D-14, DICTOTF (Object Text File)—H-Series, on page D-42

Change bars in Figure D-14 on page D-42 show where it differs from Figure D-13 on page D-41.

These tables apply to both G-series and H-series systems:

- Table D-16, DICTOTF (Object Text File) Fields, on page D-44
- Table D-17, TEXT-TYPE Codes, on page D-45

Figure D-13. DICTOTF (Object Text File)—G-Series (page 1 of 2)

```
Record OTF.
  File is "DICTOTF"                    Key-sequenced
                                       Code 203
                                       Audit.

  02 IDENTIFIER.

    03 TEXT-ID                         Pic "9(9)" COMP
                                       Null 0
                                       Display "[BZ]I10"
                                       Heading "Text Id/Number".

    03 LINE-NUMBER                     Type Binary 16 Unsigned
                                       Heading "Line/Num".

  02 CONSTANT-ID                       Pic "9(9)" COMP
                                       Null 0
                                       Display "[BZ]I10"
                                       Heading "Constant/Object #".
```

Figure D-13. DICTOTF (Object Text File)—G-Series (page 2 of 2)

```
  02 CONSTANT-TYPE-STRING            PIC "XX"
    88 CONSTANT-STRING               Value is "ST".
    88 CONSTANT-VERSION              Value is "VR".
    88 CONSTANT-BINARY-16            Value is "2S".
    88 CONSTANT-BINARY-UNSIGNED      Value is "2U".
    88 CONSTANT-BINARY-32            Value is "4S".
    88 CONSTANT-BINARY-32-UNSIGNED   Value is "4U".
    88 CONSTANT-BINARY-64            Value is "8S".
    88 CONSTANT-NATIONAL-STRING      Value is "NS".

  02 CONSTANT-TYPE Redefines CONSTANT-TYPE-STRING Type Binary 16.

  02 CONSTANT-TYPE-EXPLICIT      Type Character 1
                                 MUST BE "Y","N"
                                 UPSHIFT
                                 Display "A1"
                                 Heading "Type/Explicitly Given".

  02 LOCALE-NAME                 Type Character 16.
                                 Heading "Locale Name".

  02 LN-CONSTANT                Pic "9(9)" COMP
                                Null 0
                                Display "[BZ]I10"
                                Heading "Constant/Locale #".

  02 TEXT-LEN                   Type Binary 16 Unsigned
                                Heading "Text/Len".

  02 TEXT-TYPE                  Type Character 1
                                MUST BE "K", "N", "S", "E" or "J"
                                UPSHIFT
                                Heading "TX/TP".

  02 TEXT-LINE                  Heading "Text Line".
    03 BYTE                     Type Character 1
                                Occurs 1 to 132 times depending on TEXT-LEN.

  Key is IDENTIFIER Duplicates not allowed.

End
```

Figure D-14. DICTOTF (Object Text File)—H-Series (page 1 of 2)

```
Record OTF.
  File is "DICTOTF"                 Key-sequenced
                                    Code 203
                                    Audit
                                    MaxExtents 500.

  02 IDENTIFIER.

    03 TEXT-ID                      Type *.

    03 LINE-NUMBER                  Type Binary 16 Unsigned
                                    Heading "Line/Num".

  02 CONSTANT-ID                    Type OBJECT-NUMBER
                                    Heading "Constant/Object #".
```

Figure D-14. DICTOTF (Object Text File)—H-Series (page 2 of 2)

```
  02 CONSTANT-TYPE-STRING            PIC "XX"
    88 CONSTANT-STRING               Value is "ST".
    88 CONSTANT-VERSION              Value is "VR".
    88 CONSTANT-BINARY-16            Value is "2S".
    88 CONSTANT-BINARY-UNSIGNED      Value is "2U".
    88 CONSTANT-BINARY-32            Value is "4S".
    88 CONSTANT-BINARY-32-UNSIGNED   Value is "4U".
    88 CONSTANT-BINARY-64            Value is "8S".
    88 CONSTANT-NATIONAL-STRING      Value is "NS".

  02 CONSTANT-TYPE Redefines CONSTANT-TYPE-STRING Type Binary 16.

  02 CONSTANT-TYPE-EXPLICIT     Type ASCII-SWITCH
                                Heading "Type/Explicitly Given".

  02 LOCALE-NAME                Type Character 16.
                                Heading "Locale Name".

  02 LN-CONSTANT                Type OBJECT-NUMBER
                                Heading "Constant/Locale #".

  02 TEXT-LEN                   Type Binary 16 Unsigned
                                Heading "Text/Len".

  02 TEXT-TYPE                  Type Character 1
                                MUST BE "K", "N", "S", "E" or "J"
                                UPSHIFT
                                Heading "TX/TP".

  02 TEXT-LINE                  Heading "Text Line".
    03 BYTE                     Type Character 1
                                Occurs 1 to 132 times depending on TEXT-LEN.

  Key is IDENTIFIER Duplicates not allowed.

End
```

Table D-16. DICTOTF (Object Text File) Fields

Field	Description
IDENTIFIER	A group that uniquely identifies each record in DICTOTF, consisting of the next two fields, TEXT-ID and LINE-NUMBER.
TEXT-ID	A number that uniquely identifies a text item. The value of TEXT-ID can be used as a link to DICTODF and DICTOBL through the following fields: ● ODF.COMMENT-TEXT ● OBL.COMMENT-TEXT ● OBL.DISPLAY-TEXT ● OBL.HEADING-TEXT ● OBL.HELP-TEXT ● OBL.MUST-BE-TEXT ● OBL.PICTURE-TEXT ● OBL.VALUE-TEXT DISPLAY, HEADING, and PICTURE strings can have only one line of text for each text ID; otherwise, a text ID can have multiple lines of text associated with it.
LINE-NUMBER	A number that uniquely identifies each line of text associated with a text item. Line numbers are assigned sequentially from 0.
CONSTANT-ID	If the text element was defined by referring to a constant, contains the object number of the constant; otherwise, it is 0.
CONSTANT-TYPE-STRING	A two-character ASCII code that identifies the type of constant.
CONSTANT-TYPE	A numeric code identifying the type of constant.
CONSTANT-TYPE-EXPLICIT	Contains an ASCII character “Y” (yes) to indicate the constant type was entered explicitly or “N” (no) to indicate the constant type was inherited from another constant or by default.
LOCALE-NAME	The locale name for an internationalization item.
LN-CONSTANT	If the locale name was defined by referring to a constant, contains the object number of the constant; otherwise, it is 0.
TEXT-LEN	The number of bytes of text in TEXT-LINE.
TEXT-TYPE	A code in Table D-17 on page D-45 that identifies the type of text stored in TEXT-LINE. For more information about text types, see Text Items on page D-2.
TEXT-LINE	The text line identified by TEXT-ID and LINE number. Each line of text is a variable length string of from 0 through 132 bytes.

Table D-17. TEXT-TYPE Codes

Code	Meaning
K	Keyword
N	ASCII representation of a numeric literal
S	Alphanumeric string
E	Enumeration value name
J	National string

DICTOUF (Object Usage File)

DICTOUF (Object Usage File) is a key-sequenced file that contains one record for each object that is used by another object. This file indicates which objects are used by which other objects. For example, in the following statements, definition B uses definition A:

```
DEF A  TYPE BINARY 16.
DEF B  TYPE A.
```

DICTOUF is different on G-series and H-series systems—see:

- Figure D-15, DICTOUF (Object Usage File)—G-Series, on page D-45
- Figure D-16, DICTOUF (Object Usage File)—H-Series, on page D-46

Change bars in Figure D-16 on page D-46 show where it differs from Figure D-15 on page D-45.

These tables apply to both G-series and H-series systems:

- Table D-18, DICTOUF (Object Usage File) Fields, on page D-46
- Table D-19, OBJECT-TYPE Codes, on page D-47

Figure D-15. DICTOUF (Object Usage File)—G-Series (page 1 of 2)

```
Record OUF.
  File is "DICTOUF"                    Key-sequenced
                                       Code 208
                                       Audit.

  02 IDENTIFIER.

    03 OBJECT-USED                     Null 0
                                       Heading "Object/Used".

      04 OBJ-TYPE                      Type Character 2
                                       MUST BE "ID","RD","CD" "TT",
                                                    "TC","TM","SR","SV",
                                                    "RQ","TB"
                                       UPSHIFT
                                       Heading "Obj/Type".

      04 NAME                          Type Character 30
                                       Heading "Object Name".
```

Figure D-15. DICTOUF (Object Usage File)—G-Series (page 2 of 2)

```
    03 CONSUMER                       Null 0
                                      Heading "Consumer/Object".

      04 OBJ-TYPE                     Type Character 2
                                      MUST BE "ID","RD","CD" "TT",
                                                "TC", TM","SR","SV",
                                                "RQ","TB"
                                      UPSHIFT
                                      Heading "Obj/Type".

      04 NAME                         Type Character 30
                                      Heading "Object Name".

  02 REPLACEMENT-ALLOWED              Type Character 1.
    88 REPLACEMENT-IS-ALLOWED         Value is "Y".
    88 REPLACEMENT-NOT-ALLOWED        Value is "N".

  Key is IDENTIFIER Duplicates not allowed.
  Key "OC" is CONSUMER.

End
```

Figure D-16. DICTOUF (Object Usage File)—H-Series

```
Record OUF.
  File is "DICTOUF"                   Key-sequenced
                                      Code 208
                                      Audit
                                      MaxExtents 500.

 02 IDENTIFIER.

    03 OBJECT-USED                    Type OBJECT-IDENTIFIER
                                      Null 0
                                      Heading "Object/Used".

    03 CONSUMER                       Type OBJECT-IDENTIFIER
                                      Null 0
                                      Heading "Consumer/Object".

  02 REPLACEMENT-ALLOWED              Type Character 1.
    88 REPLACEMENT-IS-ALLOWED         Value is "Y".
    88 REPLACEMENT-NOT-ALLOWED        Value is "N".

  Key is IDENTIFIER Duplicates not allowed.
  Key "OC" is CONSUMER File is "DICTOUK".

End
```

Table D-18. DICTOUF (Object Usage File) Fields (page 1 of 2)

Field	Description
IDENTIFIER	The unique key of the OUF record, consisting of the next two group fields, OBJECT-USED and CONSUMER.
OBJECT-USED	A group field that identifies the object being used, consisting of the fields OBJECT-TYPE and NAME.
CONSUMER	A group field that identifies the using object (or consumer), consisting of the fields OBJECT-TYPE and NAME.

Table D-18. DICTOUF (Object Usage File) Fields (page 2 of 2)

Field	Description
OBJECT-TYPE	Contains a 2-character ASCII code from Table D-19 on page D-47, which indicates the type of the object being used in the DDL subsystem.
NAME	Contains the name of the object.
REPLACEMENT-ALLOWED	Contains the ASCII character Y (yes) to indicate that the object used can be replaced or deleted even if the using object (consumer) is still in the dictionary; or N (no) to indicate that the using object (consumer) must be deleted before the object used can be replaced or deleted. For DDL objects, this field is set to N. The Pathmaker application generator does not currently use this file.

Table D-19. OBJECT-TYPE Codes

Code	Object
ID	Definition
RD	Record
CD	Constant
TT	SPI Token Type
TC	SPI Token Code
TM	SPI Token Map
SR	Server
SV	Service
RQ	Requester
TB	Table

DICTOUK (Object Usage Key File)

DICTOUK (Object Usage Key File) is a key-sequenced file that contains alternate keys for DICTOUF (Object Usage File) on page D-45.

DICTRDF (Record Definition File)

DICTRDF (Record Definition File) is a key-sequenced file that contains one record for each record in the dictionary. Each RDF record contains the object number, definition number, file name, and file type of the dictionary record.

DICTRDF is different on G-series and H-series systems—see:

- Figure D-17, DICTRDF (Record Definition File)—G-Series, on page D-48
- Figure D-18, DICTRDF (Record Definition File)—H-Series, on page D-50

Change bars in Figure D-18 on page D-50 show where it differs from Figure D-17 on page D-48.

These tables apply to both G-series and H-series systems:

- Table D-20, DICTRDF (Record Definition File) Fields, on page D-53
- Table D-21, FILE-TYPE Codes, on page D-55
- Table D-22, FILE-DURATION Values, on page D-55

Figure D-17. DICTRDF (Record Definition File)—G-Series (page 1 of 3)

```
Record RDF.
  File is "DICTRDF"                    Key-sequenced
                                       Code 205
                                       Audit.

  02 OBJECT                            Pic "9(9)" COMP
                                       Null 0
                                       Display "[BZ]I10"
                                       Heading "Record/Number".

  02 DEF-NUMBER                        Pic "9(9)" COMP
                                       Null 0
                                       Display "[BZ]I10"
                                       Heading "Def/Number".

  02 RECORD-LENGTH                     Type Binary 16
                                       Heading "Record/Length".

  02 FILE-NAME                         Type Character 34
                                       UPSHIFT
                                       Heading "Physical File Name".

  02 FILE-TYPE                         Type Character 1
                                       Heading "File/Type".

  02 FILE-DURATION                     Type Character 1
                                       VALUE "P"
                                       Heading "File/Dur".

  02 FILE-CODE                         Type Binary 16 Unsigned
                                       VALUE 0
                                       Display "I5"
                                       Heading "File/Code".

  02 FILE-CODE-OBJECT                  Pic "9(9)" COMP
                                       VALUE 0
                                       Null 0
                                       Display "I5"
                                       Heading "File Code/Object".

  02 PRIMARY-EXTENT-SIZE               Type Binary 16
                                       VALUE 4
                                       Display "I5"
                                       Heading "Primary/Ext Size".

  02 PRIMARY-EXTENT-OBJECT             Pic "9(9)" COMP
                                       VALUE 0
                                       Null 0
                                       Display "I5"
                                       Heading "Pri Ext/Object".
```

Figure D-17. DICTRDF (Record Definition File)—G-Series (page 2 of 3)

```
02 SECONDARY-EXTENT-SIZE           Type Binary 16
                                   VALUE 32
                                   Display "I5"
                                   Heading "Secondary/Ext Size".

02 SECONDARY-EXTENT-OBJECT         Pic "9(9)" COMP
                                   VALUE 0
                                   Null 0
                                   Display "I5"
                                   Heading "Sec Ext/Object".

02 REFRESH                         Type Character 1
                                   MUST BE "Y", "N"
                                   UPSHIFT
                                   Display "A1"
                                   VALUE "N"
                                   Heading "Refresh".

02 AUDIT                           Type Character 1
                                   MUST BE "Y", "N"
                                   UPSHIFT
                                   Display "A1"
                                   VALUE "N"
                                   Heading "Audit".

02 BLOCK-SIZE                      Type Binary 16
                                   VALUE 4096
                                   Display "I4"
                                   Heading "Block/Size".

02 BUFFER-SIZE Redefines BLOCK-SIZE Type Binary 16
                                   Display "I4"
                                   Heading "Buffer/Size".

02 BLOCK-SIZE-OBJECT               Pic "9(9)" COMP
                                   Display "[BZ]I10"
                                   VALUE 0
                                   Null 0
                                   Display "I5"
                                   Heading "Blk Siz/Object".

02 BUFFER-SIZE-OBJECT              Pic "9(9)" COMP
                                   Display "[BZ]I10"
                                   VALUE 0
                                   Null 0
                                   Display "I5"
                                   Heading "Buf Siz/Object".

02 ICOMPRESS                       Type Character 1
                                   MUST BE "Y", "N"
                                   UPSHIFT
                                   Display "A1"
                                   VALUE "N"
                                   Heading "ICompress".

02 DCOMPRESS                       Type Character 1
                                   MUST BE "Y", "N"
                                   UPSHIFT
                                   Display "A1"
                                   VALUE "N"
                                   Heading "DCompress".
```

Figure D-17. DICTRDF (Record Definition File)—G-Series (page 3 of 3)

```
  02 MAXEXTENTS                       Type Binary 16
                                      VALUE 100
                                      Display "I4"
                                      Heading "Maxextents".

  02 MAXEXTENTS-OBJECT                Pic "9(9)" COMP
                                      VALUE 0
                                      Null 0
                                      Display "I5"
                                      Heading "Max Ext/Object".

  02 BUFFERED                         Type Character 1
                                      VALUE "D"
                                      Display "A1"
                                      Heading "Buffered".

  02 AUDIT-COMPRESS                   Type Character 1
                                      MUST BE "Y", "N"
                                      UPSHIFT
                                      Display "A1"
                                      VALUE "N"
                                      Heading "Audit/Compress".

  02 VERIFIED-WRITES                  Type Character 1
                                      MUST BE "Y", "N"
                                      UPSHIFT
                                      Display "A1"
                                      VALUE "N"
                                      Heading "Verifies/Writes".

  02 SERIAL-WRITES                    Type Character 1
                                      MUST BE "Y", "N"
                                      UPSHIFT
                                      Display "A1"
                                      VALUE "N"
                                      Heading "Serial/Writes".

  02 ODD-UNSTRUCTURED                 Type Character 1
                                      MUST BE "Y", "N"
                                      UPSHIFT
                                      Display "A1"
                                      VALUE "Y"
                                      Heading "Odd/Unstructured".

  Key is OBJECT Duplicates not allowed.
  Key "IN" is DEF-NUMBER.

End
```

Figure D-18. DICTRDF (Record Definition File)—H-Series (page 1 of 3)

```
Record RDF.
  File is "DICTRDF"                   Key-sequenced
                                      Code 205
                                      Audit
                                      MaxExtents 500.

  02 OBJECT                           Type OBJECT-NUMBER
                                      Heading "Record/Number".

  02 DEF-NUMBER                       Type OBJECT-NUMBER
                                      Heading "Def/Number".
```

Figure D-18. DICTRDF (Record Definition File)—H-Series (page 2 of 3)

```
02 RECORD-LENGTH                    Type Binary 16
                                    Heading "Record/Length".

02 FILE-NAME                        Type *.

02 FILE-TYPE                        Type Character 1
                                    Heading "File/Type".

02 FILE-DURATION                    Type Character 1
                                    VALUE "P"
                                    Heading "File/Dur".

02 FILE-CODE                        Type Binary 16 Unsigned
                                    VALUE 0
                                    Display "I5"
                                    Heading "File/Code".

02 FILE-CODE-OBJECT                 Type OBJECT-NUMBER
                                    VALUE 0
                                    Null 0
                                    Display "I5"
                                    Heading "File Code/Object".

02 PRIMARY-EXTENT-SIZE              Type Binary 16
                                    VALUE 4
                                    Display "I5"
                                    Heading "Primary/Ext Size".

02 PRIMARY-EXTENT-OBJECT            Type OBJECT-NUMBER
                                    VALUE 0
                                    Null 0
                                    Display "I5"
                                    Heading "Pri Ext/Object".

02 SECONDARY-EXTENT-SIZE            Type Binary 16
                                    VALUE 32
                                    Display "I5"
                                    Heading "Secondary/Ext Size".

02 SECONDARY-EXTENT-OBJECT          Type OBJECT-NUMBER
                                    VALUE 0
                                    Null 0
                                    Display "I5"
                                    Heading "Sec Ext/Object".

02 REFRESH                          Type ASCII-SWITCH
                                    VALUE "N"
                                    Heading "Refresh".

02 AUDIT                            Type ASCII-SWITCH
                                    VALUE "N"
                                    Heading "Audit".

02 BLOCK-SIZE                       Type Binary 16
                                    VALUE 4096
                                    Display "I4"
                                    Heading "Block/Size".

02 BUFFER-SIZE Redefines BLOCK-SIZE Type Binary 16
                                    Display "I4"
                                    Heading "Buffer/Size".
```

Figure D-18. DICTRDF (Record Definition File)—H-Series (page 3 of 3)

```
  02 BLOCK-SIZE-OBJECT                OBJECT-NUMBER
                                      VALUE 0
                                      Null 0
                                      Display "I5"
                                      Heading "Blk Siz/Object".

  02 BUFFER-SIZE-OBJECT               OBJECT-NUMBER
                                      VALUE 0
                                      Null 0
                                      Display "I5"
                                      Heading "Buf Siz/Object".

  02 ICOMPRESS                        Type ASCII-SWITCH
                                      VALUE "N"
                                      Heading "ICompress".

  02 DCOMPRESS                        Type ASCII-SWITCH
                                      VALUE "N"
                                      Heading "DCompress".

  02 MAXEXTENTS                       Type Binary 16
                                      VALUE 100
                                      Display "I4"
                                      Heading "Maxextents".

  02 MAXEXTENTS-OBJECT                Type OBJECT-NUMBER
                                      VALUE 0
                                      Null 0
                                      Display "I5"
                                      Heading "Max Ext/Object".

  02 BUFFERED                         Type Character 1
                                      VALUE "D"
                                      Display "A1"
                                      Heading "Buffered".

  02 AUDIT-COMPRESS                   Type ASCII-SWITCH
                                      VALUE "N"
                                      Heading "Audit/Compress".

  02 VERIFIED-WRITES                  Type ASCII-SWITCH
                                      VALUE "N"
                                      Heading "Verifies/Writes".

  02 SERIAL-WRITES                    Type ASCII-SWITCH
                                      VALUE "N"
                                      Heading "Serial/Writes".

  02 ODD-UNSTRUCTURED                 Type ASCII-SWITCH
                                      VALUE "Y"
                                      Heading "Odd/Unstructured".

  Key is OBJECT Duplicates not allowed.
  Key "IN" is DEF-NUMBER File is "DICTALT".

End
```

Table D-20. DICTRDF (Record Definition File) Fields (page 1 of 3)

Field	Description
OBJECT	Contains the object number of this record from DICTODF.OBJECT. The record in DICTODF contains the record name and the object-type code “RD.”
DEF-NUMBER	The object number of the definition that defines this record if the record is described with a DEFINITION IS def-name clause; otherwise, DEF-NUMBER contains the object number of the record itself.
RECORD-LENGTH	The length in bytes of the record.
FILE-NAME	This record's permanent HP file name, stored in FNAMECOLLAPSE form. For a description of FNAMECOLLAPSE, see the *Guardian Procedure Calls Reference Manual*. This field is defined only if FILE-DURATION is permanent.
FILE-TYPE	Contains a 1-character ASCII code from Table D-21 on page D-55, which indicates the record's file type.
FILE-DURATION	A value that indicates whether the file specified by FILE-NAME is permanent, dynamically assigned, or temporary. FILE-DURATION values are in Table D-22 on page D-55.
FILE-CODE	This record's file code. The default value for a user-created file is 0.
FILE-CODE-OBJECT	If file code is defined by a constant, contains the object number of the constant; otherwise, it is 0.
PRIMARY-EXTENT-SIZE	This record's primary file extent in pages. PRIMARY-EXTENT-SIZE must be an integer from 1 through 65,535. The default primary extent size for DDL is four pages. The extent size must be an integral multiple of the file's block size (for a structured file) or buffer size (for an unstructured file). For more information about extent sizes, see the *Enscribe Programmer’s Guide*.
PRIMARY-EXTENT-OBJECT	If primary extent size is defined by a constant, contains the object number of the constant; otherwise, it is 0.

Table D-20. DICTRDF (Record Definition File) Fields (page 2 of 3)

Field	Description
SECONDARY-EXTENT-SIZE	This file's secondary extent in pages. SECONDARY-EXTENT-SIZE must be an integer from 1 through 65,535. The default secondary extent size for DDL is 32 pages. Like the primary extent size, the secondary extent size must be an integral multiple of the file's block size (for a structured file) or buffer size (for an unstructured file).
SECONDARY-EXTENT-SIZE-OBJECT	If secondary extent size is defined by a constant, contains the object number of the constant; otherwise, it is 0.
REFRESH	Contains the ASCII character Y (yes) or N (no) to indicate whether the file's label will be copied to disk whenever the file's end-of-file value is changed. N is the default.
AUDIT	Contains the ASCII character Y (yes) or N (no) to indicate whether a file is audited by TMF. N is the default.
BLOCK-SIZE	Block size of a structured file in bytes. BLOCK-SIZE must be 512, 1,024, 2,048, or 4,096 bytes. The default block size for DDL is 4,096 bytes. For information about block sizes, see the *Enscribe Programmer's Guide*.
BUFFER-SIZE	Buffer size of an unstructured file in bytes. BUFFER-SIZE redefines BLOCK-SIZE. Value must be 512, 1,024, 2,048, or 4,096 bytes. 4,096 bytes is the default.
BLOCK-SIZE-OBJECT	If block size is defined by a constant, contains the object number of the constant; otherwise, it is 0.
BUFFER-SIZE-OBJECT	If buffer size is defined by a constant, contains the object number of the constant; otherwise, it is 0.
ICOMPRESS	Contains the ASCII character Y (yes) or N (no) to indicate whether the user has selected the index compression attribute for this file. N is the default.
DCOMPRESS	Contains the ASCII character Y (yes) or N to indicate whether the user has selected the data compression attribute for this file. N is the default.
MAXEXTENTS	The maximum number of extents this file can have. MAXEXTENTS contains an integer from 1 through 978. 100 is the default.
MAXEXTENTS-OBJECT	If MAXEXTENTS is defined by a constant, contains the object number of the constant; otherwise, it is 0.

Table D-20. DICTRDF (Record Definition File) Fields (page 3 of 3)

Field	Description
BUFFERED	Indicates the mode of handling write requests. BUFFERED can be Y (yes), N (no), or D (follow the default). The default value is Y for audited files and N for nonaudited files. If you select Y, then write requests are buffered in the disk-process cache rather than forced to disk at each request.
AUDIT-COMPRESS	Contains the ASCII character Y (yes) to indicate the audit-checkpoint record is to be compressed or N (no) to indicate it is not to be compressed. The audit-checkpoint record contains a copy of an audited data record both before and after an update. The audit-checkpoint record is compressed by omitting the unchanged portions of the data record. N is the default.
VERIFIED-WRITES	Contains Y (yes) to indicate disk writes are verified or N (no) to indicate they are not. N is the default.
SERIAL-WRITES	Contains Y (yes–serial) or N (no–parallel) to indicate whether mirror disk writes are serial or parallel. N is the default.
ODD-UNSTRUCTURED	Contains Y (yes–odd unstructured) or N (no–even unstructured) to indicate whether the file is to be created as odd unstructured or even unstructured. Y is the default. For information about even unstructured and odd unstructured files, see the *File Utility Program (FUP) Reference Manual*.

Table D-21. FILE-TYPE Codes

Code	File Type
U	Unstructured
R	Relative
E	Entry-sequenced
K	Key-sequenced

Table D-22. FILE-DURATION Values

Value	File Type
P	Permanent (default)
T	Temporary
A	Assigned

DICTTKN (Token Code File)

DICTTKN (Token Code File) is a key-sequenced file that contains one record for each SPI token code. Each record contains the detailed information about a token code, including the object number of the token code, the object number of the associated token type, and the value of the token number that identifies the token code within its type.

DICTTKN is different on G-series and H-series systems—see:

- Figure D-19, DICTTKN (Token Code File)—G-Series, on page D-56
- Figure D-20, DICTTKN (Token Code File)—H-Series, on page D-57

Change bars in Figure D-20 on page D-57 show where it differs from Figure D-19 on page D-56.

Table D-23, DICTTKN (Token Code File) Fields, on page D-57 applies to both G-series and H-series systems.

Figure D-19. DICTTKN (Token Code File)—G-Series (page 1 of 2)

```
Record TKN.
  File is "DICTTKN"                    Key-sequenced
                                       Code 209
                                       Audit.

  02 OBJECT-NUMBER                     Pic "9(9)" COMP
                                       Null 0
                                       Display "[BZ]I10"
                                       Heading "Token Code/Object".

  02 TOKEN-TYPE-OBJECT                 Pic "9(9)" COMP
                                       Null 0
                                       Display "[BZ]I10"
                                       Heading "Token Type/Object".

  02 TOKEN-NUMBER-VALUE                Type Binary 16
                                       Heading "Token Numb".

  02 TOKEN-NUMBER-CONSTANT             Pic "9(9)" COMP
                                       Null 0
                                       Display "[BZ]I10"
                                       Heading "Token Numb/Object".

  02 SSID-TEXT                         Pic "9(9)" COMP
                                       Null 0
                                       Display "[BZ]I10"
                                       Heading "SSID".

  02 HEADING-TEXT                      Pic "9(9)" COMP
                                       Null 0
                                       Display "[BZ]I10"
                                       Heading "Heading".
```

Figure D-19. DICTTKN (Token Code File)—G-Series (page 2 of 2)

```
  02 DISPLAY-TEXT                    Pic "9(9)" COMP
                                     Null 0
                                     Display "[BZ]I10"
                                     Heading "Display".

  Key is OBJECT-NUMBER Duplicates not allowed.

End
```

Figure D-20. DICTTKN (Token Code File)—H-Series

```
Record TKN.
  File is "DICTTKN"                  Key-sequenced
                                     Code 209
                                     Audit.
                                     MaxExtents 500.

  02 OBJECT-NUMBER                   Type *
                                     Heading "Token Code/Object".

  02 TOKEN-TYPE-OBJECT               Type OBJECT-NUMBER
                                     Heading "Token Type/Object".

  02 TOKEN-NUMBER-VALUE              Type Binary 16
                                     Heading "Token Numb".

  02 TOKEN-NUMBER-CONSTANT           Type OBJECT-NUMBER
                                     Heading "Token Numb/Object".

  02 SSID-TEXT                       Type TEXT-ID
                                     Heading "SSID".

  02 HEADING-TEXT                    Type TEXT-ID
                                     Heading "Heading".

  02 DISPLAY-TEXT                    Type TEXT-ID
                                     Heading "Display".

  Key is OBJECT-NUMBER Duplicates not allowed.

End
```

Table D-23. DICTTKN (Token Code File) Fields (page 1 of 2)

Field	Description
OBJECT-NUMBER	Contains the object number of this record from DICTODF.OBJECT. The record in DICTODF contains the token-code name and its object-type code, “TC.”
TOKEN-TYPE-OBJECT	Contains the object number of the SPI token type for the token code.

Table D-23. DICTTKN (Token Code File) Fields (page 2 of 2)

Field	Description
TOKEN-NUMBER-VALUE	Contains the token number used by a subsystem to distinguish one token code from another. Token numbers can be in the range -32,768 through 32,767. Any user-supplied token numbers must be in the range 1 through 9,998; the other token numbers are reserved by HP or are previously defined by SPI.
TOKEN-NUMBER-CONSTANT	If the token number is specified as a constant, this field contains the object number of the constant; otherwise, it is 0.
SSID-TEXT	Contains the text ID of the OTF record that contains the subsystem ID value for the token code.
HEADING-TEXT	Contains the text ID of the OTF record that contains the heading value for the token code.
DISPLAY-TEXT	Contains the text ID of the OTF record that contains the display value for the token code.

DICTTYP (Token Type File)

DICTTYP (Token Type File) is a key-sequenced file that contains one record for each SPI token type. Each record contains fields to specify the object number of the token type, its token data type, its structure, and the token length.

DICTTYP is different on G-series and H-series systems—see:

- Figure D-21, DICTTYP (Token Type File)—G-Series, on page D-58
- Figure D-22, DICTTYP (Token Type File)—H-Series, on page D-59

Change bars in Figure D-22 on page D-59 show where it differs from Figure D-21 on page D-58.

These tables apply to both G-series and H-series systems:

- Table D-24, DICTTYP (Token Type File) Fields, on page D-60
- Table D-25, TOKEN-OCCURS-VALUE Values, on page D-61

Figure D-21. DICTTYP (Token Type File)—G-Series (page 1 of 2)

```
Record TYP.
  File is "DICTTYP"                  Key-sequenced
                                     Code 209
                                     Audit.

  02 OBJECT-NUMBER                   Type *
                                     Heading "Token Type/Object".

  02 TOKEN-TYPE-VALUE                Type Binary 16
                                     Heading "Token/Value".
```

Figure D-21. DICTTYP (Token Type File)—G-Series (page 2 of 2)

```
  02 TOKEN-TYPE-CONSTANT              Pic "9(9)" COMP
                                      Null 0
                                      Display "[BZ]I10"
                                      Heading "Constant/Object".

  02 TOKEN-DEF                        Pic "9(9)" COMP
                                      Null 0
                                      Display "[BZ]I10"
                                      Heading "Def/Object".

  02 TOKEN-OCCURS-VALUE               Type Binary 16
                                      Heading "Token/Occurs".
    88 OCCURS-VARYING                 Value is -1.
    88 OCCURS-0                       Value is 0.

  02 TOKEN-OCCURS-CONSTANT            Pic "9(9)" COMP
                                      Null 0
                                      Display "[BZ]I10"
                                      Heading "Occurs/Object".

  02 STRUCTURE                        Type Binary 16
                                      Heading "Structure".

  02 TOKEN-LENGTH                     Type Binary 16 Unsigned
                                      Heading "Token/Length".

  Key is OBJECT-NUMBER Duplicates not allowed.

End
```

Figure D-22. DICTTYP (Token Type File)—H-Series (page 1 of 2)

```
Record TYP.
  File is "DICTTYP"                   Key-sequenced
                                      Code 209
                                      Audit
                                      MaxExtents 500.

  02 OBJECT-NUMBER                    Type *
                                      Heading "Token Type/Object".

  02 TOKEN-TYPE-VALUE                 Type Binary 16
                                      Heading "Token/Value".

  02 TOKEN-TYPE-CONSTANT              Type OBJECT-NUMBER
                                      Heading "Constant/Object".

  02 TOKEN-DEF                        Type OBJECT-NUMBER
                                      Heading "Def/Object".

  02 TOKEN-OCCURS-VALUE               Type Binary 16
                                      Heading "Token/Occurs".
    88 OCCURS-VARYING                 Value is -1.
    88 OCCURS-0                       Value is 0.

  02 TOKEN-OCCURS-CONSTANT            Type OBJECT-NUMBER
                                      Heading "Occurs/Object".

  02 STRUCTURE                        Type Binary 16
                                      Heading "Structure".
```

Figure D-22. DICTTYP (Token Type File)—H-Series (page 2 of 2)

```
  02 TOKEN-LENGTH                    Type Binary 16 Unsigned
                                     Heading "Token/Length".

  Key is OBJECT-NUMBER Duplicates not allowed.

End
```

Table D-24. DICTTYP (Token Type File) Fields

Field	Description
OBJECT-NUMBER	Contains the object number of this record from DICTODF.OBJECT. A record in DICTODF contains the token-type name and its object-type code, “TT.”
TOKEN-TYPE-VALUE	Contains the numeric value used by subsystems to identify the token type. This value must be a positive integer in the range 0 through 254.
TOKEN-TYPE-CONSTANT	If TOKEN-TYPE-VALUE is specified as a constant, this field contains the object number of the constant; otherwise, it is 0.
TOKEN-DEF	If the token structure is defined by reference to a definition, this field contains the object number of the specified definition; otherwise, it is 0.
TOKEN-OCCURS-VALUE	Specifies the number of times the token data structure occurs; possible values are in Table D-25 on page D-61.
TOKEN-OCCURS-CONSTANT	If the number of occurrences in an OCCURS clause is specified as a constant, this field contains the object number of the constant; otherwise, it is 0.
STRUCTURE	If the structure of the token is defined by reference to a definition, this field contains a code indicating the data type of the first element of the definition; otherwise, this field is set to 0. For a description of the possible codes this field can contain and their meanings, see DICTOBL (Object Build List) on page D-15.
TOKEN-LENGTH	Contains the length of the token derived from either the TOKEN-OCCURS-VALUE or the STRUCTURE field, as follows: ● If TOKEN-OCCURS-VALUE is a positive integer in the range 1 through 254 and if a DEF IS clause was specified, the length from the definition (DEF) is multiplied by the OCCURS value and stored in TOKEN-LENGTH. ● If no DEF IS clause was specified, the TOKEN-OCCURS-VALUE is stored in TOKEN-LENGTH. A token length of 0 or -1 is considered valid.

Table D-25. TOKEN-OCCURS-VALUE Values

Value	Meaning
255	OCCURS VARYING
0	OCCURS 0 TIMES
n	OCCURS *n* TIMES when 1 <=*n* <=254

DICTVER (Token Map Field Version File)

DICTVER (Token Map Field Version File) is a key-sequenced file that associates product version numbers from VERSION clauses in an SPI token-map definition with single fields or sequences of fields in a structured token.

DICTVER is different on G-series and H-series systems—see:

- Figure D-23, DICTVER (Token Map Field Version File)—G-Series, on page D-61
- Figure D-24, DICTVER (Token Map Field Version File)—H-Series, on page D-62

Change bars in Figure D-24 on page D-62 show where it differs from Figure D-23 on page D-61.

Table D-26, DICTVER (Token Map Field Version File) Fields, on page D-62 applies to both G-series and H-series systems.

Figure D-23. DICTVER (Token Map Field Version File)—G-Series

```
Record VER.
  File is "DICTVER"                    Key-sequenced
                                       Code 209
                                       Audit.

  02 IDENTIFIER.

    03 MAP-OBJECT                      Pic "9(9)" COMP
                                       Null 0
                                       Display "[BZ]I10"
                                       Heading "Token Map/Object".

    03 MAP-ELEMENT                     Type Binary 16
                                       Heading "Element".

  02 VERSION                           Type Binary 16 Unsigned
                                       Heading "Version".

  02 VERSION-CONSTANT                  Pic "9(9)" COMP
                                       Null 0
                                       Display "[BZ]I10"
                                       Heading "Version/Object".

  02 BEG-ELEMENT                       Type Binary 16
                                       Heading "Beginning/Element".

  02 END-ELEMENT                       Type Binary 16
                                       Heading "Ending/Element".
```

Figure D-23. DICTVER (Token Map Field Version File)—G-Series

```
  02 VERSION-TEXT                    Type Character 3
                                     Heading "Version/String".

  Key is IDENTIFIER Duplicates not allowed.

End
```

Figure D-24. DICTVER (Token Map Field Version File)—H-Series

```
Record VER.
  File is "DICTVER"                  Key-sequenced
                                     Code 209
                                     Audit
                                     MaxExtents 500.

  02 IDENTIFIER.

    03 MAP-OBJECT                    Type OBJECT-NUMBER
                                     Heading "Token Map/Object".

    03 MAP-ELEMENT                   Type Binary 16
                                     Heading "Element".

  02 VERSION                         Type Binary 16 Unsigned
                                     Heading "Version".

  02 VERSION-CONSTANT                Type OBJECT-NUMBER
                                     Heading "Version/Object".

  02 BEG-ELEMENT                     Type Binary 16
                                     Heading "Beginning/Element".

  02 END-ELEMENT                     Type Binary 16
                                     Heading "Ending/Element".

  02 VERSION-TEXT                    Type Character 3
                                     Heading "Version/String".

  Key is IDENTIFIER Duplicates not allowed.

End
```

Table D-26. DICTVER (Token Map Field Version File) Fields (page 1 of 2)

Field	Description
IDENTIFIER	Contains a unique identifier for each record consisting of two fields, MAP-OBJECT and MAP-ELEMENT.
MAP-OBJECT	Contains the object number that uniquely identifies the token map (OBJECT-NUMBER from DICTMAP).
MAP-ELEMENT	Contains an element number sequentially assigned by the DDL compiler.
VERSION	Contains the product version number from the VERSION clause of the TOKEN-MAP statement; or 0 for NOVERSION.
VERSION-CONSTANT	If the product version number was specified as a constant, this field contains the object number of the constant; otherwise, it is 0.

Table D-26. DICTVER (Token Map Field Version File) Fields (page 2 of 2)

Field	Description
BEG-ELEMENT	Contains the element number from DICTOBL for the first element specified in the sequence of fields with this product version.
END-ELEMENT	Contains the element number from DICTOBL for the last element in the sequence of fields with this product version.
VERSION-TEXT	Contains the 3-character product version number in the form ann, in which a is a letter of the alphabet and nn is a 2-digit number.

Definition and Record Storage in the Dictionary

This topic explains how the dictionary database files are structured, showing how definitions and records are stored in the dictionary. For simplicity, only a subset of the dictionary fields is shown. The focus is on primary and alternate key fields, because these fields show how the files are related.

A schema consisting of two objects, a definition and a record, is used to construct a sample dictionary (see Example D-4 on page D-63).

Example D-4. Sample Dictionary Schema for a Definition and a Record

```
DEF partname          PIC X (18)
                      HEADING "Part/Name".

RECORD parts.

FILE IS "$data.sales.parts" KEY-SEQUENCED
  02 PARTNAME    TYPE *.
  02 inventory   PIC 999 COMP
                 VALUE ALL ZEROES
  02 location    PIC XXX
                 88 san-francisco
                 UPSHIFT VALUE "SFO".

  KEY IS parts.partname.
END
```

Topics:

- DICTKDF (Key Definition File) on page D-67
- Dictionary Structure Link Diagram on page D-68

DICTDDF (Dictionary Definition File)

DICTDDF, an unstructured file, contains only one record. The most important fields in this file for record and definition storage are NEXT-OBJ and NEXT-TEXT-ID.

Field	Description
NEXT-OBJ	NEXT-OBJ is used by the DDL compiler to assign object numbers to objects as they are entered in the dictionary. NEXT-OBJ has an initial value of 1. When an object is entered in the dictionary, it is given the current value of NEXT-OBJ. NEXT-OBJ is then incremented by 1.
NEXT-TEXT-ID	NEXT-TEXT-ID is used by the DDL compiler to assign text numbers to text items as they are added to the dictionary. Like NEXT-OBJ, NEXT-TEXT-ID has an initial value of 1 and is incremented after a text item is entered in the dictionary.
VERSION	VERSION contains the product version number of the dictionary. This value will not change unless you regenerate the dictionary with a different product version of the DDL compiler.

After PARTNAME and PARTS from Example D-4 on page D-63 are added to a new dictionary, the fields of DICTDDF have these values:

Field	Value
NEXT-OBJ	3
NEXT-TEXT-ID	7
VERSION	4

DICTODF (Object Definition File)

DICTODF, a key-sequenced file, contains one record for every object (definition, record, service, server, requester, screen) entered in the dictionary. The three most important fields of this file are OBJECT, IDENTIFIER.NAME, and IDENTIFIER.OBJECT-TYPE.

For the sample dictionary in Example D-4 on page D-63, the fields of DICTODF have these values:

OBJECT	IDENTIFIER.OBJECT-TYPE	IDENTIFIER.NAME	COMMENT-TEXT
1	ID	partname	0
2	RD	parts	0

DICTOBL (Object Build List)

DICTOBL, a key-sequenced file, contains one record for each element of each DDL object (record or definition) in the dictionary. The primary key of DICTOBL is the object number (IDENTIFIER.OBJECT) and an element number (IDENTIFIER.ELEMENT) that identifies each element within an object. Each DICTOBL record contains most of the information needed to describe an element: the element's name, data type, size, offset within the object, text ID number, and other information.

For the sample dictionary in Example D-4 on page D-63, the fields of DICTOBL have these values:

IDENTIFIER. OBJECT	IDENTIFIER. ELEMENT	LOCAL-NAME	PICTURE-TEXT	HEADING-TEXT	VALUE-TEXT	UPSHIFT
1	0	partname	1	2		N
2	0	parts				N
2	1	partname				N
2	2	standard-price	3		4	N
2	3	location	5			Y
2	4	san-francisco			6	N

DICTOTF (Object Text File)

DICTOTF contains one record for each text item entered in the dictionary. Text items can have one of these types:

Code	Type
N	ASCII representation of a number
K	Keyword in a MUST BE or VALUE clause
S	ASCII character string
E	Enumeration value name
J	National string

For more information about text types, see Text Items on page D-2.

For the sample dictionary shown in Example D-4 on page D-63, the fields of DICTOTF have these values:

IDENTIFIER.TEXT-ID	IDENTIFIER.LINE-NUMBER	TEXT-TYPE	TEXT-LEN	TEXT-LINE
1	0	S	5	X (18)
2	0	S	9	Part/Name
3	0	N	3	999
4	0	K	3	ALL
4	1	K	6	ZEROES
5	0	S	3	XXX
6	0	S	3	SFO

DICTRDF (Record Definition File)

DICTRDF has one record for each record in the dictionary containing the record's object number, definition number, and file information.

Field	Description
OBJECT	The record’s unique object number.
DEF-NUMBER	Either the object number of the record (if it is not defined with a DEFINITION IS *def-name* clause) or (if it is defined with a DEFINITION IS *def-name* clause) the object number of the referenced definition.
FILE-NAME	The file name.

For the sample dictionary in Example D-4 on page D-63, the fields of DICTRDF contain these values:

Field	Value
OBJECT	2
DEF-NUMBER	2
FILE-NAME	$data.sales.parts

Suppose that the following record is added to the dictionary:

```
RECORD newparts.
FILE IS "$data.sales.newpart".
DEFINITION IS partname.
END
```

The fields of DICTRDF have these values:

OBJECT	DEF-NUMBER	FILE-NAME
2	2	$data.sales.parts
3	1	$data.sales.newpart

When a record is declared with a DEFINITION IS clause, DICTOBL has no entry for the new record. Instead, the data structure is found in DICTOBL by looking up the referenced definition number in DICTRDF.

DICTKDF (Key Definition File)

DICTKDF contains one record for each primary key and each alternate key, or each SEQUENCE IS field declared for each record in the dictionary. DICTKDF records are uniquely identified by the object number of the record that contains the key and an element number.

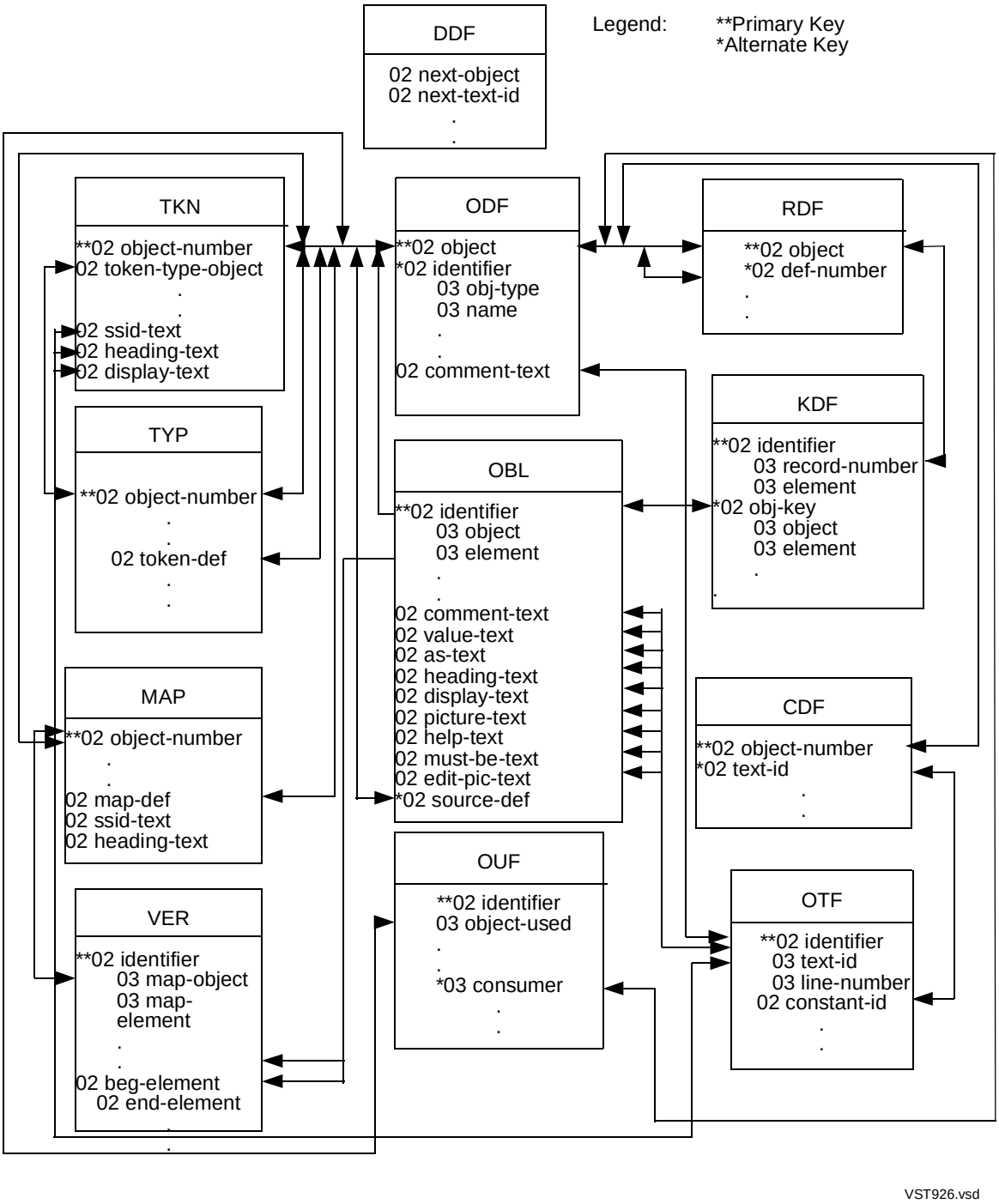
For the sample dictionary shown in Example D-4 on page D-63, the fields of DICTKDF have these values.

Field	Value
IDENTIFIER.RECORD-NUMBER	2
OBL-KEY.OBJECT	2
KEYTAG-VALUE	0

Dictionary Structure Link Diagram

Figure D-25 on page D-68 shows the main database links in the dictionary database.

Figure D-25. Main Links Among Dictionary Files



E Dictionary Reports

HP supplies a set of Enform Plus queries that you can use to get information about any dictionary. These queries produce a set of reports that provide the following information:

- A list of all the objects in the dictionary, including any constants, definitions, records, and Subsystem Programmatic Interface (SPI) token codes, token maps, and token types.
- A description of the structure of each definition and record.
- A list of the records and definitions that refer to definitions and of the definitions that are referenced by records and other definitions.
- Information on all records in the dictionary, their key fields, and how they are defined.
- A list of all the comment, display, and heading text for any definition or record that has such text.

In addition to the standard reports, you can produce customized reports, tailored to answer specific questions, by editing the Enform Plus source code supplied by HP.

This appendix explains how you can obtain dictionary reports.

Topics in this appendix:

Using Enform Plus Queries for Dictionary Reports

The Enform Plus source code for the dictionary reports is stored in the file $SYSTEM.SYSTEM.DDQUERYS.

For any of the standard reports summarized in Table E-1 on page E-2, use the source code as is. For customized reports, copy DDQUERYS to your own subvolume and edit your copy.

DDQUERYS contains source code for 16 Enform Plus queries that produce 16 different dictionary reports. Each query is a separate section. You can run the queries as a complete group, individually, or in any combination.

Table E-1. Dictionary Report Queries (page 1 of 2)

Query Name		Report Description
R1	DICTIONARY OBJECTS	Describes every constant, definition, record, token code, token map, and token type in the dictionary, giving the time and date of its creation, the time and date of its last modification, and its product version number.
R2	DEFINITION STRUCTURE	Lists every component group and field in every definition in the dictionary.
R3	RECORD STRUCTURE	Lists every component group and field in every record in the dictionary.
R4	DEFINITIONS USING DEFINITIONS	Lists every definition and every element within that definition that refers to another definition and lists the source definition for each reference.
R5	RECORDS USING DEFINITIONS	Lists every record and every element within that record that refers to a definition and lists the source definition for each reference.
R6	DEFINITIONS WHERE USED	Lists every definition that is referenced by a record or by another definition and lists the referring definition or record.
R7	RECORD ACCESS	Lists the file name and any primary and alternate keys for each record in the dictionary.
R8	RECORD DEFINITION METHOD	Shows the method used to define each record and gives the source definition for any record defined with a DEFINITION IS *def-name* clause.
R9	REPORT HEADINGS	Lists the Enform Plus report heading for any field or group that is defined with a HEADING clause.
R10	DISPLAY FORMATS	Lists the Enform Plus display format for any field or group that is defined with a DISPLAY clause.
R11	RECORD COMMENTS	Lists all comments that immediately precede any record in the dictionary.
R12	DEFINITION COMMENTS	Lists all comments that immediately precede any definition in the dictionary.
R13	CONSTANTS	Lists the type and value of each constant.

Table E-1. Dictionary Report Queries (page 2 of 2)

Query Name		Report Description
R14	TOKEN CODES	Lists the token type, value, and subsystem ID of each token code.
R15	TOKEN MAPS	Lists the value, definition, subsystem ID, and product version of each token map.
R16	TOKEN TYPES	Lists the value, definition, number of occurrences of the definition, and length of each token type.

Each report begins with a brief description of what the report does and the meanings of the report fields.

Producing Dictionary Reports

The Enform Plus report queries use the dictionary that describes the structure of the 14 dictionary files. This dictionary must be compiled before you can produce the Enform Plus dictionary reports. For a description of this dictionary, see Appendix D, Dictionary Database Structure.

After the dictionary has been compiled, you can run any or all of the 16 Enform Plus queries to report on any dictionary on any subvolume in the system.

Both the dictionary schema (DDSCHEMA) and the Enform Plus source (DDQUERYS) reside on $SYSTEM.SYSTEM. When DDSCHEMA is compiled, the DDL compiler creates the dictionary on the subvolume $SYSTEM.DDL.

To produce a report:

1. Compile the dictionary schema, thereby creating a dictionary for the 14 dictionary files. (For details, see Compiling the Dictionary Schema on page E-4.)
2. Establish your dictionary volume and subvolume as the default for the terminal on which you request the reports.
3. Generate reports.

   To generate all 16 reports, run the Enform Plus program noninteractively using $SYSTEM.SYSTEM.DDQUERYS as the source file.

   To select particular reports, run the Enform Plus program interactively and specify only the $SYSTEM.SYSTEM.DDQUERYS sections that generate the reports you want.

   For details, see Requesting Reports on page E-5.

Compiling the Dictionary Schema

Each DDQUERYS query begins with this statement:

```
?DICTIONARY $SYSTEM.DDL
```

The statement opens the dictionary on subvolume $SYSTEM.DDL. Before you can run any query, you must create a dictionary describing the dictionary files on subvolume $SYSTEM.DDL. You can create the dictionary on any volume that does not already have a dictionary, but to conform to the Enform Plus query, you must create it on subvolume $SYSTEM.DDL. You create the dictionary by compiling DDSCHEMA for example:

```
13> DDL/IN $system.system.ddschema, OUT/ DICT $system.ddl
```

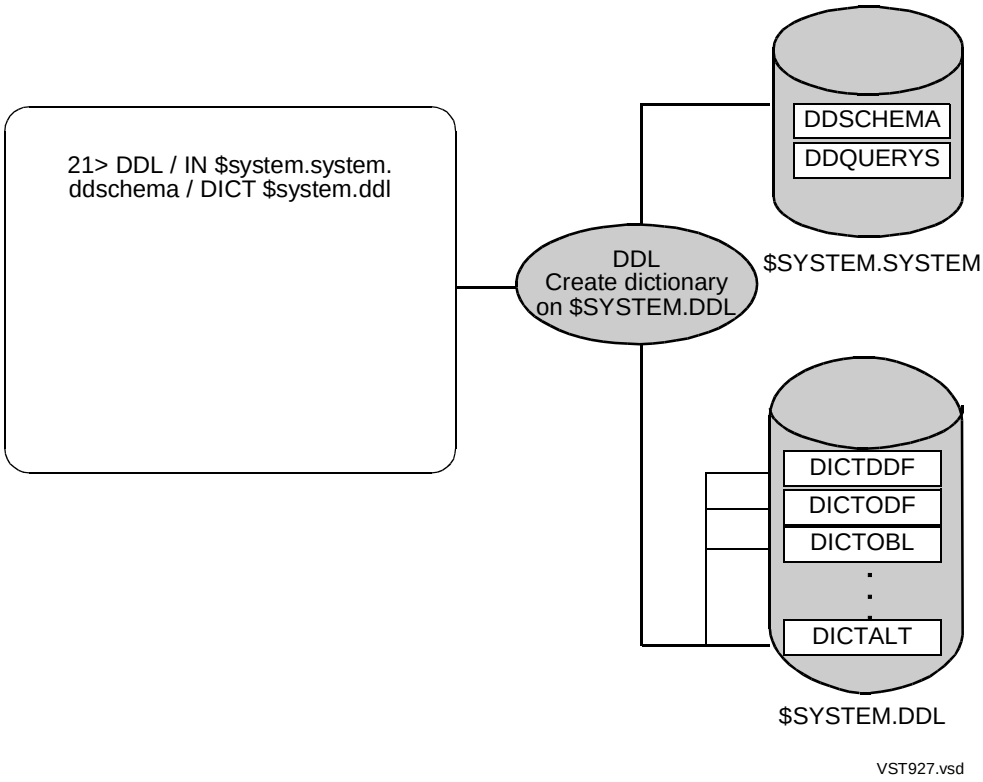
The preceding command creates a dictionary describing the 14 dictionary files on $SYSTEM.DDL and suppresses the listing. You can, of course, get a listing at your terminal or list DDSCHEMA on a printer. Depending on how your dictionary is managed, you might be able to omit this step. If the dictionary is already compiled on subvolume $SYSTEM.DDL, you need not recompile it.

The DDQUERYS queries contain the following command:

```
?ASSIGN QUERY-COMPILER-LISTING to $NULL
```

If you do not have a $NULL process on your system, or if you want to redirect this output, you can remove or change this line.

Figure E-1. Creating a Dictionary for DDSCHEMA



Requesting Reports

After you have created a dictionary, you can request reports about any dictionary on any subvolume in your system.

The commands in Example E-1 on page E-5 list all of the dictionary reports that describe the $DATA.SALES dictionary on the printer identified as $S.#PRINTER.

Example E-1. Requesting All 16 Dictionary Reports

```
14> VOLUME $data.sales
15> ENFORM /IN $system.system.ddquerys, OUT $S.#printer /
```

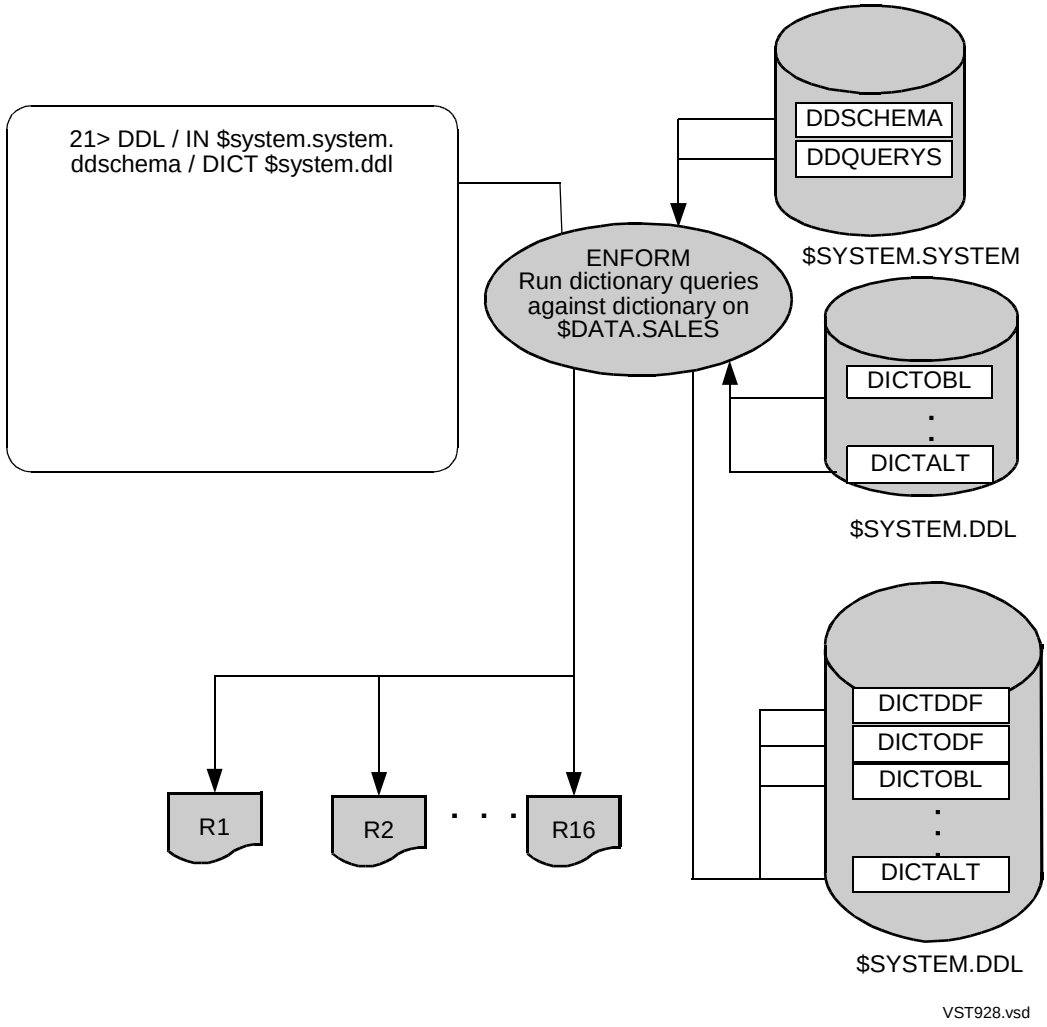
To select particular reports, you must run the Enform Plus program in the interactive mode, specify the output device with the OUT command, and specify the reports you want with the SOURCE command.

The commands in Example E-2 on page E-6 list two reports on $S.#PRINTER. The first (R2) lists the structure of every definition in the $DATA.SALES dictionary; the second (R6) shows every definition in the $DATA.SALES dictionary that is referenced by records and other definitions and lists the referring structures.

Example E-2. Requesting Selected Dictionary Reports

```
16> VOLUME $data.sales
17> ENFORM
18> ?OUT $S.#printer
19> ?SOURCE $system.system.ddquerys (R2, R6)
```

Figure E-2. Running DDQUERYS to Produce Reports



As Figure E-2 on page E-6 shows, the Enform Plus program reads its source statements from $SYSTEM.SYSTEM.DDQUERYS, reads the dictionary on $SYSTEM.DDL for the structure of the dictionary files; and finally, using the record names, field names, and field offsets from $SYSTEM.DDL, reads the dictionary files on $DATA.SALES to produce the reports.

The queries in DDQUERYS use the dictionary files on $DATA.SALES, because the dictionary file names stored in the dictionary on $SYSTEM.DDL do not specify a volume or a subvolume. When the Enform Plus program reads $SYSTEM.DDL, it uses the current default volume and subvolume to qualify the dictionary file names.

In the case of the sample database, the file name in the Enform Plus OPEN statement in DDQUERYS is fully qualified by the current default volume and subvolume. For example:

```
OPEN DICTOBL;                ! Statement in DDQUERYS
OPEN $data.sales.DICTOBL;    ! Statement as executed
```

The same default volume and subvolume insertion takes place for every OPEN statement in DDQUERYS. So, to query any dictionary on any subvolume, you can use the same dictionary describing the fourteen dictionary files, $SYSTEM.DDL, and the same Enform Plus dictionary report source, $SYSTEM.SYSTEM.DDQUERYS.

F Syntax Summary

Topics in this appendix:

RUN DDL Command

```
[RUN] DDL [ / run-option [ , run-option ] .../ ]
   [ compiler-command [ , compiler-command ] ...
```

CONSTANT Statement

```
CONSTANT constant-name { num-value-clause [ TYPE type ] }
                       { [ TYPE type ] num-value-clause }
                       { value-clause                   }.
```

num-value-clause

```
   VALUE [ IS ] { { constant-number   } [ LN-clause ]... }
                { { national-literal  }                   }
                { { existing-constant }                   }
```

type

```
   BINARY { [16] } [ UNSIGNED ]
          {  32  }
```

Default: BINARY 16

value-clause

```
   VALUE [ IS ] { { constant-number   } [ LN-clause ]... }
                { { "string"          }                   }
                { { national-literal  }                   }
                { { existing-constant }                   }
                {                                         }
                { VERSION "Lnn"                           }
```

national-literal

```
      { N }{ "2-byte-character ..." }
      { n }{ '2-byte-character ...' }
```

DEFINITION Statement

Field Definition

```
DEF[INITION] def-name
   { PICTURE-clause | TYPE-clause }
   [ AS-clause ]
   [ BEGIN ]
   [ DISPLAY-clause ]
   [ EDIT-PIC-clause ]
   [ EXTERNAL-clause ]
   [ HEADING-clause ]
   [ HELP-clause ]
   [ JUSTIFIED-clause ]
   [ MUST-BE-clause ]
   [ NULL-clause ]
   [ SPI-NULL-clause ]
   [ SQLNULLABLE-clause ]
   [ TACL-clause ]
   [ UPSHIFT-clause ]
   [ USAGE-clause ]
   [ VALUE-clause ] .
   [ 88-condition-name-clause . ] ...
   [ 89-enumeration-clause . ] ...
   [ END [ . ] ]
```

Group Definition

```
DEF[INITION] def-name

   [ DISPLAY-clause ]

   [ EXTERNAL-clause ]

   [ HEADING-clause ]

   [ HELP-clause ]

   [ NULL-clause ]

   [ SQLNULLABLE-clause ]

   [ USAGE-clause ]

   [ VALUE-clause ] .

   line-item specification ...

   [ 66-RENAMES-clause . ] ...

END [ . ]
```

line-item specification

```
   level-number { field-name | group-name | FILLER }
   { PICTURE-clause | TYPE-clause }
   [ AS-clause ]
   [ DISPLAY-clause ]
   [ EDIT-PIC-clause ]
   [ HEADING-clause ]
   [ HELP-clause ]
   [ JUSTIFIED-clause ]
   [ LN-clause ] ...
   [ MUST-BE-clause ]
   [ NULL-clause ]
   [ { OCCURS-clause | OCCURS-DEPENDING-ON-clause } ]
   [ REDEFINES-clause ]
   [ SPI-NULL-clause ]
   [ SQLNULLABLE-clause]
   [ TACL-clause ]
   [ USAGE-clause ]
   [ VALUE-clause ] .
   [ 88-condition-name-clause . ] ...
   [ 89-enumeration-clause . ] ...
```

Reference Definition

```
DEF[INITION] def-name-1 TYPE def-name-2

   [ AS-clause ]

   [ BEGIN ]

   [ DISPLAY-clause ]

   [ EDIT-PIC-clause ]

   [ EXTERNAL-clause ]

   [ HEADING-clause ]

   [ HELP-clause ]

   [ MUST-BE-clause ]

   [ NULL-clause ]

   [ SPI-NULL-clause ]

   [ TACL-clause ]

   [ UPSHIFT-clause ]

   [ USAGE-clause ]

   [ VALUE-clause ] .

   [ 88-condition-name-clause . ] ...

   [ END [ . ] ]
```

DELETE Statement

```
        { DEF[INITION] def-name ... }
        { RECORD record-name ...    }
DELETE  { TOKEN-CODE token-name ... } .
        { TOKEN-MAP map-name ...    }
        { TOKEN-TYPE type-name ...  }
```

EXIT Statement

```
EXIT [ . ]
```

OUTPUT Statement

```
OUTPUT { CONSTANT { constant-name ... } }
       {          { *                 } }
       {                                 }
       { DEF[INITION] { def-name ... }   }
       {              { *            }   }
       {                                 }
       { RECORD { record-name ... }      }
       {        { *               }      }
       {                                 }
       { TOKEN-CODE { token-name ... }   }
       {            { *              }   }
       {                                 }
       { TOKEN-MAP { map-name ... }      }
       {           { *            }      }
       {                                 }
       { TOKEN-TYPE { type-name ... }    }
       {            { *             }    }
       {                                 }
       { *                               } .
```

OUTPUT UPDATE Statement

```
OUTPUT UPDATE { CONSTANT constant-name ...     }
              { [ DEF[INITION] ] def-name ... }
              { TOKEN-TYPE type-name ...       } .
```

RECORD Statement

```
RECORD record-name .

  [ file-creation ]

  { record-structure | record-reference }

  [ key-assignment ]

END [ . ]
```

file-creation

```
  FILE IS { ["]file-name["] } [ creation-attribute ] ...
          { TEMPORARY       }
          { ASSIGNED        }
```

```
creation-attribute

   { KEY-SEQUENCED   }
   { RELATIVE        }
   { ENTRY-SEQUENCED }
   ( UNSTRUCTURED    }

   [ AUDIT ]

   [ AUDITCOMPRESS]

   [ BLOCK block-length ]

   [ [NO]BUFFERED ]

   [ BUFFERSIZE buffer-size ]

   [ CODE file-code ]

   { COMPRESS | DCOMPRESS | ICOMPRESS }

   [     { extent-size                                 ]
   [ EXT {                                             ]
   [     { ( pri-extent-size [, sec-extent-size ] ) ]

   [ MAXEXTENTS maximum-extents ]

   [ NO ODDUNSTR ]

   [ REFRESS ]

   [ SERIALWRITES ]

   [ VERIFYWRITES ]
```

```
record-structure

   line-item specification ...
   [ 66 RENAMES clause . ] ...

      line-item specification

         level-number { field-name | group-name | FILLER }
         { PICTURE-clause | TYPE-clause }
         [ AS-clause ]
         [ DISPLAY-clause ]
         [ EDIT-PIC-clause ]
         [ HEADING-clause ]
         [ HELP-clause ]
         [ JUSTIFIED-clause ]
         [ LN-clause ]
         [ MUST-BE-clause ]
         [ NULL-clause ]
         { OCCURS-clause | OCCURS-DEPENDING-ON-clause }
         [ REDEFINES-clause ]
         [ SPI-NULL-clause ]
         [ SQLNULLABLE-clause]
         [ TACL-clause ]
         [ USAGE-clause ]
         [ VALUE-clause ] .
         [ 88-condition-name- clause . ] ...
         [ 89-enumeration-clause . ] ...

record-reference

   DEF[INITION] IS def-name

key-assignment

   KEY key-specifier IS { group-name | field-name }

      [ FILE IS ["]file-name["] ]

      [ DUPLICATES [ NOT ] ALLOWED ] . ] ...

      [ UPDATE [ NOT ] ALLOWED ]

      [ SEQUENCE IS [ ASCENDING  ] { group-name }  ]
      [             [ DESCENDING ] { field-name } .]
```

SHOW USE OF Statement

```
SHOW USE OF { CONSTANT constant-name [, constant-name ]... }
            { [ DEF[INITION] ] def-name [, def-name ]...   }
            { TOKEN-TYPE type-name [, type-name ]...       }
```

TOKEN-CODE Statement

```
TOKEN-CODE token-name

  VALUE [ IS ] token-number

  TOKEN-TYPE [ IS ] type-name

  [ SSID subsystem-id ]

  [ HEADING label ]

  [ DISPLAY display-format ]
```

TOKEN-MAP Statement

```
TOKEN-MAP map-name

  VALUE [ IS ] token-number

  DEF [ IS ] def-name

  [ SSID subsystem-id ]

  [ HEADING label ]

  { { VERSION { number        } }                          }
  { {         { "Lnn"         } }                          }
  { {         { constant-name } }                          }
  { {                          }                           }
  { { NONVERSION               }                           }
  {                                                        }
  {     FOR { field-name [ { THROUGH } field-name ] }      }
  {         {            [ {   THRU  }          ] }      }
  {         {                                     }      }
  {         { group-name [ { THROUGH } group-name ] }      }
  {         {            [ { THRU    }          ] }. } ...

  END [ . ]
```

TOKEN-TYPE Statement

```
TOKEN-TYPE type-name

  VALUE [ IS ] token-data-type

  { DEF [ IS ] def-name [ OCCURS number TIMES ] }
  {                                             }
  { OCCURS { VARYING [ DEF [ IS ] def-name ] }  }
  {        { 0 TIMES                        }   }
```

DEFINITION and RECORD Statement Clauses

AS Clause

```
AS display-string  [ LN-clause ]...
```

DISPLAY Clause

```
DISPLAY display-format
```

EDIT-PIC Clause

```
EDIT-PIC edit-picture-string
```

EXTERNAL Clause

```
EXTERNAL
```

FILLER Clause

```
FILLER
```

HEADING Clause

```
HEADING report-heading  [ LN-clause ]...
```

HELP Clause

```
HELP help-text [ [,] help-text ]...
```

JUSTIFIED Clause

```
JUST[IFIED] RIGHT
```

KEYTAG Clause

```
KEYTAG key-specifier [ DUPLICATES [NOT] ALLOWED ]
```

LN Clause

```
{ LN"language-code[_territory-code][.charset]" }
{ constant-name                                }
```

MUST BE Clause

```
MUST BE { value                                  }
        { value-1 { THROUGH | THRU } value-2     }
```

value
value-1
value-2

```
{ "character-string"  }
{ constant-name       }
{ figurative-constant }
{ national-literal    }
{ number              }
{ symbolic-literal    }
{ value-name          }
```

NULL Clause

```
NULL { "character" | number | constant-name }
```

OCCURS Clause

```
OCCURS max [ TIMES ] [ INDEXED BY index-name ]
```

OCCURS DEPENDING ON Clause

```
OCCURS min TO max TIMES DEPENDING ON field-name
  [ INDEXED BY index-name ]
```

PICTURE Clause

```
PIC[TURE] { "{picture-string          }" }
          {  {national-picture-string }  }
          {                              }
          {  {picture-string          }  }
          {  {national-picture-string }  }
```

picture-string

```
{ alphanumeric-string | numeric-string }
```

alphanumeric-string

```
{ A | X | 9 }...[(length)]
```

numeric-string

```
{ [S]9...[(length)[V[9...[(length)]]]] }
{ T[9...[(length)[V[9...[(length)]]]]] }
{ 9...[(length)[V[9...[(length)]]]]S   }
{ 9...[(length)[V[9...[(length)]]]]T   }
```

national-picture-string

```
{ { N | n } [(length)] }
{ { N | n }...         }
```

REDEFINES Clause

```
REDEFINES { field-name | group-name }
```

SPI-NULL Clause

```
SPI-NULL {"character" | number | constant-name }
```

SQLNULLABLE Clause

```
[NOT]SQLNULLABLE
```

TACL Clause

```
TACL type
```

type

```
{ CRTPID   }
{ DEVICE   }
{ ENUM     }
{ FNAME    }
{ FNAME32  }
{ PHANDLE  }
{ SSID     }
{ SUBVOL   }
{ TRANSID  }
{ TSTAMP   }
{ USERNAME }
```

TYPE Clause

```
TYPE { data-type | def-name  | * }
```

data type

```
{ CHARACTER length                                          }
{                                                           }
{ BINARY { 8                    } [ UNSIGNED ]              }
{        { [ 16 [ , scale ] ]   }                           }
{        { 32 [ , scale ]       }                           }
{        { 64 [ , scale ]       }                           }
{                                                           }
{ FLOAT { [ 32 ] }                                          }
{       { 64     }                                          }
{                                                           }
{ COMPLEX                                                   }
{                                                           }
{ LOGICAL { 1     }                                         }
{         { [ 2 ] }                                         }
{         { 4     }                                         }
{                                                           }
{ ENUM                                                      }
{                                                           }
{ SQL-data-type                                             }
{                                                           }
{ BIT bit-length [ UNSIGNED ] [ ENUM enum-name ]            }
```

UPSHIFT Clause

```
UPSHIFT
```

USAGE Clause

```
[ USAGE [ IS ] ] { COMP[UTATIONAL]    }
                 { INDEX              }
                 { COMP[UTATIONAL]-3  }
                 ( PACKED-DECIMAL     }
```

VALUE Clause

```
{ VALUE [ IS ] value }
{ NOVALUE            }
```

value

```
  { { "character-string" } [ LN clause ]... }
  { { constant-name      }                   }
  { { national-literal   }                   }
  { { number             }                   }
  {                                          }
  { figurative-constant                      }
  { sql-datetime-literal                     }
  { sql-interval-literal                     }
  { symbolic-literal                         }
  { value-name                               }
```

66 RENAMES Clause

```
66 renames-name RENAMES
  { field-name [ { THROUGH } field-name ] }
  {            [ { THRU    }            ] }
  {                                       }
  { group-name [ { THROUGH } group-name ] }
  {            [ { THRU    }            ] }
```

88 Condition-Name Clause

```
88 condition-name { VALUE [ IS ]   }
                  { VALUES [ ARE ] }

 { value                      } [, value                      ]
 {                            } [                             ]
 { value { THROUGH } value    } [, value { THROUGH } value    ]
 {       { THRU    }          } [        { THRU    }          ] ...
```

value

```
  { { "character-string" } [ LN clause ] }
  { { constant-name      }               }
  { { national-literal   }               }
  { { number             }               }
  {                                      }
  { figurative-constant                  }
  { sql-datetime-literal                 }
  { sql-interval-literal                 }
  { symbolic-literal                     }
  { value-name                           }
```

89 Enumeration Clause

```
89 value-name [ VALUE value ] [ AS-clause ]
```

Commands

```
[NO]ANSICOBOL
```

Default: NOANSICOBOL

```
{ C [ c-source-file [ ! ] ] }
{ NOC                       }
```

Default: NOC

```
[NO]C00CALIGN
```

Default: C00CALIGN

```
[NO]CCHECK
```

Default: CCHECK if a C source code file is open, otherwise NOCCHECK

```
[NO]CDEFINEUPPER
```

Default: CDEFINEUPPER

```
?CFIELDALIGN_MATCHED2
```

```
{ CIFNDEF } identifier_name
{ CIFDEF  }

CENDIF
```

```
[NO]CLISTIN
```

Default: CLISTIN

```
{ [NO]CLISTOUT | CLISTOUTDETAIL }
```

Default: CLISTOUT

```
[NO]COBCHECK
```

Default: COBCHECK if a COBOL source code file is open, otherwise NOCOBCHECK

```
COBLEVEL [ base [ , increment ] ]
```

Default: *base* =1, *increment* = 1

```
{ COBOL [ cobol-source-file [ ! ] ] }
{ NOCOBOL                           }
```

Default: NOCOBOL

```
COLUMNS num
```

Default: *num* = 132

```
[NO]COMMENTS
```

Default: NOCOMMENTS

```
[NO]CPRAGMA
```

Default: CPRAGMA

```
[NO]CTOKENMAP_ASDEFINE
```

Default: NOCTOKENMAP_ASDEFINE

```
CUNDEF identifier_name
```

```
[NO]C_DECIMAL
```

Default: NOC_DECIMAL

```
[NO]C_MATCH_HISTORIC_TAL
```

Default: NOC_MATCH_HISTORIC_TAL

```
{ DDL [ ddl-source-file [ ! ] ] }
{ NODDL                         }
```

Default: NODDL

```
[NO]DEFLIST
```

Default: NODEFLIST

```
{ DICT [ dict-subvol-name ] [ ! ] }
{ NODICT                          }
```

Default: NODICT

```
DICTN [ dict-subvol-name ] [ ! ]
```

```
DICTR [ dict-subvol-name ]
```

```
{ DO_PTAL_ON | DO_PTAL_OFF }
```

Default: DO_PTAL_ON

```
EDIT [ edit-file-name [ ; edit-parameter ] ... ]
```

```
ERRORS [ max-errors ]
```

Default: Compilation continues until the end of the source code file regardless of the number of errors

```
[NO]EXPANDC
```

Default: NOEXPANDC

```
FIELDALIGN_SHARED8
```

```
FILLER { 1 | 0 }
```

Default: FILLER 1

```
[NO]FORCHECK
```

Default: FORCHECK if a FORTRAN source code file is open, otherwise NOFORCHECK

```
{ FORTRAN [ fortran-source-file [ ! ] ] }
{ NOFORTRAN                            }
```

Default: NOFORTRAN

```
[NO]FORTRANUNDERSCORE
```

Default: NOFORTRANUNDERSCORE

```
{ FUP [ fup-source-file [ ! ] ] }
{ NOFUP                        }
```

Default: NOFUP

```
HELP [ command ]
```

Default: all DDL commands

```
LINECOUNT number
```

Default: LINECOUNT 56

```
[NO]LIST
```

Default: LIST

```
{ NCLCONSTANT [ NCL-source-file [ ! ] ] }
{ NONCLCONSTANT                         }
```

Default: NONCLCONSTANT

```
{ NEWFUP_FILEFORMAT | OLDFUP_FILEFORMAT | NOFILEFORMAT }
```

Default: NOFILEFORMAT

```
OUT [ listing-destination ]
```

Default: destination specified in the OUT run option of the RUN DDL Command on page F-2

```
[NO]OUTPUT_SENSITIVE
```

Default: NOOUTPUT_SENSITIVE

```
PAGE [ "listing-title" ]
```

```
{ PASCAL [ pascal-source-file { ! ] ] }
{ NOPASCAL                          }
```

Default: NOPASCAL (D-series systems only)

```
PASCALBOUND { 0 | 1 }
```

Default: PASCALBOUND 1 (D-series systems only)

```
[NO] PASCALCHECK
```

Default: PASCALCHECK if a Pascal source code file is open, otherwise NOPASCALCHECK (D-series systems only)

```
[NO] PASCALNAMEDVARIANT
```

Default: NOPASCALNAMEDVARIANT (D-series systems only)

```
{ REPORT [ report-destination [ ! ] ] }
{ NOREPORT                            }
```

Default: NOREPORT

```
RESET
```

```
[NO] SAVE
```

Default: SAVE

```
SECTION section-name
```

```
SETLOCALENAME [ locale-name ]
```

Default: default system locale

```
SETSECTION [ section-name ]
```

```
SOURCE source-name [ ( section-name [ , section-name ] ...) ]
```

```
SPACING { 0 | 1 | 2 }
```

Default: SPACING 0

```
{ TACL [ tacl-source-file [ ! ] }
{ NOTACL                        }
```

Default: NOTACL

```
TACLGEN 0
```

Default: TACLGEN 0

```
{ TAL [ tal-source-file [ ! ] }
{ NOTAL                       }
```

Default: NOTAL

```
[NO]TALALLOCATE
```

Default: TALALLOCATE

```
TALBOUND { 0 | 1 }
```

Default: TALBOUND 1

```
[NO]TALCHECK
```

Default: TALCHECK if a TAL or pTAL source code file is open, otherwise NOTALCHECK

```
[NO]TALUNDERSCORE
```

Default: NOTALUNDERSCORE

```
TEDIT [ edit-file-name [ ; edit-parameter ] ... ]
```

```
[NO]TIMESTAMP
```

Default: TIMESTAMP

```
[NO]VALUES
```

Default: VALUES

```
[NO]WARN
```

Default: WARN

```
WARNINGS [ max-warnings ]
```

Default: Compilation continues until the end of the source code file regardless of the number of warnings

G Pathmaker and DDL

The Pathmaker product is a NonStop Transaction Services/MP (NonStop TS/MP) application systems generator. When you start a Pathmaker project, the Pathmaker program installs a dictionary for you as part of the application catalog, which is an integrated system directory for the Pathmaker project.

After the Pathmaker dictionary has been installed, you can start a DDL process from within the Pathmaker environment and enter definitions and records just as if you had created the dictionary from the command interpreter; however, there are differences in the information stored in the dictionary and this affects the way the dictionary is maintained.

DDL is used to specify definitions and records used by Pathmaker dictionaries. The DDL compiler can add these objects to and delete them from the dictionary, as well as perform other operations on them. The Pathmaker product contains other objects in its dictionaries (servers, services, requesters, and screens) that are not defined by using DDL. Only the Pathmaker software can add these objects to or delete them from a Pathmaker dictionary (or catalog). In addition, a Pathmaker dictionary contains application design information provided by the Pathmaker product, not by DDL.

Table G-1. DDL Features That Differ in the Pathmaker Environment (page 1 of 2)

DDL Feature	Pathmaker Action	Manual Reference
HELP Clause	The Pathmaker product displays help text on the screen when requested from a Pathmaker application.	HELP on page 6-10
MUST BE Clause	The Pathmaker product enforces MUST BE constraints; the DDL compiler does not.	MUST BE on page 6-15
UPSHIFT Clause	The Pathmaker product shifts specified character strings to uppercase; the DDL compiler does not.	UPSHIFT on page 6-69
OUTPUT* Statement	Pathmaker objects are not generated by this statement. As a result, you cannot use OUTPUT* to rebuild a Pathmaker dictionary.	OUTPUT on page 8-5
OUTPUT UPDATE Statement	Pathmaker objects that refer to specified DDL objects are ignored by OUTPUT UPDATE; the Pathmaker product makes any changes to Pathmaker objects that refer to DDL objects.	OUTPUT UPDATE on page 8-7

Table G-1. DDL Features That Differ in the Pathmaker Environment (page 2 of 2)

DDL Feature	Pathmaker Action	Manual Reference
DDL Command	The Pathmaker product cannot use DDL source code created by the DDL command to rebuild a Pathmaker dictionary; the DDL source code does not contain essential Pathmaker application design information.	DDL on page 9-42
DICT! Command	Pathmaker objects are not deleted from the dictionary with this command; only the Pathmaker product can modify or delete Pathmaker objects.	DICT on page 9-47
PURGE Command	Pathmaker dictionary files cannot be purged with the command interpreter PURGE command; only the Pathmaker product can purge Pathmaker dictionary files.	Purging a Dictionary on page 10-18

DDL Alignment Rules for C

This section provides information about alignment rules used by the DDL compiler when generating C code.

The DDL compiler supports four types of alignment rules:

Rules	For	Command
C00CALIGN Alignment Rules	C00 and later versions of the C compiler	C00CALIGN (default)
NOC00CALIGN Alignment Rules	Versions of the C compiler earlier than C00	NOC00CALIGN
C_MATCH_HISTORIC_TAL Alignment Rules	Compatibility with pTAL or TAL structures	CMATCH_HISTORIC_TAL
FIELDALIGN_SHARED8 Alignment Rules	Default alignment rules for compatibility with C structures	FIELDALIGN_SHARED8

Note. The DDL compiler does not generate C output for a structure definition if the alignment is not compatible to that of other languages. Incompatibility can occur if, for example, the C compiler adds implicit FILLER bytes to a structure wherever the DDL compiler does not add FILLER bytes.

C00CALIGN Alignment Rules

These are the default alignment rules.

C00 and later versions of the C compiler follow these rules:

- All structures and nested substructures begin and end on an even byte boundary.
- When a CHAR or CHAR ARRAY item directly follows another CHAR or CHAR ARRAY item, no filler exists between them (see Example H-1 on page H-2).

  This rule does not apply if the first CHAR data is within a substructure and the second is outside of the structure (see Example H-2 on page H-2).

Example H-1. C00CALIGN Alignment With Character Inside Structure

```
struct
{
  char a[3];
  char b;      ! Starts at offset 3
} s;
```

Example H-2. C00CALIGN Alignment With Character Outside Structure

```
struct
{
  struct
  {
    char x[3];
  } ss2;          ! C adds a filler byte at the of ss2
  char y;         ! Starts at offset 4
} s1;
```

When C00CALIGN is in effect, the DDL compiler does not generate C output for a structure that contains one of the following:

- A substructure that begins on an odd byte boundary.

  Note. The term substructure refers to a structure or union within a structure definition. The only data that the DDL compiler allocates starting on an odd byte is character data.

- A structure that ends on an odd byte boundary and is followed by a user-defined item that the DDL compiler allocates starting on an odd byte.

NOC00CALIGN Alignment Rules

Versions of the C compiler earlier than C00 follow these rules:

- If a substructure contains any word-aligned data (any data except for a CHAR or CHAR array), then C aligns the substructure on word boundaries and uses an even length (adding filler before and after the structure as needed).
- If a substructure contains no word-aligned data (only CHAR data, CHAR array data, or substructures containing only CHAR or CHAR array data, applied recursively), the DDL compiler aligns the substructure on byte boundaries and does not include implicit filler.

When NOC00CALIGN is in effect, the DDL compiler does not generate C output for a structure if the structure contains a substructure that contains a word-aligned item and one of the following is true:

- The substructure starts on an odd byte boundary.

  Note. The term substructure refers to either a structure or union within a structure definition. The only data that the DDL compiler allocates starting on an odd byte is character data.

- The substructure ends on an odd byte boundary and is followed by a user-defined item that the DDL compiler allocates starting on that odd byte.
- The DDL compiler does not insert implicit filler between byte-aligned objects except as defined by the preceding two rules.

C_MATCH_HISTORIC_TAL Alignment Rules

When you specify the CFIELDALIGN_MATCHED2 command, DDL uses the following alignment rules:

- If a substructure starts on an odd byte boundary or has an odd length, and refers to a previously defined structure, the DDL compiler inserts one or more fillers to word-align the substructure and make its length even.
- If a substructure defined in line starts on an odd byte boundary, the DDL compiler aligns the data on an odd byte boundary.

The C_MATCH_HISTORIC_TAL command allows members of a structure to have consecutive byte or word addresses. If the remaining byte in a two-byte word is not large enough to accommodate the next member, then the DDL compiler assigns the next word-aligned address. This condition also applies to a substructure that is declared inline, using the first member of the substructure.

FIELDALIGN_SHARED8 Alignment Rules

When you specify the FIELDALIGN_SHARED8 command, DDL uses the following alignment rules:

- The offset of each field (other than bit fields) in the structure from the base of the structure must begin at an address that is an integral multiple of the width of the field.
- The offset of a substructure field must be an integral multiple of the widest field in the substructure.
- The offset of an array must be an integral multiple of an element of the array.
- Bit fields are packed from the most significant bit to the least significant bit in a word and can not overlap a 16-bit word boundary.
- Explicit fillers are required to ensure that the components are properly aligned.
- The structure must begin at an address that is an integral multiple of the width of the widest field in the structure.
- The size of a structure must be an integral multiple of its alignment. Explicit fillers might be required to ensure this.
- The minimum alignment for a struct or substruct is 16 bits.
- The possible values for alignment are 1,2,4, or 8 bytes.

Glossary

alphabetic character. Any uppercase or lowercase letter or a space.

alphanumeric character. Any ASCII character.

alternate record key. A field other than the primary record key whose value identifies a record in a structured file.

command interpreter. A process that manages interactive communication between you and the operating system. In this manual, command interpreter refers to a TACL process.

compilation. The process of translating a source file to an object file.

complex instruction set computing (CISC). A processor architecture based on a large instruction set, characterized by numerous addressing modes, multicycle machine instructions, and many special-purpose instructions. Compare to reduced instruction set computing (RISC) and explicitly parallel instruction set computing (EPIC).

constant. A dictionary object that has a name, a data type, and a value. You define a constant in a CONSTANT statement, and you can refer to a constant value by name in other DDL statements.

Data Definition Language (DDL). An HP product for defining data objects in Enscribe files and translating object definitions to source code for programming languages and other products on HP NonStop systems.

DDL. See Data Definition Language (DDL).

dictionary. A DDL database that contains information about dictionary objects in a set of 14 files on the same subvolume. The name of the dictionary is the subvolume name. A subvolume can contain only one dictionary.

dictionary object. A data item defined in a schema and stored in a dictionary. Dictionary objects include:

- constants
- definitions
- records
- SPI token codes
- SPI token maps
- SPI token types

DEF. See definition.

definition. A dictionary object that describes a data structure, including the name, data type, size, and other attributes of a field (elementary item) or of a named group of fields.

ENABLE™. A product that is part of the ENCOMPASS Distributed Database Management System. ENABLE allows you to build simple applications that execute within a PATHWAY system.

Enform Plus. A language and a report generator used to retrieve information from databases. Enform Plus can use DDL to define data format.

Enscribe. The HP database record manager, which provides access to and manipulation of records in disk files.

explicitly parallel instruction set computing (EPIC). A processor architecture in which the instruction stream encodes what can be done in parallel (so that the hardware need not do this). Compare to complex instruction set computing (CISC) and reduced instruction set computing (RISC).

extensible structured token. An SPI token to which new fields can be added.

File Utility Program (FUP). A HP product for creating files and altering file attributes.

host language. Generally, a programming language available on HP NonStop systems; in this manual, a language in which the DDL compiler can generate source code.

locale. In localization, the definition of the subset of a user's environment that depends on language and cultural conventions.

Pathmaker. A NonStop Transaction Services/MP (NonStop TS/MP) application systems generator that can create and manipulate a dictionary.

record. A dictionary object that describes the structure of an Enscribe disk file; a record usually includes file creation information so that FUP can create a file from the record structure. If the file is to be key sequenced, a record also contains the key attributes.

reduced instruction set computing (RISC). A processor architecture based on a relatively small and simple instruction set, a large number of general-purpose registers, and an optimized instruction pipeline that supports high-performance instruction execution. Compare to complex instruction set computing (CISC) and explicitly parallel instruction set computing (EPIC).

simple token. An SPI token that has a single field or a fixed data structure.

source code. Input to a language compiler or other HP product.

schema. A set of DDL statements that define DDL objects.

schema file. An EDIT file that contains a schema.

SPI. See Subsystem Programmatic Interface (SPI).

SPI token. The smallest accessible unit in an SPI message; each token has a value and a code that identifies the value. See also simple token and extensible structured token.

SPI token code. The identifying code of a simple token.

SPI token map. The identifying code of an extensible structured token.

SPI token type. The data type and size of one or more tokens.

Subsystem Programmatic Interface (SPI). A set of procedures and associated definition files and a standard message protocol used to define common message-based interfaces for communication between management applications and subsystems. It includes procedures to build and decode specially formatted messages; definition files in TAL, COBOL, and HP Tandem Advanced Command Language (TACL) for inclusion in programs, macros, and routines using the interface procedures; and definition files in Data Definition Language (DDL) for programmers writing their own subsystems.

TNS architecture. NonStop Series architecture. HP computers that are based on complex instruction set computing (CISC) technology. TNS architecture implements the TNS instruction set.

TNS/E architecture. NonStop Series/Itanium® architecture. HP computers that are based on Itanium technology. TNS/E architecture implements the Itanium instruction set [explicitly parallel instruction set computing (EPIC)] and are upwardly compatible with the TNS and TNS/R architectures.

TNS/R architecture. NonStop Series/RISC architecture. HP computers that are based on reduced instruction set computing (RISC) technology. TNS/R architecture implements the RISC instruction set and are upwardly compatible with the TNS architecture.

Index

Numbers

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

Content Feedback

First Name: _____ Last Name: _____
Phone: _____ e-mail address: _____
Company: _____

(All contact information fields are required.)

If you're reporting an error or omission, is your issue:

☐ **Minor**: I can continue to work, but eventual resolution is requested.

☐ **Major**: I can continue to work, but prompt resolution is requested.

☐ **Critical**: I cannot continue to work without immediate response.

Comments (give sufficient detail to help us locate the text):

_____
_____
_____
_____
_____

Thank you for taking the time to provide us with your comments.

You can submit this form online, email it as an attachment topubs.comments@hp.com, FAX it to 408-285-5520, or mail it to:

Hewlett-Packard Company
NonStop Enterprise Division
19333 Vallco Parkway, MS 4421
Cupertino, CA 95014-2599
Attn.: Product Manager, Software Publications