

TAL Programmer's Guide

Abstract	This manual provides usage information for TAL (Transaction Application Language) and the TAL compiler for system and application programmers.
Part Number	096254
Edition	Second
Published	September 1993
Product Version	TAL C30, TAL D10, TAL D20
Release ID	D20.00
Supported Releases	This manual supports C30/D10.00 and all subsequent releases until otherwise indicated in a new edition.

Document History	Edition	Part Number	Product Version	Earliest Supported Release	Published
	First	065721	TAL D10	N/A	January 1993
	Second	096254	TAL C30 TAL D10 TAL D20	C30/D10.00	September 1993

New editions incorporate any updates issued since the previous edition.

A plus sign (+) after a release ID indicates that this manual describes function added to the base release, either by an interim product modification (IPM) or by a new product version on a .99 site update tape (SUT).

Copyright Copyright © 1993 by Tandem Computers Incorporated. Printed in the U.S.A. All rights reserved. No part of this document may be reproduced in any form, including photocopying or translation to another language, without the prior written consent of Tandem Computers Incorporated.

Export Statement Export of the information contained in this manual may require authorization from the U. S. Department of Commerce.

Examples Examples and sample programs are for illustration only and may not be suited for your particular purpose. Tandem does not warrant, guarantee, or make any representations regarding the use or the results of the use of any examples or sample programs in any documentation. You should verify the applicability of any example or sample program before placing the software into productive use.

Ordering Information For manual ordering information: Domestic U.S. customers, call 1-800-243-6886; international customers, contact your local sales representative.

New and Changed Information

The *TAL Programmer's Guide* gives guidelines for using TAL. This edition incorporates new features introduced in the D20 release and corrects text in response to reader comment forms.

This edition includes the following new features:

- For INT, REAL, and UNSIGNED data types and parameter types, a *width* argument can consist of any constant expression, including LITERALS and DEFINES.
- FIXED(*) is now a data type and a parameter type.
- A variable can have the same identifier as the encompassing BLOCK declaration.
- In addition to the implicit block named #GLOBAL, TAL creates an implicit block for each template structure declared outside a BLOCK declaration.
- The compiler ignores extra commas in, or adjacent to, procedure attribute lists.
- In CALL statements, the CALL keyword is optional.
- The RETURN statement lets you specify a condition-code value and a return expression.
- \$OPTIONAL, a new standard function, lets you conditionally pass parameters to VARIABLE and EXTENSIBLE procedures.
- DEFINETOG, a new directive, lets you create named and numeric toggles. It leaves new toggles turned off but does not change the settings of existing toggles.
- The IF, IFNOT, and ENDIF directives now support named toggles.

The operating system for Tandem NonStop systems, formerly called the Guardian operating system, is now called the Tandem NonStop Kernel. This change reflects Tandem's current and future operating system enhancements that further enable open systems and application portability.

Contents

About This Manual xxv
Notation Conventions xxxiii

Section 1 Introducing TAL

Using TAL 1-1
Major Features 1-1
System Services 1-3
System Procedures 1-3
TAL Run-Time Library 1-3
CRE Services 1-3

Section 2 Getting Started

Sample Source File 2-1
Creating Source Files 2-2
 Compiler Directives 2-2
 System Procedures 2-3
 Procedures 2-3
 Data Declarations 2-3
 Statements 2-4
 Comments 2-4
Compiling Source Files 2-4
Running Programs 2-5

Section 3 Structuring Programs

Source Files 3-1
Compilation Units 3-1
 Structuring Compilation Units 3-1
 Order of Components 3-3
Naming Compilation Units 3-4
Declaring Data 3-4
Declaring Global Data 3-5
 Declaring Unblocked Global Data 3-5
 Declaring Blocked Global Data 3-5

Declaring Procedures	3-6
Procedure Heading	3-6
Procedure Body	3-6
FORWARD Procedures	3-6
EXTERNAL Procedures	3-7
Procedure Entry Points	3-7
Local Data	3-8
Local Labels	3-8
Local Statements	3-8
Declaring Subprocedures	3-10
Subprocedure Heading	3-10
Subprocedure Body	3-10
FORWARD Subprocedures	3-10
Subprocedure Entry Points	3-11
Sublocal Data	3-12
Sublocal Labels	3-12
Sublocal Statements	3-12
Formatting Programs	3-13
Formatting With Comments	3-14
Formatting With BEGIN-END Constructs	3-16
Using Semicolons	3-17
Using Compiler Directives	3-18

Section 4 Introducing the Environment

Process Environment	4-1
Code Space	4-1
Data Space	4-2
System Code Space	4-5
System Library Space	4-5
Registers	4-5
Addressing Modes	4-5
Direct Addressing	4-6
Indirect Addressing	4-6
Indexing	4-7
Storage Allocation	4-8
Allocation in the User Data Segment	4-8
Allocation in the Automatic Extended Segment	4-10

Section 5 Using Expressions

- About Expressions 5-1
 - Complexity 5-1
 - Functionality 5-1
- Operands 5-2
 - Identifiers 5-2
 - Data Types 5-4
 - Data Type Aliases 5-6
 - Storage Units 5-6
 - Data Types of Expressions 5-6
 - Variables 5-7
 - Constants 5-7
 - LITERALS 5-11
 - Standard Functions 5-12
- Precedence of Operators 5-13
- Arithmetic Expressions 5-15
 - Operands in Arithmetic Expressions 5-15
 - Signed Arithmetic Operators 5-16
 - Unsigned Arithmetic Operators 5-18
 - Bitwise Logical Operators 5-20
- Conditional Expressions 5-21
 - Conditions 5-21
 - Boolean Operators 5-22
 - Relational Operators 5-23
 - Assigning Conditional Expressions 5-24
- Testing Hardware Indicators 5-25
 - Condition Code Indicator 5-25
 - Carry Indicator 5-26
 - Overflow Indicator 5-26
- Accessing Operands 5-27
 - Getting the Address of Variables 5-27
 - Dereferencing Simple Variables 5-27
 - Extracting Bit Fields 5-28
 - Shifting Bit Fields 5-29

Section 6 Using Simple Variables

- Declaring Simple Variables 6-1
 - Specifying Data Types 6-1
 - Initializing Simple Variables 6-2
 - Allocating Simple Variables 6-3
- Assigning Data to Simple Variables 6-4
 - Assigning Variables 6-4
 - Matching Data Types 6-4
 - Converting Data Types 6-4
 - Assigning Character Strings 6-4
 - Multiple Variables 6-5
- Simple Variables by Data Type 6-5
 - STRING Simple Variables 6-5
 - INT Simple Variables 6-6
 - INT(32) Simple Variables 6-8
 - REAL Simple Variables 6-9
 - REAL(64) Simple Variables 6-10
 - FIXED Simple Variables 6-11
 - UNSIGNED Simple Variables 6-13

Section 7 Using Arrays

- Declaring Arrays 7-1
 - Using Indirection 7-2
 - Specifying Data Types 7-2
 - Initializing Arrays 7-2
 - Arrays by Data Type 7-4
 - Allocating Arrays 7-8
 - Addressability of Arrays 7-11
- Accessing Arrays 7-12
 - Indexing Arrays 7-12
- Assigning Data to Array Elements 7-13
- Assigning the Address of Arrays 7-13
- Copying Data Into Arrays 7-14
 - Copying a Constant List Into an Array 7-14
 - Copying Data Between Arrays 7-15
 - Copying Data Within an Array 7-17
 - Using the Next Address 7-17
 - Concatenating Copy Operations 7-18

Scanning Arrays	7-19
Delimiting the Scan Area	7-19
Determining What Stopped the Scan	7-20
Scanning Bytes in Word-Aligned Arrays	7-20
Multipart Scan Example	7-20
Comparing Arrays	7-23
Using Standard Functions With Arrays	7-23
Using Read-Only Arrays	7-24
Declaring Read-Only Arrays	7-24
Accessing Read-Only Arrays	7-24

Section 8 Using Structures

Kinds of Structures	8-1
Structure Layout	8-2
Declaring Definition Structures	8-3
Specifying Structure Occurrences	8-3
Using Indirection	8-3
Allocating Definition Structures	8-4
Addressability of Structures	8-6
Declaring Template Structures	8-7
Declaring Referral Structures	8-8
Specifying Structure Occurrences	8-8
Using Indirection	8-8
Allocating Referral Structures	8-9
Addressability of Structures	8-9
Declaring Simple Variables and Arrays in Structures	8-9
Declaring Arrays That Use No Memory	8-10
Allocating Simple Variables and Arrays in Structures	8-10
Alignment of Simple Variables and Arrays	8-10
Declaring Substructures	8-12
Declaring Definition Substructures	8-12
Declaring Referral Substructures	8-15
Declaring Fillers	8-16
Declaring Filler Bytes	8-16
Declaring Filler Bits	8-16

Declaring Simple Pointers in Structures	8-17
Addresses Simple Pointers Can Contain	8-18
Allocating Simple Pointers in Structures	8-18
Declaring Structure Pointers in Structures	8-19
Addresses Structure Pointers Can Contain	8-20
Allocating Structure Pointers in Structures	8-20
Declaring Redefinitions	8-21
Simple Variables or Arrays as Redefinitions	8-21
Definition Substructures as Redefinitions	8-23
Referral Substructures as Redefinitions	8-25
Simple Pointers as Redefinitions	8-26
Structure Pointers as Redefinitions	8-26
Accessing Structure Items	8-27
Qualifying Identifiers	8-27
Indexing Structures	8-28
Assigning Values to Structure Items	8-34
Assigning Addresses to Pointers in Structures	8-35
Accessing Data Through Pointers in Structures	8-36
Copying Data in Structures	8-39
Using Standard Functions With Structures	8-43

Section 9 Using Pointers

Using Simple Pointers	9-2
Declaring Simple Pointers	9-2
Specifying a Data Type	9-2
Initializing Simple Pointers	9-3
Initializing Global Simple Pointers	9-3
Initializing Local or Sublocal Simple Pointers	9-5
Allocating Simple Pointers	9-6
Assigning Addresses to Simple Pointers	9-7
Accessing Data With Simple Pointers	9-9
Indexing Simple Pointers	9-10

Using Structure Pointers	9-12
Declaring Structure Pointers	9-12
Initializing Structure Pointers	9-13
Initializing Global Structure Pointers	9-13
Initializing Local or Sublocal Structure Pointers	9-15
Allocating Structure Pointers	9-16
Assigning Addresses to Structure Pointers	9-16
Accessing Data With Structure Pointers	9-18
Indexing Structure Pointers	9-20

Section 10 Using Equivalenced Variables

Example Diagrams	10-1
Variables You Can Equivalence	10-1
Equivalencing Simple Variables	10-2
Declaring Equivalenced Simple Variables	10-2
Accessing Equivalenced Simple Variables	10-5
Equivalencing Simple Pointers	10-6
Declaring Equivalenced Simple Pointers	10-6
Accessing Equivalenced Simple Pointers	10-8
Equivalencing Structures	10-9
Equivalencing Definition Structures	10-10
Equivalencing Referral Structures	10-14
Equivalencing Structure Pointers	10-15
Declaring Equivalenced Structure Pointers	10-16
Accessing Data Through Equivalenced Structure Pointers	10-18
Using Indexes or Offsets	10-19
Emulating Pascal Variant Parts	10-21

Section 11 Using Procedures

Procedures and Code Segments	11-1
Declaring Procedures	11-2
Calling Procedures	11-2
Declaring Parameters in Procedures	11-3
Specifying a Formal Parameter List	11-3
Declaring Formal Parameters	11-3
Passing Actual Parameters	11-4

Declaring Data in Procedures	11-4
Allocating Local Variables	11-5
Allocating Parameters and Variables	11-5
Returning From a Procedure	11-7
Using Procedure Options	11-7
Declaring the MAIN Procedure	11-7
Declaring Functions	11-8
Declaring FORWARD Procedures	11-8
Declaring EXTERNAL Procedures	11-9
Declaring VARIABLE Procedures	11-9
Declaring EXTENSIBLE Procedures	11-10
Passing Parameters to VARIABLE or EXTENSIBLE Procedures	11-11
Converting VARIABLE Procedures to EXTENSIBLE	11-13
Comparing Procedures and Subprocedures	11-14
Declaring and Calling Subprocedures	11-15
Including Formal Parameters	11-15
Sublocal Variables	11-16
Visibility of Identifiers	11-16
Sublocal Storage Limitations	11-17
Sublocal Parameter Storage Limitations	11-18
Using Parameters	11-20
Declaring Formal Parameters	11-20
Using Value Parameters	11-21
Using Reference Parameters	11-29
Parameter Pairs	11-35
Procedure Parameter Area	11-36
Subprocedure Parameter Area	11-36
Scope of Formal Parameters	11-36
Parameter Masks	11-38
VARIABLE Parameter Masks	11-38
EXTENSIBLE Parameter Masks	11-42
Using Labels	11-48
Using Local Labels	11-48
Using Undeclared Labels	11-51
Getting Addresses of Procedures and Subprocedures	11-52

Section 12 Controlling Program Flow

IF Statement 12-2

 IF Statement Execution 12-3

 IF-ELSE Pairing 12-4

CASE Statement, Labeled 12-5

 Statement Forms Generated by the Compiler 12-6

 Directives and CASE Statements 12-7

 Labeled CASE Statement Execution 12-7

WHILE Statement 12-8

DO Statement 12-10

FOR Statement 12-12

 Nesting FOR Loops 12-13

 Standard FOR Loops 12-13

 Optimized FOR Loops 12-15

ASSERT Statement 12-17

 Using ASSERT with ASSERTION 12-17

 Nullifying ASSERT Statements 12-18

CALL Statement 12-19

 Calling Procedures and Subprocedures 12-19

 Calling Functions 12-20

 Calling Procedures Declared as Formal Parameters 12-20

 Passing Parameter Pairs 12-20

RETURN Statement 12-21

 Returning From Functions 12-21

 Returning From Nonfunction Procedures 12-23

GOTO Statement 12-24

 Local Scope 12-24

 Sublocal Scope 12-24

 Usage Guidelines 12-25

Section 13 Using Special Expressions

- Assignment Expression 13-2
- CASE Expression 13-3
- IF Expression 13-4
- Group Comparison Expression 13-5
 - Comparing a Variable to a Constant List 13-5
 - Comparing a Variable to a Single Byte 13-5
 - Comparing a Variable to a Variable 13-6
 - Using the Next Address 13-7
 - Testing the Condition Code Setting 13-9

Section 14 Compiling Programs

- The Compiler 14-1
 - BINSERV 14-1
 - SYMSERV 14-1
- Compiling Source Files 14-2
 - Running the Compiler 14-2
 - Completion Codes Returned by the Compiler 14-5
- Binding Object Files 14-5
 - Binding During Compilation 14-6
 - Binding After Compilation 14-6
 - Binding at Run Time 14-6
- Using Directives in the Source File 14-7
- Using Directive Stacks 14-8
 - Pushing Directive Settings 14-8
 - Popping Directive Settings 14-8
- File Names as Directive Arguments 14-9
 - Partial File Names 14-9
 - Logical File Names 14-9
- Compiling With SOURCE Directives 14-10
 - Section Names 14-10
 - Nesting Levels 14-10
 - Effect of Other Directives 14-10
 - Including System Procedure Declarations 14-11

Compiling With Search Lists	14-12
Creating the Master Search List	14-12
Clearing the Master Search List	14-12
Searching the Master Search List	14-13
Binding the Master Search List	14-13
Compiling With Relocatable Data Blocks	14-14
Declaring Relocatable Global Data	14-14
Allocating Global Data Blocks	14-16
Sharing Global Data Blocks	14-20
Directives for Relocatable Data	14-21
Compiling With Saved Global Data	14-23
Saving Global Data	14-23
Retrieving Global Data	14-23
Saved Global Data Compilation Session	14-24
Effects of Other Directives	14-25
Compiling With Cross-References	14-26
Selecting Classes	14-26
Collecting Cross-References	14-27
Printing Cross-References	14-28

Section 15 Compiler Listing

Page Header	15-1
Banner	15-2
Directives in Compilation Commands	15-2
Compiler Messages	15-2
Source Listing	15-3
Edit-File Line Number	15-3
Code-Address Field	15-3
Lexical-Level Counter	15-4
BEGIN-END Pair Counter	15-4
Conditional Compilation Listing	15-5
Local or Sublocal Map	15-6
INNERLIST Listing	15-7
CODE Listing	15-9
ICODE Listing	15-9
Global Map	15-9
File Name Map	15-10

Cross-Reference Listings	15-11
Source-File Cross-References	15-11
Identifier Cross-References	15-11
LMAP Listings	15-13
Entry-Point Load Map	15-14
Data-Block Load Maps	15-14
Compilation Statistics	15-16
Object-File Statistics	15-16

Section 16 Running and Debugging Programs

Running Programs	16-1
Specifying Run Options	16-1
Passing Run-Time Parameters	16-3
Stopping Programs	16-3
Run-Time Errors	16-3
Debugging Programs	16-4
Using the Inspect Product	16-4
Requesting the Inspect Product	16-4
Compiling the Source File	16-4
Starting the Inspect Session	16-4
Setting Breakpoints	16-5
Stepping Through a Program	16-5
Displaying Values	16-5
Stopping the Inspect Session	16-5
Sample Inspect Session	16-6

Section 17 Mixed-Language Programming

Mixed-Language Features of TAL	17-1
NAME and BLOCK Declarations	17-1
LANGUAGE Attribute	17-2
Public Name	17-3
PROC Parameter Type	17-4
PROC(32) Parameter Type	17-5
Parameter Pairs	17-6
ENV Directive	17-8
HEAP Directive	17-8

TAL and C Guidelines	17-9
Using Identifiers	17-9
Matching Data Types	17-10
Memory Models	17-11
Calling C Routines From TAL Modules	17-12
Calling TAL Routines From C Modules	17-13
Sharing Data	17-15
Parameters and Variables	17-18
Extended Data Segments	17-33
CRE Guidelines for TAL	17-37
General Coding Guidelines	17-37
Specifying a Run-Time Environment	17-39
The User Heap	17-41
Including Library Files	17-43
Initializing the CRE	17-44
Terminating Programs	17-45
Sharing the Standard Files	17-46
Accessing \$RECEIVE	17-48
Handling Errors in CRE Math Routines	17-49
The Extended Stack	17-49
CRE Sample Program	17-50

Appendix A Sample Programs

String-Display Programs	A-1
String-Entry Program	A-3
Binary-to-ASCII Conversion Program	A-5
Modular Programming Example	A-7
Modular Structure	A-7
File-Naming Conventions	A-8
Compiling and Binding the Modular Program	A-9
Source Modules	A-9
Compilation Maps and Statistics	A-21

Appendix B Managing Addressing

- Extended Pointer Format B-1
- Accessing the Upper 32K-Word Area B-2
 - Storing Addresses in Simple Pointers B-2
 - Storing Addresses in Structure Pointers B-4
 - Managing Data Allocation in Upper 32K-Word Area B-5
 - Managing Large Blocks of Memory B-7
 - Indexing the Upper 32K-Word Boundary B-8
- Accessing the User Code Segment B-9
 - Initializing Simple Pointers B-9
 - Assigning Addresses to Simple Pointers B-9
- Using Extended Data Segments B-10
 - Using the Automatic Extended Segment B-10
 - Using Explicit Extended Segments B-10
 - C-Series Extended Segment Examples B-19

Appendix C Improving Performance

- General Guidelines C-1
- Addressing Guidelines C-1
 - Direct Addressing C-1
 - Indirect Addressing C-1
 - Extended Addressing C-1
 - STACK and STORE Statements C-2
- Indexing Guidelines C-2
- Arithmetic Guidelines C-3

Appendix D ASCII Character Set

Appendix E File Names and TACL Commands

- Disk File Names E-1
 - Parts of a Disk File Name E-2
 - Partial File Names E-3
 - Logical File Names E-3
 - Internal File Names E-4
- TACL Commands E-4

TACL DEFINE Commands	E-4
Substituting File Names	E-4
TACL DEFINE Names	E-5
Setting DEFINE CLASS Attributes	E-5
TACL PARAM Commands	E-6
PARAM BINSERV Command	E-6
PARAM SAMECPU Command	E-6
PARAM SWAPVOL Command	E-7
PARAM SYMSERV Command	E-7
Using PARAM Commands	E-7
TACL ASSIGN Commands	E-8
Ordinary ASSIGN Command	E-8
ASSIGN SSV Command	E-9

Appendix F Type Correspondence

Glossary	Glossary-1
-----------------	------------

Index	Index-1
--------------	---------

Figures	Figure 2-1.	Sample Source File	2-1
	Figure 3-1.	Structure of a Sample Compilation Unit	3-2
	Figure 4-1.	User Data Segment	4-3
	Figure 4-2.	Automatic Extended Data Segment	4-4
	Figure 4-3.	Byte and Word Addressing in User Data Segment	4-6
	Figure 4-4.	Primary and Secondary Storage in the User Data Segment	4-8
	Figure 6-1.	Simple Variable Storage Allocation	6-3
	Figure 7-1.	Allocating Direct Arrays	7-8
	Figure 7-2.	Allocating Indirect Arrays	7-10
	Figure 9-1.	Allocating Simple Pointers	9-7
	Figure 9-2.	Indexing Simple Pointers	9-11
	Figure 11-1.	Procedure Data Allocation	11-6
	Figure 11-2.	Sublocal Data Storage Limitations	11-19
	Figure 11-3.	VARIABLE Word Parameter Mask	11-39
	Figure 11-4.	Parameter Area of a VARIABLE Procedure	11-40
	Figure 11-5.	VARIABLE Doubleword Parameter Mask	11-41
	Figure 11-6.	EXTENSIBLE Word Parameter Mask	11-43
	Figure 11-7.	EXTENSIBLE Doubleword Parameter Mask	11-44
	Figure 11-8.	Parameter Area of EXTENSIBLE Procedure	11-46
	Figure 12-1.	IF-THEN Execution	12-3
	Figure 12-2.	IF-THEN-ELSE Execution	12-3
	Figure 12-3.	Labeled CASE Statement Execution	12-7
	Figure 12-4.	WHILE Statement Execution	12-9
	Figure 12-5.	DO Statement Execution	12-11
	Figure 12-6.	Standard FOR Loop Execution	12-14
	Figure 14-1.	Compiling a Source File Into an Object File	14-2
	Figure 14-2.	Binding Object Files	14-5
	Figure 14-3.	Allocating Global Data Blocks	14-19
	Figure 15-1.	Page Headers	15-1
	Figure 15-2.	Sample Banner	15-2
	Figure 15-3.	Source Listing	15-5
	Figure 15-4.	Local Map	15-7
	Figure 15-5.	INNERLIST Listing	15-8

Figure 15-6.	CODE Listing	15-9
Figure 15-7.	ICODE Listing	15-9
Figure 15-8.	Global Map	15-10
Figure 15-9.	File Name Map	15-10
Figure 15-10.	Source-File Cross-Reference Listing	15-11
Figure 15-11.	Identifier Cross-Reference Listing	15-13
Figure 15-12.	Entry-Point Load Map by Name	15-14
Figure 15-13.	Data-Block Load Map by Location	15-15
Figure 15-14.	Read-Only Data-Block Load Map by Location	15-15
Figure 15-15.	Compiler Statistics	15-16
Figure 15-16.	Object-File Statistics	15-17
Figure 16-1.	Sample Source File	16-6
Figure 16-2.	Sample Compiler Listing	16-7
Figure 16-3.	Running a Sample Inspect Session	16-9
Figure A-1.	Entry-Point Load Map for Modular Program	A-21
Figure A-2.	Data-Block Load Map for Modular Program	A-22
Figure A-3.	Compilation Statistics for Mainline Module	A-22
Figure B-1.	Format of Extended Pointer	B-1
Figure B-2.	Format of Extended Segment Base Address	B-19

Tables	Table 3-1.	Data Declarations	3-4
	Table 3-2.	TAL Statements	3-9
	Table 4-1.	Registers in Current Process Environment	4-5
	Table 5-1.	Reserved Keywords	5-3
	Table 5-2.	Nonreserved Keywords	5-3
	Table 5-3.	Data Types	5-4
	Table 5-4.	Storage Units	5-6
	Table 5-5.	Variables	5-7
	Table 5-6.	Number Base Formats	5-8
	Table 5-7.	Summary of Standard Functions	5-12
	Table 5-8.	Precedence of Operators	5-13
	Table 5-9.	Operands in Arithmetic Expressions	5-15
	Table 5-10.	Signed Arithmetic Operators	5-16
	Table 5-11.	Signed Arithmetic Operand and Result Types	5-16
	Table 5-12.	Unsigned Arithmetic Operators	5-18
	Table 5-13.	Unsigned Arithmetic Operand and Result Types	5-19
	Table 5-14.	Logical Operators and Result Yielded	5-20
	Table 5-15.	Conditions in Conditional Expressions	5-21
	Table 5-16.	Boolean Operators and Result Yielded	5-22
	Table 5-17.	Signed Relational Operators and Result Yielded	5-23
	Table 5-18.	Unsigned Relational Operators and Result Yielded	5-23
	Table 5-19.	Bit-Shift Operators	5-29
	Table 6-1.	Number Base Formats	6-2
	Table 8-1.	Kinds of Structures	8-1
	Table 8-2.	Structure Items	8-2
	Table 8-3.	Data Accessed by Simple Pointers	8-17
	Table 8-4.	Addresses in Simple Pointers	8-18
	Table 8-5.	Addresses in Structure Pointers	8-20
	Table 8-6.	Indexing Structures	8-28
	Table 9-1.	Data Accessed by Simple Pointers	9-2
	Table 9-2.	Addresses in Simple Pointers	9-3
	Table 9-3.	Addresses in Structure Pointers	9-13
	Table 9-4.	Indexing Structure Pointers	9-21

Table 10-1.	Equivalenced Variables	10-1
Table 10-2.	Indexes and Offsets	10-19
Table 11-1.	Value and Reference Parameters	11-3
Table 11-2.	Procedures and Subprocedures	11-14
Table 11-3.	Formal Parameter Specification	11-20
Table 11-4.	Value Parameter Storage Allocation	11-21
Table 11-5.	Reference Parameter Storage Allocation	11-29
Table 11-6.	Reference Parameter Address Conversions	11-35
Table 11-7.	Entry Values Loaded Onto Register Stack	11-47
Table 12-1.	Program Control Statements	12-1
Table 13-1.	Special Expressions	13-1
Table 14-1.	Completion Codes	14-5
Table 14-2.	Data Blocks Created by the Compiler	14-18
Table 15-1.	Local/Sublocal Map Information	15-6
Table 15-2.	Compiler Attributes	15-12
Table 15-3.	Entry-Point Load Map Information	15-14
Table 15-4.	Data-Block Load Map Information	15-15
Table 17-1.	Parameter Pair Type Correspondence	17-6
Table 17-2.	Compatible TAL and C Data Types	17-10
Table 17-3.	CRE Data Blocks	17-39
Table 17-4.	Including Library and External Declaration Files	17-43
Table B-1.	Extended Pointer Format	B-1
Table F-1.	Integer Types, Part 1	F-3
Table F-2.	Integer Types, Part 2	F-3
Table F-3.	Floating, Fixed, and Complex Types	F-3
Table F-4.	Character Types	F-4
Table F-5.	Structured, Logical, Set, and File Types	F-4
Table F-6.	Pointer Types	F-5

Examples	Example 17-1.	D-Series TAL and C Extended Segment Management	17-35
	Example 17-2.	D-Series CRE Sample Program	17-50
	Example A-1.	C- or D-Series String-Display Program	A-1
	Example A-2.	C-Series String-Display Program	A-2
	Example A-3.	D-Series String-Entry Program	A-3
	Example A-4.	C-Series String-Entry Program	A-4
	Example A-5.	C- or D-Series Binary-to-ASCII Conversion Program	A-6
	Example A-6a.	D-Series Mainline Module	A-10
	Example A-6b.	D-Series Initialization Module	A-12
	Example A-6c.	D-Series Input File Module	A-14
	Example A-6d.	D-Series Output File Module	A-17
	Example A-6e.	D-Series Message Module	A-19
	Example B-1.	D-Series Extended Segment Allocation Program	B-12
	Example B-2.	D-Series Extended Segment Management Program	B-16
	Example B-3.	C-Series Extended Segment Allocation Program	B-20
	Example B-4.	C-Series Extended Segment Management Program	B-22

About This Manual

The Transaction Application Language (TAL) is a high-level, block-structured language used to write systems software and transaction-oriented applications.

The TAL compiler compiles TAL source programs into executable object programs. The compiler and the object programs it generates execute under control of the Tandem NonStop Kernel.

This manual gives guidelines for using TAL and the TAL compiler. General topics described in this manual include:

- How to create, structure, compile, and run a program
- The process environment, addressing modes, and storage allocation
- How to declare and access procedures and variables

Audience This manual is intended for system programmers and application programmers familiar with Tandem systems and the operating system.

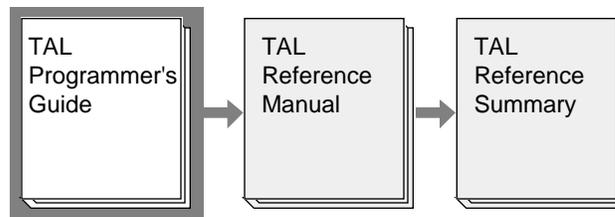
How to Use This Manual Set

The TAL manual set consists of the manuals listed in Table 1:

Table 1. TAL Manual Set

Manual	Description
<i>TAL Programmer's Guide</i>	Helps you get started in creating, structuring, compiling, running and debugging programs. Describes how to declare and access procedures and variables and how the TAL compiler allocates storage for variables.
<i>TAL Reference Manual</i>	Describes the syntax for declaring variables and procedures and for specifying expressions, statements, standard functions, and compiler directives; describes error and warning messages.
<i>TAL Reference Summary</i>	Presents a summary of syntax diagrams.

The *TAL Programmer's Guide* is a prerequisite to the *TAL Reference Manual*:



329

If you are not familiar with TAL, first read the *TAL Programmer's Guide*.

If you are familiar with TAL and the process environment, consult the *TAL Reference Manual* for the syntax for declarations, statements, and directives and for information about error messages.

If you are writing a program that mixes TAL modules with modules written in other languages, see Section 17, "Mixed-Language Programming," in the *TAL Programmer's Guide*.

Organization of Manual The *TAL Programmer's Guide* is organized as follows:

Section 1, "Introducing TAL," summarizes the features of TAL.

Section 2, "Getting Started," explains how to create, compile, and run a program.

Section 3, "Structuring Programs," describes how to structure source programs.

Section 4, "Introducing the Environment," describes the process environment, addressing modes, and storage allocation.

Section 5, "Using Expressions," describes arithmetic and conditional expressions.

Section 6, "Using Simple Variables," explains how to declare and access simple variables and how the compiler allocates storage for them.

Section 7, "Using Arrays," explains how to declare and access arrays and how the compiler allocates storage for them.

Section 8, "Using Structures," explains how to declare and access structures and how the compiler allocates storage for them.

Section 9, "Using Pointers," explains how to declare and access simple pointers and structure pointers and how the compiler allocates storage for them.

Section 10, "Using Equivalenced Variables," explains how to declare and access equivalenced variables.

Section 11, "Using Procedures," gives information about procedures and subprocedures.

Section 12, "Controlling Program Flow," explains control statements.

Section 13, "Using Special Expressions," describes assignment, CASE, IF, and group comparison expressions.

Section 14, "Compiling Programs," describes some of the compilation options.

Section 15, "Compiler Listing," describes the output generated by the compiler.

Section 16, "Running and Debugging Programs," shows how to run and debug programs.

Section 17, "Mixed-Language Programming," describes TAL mixed-language features, TAL and C guidelines, and Common Run-Time Environment (CRE) guidelines for TAL.

Appendix A, "Sample Programs," shows sample programs.

Appendix B, "Managing Addressing," gives guidelines for managing addressing, particularly in explicit (user-allocated) extended data segments.

Appendix C, "Improving Performance," gives guidelines for improving execution performance.

Appendix D, "ASCII Character Set," describes the ASCII character set.

Appendix E, “File Names and TACL Commands,” describes D-series file-naming conventions and TACL ASSIGN, DEFINE, and PARAM commands.

Appendix F, “Data Type Correspondence,” lists the data types of Tandem languages.

System Dependencies The features mentioned in this manual are supported on all currently supported systems except where noted. Table 2 lists the systems that TAL supports:

Table 2. Tandem Systems

System Name	Description	Operating System
Tandem NonStop Series (TNS) system	Based on complex instruction set computing (CISC) technology—a large instruction set, numerous addressing modes, multicycle machine instructions, and special-purpose instructions	C-series and D-series versions
Tandem NonStop Series/RISC (TNS/R) system	Based on reduced instruction set computing (RISC) technology—a small, simple instruction set, general-purpose registers, and high-performance instruction execution	C30 and D20 versions and later

Programs That Run on the TNS System

All programs written for the C-series TNS system can run on a D-series TNS system without modification. You can modify C-series application programs to take advantage of D-series features, as described in the *Guardian Application Conversion Guide*.

Programs That Run on a TNS/R System

Most programs written for TNS systems can run on a TNS/R system without modification. Low-level programs, however, might need modification as described in the *Guardian Application Conversion Guide*.

The *Accelerator Manual* tells how to accelerate a TNS program to make it run faster on a TNS/R system. An accelerated object file contains:

- The original TNS object code and related Binder and symbol information
- The accelerated (RISC) object code and related address map tables

Future Software Platforms

The storage allocation conventions described in this manual apply only to current software platforms. For portability to future software platforms, do not write programs that rely on the spatial relationships shown for variables and parameters stored in memory. More specific areas of nonportability are noted in this manual where applicable.

Compiler Dependencies The TAL compiler is a disk-resident program on each Tandem system. In general, a particular version of the compiler runs on the corresponding or later version of the operating system. For example, the D20 version of the compiler requires at least the D20 version of the operating system.

If you need to develop and maintain C-series TAL applications on a D-series system, the following files must be restored from the C-series system:

C-Series File to Restore	Description
TAL	TAL compiler
TALERROR	TAL error messages
TALLIB	TAL run-time library
TALDECS	TAL external declarations
FIXERRS	TACL macro for correcting TAL source files
BINSERV	Binder server for compilers
SYMSERV	Symbol-table server for compilers

The C-series compiler expects a C-series BINSERV and SYMSERV in the same subvolume (although you can use the PARAM command to specify a BINSERV and SYMSERV in a different subvolume). C-series tool files (such as BIND and CROSSREF) can also be restored.

To compile a C-series compilation unit on a D-series system, you must use the fully qualified name of the C-series compiler; for example:

```
$myvol.mysubvol.TAL / IN mysrc / myobj
```

Additional Information Table 3 describes manuals that provide information about Tandem systems.

Table 3. System Manuals

Manual	Description
<i>Introduction to Tandem NonStop Systems</i>	Provides an overview of the system hardware and software.
<i>Introduction to D-Series Systems</i>	Provides an overview of D-series enhancements to the Guardian operating system.
<i>System Description Manual</i>	Describes the system hardware and the process-oriented organization of the operating system.
<i>TACL Reference Manual</i>	Describes the syntax for specifying TACL command interpreter commands.
<i>D-Series System Migration Planning Guide</i>	Gives guidelines for migrating from a C-series system to a D-series system.

Table 4 describes manuals that provide information about programming in the Tandem environment.

Table 4. Programming Manuals

Manual	Description
<i>Guardian Procedure Calls Reference Manual</i>	Describes the syntax and programming considerations for using system procedures.
<i>Guardian Programmer's Guide</i>	Explains how to use the programmatic interface of the operating system.
<i>Guardian Procedure Errors and Messages Manual</i>	Describes error codes, error lists, system messages, and trap numbers for system procedures.
<i>Guardian Application Conversion Guide</i>	Gives guidelines for converting C-series TNS programs to D-series TNS programs, and for converting TNS programs to TNS/R programs.
<i>Accelerator Manual</i>	Tells how to accelerate TNS object files for a TNS/R system.
<i>Common Run-Time Environment (CRE) Programmer's Guide</i>	Explains how to use the CRE for running mixed-language programs written for D-series systems.
<i>NonStop SQL Programming Manual for TAL</i>	Describes the syntax for embedding SQL statements in TAL programs.

Table 5 describes manuals that provide information about program development tools.

Table 5. Program Development Manuals

Manual	Description
<i>PS Text Edit Reference Manual</i>	Explains how to create and edit a text file using the PS Text Edit full-screen text editor.
<i>Edit User's Guide and Reference Manual</i>	Explains how to create and edit a text file using the Edit line and virtual-screen text editor.
<i>Binder Manual</i>	Explains how to bind compilation units (or modules) using Binder.
<i>Crossref Manual</i>	Explains how to collect cross-reference information using the stand-alone Crossref product.
<i>Inspect Manual</i>	Explains how to debug programs using the Inspect source-level and machine-level interactive debugger.
<i>Debug Manual</i>	Explains how to debug programs using the Debug machine-level interactive debugger.

**Your Comments
Invited**

After you have had a chance to use this manual, please take a moment to fill out the Reader Comment Card and send it to us. The Reader Comment Card is located at the back of the printed manual and as a separate file in the CD Read Document List. You can fax the card to us at (408) 285-6660 or mail the card by using the business reply address on the back of the card in the printed manual. Many of the improvements you see in Tandem manuals are a result of suggestions from our customers. Please take this opportunity to help us improve future manuals.

Notation Conventions

This manual presents terms that represent keywords, information you supply, and identifiers as follows:

- Keywords are shown in uppercase in text and in examples. For instance, the keyword SOURCE appears in an example as follows:

```
?SOURCE . . .
```

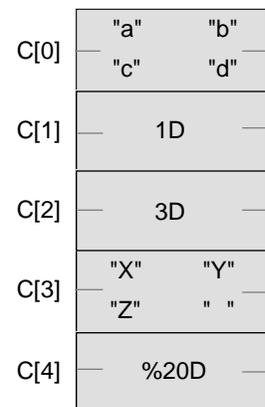
- Identifiers are shown in lowercase in examples and in uppercase in example comments and in text. For instance, the identifier TOTAL appears in an example as follows:

```
INT total;
```

- Terms that represent user-specified information are shown in lowercase italics. For instance, when you declare a variable, you specify an *identifier*.

Some of the examples in this manual include diagrams that illustrate memory allocation. The diagrams are two bytes wide. Unless otherwise noted, the diagrams refer to locations in the primary area of the user data segment. The following example shows allocation of an INT(32) array and its initializing values. In the diagram, solid lines depict borders of storage units, in this case doublewords. Short lines depict words within each doubleword:

```
INT(32) c[0:4] :=  
["abcd", 1D, 3D, "XYZ ", %20D];  
  
!Declare an array and initialize  
! the array elements with values  
! specified in a constant list
```



333

1 Introducing TAL

The Transaction Application Language (TAL) is a high-level, block-structured language that works efficiently with the system hardware to provide optimal object program performance. The TAL compiler compiles TAL source programs into executable object programs. The compiler and the object programs it generates execute under control of the Tandem NonStop Kernel.

Using TAL You use TAL most often for writing systems software or transaction-oriented applications where optimal performance has high priority. You can, for example, use TAL to write the kinds of software listed in Table 1-1.

Table 1-1. Uses of TAL

Kind of Software	Examples
Systems software	Operating system components Compilers and interpreters Command interpreters Special subsystems Special routines that support data communication activities
Applications software	Server processes used with Tandem data management software Conversion routines that allow data transfer between Tandem software and other applications Procedures that are callable from programs written in other languages Applications that require optimal performance

Many Tandem software products are written in TAL.

Major Features The major features of TAL are:

- Procedures**—Each program contains one or more procedures. A procedure is a discrete sequence of declarations and statements that performs a specific task. A procedure is callable from anywhere in the program.

Each procedure executes in its own environment and can contain local data that is not affected by the actions of other procedures. When a procedure calls another procedure, the operating system saves the caller's environment and restores the environment when the called procedure returns control to the caller.
- Subprocedures**—A procedure can contain subprocedures, callable only from within the same procedure. When a subprocedure calls another subprocedure, the caller's environment remains in place. The operating system saves the location in the caller to which control is to return when the called subprocedure terminates.
- Private data area**—Each activation of a procedure or subprocedure has its own data area. When each activation terminates, it relinquishes its private data area, thereby keeping the amount of memory used by a program to a minimum.

- Recursion—Because each activation of a procedure or subprocedure has its own data area, a procedure or subprocedure can call itself or can call other procedures that in turn calls the original procedure.
- Parameters—A procedure or subprocedure can have optional or required parameters. The same procedure or subprocedure can process different sets of variables sent by different calls to it.
- Data types—You can declare and reference the following types of data:

Data Type	Description
STRING	8-bit integer byte
INT, INT(16)	16-bit integer word
INT(32)	32-bit integer doubleword
FIXED, INT(64)	64-bit fixed-point quadrupleword
REAL, REAL(32)	32-bit floating-point doubleword
REAL(64)	64-bit floating-point quadrupleword
UNSIGNED(<i>n</i>)	<i>n</i> -bit field, where $1 \leq n \leq 31$

- Data sets—You can declare and use sets of related variables such as arrays and structures (records).
- Pointers—You can declare pointers (variables that can contain byte addresses or word addresses) and use them to access locations throughout memory. You can store addresses in them when you declare them or later in your program.
- Data operations—You can copy a contiguous group of words or bytes and compare one group with another. You can scan a series of bytes for the first byte that matches (or fails to match) a given character.
- Bit operations—You can perform bit deposits, bit extractions, and bit shifts.
- Standard functions—You can use built-in functions, for example, to convert data types and addresses, test for an ASCII character, or determine the length, offset, type, or number of occurrences of a variable.
- Compiler directives—You can use directives to control a compilation. You can, for example, check the syntax in your source code or control the content of compiler listings.
- Modular programming—You can divide a large program into modules, compile them separately, and then bind the resulting object files into a new object file.
- Mixed-language programming—You can use NAME and BLOCK declarations, procedure declaration options—such as public name, language attribute, and parameter pairs—and compiler directives in support of mixed-language programming.
- NonStop SQL features—You can use compiler directives to prepare a program in which you want to embed SQL statements.

System Services Your program can ignore many things such as the presence of other running programs and whether your program fits into memory. For example, programs are loaded into memory for you and absent pages are brought from disk into memory as needed.

System Procedures The file system treats all devices as files, including disk files, disk packs, terminals, printers, and programs running on the system. File-system procedures provide a file-access method that lets you ignore the peculiarities of devices. Your program can refer to a file by the file's symbolic name without knowing the physical address or configuration status of the file.

Your program can call system procedures that activate and terminate programs running in any processor on the system. Your program can also call system procedures that monitor the operation of a running program or processor. If the monitored program stops or a processor fails, your program can determine this fact.

System procedures are described in the *Guardian Procedure Calls Reference Manual* and the *Guardian Programmer's Guide* for your system.

TAL Run-Time Library The TAL run-time library provides routines that:

- Initialize the Common Run-Time Environment (CRE) when you use D-series compilers (as described later in this manual)
- Prepare a program for SQL statements (as described in the *NonStop SQL Programming Manual for TAL*)

CRE Services The CRE provides services that support mixed-language programs compiled on D-series compilers. A mixed-language program can consist of C, COBOL85, FORTRAN, Pascal, and TAL routines.

A **routine** is a program unit that is callable from anywhere in your program. The term routine can represent:

- A C function
- A COBOL85 program
- A FORTRAN program or subprogram
- A Pascal procedure or function
- A TAL procedure or function procedure

When you use the CRE, each routine in your program, regardless of language, can:

- Use the routine's run-time library without overwriting the data of another run-time library
- Share data in the CRE user heap
- Share access to the standard files—standard input, standard output, and standard log
- Call math and string functions provided in the CRELIB file
- Call Saved Messages Utility (SMU) functions provided in the Common Language Utility Library (CLULIB file)

Without the CRE, only routines written in the language of the MAIN routine can fully access their run-time library. For example, if the MAIN routine is written in TAL, a routine written in another language might not be able to use its own run-time library.

Section 17, "Mixed-Language Programming," gives CRE guidelines for TAL programs.

The CRE Programmer's Guide describes the services provided by the CRE, including the math, string, and SMU functions.

2 Getting Started

To get you started quickly, this section presents a sample program and describes the basic commands for developing and running a program.

Sample Source File The sample program adds two numbers together. The source code for the sample program is in a source file named MYSRC. Figure 2-1 shows the content of the sample source file.

Figure 2-1. Sample Source File

```
!This is a source file named MYSRC.
?SOURCE $SYSTEM.SYSTEM.EXTDECS (INITIALIZER)
                                           !Include system procedure

PROC myproc MAIN;                        !Declare procedure MYPROC
  BEGIN
    INT var1;                             !Declare variables
    INT var2;
    INT total;

    CALL initializer;                      !Handle start-up message
    var1 := 5;                             !Assign value to VAR1
    var2 := 10;                            !Assign value to VAR2
    total := var1 + var2;                  !Assign sum to TOTAL
  END;                                     !End MYPROC
```

This section describes how to:

1. Create the source file
2. Compile the source file into an object file
3. Run the object file

Creating Source Files A source file contains source code for your program. The source code can include declarations, statements, compiler directives, and comments.

You can use a text editor to create a file and to type source code into the file. The *PS Text Edit Reference Manual* and the *Edit User's Guide and Reference Manual* describe how to use an editor.

When you enter the source code into the source file, you can use lowercase or uppercase letters. The compiler does not distinguish between lowercase and uppercase.

The sample source file shows keywords in uppercase and identifiers in lowercase to make it easy to see which is which. Keywords are terms that have predefined meanings to the compiler. Identifiers are names you supply.

The sample source file contains the following kinds of items:

- Compiler directives
- System procedures
- Procedures
- Data declarations
- Statements
- Comments

Compiler Directives Compiler directives let you select compilation options that control various aspects of the compilation. For example, you can use compiler directives to:

- Include source code from other source files
- Control the listing
- Perform conditional compilation
- Check the syntax without generating object code

You can include compiler directives in the source file or in the command to run the compiler. In the source file, you specify compiler directives in directive lines. Each directive line begins with a question mark in the leftmost column.

In the sample source file, the SOURCE directive reads the INITIALIZER system procedure, which handles the startup message (a system message sent to a process when it starts).

- System Procedures** System procedures are procedures provided by the operating system. Declarations for system procedures are located in the EXTDECS files. You can, for instance, use system procedures when your program needs to:
- Perform input/output operations
 - Manage processes
 - Communicate with other processes
 - Interface with the command interpreter (TACL)
 - Interface with terminals, printers, magnetic tapes, and card readers
 - Send operator messages
 - Provide fault tolerance
 - Handle traps
- Appendixes A and B contain examples that include input/output and other system procedures. For information on how to use the system procedures, however, see the *Guardian Programmer's Guide* and the *Guardian Procedure Calls Reference Manual*.
- The sample source file includes one system procedure, INITIALIZER, which processes the startup message.
- Procedures** Procedures contain the executable parts of a TAL program. Procedures can contain data declarations, statements, and subprocedures.
- A TAL program consists of one or more procedures. The program must include a MAIN procedure (a procedure that has the MAIN attribute). The MAIN procedure executes first when you run the program.
- The sample source file consists of one procedure, named MYPROC. This procedure has the MAIN attribute and contains data declarations and statements.
- Data Declarations** Data declarations associate identifiers with memory locations and allocate storage space for variables. Variables contain data that can change during program execution. You can initialize variables with values when you declare the variables, or you can assign values to them later in assignment statements.
- The sample source file includes data declarations for variables named VAR1, VAR2, and TOTAL. These variables are not initialized when they are declared; they are assigned values later in the source file.

Statements Statements let you specify the actions you want a program to perform. All statements are executable. For example, you use statements to:

- Call procedures
- Return from called procedures
- Assign values to variables
- Copy data from one location to another
- Scan data for a character
- Select statements to execute based on a condition

The sample source file includes statements that call a procedure and assign values to variables. The following statements appear in the sample source file:

- A CALL statement that calls the INITIALIZER system procedure
- An assignment statement that assigns a value to VAR1
- An assignment statement that assigns a value to VAR2
- An assignment statement that assigns the sum of VAR1 and VAR2 to TOTAL

Comments Comments are notes you include in the source file to explain the source code. For example, you can use a comment to explain a construct or describe an operation. Comments in a source file can either:

- Start with two hyphens (--) and terminate with the end of the line
- Start with an exclamation point (!) and terminate with either another exclamation point or the end of the line

In the sample source file, each comment begins with an exclamation point and ends with the end of the line.

Compiling Source Files

When you compile a source file, the compiler produces an object file and a compiler listing. The compiler listing consists of source code and summary information. You can execute the object file if it contains a procedure that has the MAIN attribute.

To compile the sample source file MYSRC, issue the following compilation command at the TACL prompt:

```
TAL /IN mysrc/ myprog
```

The preceding command sends the compiler listing to your terminal and the object code to an object file named MYPROG. You can include compiler directives and additional run options in the compilation command. Section 14, "Compiling Programs," gives an overview of run options and compiler directives.

Running Programs After the compiler generates an object file for your program, you can run the program by issuing a TACL RUN command.

To run the sample program MYPROG, issue the following command at the TACL prompt:

```
RUN myprog
```

You can include run options in the TACL RUN command. Section 16, “Running and Debugging Programs,” gives an overview of some commonly used run options. For more information on run options, see the *TACL Reference Manual*.

The remainder of this manual describes the structure of a TAL source file, variable declarations, expressions, statements, procedure declarations, and other language- and compiler-specific information.

3 Structuring Programs

This section describes how you can structure and format a TAL source program. The structure is the order and level at which major components appear in a program. The format is the spacing and alignment you use to make the program readable.

-
- Source Files** A program consists of one or more source files. A source file can be a complete program or a part of a modular program. In modular programming, for instance, you can:
- Divide a large program into smaller, more manageable source files
 - Work independently on a source file while other programmers work on other source files
 - Compile and debug each source file separately.
 - Bind new object code to existing debugged object code, including general-purpose library routines
 - Use other languages, such as C or COBOL, for some of the source files
 - Group procedures into source files by the kinds of tasks the procedures perform; for example, a source file can provide input/output (I/O) processing and another can provide error processing

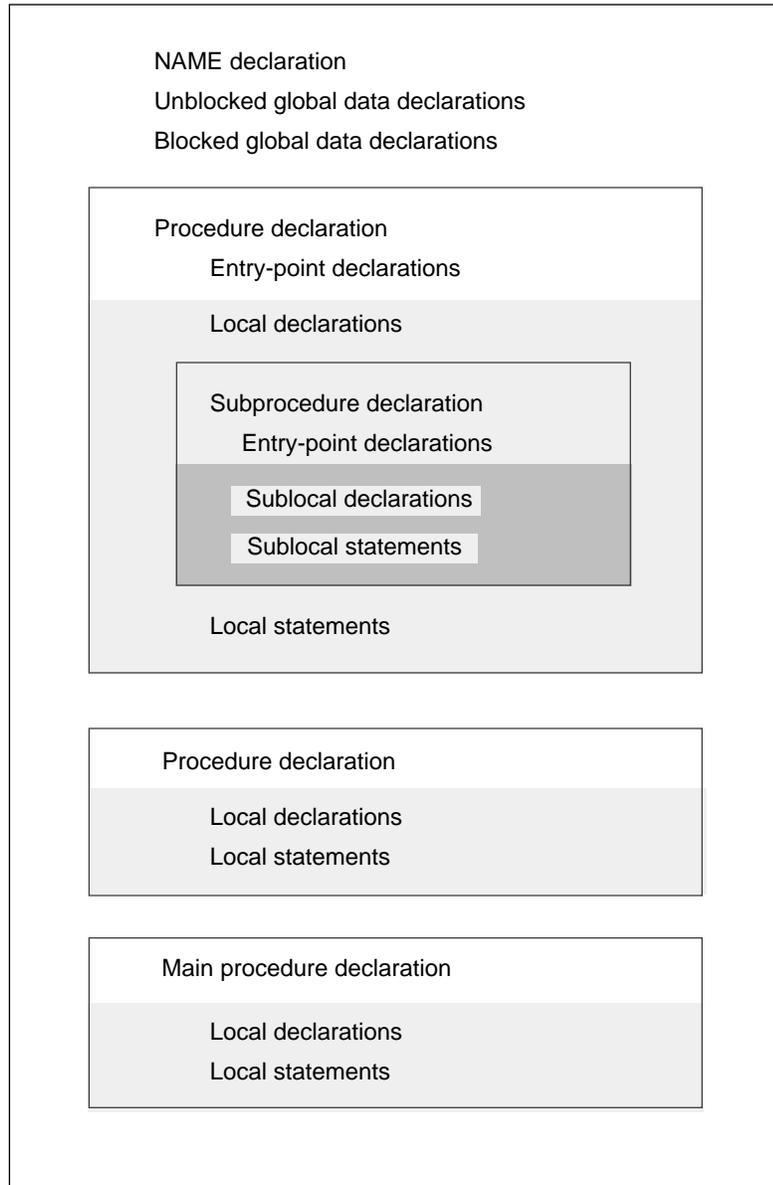
Compilation Units The input to the compiler is a single source file. The source file, however, can contain SOURCE directives that read in code from other source files. The source file together with code from other source files that are read in by SOURCE directives compose a compilation unit.

When you compile a compilation unit, the output is an object file that you can bind with other object files into a new object file, as described in the *Binder Manual*.

Structuring Compilation Units The structure of a compilation unit is the order and level at which major components appear. Figure 3-1 shows the structure of a sample compilation unit.

Not all of the components shown in the figure need to appear in a compilation unit. For example, if your compilation unit is a complete program, instead of a part of a modular program, the NAME declaration need not appear.

Figure 3-1. Structure of a Sample Compilation Unit



- Global scope
- Local scope
- Sublocal scope

Figure 3-1 shows three procedures and a subprocedure. You can declare any number of procedures in a compilation unit. You can declare any number of subprocedures within any procedure. You cannot, however, declare a subprocedure within another subprocedure.

The shading in Figure 3-1 represents the scope of identifiers—that is, the set of levels at which you can access identifiers:

- Identifiers in the unshaded areas have global scope. You can usually access global identifiers from all compilation units in the program and from within the current compilation unit.
- Identifiers in the light gray areas have local scope. You can access local identifiers only from within the encompassing procedure.
- Identifiers in the dark gray area have sublocal scope. You can access sublocal identifiers only from within the encompassing subprocedure.

Order of Components To structure a compilation unit, place declarations and statements in the following order:

1. The NAME declaration, if present
2. Any unblocked global data declarations
3. Any blocked global data declarations
4. Procedure declarations. Within each procedure:
 - a. Any local data declarations
 - b. Any subprocedure declarations. Within each subprocedure:
 - 1) Any sublocal data declarations
 - 2) Any sublocal statements
 - c. Any local statements

The TAL compiler is a single-pass compiler. The prescribed ordering enables the compiler to recognize the scope and other characteristics of your data. For example, you must declare variables before you use their identifiers in statements.

The following subsections give more information on how to specify program components.

**Naming
Compilation Units**

You can assign an identifier to the compilation unit by using the NAME declaration. If present, the NAME declaration must be the first declaration in the compilation unit. For example, you can assign the identifier INPUT_MODULE to a compilation unit as follows:

```
NAME input_module;
```

Normally, you include the NAME declaration in compilation units that you compile separately and then bind together by using Binder. You must include the NAME declaration in a compilation unit that contains blocked global data (those declared within BLOCK declarations).

Declaring Data

You declare data to associate identifiers with memory locations. Table 3-1 lists the data items you must declare before you access them.

Table 3-1. Data Declarations

Data Item	Description
LITERAL	A named constant
DEFINE	A named sequence of text
Simple variable	A variable that contains one element of a specified data type
Array	A variable that contains multiple elements of the same data type
Structure	A variable that contains variables of different data types
Simple pointer	A variable that contains a memory address, usually of a simple variable or an array element, which you can access with this simple pointer
Structure pointer	A variable that contains the memory address of a structure, which you can access with this structure pointer
Equivalenced variable	An alternate identifier and description for a previously declared variable

When you declare a variable, you specify at least a data type and an identifier. For example, you can declare a simple variable named TOTAL of data type INT as follows:

```
INT total;
```

Declaring Global Data Global data is data that can be accessed by all compilation units in the program. Global data can be any data item listed in Table 3-1 earlier in this section.

All global data declarations except LITERALS and DEFINES must appear before any procedure declarations. Global LITERAL and DEFINE declarations can appear between procedures as well.

You can declare unblocked or blocked data.

Declaring Unblocked Global Data Unblocked global data are those you declare outside of BLOCK declarations (described next). Place all unblocked data declarations after the NAME declaration, if present, and before any BLOCK declarations. Identifiers of unblocked global data are accessible to all compilation units in the program.

Declaring Blocked Global Data Blocked global data are those you declare within BLOCK declarations. The BLOCK declaration lets you group global data into named or private data blocks. All BLOCK declarations must appear after the NAME declaration and any unblocked data declarations and before the first procedure declaration.

Declaring Named Data Blocks

Use a named data block for global data you want to share with other compilation units in the program. You can include any number of named data blocks in a compilation unit. The data declarations in a data block can be any data type:

```
BLOCK shared_data;  
  INT flag := True;  
  INT(32) index := 0;  
  STRING count := 0;  
END BLOCK;
```

Declaring Private Data Blocks

Use a private data block for global data you want to share only with procedures within the current compilation unit. You can declare only one private data block in a compilation unit. The private data block inherits the identifier you specify in the NAME declaration.

```
BLOCK PRIVATE;  
  INT average;  
  INT total;  
END BLOCK;
```

For more information about global data blocks, see Section 14, “Compiling Programs.”

Declaring Procedures You declare procedures to define routines that are callable from routines in any compilation unit in the program. You can declare procedures that perform discrete operations such as I/O or error handling. Normally, a procedure declaration includes a procedure heading and a procedure body.

Procedure Heading For the procedure heading, include the following:

- The keyword PROC
- The procedure identifier
- Identifiers of formal parameters, if any
- Procedure attributes, if any
- Declarations of formal parameters, if any

A procedure heading that has no formal parameters or attributes looks like this:

```
PROC my_heading;
```

The following procedure heading has one parameter (named PARAM) and the MAIN attribute. MAIN means execute this procedure first. The second line declares the formal parameter as being of data type INT:

```
PROC my_proc (param) MAIN;
    INT param;
```

A function is a procedure that returns a value to the caller. You declare a function as you do any other procedure except that in the function heading you specify a data type for the return value. Here is a function heading that specifies a return data type of INT:

```
INT PROC my_function (param);
    INT param;
```

Procedure Body The procedure body is a BEGIN-END construct that can contain local data declarations, subprocedure declarations, and local statements. Here is an example:

```
PROC myproc;
    BEGIN
        INT var1;                !Local data declarations
        INT var2;
        !Some code here
        var1 := var1 - var2;      !Local assignment statement
    END;
```

FORWARD Procedures If you need to call a procedure before you declare the procedure body, first declare a procedure heading that includes the FORWARD keyword. Once you declare a FORWARD procedure, you can call the procedure from anywhere in the current compilation unit. For example, you can declare a FORWARD procedure named TO_COME like this:

```
PROC to_come;
    FORWARD;
```

EXTERNAL Procedures If you want to call a procedure that is compiled in another compilation unit, first declare a procedure heading that includes the EXTERNAL keyword. Once you declare an EXTERNAL procedure, you can call the procedure from anywhere in the current compilation unit. For example, you can declare an EXTERNAL procedure named FARAWAY like this:

```
PROC faraway;
  EXTERNAL;
```

Procedure Entry Points A procedure entry point is an identifier by which callers can call a procedure. The primary entry point is the procedure identifier.

Secondary entry points are entry-point identifiers declared within the procedure and then placed at statements where the procedure can begin executing. The following example declares entry-point identifier ENTRY1 and places the identifier at a statement:

```
PROC myproc (param);           !Declare the procedure
  INT param;
  BEGIN
  ENTRY entry1;                !Declare entry-point
                                ! identifier
  INT var;
  !Some code here
entry1:                          !Apply entry-point
  var := var - param;          ! identifier to statement
  !More code
END;
```

Any procedure or subprocedure in the compilation unit can call an entry-point identifier. The caller must pass parameters as if the procedure identifier were being called. The called procedure begins executing at the location of the entry-point identifier. At each activation of an entry-point identifier, all local variables receive their initial values.

The FORWARD declaration for the preceding ENTRY1 entry point is:

```
PROC entry1 (param);
  INT param;
  FORWARD;
```

The EXTERNAL declaration for the ENTRY1 entry point is:

```
PROC entry1 (param);
  INT param;
  EXTERNAL;
```

Local Data Local data is data you declare inside a procedure and can access only from within that procedure. Local data can be any data item described in Table 3-1 earlier in this section. Within a procedure, all local data declarations must appear before any subprocedure declarations or local statements. Following is an example of how you declare local data:

```
PROC p;
  BEGIN
    INT total;                !Declare local data
    INT num;

    !Subprocedures go here, if any

    !Local statements go here

  END;
```

Local Labels You can declare labels to reserve identifiers for later use as identifiers of locations in the procedure. For example, you can declare a label named LABEL_ONE and place it at a statement. Here is an example:

```
PROC x;
  BEGIN
    INT var;
    LABEL label_one;        !Declare a local label

    !Lots of statements

  label_one :              !Place the label at this
    var := 5;              ! assignment statement
    !More statements

  END;
```

You can place the label identifier at the beginning of any statement, and then access the label identifier from within the encompassing procedure. (You can apply label identifiers without declaring them, but declaring them reserves their identifiers.)

Local Statements Statements perform specific operations. Local statements are those you include in a procedure but outside a subprocedure. For example, to store a value in a variable, you can use an assignment statement as follows:

```
PROC nonsense;
  BEGIN
    INT local_var;

    local_var := 1000;      !Local assignment statement

  END;
```

Table 3-2 lists the statements you can include in a TAL program. (Statement names in uppercase reflect a keyword. The assignment and move statements have no keywords in them; their names are in lowercase.)

Table 3-2. TAL Statements

Statement	Operation
ASSERT	Conditionally calls an error-handling procedure
Assignment	Stores a value in a variable
CALL	Invokes a procedure or a subprocedure
CASE	Selects a set of statements based on a selector value
CODE *	Specifies machine codes or constants for inclusion in the object code
DO	Executes a posttest loop until a true condition occurs
DROP	Frees an index register or removes a label from the symbol table
FOR	Executes a pretest loop for <i>n</i> times
GOTO	Unconditionally branches to a label within a procedure or subprocedure
IF	Conditionally selects one of two possible statements
Move	Copies a contiguous group of items from one location to another
RETURN	Returns from a procedure or a subprocedure to the caller; returns a value from a function. As of the D20 release, it also can return a condition code value
RSCAN	Scans a sequence of bytes, right to left, for a test character
SCAN	Scans a sequence of bytes, left to right, for a test character
STACK *	Loads a value onto the register stack
STORE *	Stores a register stack value in a variable
USE	Reserves an index register
WHILE	Executes a pretest loop while a condition is true

* Not portable to future software platforms.

Local statements can refer to:

- Global identifiers, including procedure identifiers, anywhere in the program
- Local identifiers, including subprocedure identifiers, in the encompassing procedure

Declaring Subprocedures

You declare subprocedures to specify discrete portions of source code within a procedure. You can call subprocedures only from within the encompassing procedure. You can declare any number of subprocedures within a procedure, but you cannot declare subprocedures within subprocedures.

Place any subprocedure declarations following the procedure's local declarations. In a subprocedure declaration, you normally specify a heading and a body.

Subprocedure Heading

In the subprocedure heading, include:

- The keyword `SUBPROC`
- The subprocedure identifier
- Identifiers of formal parameters, if any
- The attribute `VARIABLE`, if needed
- Declarations of formal parameters, if any

For example, a subprocedure heading with no formal parameters or attributes looks like this:

```
SUBPROC my_heading;
```

A subprocedure heading with one parameter (named `PARAM`) and the `VARIABLE` attribute looks like this:

```
SUBPROC my_heading (param) VARIABLE;
      INT param;
```

Subprocedure Body

For the subprocedure body, specify a `BEGIN-END` construct that can contain sublocal data declarations and statements. For example, within the procedure named `MYPROC` you can declare a subprocedure named `MYSUB`, declare sublocal data `VAR1` and `VAR2`, and assign a value to `VAR1` in a sublocal assignment statement:

```
PROC myproc;                                !Declare MYPROC
  BEGIN
    !Local data declarations

    SUBPROC mysub;                            !Declare MYSUB
      BEGIN
        INT var1;
        INT var2;

        var1 := var1 - var2;
      END;                                    !End MYSUB

    !Local statements
  END;                                       !End MYPROC
```

FORWARD Subprocedures

If you need to call a subprocedure before you declare its body, declare a `FORWARD` subprocedure. You can then call it from anywhere in the encompassing procedure.

```
SUBPROC to_come;
  FORWARD;
```

Subprocedure Entry Points A subprocedure entry point is an identifier by which callers can call a subprocedure. The primary entry point is the subprocedure identifier.

Secondary entry points are entry-point identifiers declared within the subprocedure and then placed at statements where the subprocedure can begin executing. The following example declares entry-point identifier `SUB_ENTRY` and places the identifier at a statement:

```
PROC myproc;
  BEGIN
    !Local data declarations

    SUBPROC some_sub (param);      !Declare subprocedure
      INT param;
      ENTRY sub_entry;            !Declare entry-point
                                  ! identifier

      INT var;
      !Some code here
    sub_entry:                    !Apply entry-point
      var := var - param;         ! identifier to statement
      !More code
    END;                          !End subprocedure

    !More subprocedures

    !Local statements
    CALL sub_entry (1);          !Call entry-point SUB_ENTRY
  END;
```

A subprocedure can call an entry-point identifier of any subprocedure within the same procedure. The caller must pass parameters to the entry-point identifier as if it were calling the subprocedure identifier. The called subprocedure begins executing at the location of the entry-point identifier. All the sublocal variables of the called subprocedure receive their initial values each time the subprocedure is activated.

If you need to reference a subprocedure entry-point identifier before you declare the subprocedure, a `FORWARD` declaration is required:

```
SUBPROC sub_entry (param);
  INT param;
  FORWARD;
```

Sublocal Data Sublocal data is data you declare inside a subprocedure and can access only from within the same subprocedure. Sublocal data can be any data item described in Table 3-1 earlier in this section. Within a subprocedure, all sublocal data declarations must appear before any sublocal statements. Also, sublocal arrays and structures must be directly addressed. (Addressing is explained in Section 4, “Introducing the Environment.”) Following is an example of how you declare sublocal data:

```
PROC myproc;  
  BEGIN  
    !Local data declarations  
  
    SUBPROC mysub;  
      BEGIN  
        INT var1;           !Sublocal data declarations  
        INT var2;  
  
        var1 := var1 - var2; !Sublocal statement  
      END;  
  
    !Local statements  
  END;
```

Sublocal Labels You declare sublocal labels as you do local labels, except that you declare sublocal labels within a subprocedure. You can access sublocal labels from anywhere within the encompassing subprocedure.

Sublocal Statements Within a subprocedure, you can specify any statement described in Table 3-2 earlier in this section. Sublocal statements can access:

- Global identifiers, including procedures, anywhere in the program
- Local identifiers, including subprocedures, in the encompassing procedure
- Sublocal identifiers in the encompassing subprocedure

Formatting Programs You can format a program to make it easier to understand and maintain. The TAL compiler allows almost a free format for source code. You can use any format that serves your purposes. The only limitation is that the maximum line length for source code is 132 characters.

Here is an example of a format that is difficult to read:

```
INT num1;INT num2;INT num3;STRING char1;STRING char2;STRING
char3;PROC format_example MAIN; BEGIN num1 := 8; num2 := 5;
num3 := num1 + num2; char1 := "A"; char2 := "B"; char3 :=
"C"; END;
```

Here is an example of a format that is easy to read:

```
INT num1;
INT num2;
INT num3;
STRING char1;
STRING char2;
STRING char3;

PROC format_example MAIN;
  BEGIN
    num1 := 8;
    num2 := 5;
    num3 := num1 + num2;
    char1 := "A";
    char2 := "B";
    char3 := "C";
  END;
```

In the second format, you can readily see each declaration. You can tell where the global declarations end and the procedure begins. You can quickly see what the procedure body contains. You have space to add comments to clarify what is going on in the program.

Formatting With Comments You can insert comments anywhere in the source code to make the code easier to understand and maintain. For instance, you can explain the purpose of certain constructs or variables. During compilation, the compiler ignores comments.

You can specify comments using either of two forms or can use both forms in the same compilation unit:

Beginning Delimiter	Ending Delimiter	Example
Two hyphens	End of line	--One form of comments
Exclamation point	Exclamation point or end of line	!Another form of comments

Explaining the Purpose of Variables

Comments can explain the purpose of variables:

```

INT num1;           --16-bit simple variables
INT num2;           -- to use for processing
INT num3;           -- integer values

STRING char1;       --8-bit simple variables
STRING char2;       -- to use for processing
STRING char3;       -- ASCII characters

PROC format_proc MAIN;  --Declare FORMAT_PROC
  BEGIN
  --Assign values to variables
  num1 := 8;
  num2 := 5;
  num3 := num1 + num2;
  char1 := "A";
  char2 := "B";
  char3 := "C";
  --Code to process the variables
END;                --End FORMAT_PROC

```

Documenting Omitted Parameters

Comments within a CALL statement can help identify omitted parameters:

```

PROC some_proc (index, num, length, limit, total)
  EXTENSIBLE;
  INT index, num, length, limit, .total;
  BEGIN
  !Lots of code
  END;

PROC caller_proc;
  BEGIN
  INT total;
  !Some code
  CALL some_proc (0, !num!, !length!, 40, total);
  END;

```

Skipping Parts of Code

When you want the compiler to ignore a portion of the code, you can either:

- Comment it out
- Use conditional compilation

For example, when your code contains `!` comments, you can use `--` to comment out the portion you want the compiler to ignore:

```
PROC my_proc;
  BEGIN
    !Lots of code
  END;

--Comment out the following portion of code:
--PROC no_proc;
--  BEGIN
--  !Lots of code
--  END;
--End of commented-out portion of code

PROC your_proc;
  BEGIN
    !Lots of code
  END;
```

Alternatively, you can use conditional compilation for the portion you want the compiler to ignore. Here is the preceding example shown with the `DEFINETO`, `IF`, and `ENDIF` conditional compilation directives:

```
PROC my_proc;
  BEGIN
    !Lots of code
  END;

?DEFINETO omit                !Define toggle OMIT without
                               ! changing its off state

?IF omit                       !If OMIT is off,
PROC this_proc;                ! skip THIS_PROC
  BEGIN
    !Lots of code
  END;
?ENDIF omit                    !End of skipped portion

PROC your_proc;
  BEGIN
    !Lots of code
  END;
```

In the preceding example, `DEFINETO` and named toggles are D20 or later features. For pre-D20 systems, you can use `RESETTO` with numeric toggles instead. For more information on these directives, see the *TAL Reference Manual*.

**Formatting With
BEGIN-END Constructs**

You use BEGIN-END constructs to group various items into a single entity. If you align the BEGIN and END keywords vertically, the program is easier to understand. Following are some uses for BEGIN-END constructs.

Procedure or Subprocedure Body

Within a procedure (or subprocedure) declaration, enclose the procedure (or subprocedure) body in a BEGIN-END construct:

```
PROC add;
  BEGIN
    !Procedure body
  END;
```

Structure Layout

Within a structure declaration, enclose the structure layout in a BEGIN-END construct:

```
STRUCT inventory;
  BEGIN
    !Structure item
    !Structure item
    !Structure item
  END;
```

Compound Statements

Place a compound statement in a BEGIN-END construct. A compound statement consists of multiple statements you want treated as a single logical statement:

```
BEGIN
!Statement
!Statement
!Statement
END;
```

For example, you can specify a choice of compound statements in an IF statement as follows:

```
IF a < b THEN
  BEGIN                               !Begin compound statement
    a := 1;
    b := 2;
    c := a + b;
  END                                 !End compound statement
ELSE
  BEGIN                               !Begin compound statement
    a := 5;
    b := 6;
    c := a + b;
  END                                 !End compound statement
```

Using Semicolons Use semicolons to end each data declaration and to separate successive statements.

Ending Data Declarations

Here is an example of using semicolons to end data declarations:

```
INT    num1;
INT    num2;
INT    num3;
STRING char1;
STRING char2;
```

The following example is equivalent to the preceding example:

```
INT    num1, num2, num3;
STRING char1, char2;
```

Separating Successive Statements

You must use a semicolon between successive statements. A semicolon before the last END keyword of a procedure or subprocedure is optional.

```
PROC myproc;
  BEGIN
    INT num1;
    INT num2 := 8;
    INT total;

    num1 := 9;           !Required semicolon
    total := num1 + num2; !Optional semicolon
  END;
```

Using Semicolons Within Statements

Within a statement:

- You can use a semicolon before the END keyword that terminates a compound statement.
- You cannot use a semicolon just before an ELSE or UNTIL keyword.

```
IF a < b THEN
  BEGIN
    a := 1;
    b := 2;           !Optional semicolon
  END               !No semicolon here
ELSE
  a := 0;

DO
  a := a + b         !No semicolon here
UNTIL
  a < b;
```

Null Statements

You can use a semicolon without a statement to create a null statement, which means do nothing. The compiler generates no code for null statements. You can use a null statement anywhere you can use a statement, except immediately before an ELSE or UNTIL keyword.

Here is an example of a null statement embedded in a labeled CASE statement:

```

CASE var OF
  BEGIN
    0 ->
      ;                               !Null statement
    1 ->
      CALL fixit;                     !CALL statement
  END;

```

Using Compiler Directives

Compiler directives are options provided by the TAL compiler so you can control the compilation. For example, compiler directives let you:

- Specify files from which to read in source code
- Specify the content of compiler listings and object files
- Conditionally compile portions of source code

You can specify most compiler directives either in the compilation command (that runs the compiler) or in your source code.

In the compilation command, you can specify directives following the semicolon. For example, to compile the source file MYSRC, send the object code to object file MYOBJ, and specify the NOLIST directive to suppress the compiler listing, issue the following command at the TACL prompt:

```
TAL /IN mysrc/ myobj; NOLIST
```

In your source file, you specify directives in directive lines. Start each directive line with a question mark (?) in column 1 as follows:

```

?LIST
!Some code here
?NOLIST, NOCODE, INSPECT, SYMBOLS, NOMAP, NOLMAP, GMAP
?CROSSREF, INNERLIST

```

The following directive line has a continuation line for the argument list of the SEARCH directive:

```

?SEARCH (file1, file2, file3, file4,
?      file5, file6)

```

When the argument list of a directive continues on a subsequent line, you must specify at least the leading parenthesis of the argument list on the same line as the directive name:

```
?PUSHLIST, NOLIST, SOURCE $SYSTEM.SYSTEM.EXTDECS (  
?                                     PROCESS_GETINFO_, PROCESS_STOP_)  
?POPLIST
```

4 Introducing the Environment

This section introduces you to:

- The process environment in which your program runs
- Addressing modes you can use in this environment
- Allocation of data storage by the compiler
- System dependencies

Process Environment Your object programs execute as individual processes on a Tandem system. A program is a static group of machine instructions and initialized data that reside in a file. The same program can execute concurrently many times, and each execution comprises a separate process.

A process is a dynamically running program. Each process has its own user data space in memory and process information maintained by the operating system. The instruction codes of a process reside in the code space; they manipulate variable data that reside in the data space.

The environment for your process includes:

- Code space (user and library)
- Data space (user and extended)
- System code space
- System library space
- Registers

Code Space The code space of your process consists of:

- An optional library code space
- A user code space

If your process does not include library space, the user code space can contain 1 to 32 code segments. If your process includes library space, the user code space can contain 1 to 16 user code segments and 1 to 16 library code segments.

During program execution, the operating system automatically allocates memory for code segments as needed, keeps track of which code segment is current, and performs segment switching when necessary.

A code segment contains instruction codes and some program constants. During execution, processes can read, but not modify, the content of a code segment.

A code segment consists of up to 65,536 words, which have consecutive addresses from C[0] through C[65535]. (C represents the code space.)

Data Space The current data space of your process consists of:

- A user data segment
- An automatic extended data segment if needed
- Any user-defined (explicit) extended data segments

(The term **segment** refers a nonextended segment except where the word **extended** is specifically used.)

User Data Segment

The user data segment provides a private storage area for the variables of your process. Your process can modify the content of the user data segment.

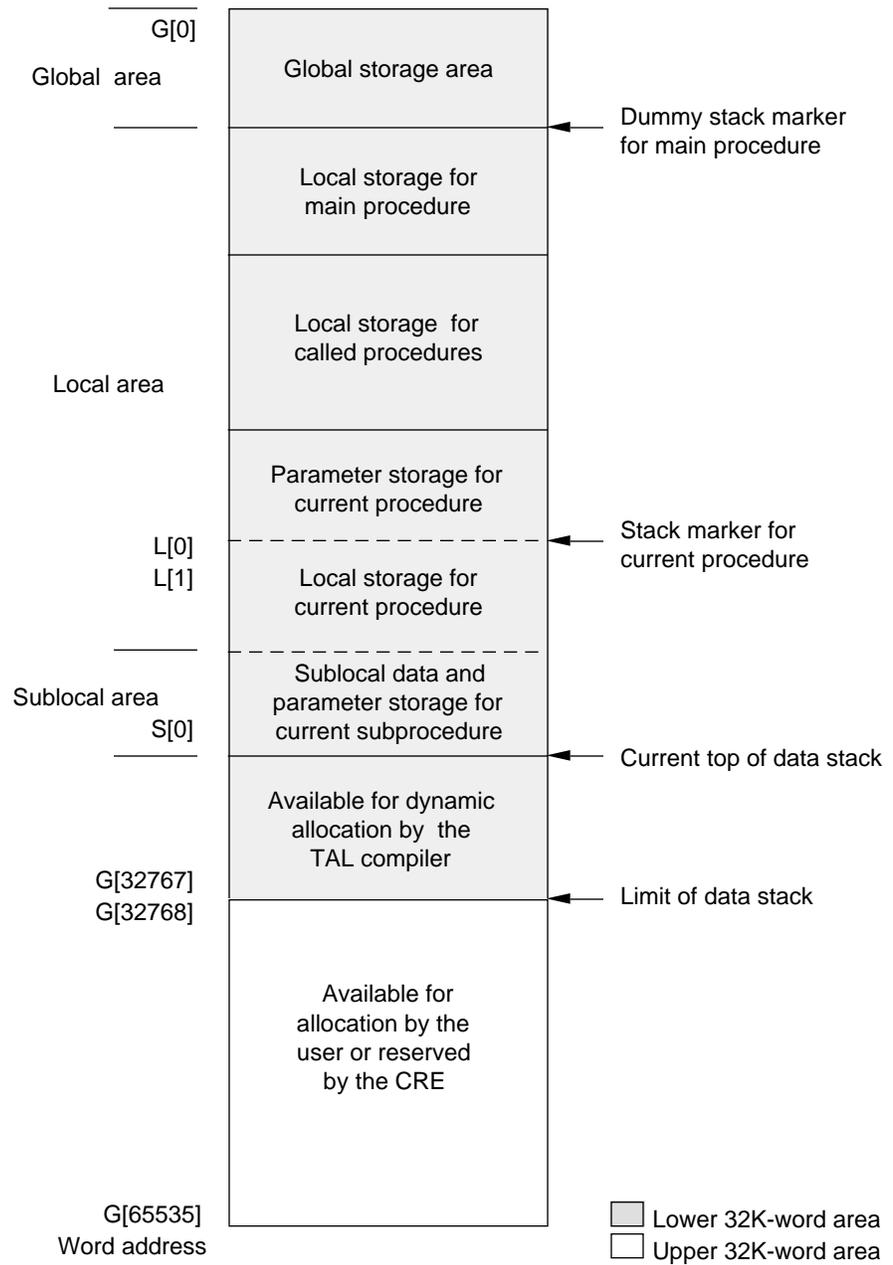
The system provides the user data segment automatically. This segment contains up to 65,536 words, addressed consecutively from G[0] through G[65535]. (G represents the global storage area.)

The lower half of the user data segment contains the global storage area and the data stack. During the execution of your process, the system stores the process' global data in the global area. During activation of a procedure, the system stores the procedure's data in the local area of the data stack. During activation of a subprocedure, the system stores the subprocedure's data in the sublocal area of the data stack.

The upper half of the user data segment provides memory that you can allocate for your data if you do not use the CRE. Appendix B, "Managing Addressing," gives information on using the upper half of the user data segment without the CRE.

Figure 4-1 shows the organization of the user data segment.

Figure 4-1. User Data Segment



Extended Data Segments

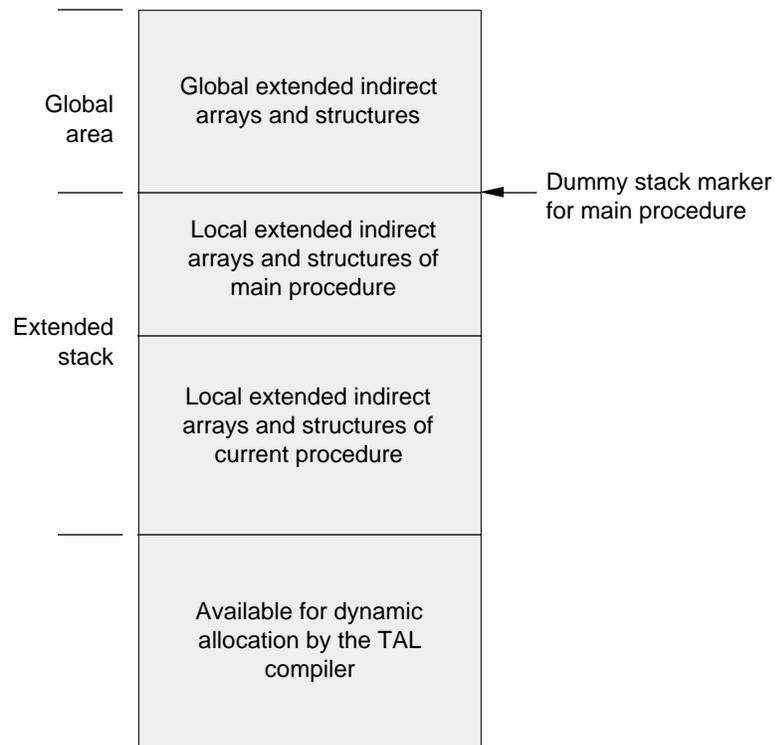
Extended data segments provide indirect data storage in addition to storage in the user data segment. References to extended data segments are not as fast as those to the user data segment. An extended data segment can be as large as 127.5 megabytes.

When you declare arrays and structures by specifying the extended indirection symbol (.EXT), the compiler automatically allocates and deallocates an appropriately sized extended segment. The segment size is fixed during compilation, and the full segment is allocated in one step. If you need a larger extended segment, use the LARGESTACK directive (described in the *TAL Reference Manual*).

As shown in Figure 4-2, the automatic extended segment contains:

- A global area—for global extended indirect arrays and structures
- An extended stack—for local extended indirect arrays and structures

Figure 4-2. Automatic Extended Data Segment



It is recommended that you use only the automatic extended data segment if possible. If you must also allocate explicit extended data segments, follow the instructions in Appendix B, “Managing Addressing.”

- System Code Space** The current system code space available to your process contains one segment of system code.
- System Library Space** The current system library space contains up to 31 segments of system library code, available to your process one at a time.
- Registers** Table 4-1 lists the registers that describe your process in the current process environment:

Table 4-1. Registers in Current Process Environment

Register	Content
Program (P) register	Contains the address of the next instruction to execute in the current code segment
Instruction (I) register	Contains the instruction currently executing in the current code segment
Local (L) register	Contains the address of the beginning of the local data area for the current procedure
Stack (S) register	Contains the address of the last allocated word in the data stack for the current procedure or subprocedure
Register stack	Contains registers (R0 through R7) that the compiler uses for arithmetic and other operations. The compiler also uses R5, R6, and R7 as index registers.
Environment (E) register	Contains information about the current process, such as the current RP value and whether traps are enabled
Register pointer (RP)	A field of the environment register that points to the current top of the register stack

Addressing Modes When you declare a variable, you specify its addressing mode and an identifier and a data type. You can specify these addressing modes:

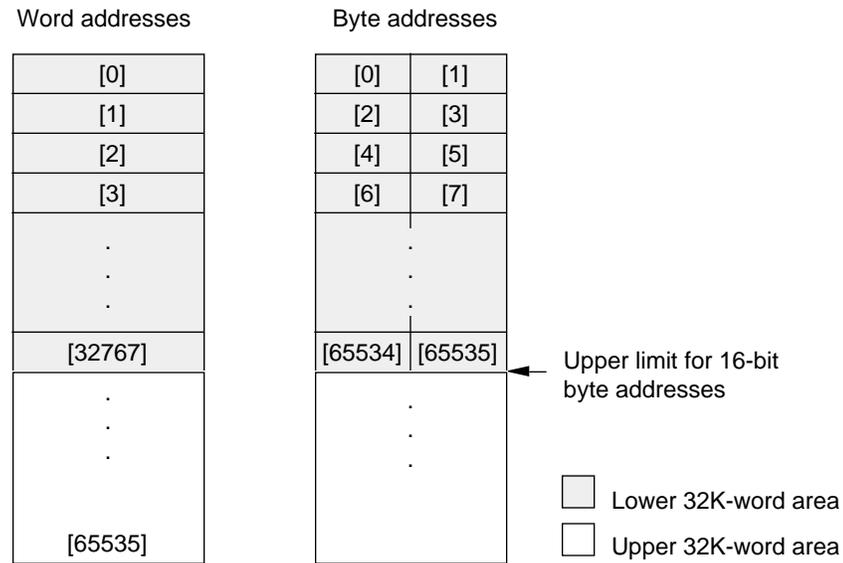
- Direct addressing
- Standard indirect addressing
- Extended indirect addressing

The addressing mode and scope of a variable determines:

- The storage area in which the compiler allocates space for the variable
- The kind of instructions the compiler generates to access the variable

The data type of a variable determines the byte or word addressing mode of the variable. Figure 4-3 shows byte and word addresses in the user data segment. (Extended segments are always byte addressed.)

Figure 4-3. Byte and Word Addressing in User Data Segment



Direct Addressing Direct addressing uses 16-bit addresses and requires only one memory reference. Direct addressing is not absolute; it is relative to the base of the global, local, or sublocal area of the user data segment.

You can use direct addressing only in the lower 32K-word area of the user data segment. To access data in any other area, use standard indirect addressing or extended indirect addressing, described next.

Indirect Addressing Indirect addressing requires two memory references, first to a location that contains an address and then to the data located at the address. You can use indirect addressing to save space in limited storage areas, as described in “Storage Allocation” later in this section.

You specify indirect addressing by using an indirection symbol when you declare indirect arrays, indirect structures, or pointers (including simple pointers and structure pointers). Simple variables are always direct.

- When you declare an indirect array or structure, the compiler automatically provides an implicit pointer, stores a memory address in it, and then allocates the data at that address.
- When you declare a pointer, you must store the memory address of data in the pointer and must manage allocation of the data itself.

You can specify standard indirect addressing or extended indirect addressing.

Standard Indirect Addressing

Standard indirect addressing is 16-bit indirect addressing in the user data segment, including:

- Byte or word addresses in the lower 32K-word area
- Word addresses in the upper 32K-word area

If you use the CRE, however, the upper 32K-word area is not available for your data.

Extended Indirect Addressing

Extended indirect addressing is relocatable 32-bit indirect addressing anywhere in virtual memory, usually in an extended data segment.

(Absolute extended indirect addressing is described in the *System Description Manual* for your system.)

Indexing Indexing can be thought of as being an addressing mode. You can access variables by appending an index to a variable identifier. The index represents an offset, for example, as follows:

- For an array, the index is an element offset from the location of the zeroth element of the array. The element size—byte, word, doubleword, or quadrupleword—depends on the data type of the array.
- For a simple pointer, the index is an element offset from the address contained in the pointer. The element size depends on the data type of the simple pointer.
- For a structure or substructure, the index is an occurrence offset from the location of the zeroth occurrence of the structure or substructure. The occurrence size is the total number of bytes in one occurrence of the structure or substructure, including pad bytes.

In the following example, the index [1] lets you access the second element of MY_ARRAY:

```
INT my_array[0:2];    !Declare MY_ARRAY, a three-element array
my_array[1] := 5;    !Assign 5 to second element of MY_ARRAY
```

The specifics of indexing arrays, structures, pointers, and equivalenced variables are discussed in Sections 7 through 10, respectively.

Storage Allocation The compiler generates code during compilation to allocate storage for variables. The compiler allocates storage as follows:

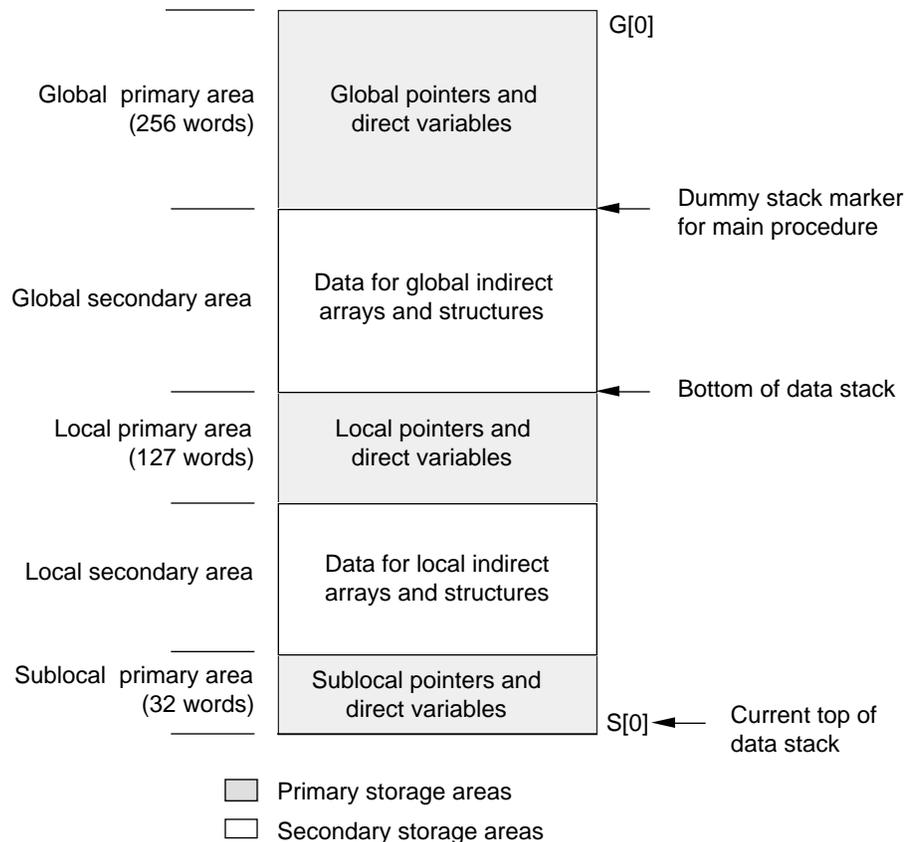
- For global variables—when compilation begins
- For local or sublocal variables—when the encompassing procedure or subprocedure is invoked

The compiler allocates each variable either in the user data segment or in the automatic extended segment, depending on how you declared the variable.

Allocation in the User Data Segment

In general, the compiler allocates storage for pointers and other variables in the lower 32K-word area of the user data segment. Figure 4-4 shows the primary and secondary areas of the lower 32K-word area and the variables each of these areas can contain.

Figure 4-4. Primary and Secondary Storage in the User Data Segment



Global Primary Area

The global primary area can store up to 256 words of the following kinds of global variables:

- Direct variables—simple variables (which are always direct), direct arrays, and direct structures
- Pointers—simple pointers and structure pointers—that you declare
- Implicit pointers—pointers that the compiler provides when you declare indirect arrays or indirect structures
- Two implicit extended-stack pointers (described in “Extended Stack” later in this section)

The compiler allocates storage for each global pointer and each direct global variable at an increasingly higher offset from the beginning of the encompassing global data block. Global data blocks are relocatable during binding.

Local Primary Area

The local primary area for each procedure can store up to 127 words of the following kinds of local variables:

- Direct variables—simple variables (which are always direct), direct arrays, and direct structures
- Pointers—simple pointers and structure pointers—that you declare
- Implicit pointers—pointers that the compiler provides when you declare indirect arrays and indirect structures

When a call to the encompassing procedure occurs, the compiler allocates storage at the current top of the data stack for each local pointer and each direct local variable. The addresses of the variables are pushed to an increasingly higher offset from L[1]. (L represents the local storage area.)

Sublocal Primary Area

The sublocal primary area for each subprocedure can store up to 32 words of the following kinds of sublocal variables:

- Direct variables—simple variables (which are always direct), direct arrays, and direct structures
- Pointers—simple pointers and structure pointers—that you declare

When a call to the encompassing subprocedure occurs, the compiler allocates storage at the current top of the data stack for each sublocal pointer and each direct sublocal variable. The addresses of previously allocated variables are pushed to increasingly negative offsets from S[0]. (S represents the sublocal storage area.)

When all sublocal variables are allocated, the compiler adjusts the top-of-data-stack pointer to point immediately below the last variable allocated. The top of the data stack is illustrated in Figure 4-4 earlier in this section.

Secondary Storage Areas

The secondary storage areas are:

- A global secondary area, which begins following the location of the last global variable allocated in the global primary area
- A local secondary area, which begins following the location of the last local variable allocated in the local primary area

The secondary areas have no explicit size, but the total of all primary and secondary areas cannot exceed the lower 32K-word area of the user data segment.

For each standard indirect array or structure that you declare:

1. The compiler provides an implicit standard pointer and allocates a word of storage for the pointer in the global or local primary area.
2. The compiler allocates storage for the array or structure in the global or local secondary area, which begins immediately following the last direct item.

If you declare indirect arrays and structures within BLOCK declarations, however, the compiler allocates such data blocks as described in Section 14, "Compiling Programs."

3. The compiler initializes the implicit pointer (provided in step 1) with the allocated address of the array or structure in the secondary area, as follows:
 - For a STRING array, the pointer contains the byte address of the array.
 - For any other array, the pointer contains the word address of the array.
 - For a structure, the pointer contains the word address of the structure.

Allocation in the Automatic Extended Segment

Extended indirect allocation is limited to the size of an extended segment, which can be as large as 127.5 megabytes.

If you declare any extended indirect array or structure, the compiler automatically allocates and manages an extended data segment. (You can also allocate and manage explicit extended segments as described in Appendix B, "Managing Addressing.")

For each extended indirect array or structure that you declare:

1. The compiler provides an implicit extended pointer and allocates a doubleword of storage for the pointer in the global or local primary area of the user data segment. (Because the local primary area is limited to 127 words, you can declare at most 63 extended local variables in any procedure.)
2. The compiler allocates storage for the array or structure in the automatic extended data segment.
3. The compiler initializes the implicit pointer (provided in step 1) with the byte address of the array or structure in the extended data segment.

The compiler also allocates and manages:

- An extended stack
- Two extended stack pointers

Extended Stack

The compiler allocates the extended stack in the automatic extended data segment in a data block named `$EXTENDED#STACK`. The default size of the extended stack is 64K bytes or the maximum space required by a procedure, whichever is greater.

When you use recursion or compile compilation units separately, the compiler cannot calculate the size requirements of the extended stack precisely. You can increase its size by using the `LARGESTACK` directive, described in the *TAL Reference Manual*.

Extended Stack Pointers

The compiler allocates two implicit extended stack pointers in a data block named `EXTENDED#STACK#POINTERS` in the global primary area of the user data segment. These pointers are as follows:

Extended Stack Pointer	Content
<code>#SX</code>	Contains the address of the first free location in the current activation record in the extended stack. The current activation record contains all the information needed to execute the current procedure.
<code>#MX</code>	Contains the maximum value allowed for <code>#SX</code> , less eight bytes.

When the value of `#SX` is greater than or equal to the value of `#MX`, the extended stack overflows. The end of the compilation listing reports the memory position of the two stack pointers.

5 Using Expressions

This section describes how you use arithmetic and conditional expressions. It gives information about operands (identifiers, data types, variables, constants) and about arithmetic and conditional operators and their effect on operands.

For information about assignment, CASE, IF, and group comparison expressions, see Section 13, “Using Special Expressions.”

About Expressions An **expression** is a sequence of operands and operators that, when evaluated, produces a single value. Operands in an expression include variables, constants, and function identifiers. Operators in an expression perform arithmetic or conditional operations on the operands.

Expressions, for example, can appear in:

- LITERAL declarations
- Variable initializations and assignments
- Array and structure bounds
- Indexes to variables
- Conditional program execution
- Parameters to procedures or subprocedures

Complexity An expression can be:

- A single operand, such as 5
- A unary plus or minus operator applied to a single operand, such as -5
- A binary operator applied to two operands, such as 5 * 8
- A complex sequence, such as:

`((alpha + beta) / chi) * (delta - 145.9)) / zeta`

Functionality The compiler at times requires arithmetic or conditional expressions. Where indicated, specify one of the following kinds of expressions:

Expression	Description	Examples
Arithmetic expression	An expression that computes a single numeric value and that consists of operands and arithmetic operators.	398 + num / 84 10 LOR 12
Constant expression	An arithmetic expression that contains only constants, LITERALS, and DEFINES as operands.	398 + 46 / 84
Conditional expression	An expression that establishes the relationship between values and that results in a true or false value. It consists of relational or Boolean conditions and conditional operators.	Relational: a < c Boolean: a OR b

Operands Operands in expressions can be items such as variables, constants, LITERALS, and function invocations.

The following subsections describe identifiers, data types, variables, constants, LITERALS, and functions, followed by arithmetic expressions and conditional expressions.

Identifiers Identifiers are names you declare for objects such as variables, LITERALS, and procedures (including functions). You can form identifiers that:

- Are up to 31 characters long
- Begin with an alphabetic character, an underscore (`_`), or a circumflex (`^`)
- Contain alphabetic characters, numeric characters, underscores, or circumflexes
- Contain lowercase and uppercase characters (the compiler treats them all as uppercase)
- Are not reserved keywords, which are listed in Table 5-1.
- Can be nonreserved keywords, except as noted in Table 5-2.

To separate words in identifiers, use underscores rather than circumflexes. International character-set standards allow the character printed for the circumflex to vary with each country.

Do not end identifiers with an underscore. The trailing underscore is reserved for identifiers supplied by the operating system.

The following identifiers are correct:

```
a2
myprog
_23456789012_00
name_with_exactly_31_characters
```

The following identifiers are incorrect:

```
2abc                !Begins with number
ab%99              !Illegal symbol
Variable            !Reserved word
This_name_is_too_long_so_it_is_invalid !Too long
```

Though allowed as TAL identifiers, avoid identifiers such as:

```
Name^Using^Circumflexes
Name_Using_Trailing_Underscore_
```

Keywords

Keywords have predefined meanings to the compiler when used as described in this manual. Table 5-1 lists reserved keywords, which you cannot use as identifiers.

Table 5-1. Reserved Keywords

AND	DO	FORWARD	MAIN	RETURN	TO
ASSERT	DOWNT0	GOTO	NOT	RSCAN	UNSIGNED
BEGIN	DROP	IF	OF	SCAN	UNTIL
BY	ELSE	INT	OR	STACK	USE
CALL	END	INTERRUPT	OTHERWISE	STORE	VARIABLE
CALLABLE	ENTRY	LABEL	PRIV	STRING	WHILE
CASE	EXTERNAL	LAND	PROC	STRUCT	XOR
CODE	FIXED	LITERAL	REAL	SUBPROC	
DEFINE	FOR	LOR	RESIDENT	THEN	

Table 5-2 lists nonreserved keywords, which you can use as identifiers anywhere identifiers are allowed, except as noted under Restrictions.

Table 5-2. Nonreserved Keywords

Keyword	Restrictions
AT	
BELOW	
BIT_FILLER	Do not use as an identifier within a structure.
BLOCK	Do not use as an identifier in a source file that contains the NAME declaration.
BYTES	Do not use as an identifier of a LITERAL or DEFINE.
C	
COBOL	
ELEMENTS	Do not use as an identifier of a LITERAL or DEFINE.
EXT	
EXTENSIBLE	
FILLER	Do not use as an identifier within a structure.
FORTTRAN	
LANGUAGE	
NAME	
PASCAL	
PRIVATE	Do not use as an identifier in a source file that contains the NAME declaration.
UNSPECIFIED	
WORDS	Do not use as an identifier of a LITERAL or DEFINE.

Data Types When you declare most kinds of variables, you specify a data type, which dictates:

- The kind of values the variable can store
- The amount of storage the compiler allocates for the variable
- The operations you can perform on the variable
- The byte or word addressing mode of the variable

Table 5-3 gives information about each data type.

Table 5-3. Data Types

Data Type	Storage Unit	Kind of Values the Data Type Can Represent
STRING	Byte	An ASCII character. An 8-bit integer in the range 0 through 255 unsigned.
INT	Word	One or two ASCII characters. A 16-bit integer in the range 0 through 65,535 (unsigned) or -32,768 through 32,767 (signed). A standard (16-bit) address (0 through 65,535).
INT(32)	Doubleword	A 32-bit integer in the range -2,147,483,648 through +2,147,483,647. An extended (32-bit) address (0 through 127.5K).
UNSIGNED	<i>n</i> -bit field *	UNSIGNED(1-15) and UNSIGNED(17-31) can represent a positive unsigned integer in the range 0 through $(2^n - 1)$. UNSIGNED(16) can represent an integer in the range 0 through 65,535 unsigned or -32,768 through 32,767 signed; it can also represent a standard address
FIXED	Quadrupleword	A 64-bit fixed-point number. For FIXED(0) and FIXED (*), the range is -9,223,372,036,854,775,808 through +9,223,372,036,854,775,807.
REAL	Doubleword	A 32-bit floating-point number in the range $\pm 8.6361685550944446E-78$ through $\pm 1.15792089237316189E77$ precise to approximately 7 significant decimal digits.
REAL(64)	Quadrupleword	A 64-bit floating-point number in the same range as data type REAL but precise to approximately 17 significant decimal digits.

* For an UNSIGNED simple variable, the bit field can be 1 to 31 bits wide.
For an UNSIGNED array, the element bit field can be 1, 2, 4, or 8 bits wide.

As shown in Table 5-3, a data type consists of a keyword possibly followed by a value enclosed in parentheses. This value is the width or *point* of the data type.

Specifying Widths

For INT, REAL, and UNSIGNED data types, the value in parentheses is a constant expression that specifies the *width*, in bits, of the variable. As of the D20 release, the constant expression can include LITERALS and DEFINES (previously declared constants and text). The result of the constant expression must be one of the following values:

Data Type Prefix	<i>width</i> , in bits
INT	16*, 32, or 64*
REAL	32* or 64
UNSIGNED—simple variable, parameter, or function result	A value in the range 1 through 31
UNSIGNED—array	1, 2, 4, or 8

* INT(16), INT(64), and REAL(32) are data type aliases, as described in "Data Type Aliases" later in this section.

Here is an example of a *width* that includes a LITERAL:

```
LITERAL dbwd_size = (4 * 8);      !INT_SIZE equals 32
INT(dbwd_size) num;              !Data type is INT(32)
```

LITERALS are described later in this section. DEFINES are described in the *TAL Reference Manual*.

Specifying *fpoint*s

For the FIXED data type, *fpoint* is the implied fixed-point setting. *fpoint* is an integer in the range -19 through 19. If you omit *fpoint*, the default *fpoint* is 0 (no decimal places).

A positive *fpoint* specifies the number of decimal places to the right of the decimal point:

```
FIXED(3) x := 0.642F;           !Stored as 642
```

A negative *fpoint* specifies a number of integer places to the left of the decimal point. To store a FIXED value, a negative *fpoint* truncates the value leftward from the decimal point by the specified number of digits. When you access the FIXED value, zeros replace the truncated digits:

```
FIXED(-3) y := 642945F;         !Stored as 642; accessed
                                ! as 642000
```

Specifying Asterisks

As of the D20 release, FIXED(*) is a data type notation. If you declare a FIXED(*) variable, the value stored in the variable is not scaled.

Data Type Aliases The compiler accepts the following aliases for the listed data types:

Data Type	Alias
INT	INT(16)
REAL	REAL(32)
FIXED(0)	INT(64)

For consistency, the remainder of this manual avoids using data type aliases. For example, although the following declarations are equivalent, the manual uses `FIXED(0)`:

```
FIXED(0) var;
INT(64) var;
```

Storage Units Storage units are the containers in which you can access data stored in memory. The system fetches and stores all data in 16-bit words, but you can access data as any of the storage units listed in Table 5-4.

Table 5-4. Storage Units

Storage Unit	Number of Bits	Data Type	Description
Byte	8	STRING	One of two bytes that make up a word
Word*	16	INT	Two bytes, with byte 0 (most significant) on the left and byte 1 (least significant) on the right
Doubleword	32	INT(32), REAL	Two contiguous words
Quadword	64	REAL(64), FIXED	Four contiguous words
Bit field	1–16	UNSIGNED	Contiguous bit fields within a word
Bit field	17–31	UNSIGNED	Contiguous bit fields within a doubleword

* In TAL a word is always 16 bits regardless of the word size used by the system hardware.

Data Types of Expressions The result of an expression can be of any data type except `STRING` or `UNSIGNED`. The compiler determines the data type of the result from the data type of the operands in the expression. All operands in an expression must have the same data type, with the following exceptions:

- An `INT` expression can include `STRING`, `INT`, and `UNSIGNED(1–16)` operands. The system treats `STRING` and `UNSIGNED(1–16)` operands as if they were 16-bit values. That is, the system:
 - Puts a `STRING` operand in the right byte of a word and sets the left byte to 0.
 - Puts an `UNSIGNED(1–16)` operand in the right bits of a word and sets the unused left bits to 0, with no sign extension. For example, for an `UNSIGNED(2)` operand, the system fills the 14 leftmost bits of the word with zeros.

- An INT(32) expression can include INT(32) and UNSIGNED(17–31) operands. The system treats UNSIGNED(17–31) operands as if they were 32-bit values. It places an UNSIGNED(17–31) operand in the right bits of a doubleword and sets the unused left bits to 0, with no sign extension. For example, for an UNSIGNED(29) operand, the system fills the three leftmost bits of the doubleword with zeros.

In all other cases, if the data types do not match, use type transfer functions (described in the *TAL Reference Manual*) to make them match.

Variables A variable is a symbolic representation of an item or a group of elements. You use variables to store data that can change during program execution. Table 5-5 lists the kinds of variables you can declare.

Table 5-5. Variables

Variable	Description
Simple variable	A variable that contains one element of a specified data type
Array	A variable that contains multiple elements of the same data type
Structure	A variable that can contain variables of different data types
Substructure	A structure nested within a structure or substructure
Structure data item	A simple variable, array, simple pointer, substructure, or structure pointer declared in a structure or substructure; also known as a structure field
Simple pointer	A variable that contains the memory address, usually of a simple variable or array element, which you can access with this simple pointer
Structure pointer	A variable that contains the memory address of a structure, which you can access with this structure pointer

Constants A constant is a value you can store in a variable, declare as a LITERAL, or use as part of an expression. Constants can be numbers or character strings. The kind and size of constants a variable can accommodate depend on the data type of the variable.

Examples are:

```

255           !Integer number
1.02E12      !Floating-point number
2.5F         !Fixed-point number
"xyz"        !Character string
2 * 5        !Constant expression

```

The following subsections describe numeric constants (integer, fixed-point, and floating-point) and character string constants.

Integer Constants

Integer constants include STRING, INT, INT(32), and FIXED(0) numbers.

Integer constants can be in binary, octal, decimal, or hexadecimal base. Decimal is the default number base. Specify the base as shown in Table 5-6.

Table 5-6. Number Base Formats

Number Base	Prefix	Digits Allowed	Example
Decimal	None	0 through 9	46
Octal	%	0 through 7	%57
Binary	%B	0 or 1	%B101111
Hexadecimal	%H	0 through 9, A through F	%H2F

STRING. A STRING numeric constant is an unsigned 8-bit integer in the range 0 through 255. Examples are:

```
59                !Decimal base
%12              !Octal base
%B101            !Binary base
%h2A            !Hexadecimal base
```

INT. An INT numeric constant is a signed or unsigned 16-bit integer in the range 0 through 65,535 (unsigned) or -32,768 through 32,767 (signed). Examples are:

```
45550            !Decimal base (unsigned)
-8987           !Decimal base (signed)
%177            !Octal base (unsigned)
-%5             !Octal base (signed)
%B1001111000010001 !Binary base
%h2f           !Hexadecimal
```

INT(32). An INT(32) numeric constant is a signed or unsigned 32-bit integer in the range -2,147,483,648 through 2,147,483,647, suffixed by **D** for decimal, octal, or binary integers, and by **%D** for hexadecimal integers. Examples are:

```
0D              !Decimal base
+14769D
-327895066d
%1707254361d   !Octal base
-%24700000221D
%B000100101100010001010001001d !Binary base
%h096228d%d    !Hexadecimal base
-%H99FF29%D
-%H99FF29 D    !This form is allowed but not
                ! recommended; always include
                ! the % in the %D suffix
```

FIXED(0). FIXED(0) numeric constants are discussed under “Fixed-Point Constants,” described next.

Fixed-Point Constants

A fixed-point constant is type **FIXED** and is a signed 64-bit fixed-point number. The range of a **FIXED** constant is determined by its *fpoint*. For example, the ranges for **FIXED(0)** and **FIXED(2)** are:

<i>fpoint</i>	Range
FIXED(0)	-9,223,372,036,854,775,808 through 9,223,372,036,854,775,807
FIXED(2)	-92,233,720,368,547,758.08 through 92,233,720,368,547,758.07

Fixed-point numbers except **FIXED(0)** must be in decimal base. A decimal fixed-point number can include a fractional part preceded by a decimal point. Append **F** to decimal, octal, or binary **FIXED** numbers. Append **%F** to hexadecimal numbers. Here are examples:

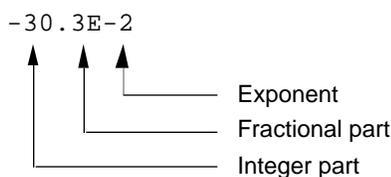
1200.09F	!Decimal base; FIXED(2)
0.1234567F	!Decimal base; FIXED(7)
239840984939873494F	!Decimal base; FIXED(0)
-10.09F	!Decimal base; signed FIXED(2)
%B1010111010101101010110F	!Binary base; FIXED(0)
-%765235512F	!Octal base; signed FIXED(0)
%H298756%F	!Hexadecimal base; FIXED(0)

Floating-Point Constants

A **REAL** or **REAL(64)** numeric constant is a signed floating-point number in the range $\pm 8.6361685550944446 * 10^{-78}$ through $\pm 1.15792089237316189 * 10^{+77}$.

- A **REAL** numeric constant is a 32-bit value that is precise to approximately seven significant digits.
- A **REAL(64)** numeric constant is a 64-bit value that is precise to approximately 17 significant digits.

The format of a floating-point constant includes an integer part, a fractional part suffixed by **E** for a **REAL** constant or **L** for a **REAL(64)** constant, and an exponent as follows:



Here are examples:

Decimal Value	REAL	REAL(64)
0	0.0E0	0.0L0
2	2.0e0	2.0L0
2	0.2E1	0.2L1
2	20.0E-1	20.0L-1
-17.2	-17.2E0	-17.2L0
-17.2	-1720.0E-2	-1720.0L-2

Character String Constants

A character string constant consists of one or more contiguous ASCII characters enclosed in quotation mark delimiters, as in:

```
"Now is the time for good parties."
```

If a quotation mark is a character within the string, use two quotation marks (in addition to the quotation mark delimiters), as in:

```
"The title is ""East of Eden""."
```

The compiler does not upshift lowercase characters.

When specifying character string constants, you can use any character in the ASCII character set (shown in Appendix D), including:

- Upper and lowercase alphabetic characters
- Numerics 0 through 9
- Special characters

Each character in a character string requires one byte of contiguous storage. When you initialize variables with character strings, follow these guidelines:

- You can initialize simple variables or arrays of any data type with character strings.
- When you initialize a simple variable, specify a character string that contains the same number of bytes as the simple variable or fewer.
- When you initialize an array, specify a character string that contains up to 127 characters and that fits on one line. If a character string is too long for one line, use a constant list and break the character string into smaller character strings. (Constant lists are described in Section 7, "Using Arrays.")

When you assign character strings to variables, follow these guidelines:

- You can assign character strings to STRING, INT, and INT(32) variables, but not to FIXED, REAL, or REAL(64) variables.
- In an assignment statement, specify a character string that contains up to four characters, depending on the data type of the variable.

LITERALS A LITERAL declaration specifies one or more identifiers and associates each with a constant expression. Each identifier in a LITERAL declaration is known as a LITERAL. You can define a LITERAL once and then reference it by identifier many times in the program. When you need to change a LITERAL, you only change the declaration, not every reference to it.

LITERALS also make the source code more meaningful. For example, identifiers such as BUFFER_LENGTH and TABLE_SIZE are more meaningful than their respective constant values of 80 and 128.

When you declare LITERALS, you can specify a constant expression for each identifier or you can let the compiler supply some or all of the constants.

Declaring LITERALS With Constants

To include constants in a LITERAL declaration, specify the keyword LITERAL and one or more *identifiers*, each followed by an equal sign (=) and a *constant expression*. Separate consecutive *identifier* and *constant* combinations with commas.

The constant expressions in a LITERAL declaration:

- Can be numeric constants of any data type except STRING or UNSIGNED
- Can be character strings that each contain at most four characters long
- Must not be the address of a global variable

Here are examples of LITERAL declarations that include constant expressions:

```
LITERAL buffer_length = 80;

LITERAL true = -1,
         false = 0,
         chars = "AB";
```

Declaring LITERALS Without Constants

You can omit constants for one or more identifiers in a LITERAL declaration. The compiler computes the omitted constants, using unsigned arithmetic:

- If you omit the first constant in the declaration, the compiler supplies a zero.
- If you omit a constant that follows an INT constant, the compiler supplies an INT constant that is one greater than the preceding constant. If you omit a constant that follows a constant of any data type except INT, an error message results.

This example shows how the compiler supplies constants in a LITERAL declaration:

```
LITERAL a,           -- The compiler supplies 0
         b,           -- The compiler supplies 1
         c,           -- The compiler supplies 2
         d = 0,       -- You specify 0
         e,           -- The compiler supplies 1
         f = 17,      -- You specify 17
         g,           -- The compiler supplies 18
         h;          -- The compiler supplies 19
```

Using LITERALS

You can use LITERALS in declarations and statements:

```

LITERAL array_length = 50;           !Length of array
INT .buffer[0:array_length - 1];    !Declare array

```

You can also use LITERAL identifiers in subsequent LITERAL declarations:

```

LITERAL number_of_file_extents = 16,
file_extent_size_in_pages = 32,
file_size_in_bytes = (number_of_file_extents '*'
file_extent_size_in_pages) * 2048D !bytes per page!;

```

Standard Functions A function is a procedure or subprocedure that returns a value to the calling procedure or subprocedure. Table 5-7 summarizes the kinds of operations that standard (built-in) functions perform.

Table 5-7. Summary of Standard Functions

Category	Operation
Type transfer	Converts an expression from one data type to another
Address conversion	Converts standard addresses to extended addresses or extended addresses to standard addresses
Character test	Tests for an alphabetic, numeric, or special ASCII character; returns a true value if the character passes the test or a false value if the character fails the test
Minimum-maximum	Returns the minimum or maximum of two expressions
Carry and overflow	Tests the state of the carry or overflow indicator in the environment register; returns a true value if the indicator is on or a false value if it is off
FIXED expression	Returns the <i>fpoint</i> , or moves the position of the implied decimal point, of a FIXED expression
Variable	Returns the unit length, offset, data type, or number of occurrences of a variable
Miscellaneous	Tests for receipt of actual parameter; returns the absolute value or one's complement from expressions; returns the setting of the system clock or internal register pointer

For example, the \$DBL standard function converts an expression from any data type to a signed INT(32) expression. In the following example, \$DBL converts an INT expression to a signed INT(32) expression:

```

INT a;           !Declare an INT variable
INT(32) b;       !Declare an INT(32) variable
!Some code here
b := $DBL (a);   !Convert A to INT(32) expression
                 ! and store it in B

```

Other examples are shown in various sections of this manual. Each standard function is described in the *TAL Reference Manual*.

Precedence of Operators Operators in expressions can be arithmetic (signed, unsigned, or logical) or conditional (Boolean or relational, signed or unsigned).

Within an expression, the compiler evaluates operators in order of precedence. Within each level of precedence, the compiler evaluates operators from left to right. Table 5-8 shows the level of precedence for each operator, from highest (0) to lowest (9).

Table 5-8. Precedence of Operators (Page 1 of 2)

Operator	Operation	Precedence
[<i>n</i>]	Indexing	0
.	Dereferencing *	0
@	Address of identifier	0
+	Unary plus	0
-	Unary minus	0
.< . . . >	Bit extraction	1
<<	Signed left bit shift	2
>>	Signed right bit shift	2
' << '	Unsigned left bit shift	2
' >> '	Unsigned right bit shift	2
*	Signed multiplication	3
/	Signed division	3
' * '	Unsigned multiplication	3
' / '	Unsigned division	3
' \ '	Unsigned modulo division	3
+	Signed addition	4
-	Signed subtraction	4
' + '	Unsigned addition	4
' - '	Unsigned subtraction	4
LOR	Bitwise logical OR	4
LAND	Bitwise logical AND	4
XOR	Bitwise exclusive OR	4
<	Signed less than	5
=	Signed equal to	5
>	Signed greater than	5
<=	Signed less than or equal to	5
>=	Signed greater than or equal to	5
<>	Signed not equal to	5

* Not portable to future software platforms.

Table 5-8. Precedence of Operators (Page 2 of 2)

Operator	Operation	Precedence
' < '	Unsigned less than	5
' = '	Unsigned equal to	5
' > '	Unsigned greater than	5
' <= '	Unsigned less than or equal to	5
' >= '	Unsigned greater than or equal to	5
' <> '	Unsigned not equal to	5
NOT	Boolean negation	6
AND	Boolean conjunction	7
OR	Boolean disjunction	8
:=	Assignment	9
. < . . . > :=	Bit deposit **	9

** Described in the *TAL Reference Manual*.

You can use parentheses to override the precedence of operators. You can nest the parenthesized operations. The compiler evaluates nested parenthesized operations outward starting with the innermost level. Here are examples:

$c * (a + b)$

Result

$c * ((a + b) / d)$

Result

$(a \text{ OR } b) \text{ AND } c$

Result

Arithmetic Expressions An arithmetic expression is a sequence of operands and arithmetic operators that computes a single numeric value of a specific data type. Following are examples of arithmetic expressions:

```
var1                !operand
var1 / var2         !operand arithmetic-operator operand
var1 * (-var2)     !operand arithmetic-operator operand
```

You can append a unary plus operator or a unary minus operator to the leftmost operand in the expression:

```
+var1 * 2          !unary plus
-var1 / var2       !unary minus
```

Arithmetic operators can be signed, unsigned, or logical, as described in this section.

Operands in Arithmetic Expressions An operand consists of one or more elements that evaluate to a single value. Table 5-9 describes the operands that can make up an arithmetic expression.

Table 5-9. Operands in Arithmetic Expressions

Element	Description	Example
Variable	The identifier of a simple variable, array element, pointer, structure data item, or equivalenced variable, with or without @ or an index	var[10]
Constant	A character string or numeric constant	103375
LITERAL	The identifier of a named constant	file_size
Function invocation (expression)	The invocation of a procedure that returns a value Any expression, enclosed in parentheses	\$LEN (x) (x := y)
Code space item	The identifier of a procedure, subprocedure, or label prefixed with @ or a read-only array optionally prefixed with @, with or without an index	@label_a

Signed Arithmetic Operators Signed arithmetic operators and the operand types on which they can operate are shown in Table 5-10.

Table 5-10. Signed Arithmetic Operators

Operator	Operation	Operand Type*	Example
+	Unary plus	Any data type	+5
-	Unary minus	Any data type	-5
+	Binary signed addition	Any data type	alpha + beta
-	Binary signed subtraction	Any data type	alpha - beta
*	Binary signed multiplication	Any data type	alpha * beta
/	Binary signed division	Any data type	alpha / beta

* The data type of the operands must match except as noted in "Data Types of Expressions" earlier in this section.

Table 5-11 shows the combinations of operand types you can use with a binary signed arithmetic operator and the result type yielded by such operators. In each combination, the order of the data types is interchangeable.

Table 5-11. Signed Arithmetic Operand and Result Types

Operand Type	Operand Type	Result Type	Example
STRING	STRING	INT	byte1 + byte2
INT	INT	INT	word1 - word2
INT(32)	INT(32)	INT(32)	dbl1 * dbl2
REAL	REAL	REAL	real1 + real2
REAL(64)	REAL(64)	REAL(64)	quad1 + quad2
FIXED	FIXED	FIXED	fixed1 * fixed2
INT	STRING	INT	word1 / byte1
INT	UNSIGNED(1-16)	INT	word + unsign12
INT(32)	UNSIGNED(17-31)	INT(32)	double + unsign20
UNSIGNED(1-16)	UNSIGNED(1-16)	INT	unsign6 + unsign9
UNSIGNED(17-31)	UNSIGNED(17-31)	INT(32)	unsign26 + unsign31

The compiler treats a STRING or UNSIGNED(1-16) operand as an INT operand. If bit <0> contains a 0, the operand is positive; if bit <0> contains a 1, the operand is negative.

The compiler treats an UNSIGNED(17-31) operand as a positive INT(32) operand.

Scaling of FIXED Operands

When you declare a FIXED variable, you can specify an implied fixed-point setting (*fpoint*)—an integer in the range -19 through 19, enclosed in parentheses following the keyword FIXED. If you do not specify an *fpoint*, the default *fpoint* is 0 (no decimal places).

A positive *fpoint* specifies the number of decimal places to the right of the decimal point:

```
FIXED(3)  x := 0.642F;           !Stored as 642
```

A negative *fpoint* specifies a number of integer places to the left of the decimal point. To store a FIXED value, a negative *fpoint* truncates the value leftward from the decimal point by the specified number of digits. When you access the FIXED value, zeros replace the truncated digits:

```
FIXED(-3) y := 642945F;         !Stored as 642; accessed
                                   ! as 642000
```

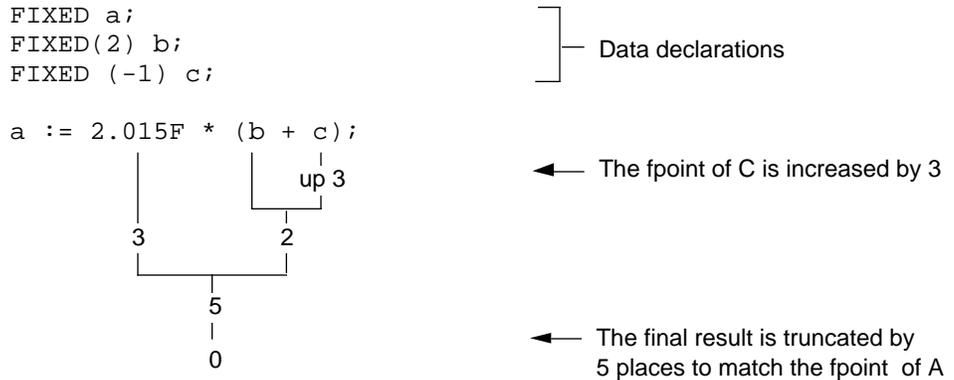
When FIXED operands in an arithmetic expression have different *fpoints*, the system makes adjustments depending on the operator.

- In addition or subtraction, the system adjusts the smaller *fpoint* to match the larger *fpoint*. The result inherits the larger *fpoint*. For example, the system adjusts the smaller *fpoint* in $3.005F + 6.01F$ to $6.010F$, and the result is $9.015F$.
- In multiplication, the *fpoint* of the result is the sum of the *fpoints* of the two operands. For example, $3.091F * 2.56F$ results in the FIXED(5) value $7.91296F$.
- In division, the *fpoint* of the result is the *fpoint* of the dividend minus the *fpoint* of the divisor. (Some precision is lost.) For example, $4.05F / 2.10F$ results in the FIXED(0) value 1.

To retain precision when you divide operands that have nonzero *fpoints*, use the \$SCALE standard function to scale up the *fpoint* of the dividend by a factor equal to the *fpoint* of the divisor; for example:

```
FIXED(3) result, a, b;         !fpoint of 3
result := $SCALE(a,3) / b;     !Scale A to FIXED(6); result
                                   ! is a FIXED(3) value
```

The following example shows how the system makes automatic adjustments when operands in an expression have different *fpoints*:



415

Effect on Hardware Indicators

Signed arithmetic operators affect the hardware indicators, as described in “Testing Hardware Indicators” later in this section.

Unsigned Arithmetic Operators

You can use binary unsigned arithmetic on operands with values in the range 0 through 65,535. For example, you can use unsigned arithmetic with pointers that contain standard addresses. Table 5-12 lists unsigned arithmetic operators and the operand types on which they can operate.

Table 5-12. Unsigned Arithmetic Operators

Operator	Operation	Operand Type	Example
' + '	Unsigned addition	STRING, INT, or UNSIGNED(1-16)	alpha '+' beta
' - '	Unsigned subtraction	STRING, INT, or UNSIGNED(1-16)	alpha '-' beta
' * '	Unsigned multiplication	STRING, INT, or UNSIGNED(1-16)	alpha '*' beta
' / '	Unsigned division	INT(32) or UNSIGNED (17-31) dividend and STRING, INT, or UNSIGNED(1-16) divisor	alpha '/' beta
' \ '	Unsigned modulo division *	INT(32) or UNSIGNED (17-31) dividend and STRING, INT, or UNSIGNED(1-16) divisor	alpha '\' beta

* Unsigned modulo operations return the remainder. If the quotient exceeds 16 bits, an overflow condition occurs and the results will have unpredictable values. For example, the modulo operation 200000D \ 2 causes an overflow because the quotient exceeds 16 bits.

Table 5-13 shows the combinations of operand types you can use with binary unsigned arithmetic operators and the result types yielded by such operators. The order of the operand types in each combination is interchangeable except in the last case.

Table 5-13. Unsigned Arithmetic Operand and Result Types

Operator	Operand Type	Operand Type	Result Type	Example
'+' '-'	STRING	STRING	INT	byte1 '-' byte2
	INT	INT	INT	word1 '+' word2
	INT	STRING	INT	byte1 '-' word1
	INT	UNSIGNED (1-16)	INT	word1 '+' uns8
	STRING	UNSIGNED (1-16)	INT	byte1 '-' uns5
	UNSIGNED(1-16)	UNSIGNED(1-16)	INT	uns1 '+' uns7
'*'	STRING	STRING	INT(32)	byte1 '*' byte2
	INT	INT	INT(32)	word1 '*' word2
	STRING	INT	INT(32)	byte1 '*' word1
	INT	UNSIGNED (1-16)	INT(32)	word1 '*' uns9
	STRING	UNSIGNED (1-16)	INT(32)	uns1 '*' uns7
	UNSIGNED(1-16)	UNSIGNED(1-16)	INT(32)	uns1 '*' uns7
'/' '\'	UNSIGNED(17-31) or INT(32) dividend	STRING, INT, or UNSIGNED(1-16) divisor	INT	dbword '\ word1

Effect on Hardware Indicators

Unsigned add and subtract operators affect the carry and condition code indicator, as described in “Testing Hardware Indicators” later in this section.

Bitwise Logical Operators You use logical operators—LOR, LAND, and XOR—to perform bit-by-bit operations on STRING, INT, and UNSIGNED(1–16) operands only. Logical operators always return 16-bit results. Table 5-14 gives information about these operators.

Table 5-14. Logical Operators and Result Yielded

Operator	Operation	Operand Type	Bit Operations	Example
LOR	Bitwise logical OR	STRING, INT, or UNSIGNED(1–16)	1 LOR 1 = 1 1 LOR 0 = 1 0 LOR 0 = 0	$10 \text{ LOR } 12 = 14$ <pre> 10 1 0 1 0 12 1 1 0 0 ----- 14 1 1 1 0 </pre>
LAND	Bitwise logical AND	STRING, INT, or UNSIGNED(1–16)	1 LAND 1 = 1 1 LAND 0 = 0 0 LAND 0 = 0	$10 \text{ LAND } 12 = 8$ <pre> 10 1 0 1 0 12 1 1 0 0 ----- 8 1 0 0 0 </pre>
XOR	Bitwise exclusive OR	STRING, INT, or UNSIGNED(1–16)	1 XOR 1 = 0 1 XOR 0 = 1 0 XOR 0 = 0	$10 \text{ XOR } 12 = 6$ <pre> 10 1 0 1 0 12 1 1 0 0 ----- 6 0 1 1 0 </pre>

The Bit Operations column in the table shows the bit-by-bit operations that occur on 16-bit values. Each 1-bit operand pair results in a 1-bit result. The bit operands are commutative.

Effect on Hardware Indicators

Logical operators set the condition code indicator, as described in “Testing Hardware Indicators” later in this section. Logical operators are always unsigned, however, so condition codes are not meaningful.

Conditional Expressions A conditional expression is a sequence of conditions and Boolean or relational operators that establishes the relationship between values. You can use conditional expressions to direct program flow.

Following are examples of conditional expressions:

```

a                               !condition
NOT a                           !NOT condition
a OR b                          !condition OR condition
a AND b                         !condition AND condition
a AND NOT b OR c               !condition AND NOT condition ...

```

Conditions A condition is an operand in a conditional expression that represents a true or false state. A condition can consist of one or more of the elements listed in Table 5-15.

Table 5-15. Conditions in Conditional Expressions

Element	Description	Example
Relational expression	Two conditions connected by a relational operator. The result type is INT; a -1 if true or a 0 if false. The example is true if A equals B.	IF a = b THEN ...
Group comparison expression	Unsigned comparison of a group of contiguous elements with another. The result type is INT; a -1 if true or a 0 if false. The example compares 20 words of two INT arrays.	IF a = b FOR 20 WORDS THEN ...
(conditional expression)	A conditional expression enclosed in parentheses. The result type is INT; a -1 if true or a 0 if false. The example is true if both B and C are false. The system evaluates the parenthesized condition first, then applies the NOT operator.	IF NOT (b OR c) THEN ...
Arithmetic expression	An arithmetic, assignment, CASE, or IF expression that has an INT result*. The expression is treated as true if its value is not 0 and false if its value is 0. The example is true if the value of X is not 0.	IF x THEN ...
Relational operator	A signed or unsigned relational operator that tests a condition code. Condition code settings are CCL (negative), CCE (0), or CCG (positive). The example is true if the condition code setting is CCL.	IF < THEN ...

* If an arithmetic expression has a result other than INT, use a signed relational expression.

Boolean Operators You use Boolean operators—NOT, OR, and AND—to set the state of a single value or the relationship between two values. Table 5-16 describes the Boolean operators, the operand types you can use with them, and the results that such operators yield.

Table 5-16. Boolean Operators and Result Yielded

Operator	Operation	Operand Type	Result	Example
NOT	Boolean negation; tests condition for false state	STRING, INT, or UNSIGNED(1-16)	True/False	NOT a
OR	Boolean disjunction; produces true state if either adjacent condition is true	STRING, INT, or UNSIGNED(1-16)	True/False	a OR b
AND	Boolean conjunction; produces true state if both adjacent conditions are true	STRING, INT, or UNSIGNED(1-16)	True/False	a AND b

Evaluation of Boolean Operations

Conditions connected by the OR operator are evaluated from left to right only until a true condition occurs.

Conditions connected by the AND operator are evaluated from left to right until a false condition occurs. The next condition is evaluated only if the preceding condition is true. In the following example, function F will not be called because `A <> 0` is false:

```
a := 0;
IF a <> 0 AND f(x) THEN ... ;
```

Effect on Hardware Indicators

Boolean operators set the condition code indicator, as described in “Testing Hardware Indicators” later in this section.

Relational Operators Relational operators can be signed or unsigned.

Signed Relational Operators

Signed relational operators perform signed comparison of two operands and return a true or false state. Table 5-17 describes signed relational operators, operand data types, and the results yielded by such operators.

Table 5-17. Signed Relational Operators and Result Yielded

Operator	Operation	Operand Type*	Result
<	Signed less than	Any data type	True/False
=	Signed equal to	Any data type	True/False
>	Signed greater than	Any data type	True/False
<=	Signed less than or equal to	Any data type	True/False
>=	Signed greater than or equal to	Any data type	True/False
<>	Signed not equal to	Any data type	True/False

* The data type of the operands must match except as noted in "Data Types of Expressions" earlier in this section.

Unsigned Relational Operators

Unsigned relational operators perform unsigned comparison of two operands and return a true or false state. Table 5-18 describes unsigned relational operators, operand data types, and the results yielded by such operators.

Table 5-18. Unsigned Relational Operators and Result Yielded

Operator	Operation	Operand Type	Result
' < '	Unsigned less than	STRING, INT, UNSIGNED (1-16)	True/False
' = '	Unsigned equal to	STRING, INT, UNSIGNED (1-16)	True/False
' > '	Unsigned greater than	STRING, INT, UNSIGNED (1-16)	True/False
' <= '	Unsigned less than or equal to	STRING, INT, UNSIGNED (1-16)	True/False
' >= '	Unsigned greater than or equal to	STRING, INT, UNSIGNED (1-16)	True/False
' <> '	Unsigned not equal to	STRING, INT, UNSIGNED (1-16)	True/False

Effect on Hardware Indicators

Relational operators set the condition code indicator, as described in "Testing Hardware Indicators" later in this section.

Controlling Program Execution

You use relational expressions in control statements to determine the flow of execution. This example shows how you can direct program execution based on comparisons using signed and unsigned operators:

```

INT a := -2,           !Unsigned value = %177776
   c :=  3,           !Unsigned value = %000003
   x := 271;

IF a '<' c THEN x := 314;  !False; X still contains 271
IF a < c THEN x := 313;   !True; X is assigned 313
IF a <> c THEN            !True; this is an arithmetic
   IF < THEN x := 314;    ! comparison; since -2 < 3,
                           ! CCL is set; x is assigned
                           ! 314

IF a '<>' c THEN          !True; this is a logical
   IF > THEN x := 315;   ! comparison; since
                           ! %177776 '>' %3, CCG is set;
                           ! X is assigned 315

```

Assigning Conditional Expressions

You can assign the value of a conditional expression to a variable. The value assigned is a -1 for the true state or a 0 for the false state.

For example, you can assign the result of a comparison to a variable:

```

INT neg := -1;         !Value = %177777
INT pos :=  1;         !Value = %000001
INT result;

result := neg < pos;   !Signed comparison produces -1
result := neg '<' pos;  !Unsigned comparison produces 0

```

You can assign a -1 if either X or Y is a nonzero value (true), or a 0 if both X and Y are zeros (false):

```

INT x, y, answer;
answer := x OR y;      !Assign -1 or 0 to ANSWER

```

Testing Hardware Indicators

Hardware indicators include condition code, carry, and overflow settings. Arithmetic and conditional operations, assignments, and some file-system calls affect the setting of the hardware indicators. To check the setting of a hardware indicator, use an IF statement immediately after the operation that affects the hardware indicator.

Condition Code Indicator

The condition code indicator is set by a zero or a negative or positive result:

Result	State of Condition Code Indicator
Negative	CCL
0	CCE
Positive	CCG

To check the state of the condition code indicator, use a relational operator (with no operands) in a conditional expression. Using a relational operator with no operands is equivalent to using the relational operator in a signed comparison against zero. When used with no operands, signed and unsigned operators are equivalent. The result returned by such a relational operator is as follows:

Relational Operator	Result Returned
< or '<'	True if CCL
> or '>'	True if CCG
= or '='	True if CCE
<> or '<>'	True if not CCE
<= or '<='	True if CCL or CCE
>= or '>='	True if CCE or CCG

An example is:

```
IF < THEN ... ;
```

File-System Errors

File-system procedures signal their success or failure by returning an error number or a condition code. Your program can preserve the returned condition code for later operation as follows:

```
CALL WRITE( ... );
IF >= THEN
    system_message := -1;           !True
ELSE
    system_message := 0;           !False
IF system_message = -1 THEN ... ;
```

Carry Indicator The carry indicator is bit 9 in the environment register (ENV.K). The carry indicator is affected as follows:

Operation	Carry Indicator
Integer addition	On if carry out of bit <0>
Integer subtraction or negation	On if no borrow out from bit <0>
INT(32) multiplication and division	Always off
Multiplication and division except INT(32)	Preserved
SCAN or RSCAN operation	On if scan stops on a 0 (zero) byte
Array indexing and extended structure addressing	Undefined
Shift operations	Preserved

To check the state of the carry indicator, use `$CARRY` in an IF statement immediately after the operation that affects the carry bit. If the carry indicator is on, `$CARRY` is -1 (true). If the carry indicator is off, `$CARRY` is 0 (false). The following example tests the state of the carry indicator after addition:

```

INT i, j, k;                !Declare variable
i := j + k;
IF $CARRY THEN ... ;      !Test state of carry bit from +

```

The following operations are not portable to future software platforms:

- Testing `$CARRY` after multiplication or division
- Passing the carry bit as an implicit parameter into a procedure or subprocedure
- Returning the carry bit as an implicit result from a procedure or subprocedure

Overflow Indicator The overflow indicator is bit 8 in the environment register (ENV.V). The overflow indicator is affected as follows:

Operation	Overflow Indicator
Unsigned INT addition, subtraction, and negation	Preserved
Addition, subtraction, and negation except unsigned INT	On or off
Division and multiplication	On or off
Type conversions	On, off, or preserved
Array indexing and extended structure addressing	Undefined
Assignment or shift operation	Preserved

For example, the following operations turn on the overflow indicator (and interrupt the system overflow trap handler if the overflow trap is armed through ENV.T):

- Division by 0
- Floating-point arithmetic result in which the exponent is too large or too small
- Signed arithmetic result that exceeds the number of bits allowed by the data type of the expression

For overflowed integer addition, subtraction, or negation, the result is truncated. For overflowed multiplication, division, or floating-point operation, the result is undefined.

A program can deal with arithmetic overflows in one of four ways:

Desired Effect	Method
Abort on all overflows	Use the system's default trap handler.
Recover globally from overflows	Use a user-supplied trap handler.
Recover locally from statement overflows	Turn off overflow trapping and use \$OVERFLOW.
Ignore all overflows	Turn off overflow trapping throughout the program.

For information on turning off overflow trapping and using \$OVERFLOW, see the description of \$OVERFLOW in the *TAL Reference Manual*.

The following operations are not portable to future software platforms:

- Passing the overflow bit as an implicit parameter into a procedure or subprocedure
- Returning the overflow bit as an implicit result from a procedure or subprocedure

Accessing Operands The remainder of this section discusses different ways of accessing operands:

- Getting the address of a variable
- Dereferencing a simple variable
- Extracting a bit field
- Shifting a bit field

For information on bit deposits, see the *TAL Reference Manual*.

Getting the Address of Variables To get the address of a variable, prefix the variable identifier with @. For example, you can assign the address of an array element to a simple variable as follows:

```
INT .array[0:2];      !Declare array
INT var;             !Declare simple variable
var := @array[2];    !Assign address of ARRAY[2] to VAR
```

Dereferencing Simple Variables You can dereference an INT simple variable in a statement by prefixing the variable identifier with the dereferencing operator (.). The content of the INT simple variable then becomes the standard word address of another data item. You can use the dereferencing operator in any INT arithmetic expression.

The dereferencing operator is not portable to future software platforms and is described here only to explain its use in existing programs.

The following example uses the dereferencing operator to store data at the standard word address that is saved in A:

```
INT a := 5;          !Declare A; initialize it with 5
.a := 0;            !The 5 contained in A becomes an address;
                    ! thus 0 is stored at address G[5]
```

Using a variable in two different ways can make your program more difficult to understand and maintain. The following example:

- Changes the data located at the address stored in VAR by using the dereferencing operator
- Changes the address stored in VAR by omitting the dereferencing operator

```

INT i;                !G[0]; declare I
INT a := 5;          !G[1]; declare A and initialize it with 5
INT var;            !G[2]; declare VAR

var := @a;          !Assign 1 (the address of A) to VAR

i := .var;          !Assign 5 (the content of A to which
                   ! VAR points) to I

i := var;           !Assign 1 (the content of VAR or the
                   ! address of A) to I

```

Extracting Bit Fields You can access a bit extraction field in an INT expression without altering the expression. (Do not use a bit extraction field to compress data. Instead, declare an UNSIGNED variable, specifying the appropriate number of bits in the bit field.)

To access a bit extraction field, specify an INT expression followed by a period (.) and a bit-extraction field enclosed in angle brackets. For example, to access bit <5> of an INT variable named VAR, specify the following construct with no intervening spaces:

```
var.<5>
```

To access bits <2> through <11> of VAR, specify the following construct with no intervening spaces:

```
var.<2:11>
```

Specify the leftmost and rightmost bits of the field as INT constants. The constant specifying the rightmost bit must be equal to or greater than the constant specifying the leftmost bit.

The INT expression in a bit extraction operation can consist of STRING, INT, or UNSIGNED(1-16) operands. The system stores a STRING value in the right byte of a word and treats it as a 16-bit value, so you can access only bits <8> through <15> of the STRING value. For example, to access bits <11> and <12> of a STRING simple variable named BYTE_VAR, specify:

```
byte_var.<11:12>
```

You can assign the bits extracted from an array element as follows:

```

LITERAL len = 8;
STRING right_byte;
INT array[0:len - 1];

right_byte := array[5].<8:15>;

```

You can use bit extraction in conditional expressions:

```
INT word;
STRING var;

IF word.<0:7> = "A" THEN ... ; !Check for "A" in 8-bit field
IF var.<15> THEN ... ; !Check for nonzero value
! in bit <15>
```

To access bits in the result of an expression, enclose the expression in parentheses:

```
INT result;
INT num1 := 51;
INT num2 := 28;

result := (num1 + num2).<4:7>;
```

Shifting Bit Fields You can shift a bit field a specified number of positions to the left or to the right within an INT or INT(32) expression. You can then use the result of the shift as an operand in an expression.

To shift a bit field, specify an INT or INT(32) expression, a left or right shift operator, and the number of positions to shift. For example, to shift the bits in an INT simple variable named VAR two positions to the left, specify:

```
var '<<' 2
```

For an INT expression, the shift occurs within a word. An INT expression can consist of STRING, INT, or UNSIGNED(1–16) operands.

For an INT(32) expression, the shift occurs within a doubleword. An INT(32) expression can consist of INT(32) and UNSIGNED(17–31) operands.

Bit-Shift Operators

Table 5-19 lists the bit-shift operators you can specify.

Table 5-19. Bit-Shift Operators

Operator	Function	Result
'<<'	Unsigned left shift through bit <0>	Zeros fill vacated bits from the right
'>>'	Unsigned right shift	Zeros fill vacated bits from the left.
<<	Signed left shift through bit <0> or bit <1>	Zeros fill vacated bits from the right. In arithmetic overflow cases, the final value of bit <0> is undefined (different for TNS/R accelerated mode than for TNS systems).
>>	Signed right shift	Sign bit (bit <0>) unchanged; sign bit fills vacated bits from the left

For signed left shifts (<<), programs that run on TNS/R systems use unsigned left shifts ('<<').

Number of Positions to Shift

Specify the number of bit positions to shift as an INT expression. A value greater than 31 gives undefined results (different on TNS and TNS/R systems).

Effect on Hardware Indicators

The bit-shift operation sets the condition code indicator, described under “Testing Hardware Indicators” earlier in this section.

Bit-Shift Operations

Bit-shift operations include:

Operation	User Action
Multiplication by powers of 2	For each power of 2, shift the field one bit to the left. (Some data might be lost.)
Division by powers of 2	For each power of 2, shift the field one bit to the right (Some data might be lost.)
Word-to-byte address conversion	Shift the word address one bit to the left, using an unsigned shift operator.

To multiply by powers of two, shift the field one position to the left for each power of 2 (Some data might be lost.) Here are examples:

```
a := b << 1;           !Multiply by 2
a := b << 2;           !Multiply by 4
a := b << 5;           !Multiply by 32
```

To divide by powers of two, shift the field one position to the right for each power of 2 (Some data might be lost.) Here are examples:

```
a := b >> 3;           !Divide by 8
a := b >> 4;           !Divide by 16
a := b >> 6;           !Divide by 64
```

To convert a word address to a byte address, use an unsigned shift operator. For example, you can convert the word address of an INT array to a byte address and initialize a STRING simple pointer with the byte address. You can then access the INT array as bytes and as words:

```
INT a[0:5];           !Declare INT array
STRING .p := @a[0] '<<' 1; !Declare and initialize
                        ! STRING simple pointer with
                        ! array byte address
p[3] := 0;           !Assign 0 to fourth byte
                        ! of A
```

You can shift the right byte of a word into the left byte and set the right byte to zero:

```

INT b;                !Declare variable
b := b '<<' 8;        !Shift right byte into left
                    ! byte, leaving zero in
                    ! right byte

```

The following unsigned left shift shows how zeros fill the vacated bits from the right:

```

Initial value = 0 010 111 010 101 000
'<<' 2 = 1 011 101 010 100 000

```

The following unsigned right shift shows how zeros fill the vacated bits from the left:

```

Initial value = 1 111 111 010 101 000
'>>' 2 = 0 011 111 110 101 010

```

The following signed left shift shows how zeros fill the vacated bits from the right, while the sign bit remains the same (TNS systems only):

```

Initial value = 1 011 101 010 100 000
<< 1 = 1 111 010 101 000 000

```

The following signed right shift shows how the sign bit fills the vacated bits from the left:

```

Initial value = 1 111 010 101 000 000
>> 3 = 1 111 111 010 101 000

```

6 Using Simple Variables

A simple variable is a single-element data item of a specified data type. You use simple variables to store data that can change during program execution.

This section describes:

- Declaring simple variables
- Initializing simple variables
- Allocating storage for simple variables
- Assigning data to simple variables
- Using simple variables, discussed by data type

Declaring Simple Variables

Before you access a simple variable, you must declare it. Declaring a variable associates an identifier with a memory address and informs the compiler how much memory storage to allocate for the variable.

You declare a simple variable by specifying a *data type* and an *identifier*, using the type and identifier formats described in Section 5, “Using Expressions.” For example, you can declare a simple variable named NUM of data type INT as follows:

```
INT num;
```

You can declare more than one variable in the same declaration. In this format, separate the variables with commas:

```
INT num1 ,  
    num2 ,  
    num3 ;
```

Simple variables are always directly addressed.

Specifying Data Types

When you declare a simple variable, you can specify any of the following data types. The data type determines the storage unit the compiler allocates for each simple variable:

Data Type	Storage Unit
STRING	Word
INT	Word
INT(32) or REAL	Doubleword
REAL(64) or FIXED	Quadrupleword
UNSIGNED(<i>n</i>)	Bit sequence of specified width

“Simple Variables by Data Types” in this section gives more information on specific data types.

Initializing Simple Variables You can initialize a simple variable of any data type (except UNSIGNED) when you declare it. Following the identifier in the declaration, specify an assignment operator (:=) and an *initialization value*:

```
INT var := 45;                                !Declare VAR and initialize
                                              ! it with the value 45
```

You can initialize a simple variable with character strings or numbers.

Initializing With Character Strings

When you initialize with a character string, specify a character string that has the same number of bytes as the simple variable or fewer. Each character in a character string requires one byte of contiguous storage. The value of any uninitialized bytes are undefined. In the following diagram, the question mark denotes an undefined value:

```
INT(32) chars := "ABC";
```



350

Initializing With Numbers

When you initialize with a number, you must specify a value of the same data type as the variable. In other words, specify a value that is in the range and format described for each data type in “Simple Variables by Data Type” in this section.

For example, to initialize a REAL simple variable, specify a REAL value. To initialize an INT(32) simple variable, specify an INT(32) value:

```
REAL flt_num := 365335.6E-3;
INT(32) dbl_num := 256D;
```

Specifying Number Bases

When you initialize a STRING, INT, or INT(32) variable with a number, you can specify integer constants in binary, octal, decimal, or hexadecimal base. The default number base in TAL is decimal. Table 6-1 describes the format of each number base.

Table 6-1. Number Base Formats

Number Base	Prefix	Digits Allowed	Example
Decimal	None	0 through 9	46
Octal	%	0 through 7	%57
Binary	%B	0 or 1	%B101111
Hexadecimal	%H	0 through 9, A through F	%H2F

Global, Local, and Sublocal Initializations

At the global level, initialize variables with expressions that contain only constants or LITERALS as operands. At the local or sublocal level, you can use any arithmetic expression, including previously declared variables:

```

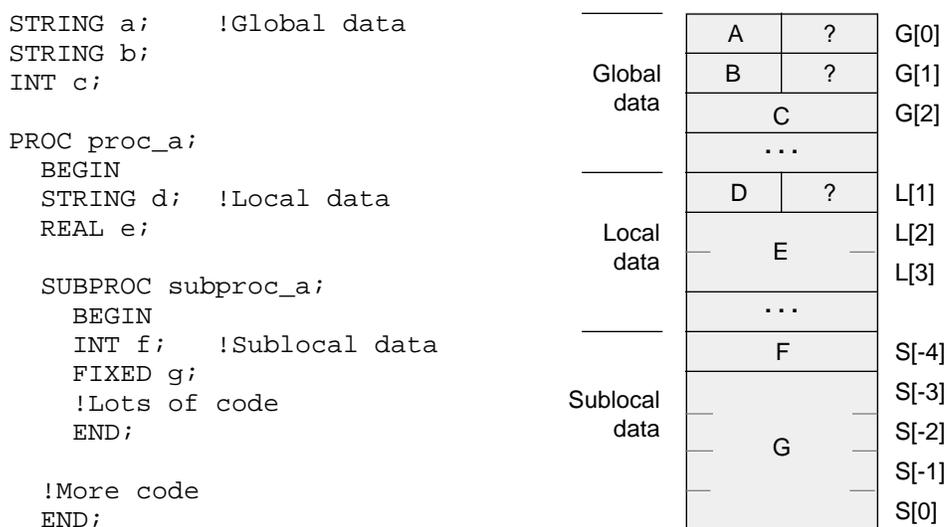
INT global := 34;                !Only constants allowed in
                                ! global initializations

PROC mymain MAIN;
  BEGIN
    INT local := global + 10;    !Any expression allowed in
    INT local2 := global * local; ! local and sublocal
    FIXED local3 := $FIX(local2); ! initializations
    !Lots of code
  END;                          !End of MYMAIN procedure

```

Allocating Simple Variables The compiler allocates storage in the global, local, or sublocal storage area based on the level at which you declare the variable, as shown in Figure 6-1. The question marks in the diagram denote undefined bytes.

Figure 6-1. Simple Variable Storage Allocation



351

Storage allocation specific to each data type is described in “Simple Variables by Data Type” later in this section. Storage allocation in programs that include BLOCK declarations is described in Section 14, “Compiling Programs.”

Assigning Data to Simple Variables

After you declare a variable, you can assign a value to it. In the assignment statement, specify the *identifier* of a previously declared simple variable, an assignment operator (`:=`), and an *expression*:

```
INT var;                !Declare VAR
var := 345;            !Assignment statement
```

Assigning Variables

You can assign an arithmetic expression that contains variables:

```
INT var1 := 5;         !Declare and initialize VAR1
INT var2;             !Declare VAR2
var2 := var1 + 10;    !Assign result of arithmetic
                    ! expression to VAR2
```

Matching Data Types

Assign values that match the data type of the variable. The assignment value should be in the range and format described for each data type in “Simple Variables by Data Type” in this section. For example, if you declare a REAL simple variable, specify an assignment value in the correct range and format for REAL constants:

```
REAL num;              !Declare NUM, a REAL variable
num := 36.6E-3;       !Assign REAL value to NUM
```

Converting Data Types

If necessary you can use type transfer functions to convert the data type of the assignment value to match the variable. For example, to convert an INT assignment value to an INT(32) value, use the \$DBL standard function:

```
INT(32) dblwd;
dblwd := $DBL(256);   !Convert assignment value to
                    ! match data type of variable
```

Assigning Character Strings

You can assign character strings to STRING, INT, and INT(32) variables in assignment statements. In assignments, the character string can contain one to four ASCII characters, depending on the variable’s data type. (You can also initialize such variables with character strings when you declare the variables.)

You cannot, however, assign character strings to FIXED, REAL, or REAL(64) variables, although you can initialize such variables with character strings when you declare the variables.

Multiple Variables You can assign a value to more than one simple variable at a time. In the following example, the first assignment statement is equivalent to the three assignment statements that follow it:

```

INT int1;
INT int2;
INT int3;                                !Declarations

int1 := int2 := int3 := 16;             !First assignment statement

int1 := 16;                               !These three assignment
int2 := 16;                               ! statements are equivalent to
int3 := 16;                               ! the first assignment statement

```

Simple Variables by Data Type The following subsections present information about simple variables depending on their data type.

STRING Simple Variables A STRING simple variable can contain an unsigned 8-bit integer in the range 0 through 255 or a one-character character string:

```

STRING a := 59;                            !Decimal number
STRING b := %12;                            !Octal number
STRING c := %B101;                          !Binary number
STRING d := %h2A;                            !Hexadecimal number
STRING e := "A";                             !Character string

```

Storage Allocation

A STRING simple variable represents a byte value, but the compiler allocates a word. The compiler allocates the initialization value in the left byte of the word. The right byte is undefined. Here are allocation examples of initializations with a character string and a number:

```

STRING a_char := "A";
STRING byte_num := 254;

```

"A"	?
254	?

352

INT Simple Variables An INT simple variable can contain a signed or unsigned 16-bit integer in the range 0 through 65,535 (unsigned) or -32,768 through 32,767 (signed). It can also contain a character string of up to two characters:

```

INT a := 5;                !Unsigned decimal number
INT b := -%5;              !Signed octal number
INT c := %B1001111000010001; !Binary number
INT d := %h2f;             !Hexadecimal number
INT e := "AB";             !Character string

```

You can initialize INT variables with the standard addresses of simple variables, arrays, or structures. The @ operator fetches the address of the variable:

```

INT var;                   !Declare word-addressed VAR

INT var_addr := @var;      !Declare VAR_ADDR; initialize
                          ! it with word address of VAR

```

You can convert a word address to a byte address by using a logical left bit-shift operation ('<<' 1):

```

INT var;                   !Declare word-addressed VAR

STRING .var_ptr := @var '<<' 1;
                          !Declare VAR_PTR; initialize
                          ! it with converted byte
                          ! address of VAR

```

You can convert a byte address to a word address by using a logical right bit-shift operation ('>>' 1):

```

STRING var;                !Declare byte-addressed VAR

INT var_addr := @var '>>' 1; !Declare VAR_ADDR; initialize
                          ! it with converted word
                          ! address of VAR

```

Storage Allocation

The compiler allocates a word for each INT simple variable. Here are examples of numeric and character string initializations:

```
INT int_num := %110;
INT two_chars := "AB";
```

%110	
"A"	"B"

353

For INT simple variables, a one-byte initialization behaves differently from a one-byte assignment as follows:

- If you **initialize** an INT variable with a one-byte character string, the compiler allocates the character in the **left** byte of the word. The right byte is undefined.
- If you **assign** a one-byte character string to the variable, at run-time the system places it in the **right** byte and sets the left byte to zero. If you want the character to be placed in the left byte, assign a two-byte character string that consists of a character and a blank space.

The following example contrasts how the system stores a byte initialization in INT simple variables as opposed to how the system stores a byte assignment:

```
INT i1, i2;
INT i3 := "A"; !Initialize with "A"
i1 := "A"; !Assign "A"
i2 := "A "; !Assign "A" and a blank
```

"A"	?
0	"A"
"A"	" "

354

INT(32) Simple Variables An INT(32) simple variable can contain a signed or unsigned 32-bit integer in the range $-2,147,483,648$ through $2,147,483,647$, suffixed by **D** for decimal, octal, or binary integers or **%D** for hexadecimal integers. It also can contain a character string of up to four characters.

```

INT(32) a := 0D;           !Decimal number
INT(32) b := -327895066D; !Decimal number
INT(32) c := %1707254361D; !Octal number
INT(32) d := %B000100101100010001010001001D;
                           !Binary number
INT(32) e := -%H99FF29%D; !Hexadecimal number
INT(32) f := "ABCD";      !Character string

```

You can initialize INT(32) simple variables with the 32-bit addresses of extended indirect variables:

```

INT .EXT x_array[0:2];    !Declare X_ARRAY
INT(32) x_addr := @x_array[0]; !Declare X_ADDR; initialize
                           ! it with address of X-ARRAY

```

Storage Allocation

For an INT(32) simple variable, the compiler allocates a doubleword. The compiler allocates numeric and character string initialization values as follows:

```

INT(32) dbl_num := 256D;

INT(32) dbl_chars := "ABC";

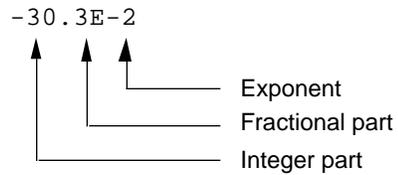
```

256D	
"A"	"B"
"C"	?

355

REAL Simple Variables A REAL simple variable can contain a signed 32-bit floating-point number in the range $\pm 8.6361685550944446 * 10^{-78}$ through $\pm 1.15792089237316189 * 10^{+77}$, precise to approximately seven significant digits.

The format of a REAL constant includes an integer part, a fractional part suffixed by E, and an exponent. Here is an example of a REAL constant value:



356

Here are more examples of REAL constant values:

Decimal Value	REAL
0	0.0E0
2	2.0e0
2	0.2E1
2	20.0E-1
-17.2	-17.2E0
-17.2	-1720.0E-2

Storage Allocation

The compiler allocates a doubleword of storage for each REAL simple variable:

```
REAL num := 2.0E0;
```



357

REAL(64) Simple Variables A REAL(64) simple variable can contain a signed 64-bit floating-point number in the same range as REAL numbers, but precise to approximately 17 significant digits.

The format of a REAL(64) constant is the same as for REAL constants, except that the suffix is L instead of E. Here are examples of REAL(64) values:

Decimal Value	REAL(64)
0	0.0L0
2	2.0L0
2	0.2L1
2	20.0L-1
-17.2	-17.2L0
-17.2	-1720.0L-2

Storage Allocation

The compiler allocates a quadrupleword of storage for each REAL(64) simple variable:

```
REAL(64) num := 2718.2818284590452L-3;
```

%26770
%52130
%121273
%45401

FIXED Simple Variables A **FIXED** simple variable can contain a signed 64-bit fixed-point number in the range $-9,223,372,036,854,775,808$ through $9,223,372,036,854,775,807$, suffixed by **F** for decimal, octal, or binary numbers or **%F** for hexadecimal numbers. For decimal numbers, you can also specify a fractional part, preceded by a decimal point:

```
300.667F                                !FIXED decimal number with
                                           ! fractional part
```

You can initialize a **FIXED** variable with a character string when you declare the variable. You cannot, however, use an assignment statement to assign a character string to a **FIXED** variable.

Fixed-Point Settings (fpoints)

When you declare a **FIXED** variable, you can specify the implied fixed-point setting (*fpoint*) for values stored in the variable. The *fpoint* is an integer in the range -19 through 19 , enclosed in parentheses, following the **FIXED** keyword. The default *fpoint* is 0 . You can specify a positive or negative *fpoint*.

Positive *fpoints*. A positive *fpoint* specifies the number of decimal places to the right of the decimal point:

```
FIXED(3) x := 0.642F;                    !Stored as 642; accessed
                                           ! as 0.642
```

Take care to specify an *fpoint* that allows enough decimal places for all the data that you might assign to the variable. The system truncates any assignment value that does not fit. For example, if you declare a **FIXED(2)** variable and then assign a value that has three decimal places, the rightmost digit is lost:

```
FIXED(2) some_num;                       !fpoint is 2
some_num := 2.348F;                      !Stored as 234; accessed
                                           ! as 2.34
```

Negative *fpoints*. A negative *fpoint* specifies a number of integer places to the left of the decimal point. To store a value, a negative *fpoint* truncates digits leftward from the decimal point in accordance with the *fpoint*. When you access the value, zeros replace the truncated digits:

```
FIXED(-3) y := 642913F;                  !Stored as 642; accessed
                                           ! as 642000
```

FIXED(*)

If you declare a **FIXED(*)** simple variable, the value stored in the variable is not scaled.

Stored Forms

The following examples compare the stored form of values having various fpoints:

```

FIXED(-3) a := 643000F;           !Stored as 000000643
FIXED(-3) b := .643F;           !Stored as 000000000
FIXED      c := 643000F;           !Stored as 000643000
FIXED      d := .643F;           !Stored as 000000000
FIXED(3)   e := 643000F;           !Stored as 643000000
FIXED(3)   f := .643F;           !Stored as 000000643
    
```

Number Bases

The following initialization examples illustrate different number bases:

```

FIXED      a := 239840984939873494F;           !Decimal number
FIXED(3)   b := %B1010111010101101010110F;    !Binary number
FIXED(5)   c := %765235512F;                   !Octal number
FIXED      d := %H298756F;                       !Hexadecimal number
    
```

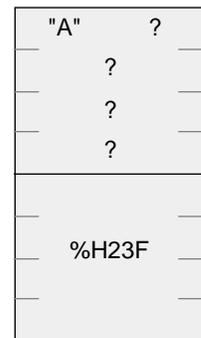
Storage Allocation

The compiler allocates a quadrupleword for each FIXED variable. It allocates character string and numeric initialization values as follows:

```

FIXED char := "A";

FIXED num := %H23F;
    
```



359

You can initialize a FIXED variable with a character string when you declare the variable (as shown in the preceding example), but you cannot assign a character string to such a variable.

UNSIGNED Simple Variables When you declare an UNSIGNED simple variable, you must specify the *width*, in bits, of the simple variable. Specify the *width* as a constant expression in the range 1 through 31, enclosed in parentheses, following the UNSIGNED keyword. The constant expression can include LITERALS and DEFINES:

```
UNSIGNED(5) bit_var;           !Width of BIT_VAR is 5 bits
```

You cannot initialize UNSIGNED variables when you declare them.

Storage Allocation

The compiler packs consecutive UNSIGNED simple variables where possible. That is, the compiler allocates the first UNSIGNED variable starting on a word boundary, and then allocates each successive UNSIGNED variable in the remaining bits of the same word as the preceding variable if:

- The variable contains 1 to 16 bits and fits in the same word
- The variable contains 17 to 31 bits and fits in the same word plus the next word

If an UNSIGNED variable does not fit in the same word or doubleword, the compiler starts the variable on a word boundary.

```
STRING a;
UNSIGNED(8) d, e;
UNSIGNED(3) f;
UNSIGNED(5) g;
UNSIGNED(6) h;
UNSIGNED(25) j;
UNSIGNED(2) k;
```

A		?	
D		E	
F	G	H	?
J			
		K	?

360

7 Using Arrays

An array is a collectively stored set of elements of the same data type. You use arrays to store constants, especially character strings. You can use the array identifier to access the elements individually or as a group.

This section describes:

- Declaring arrays
- Initializing arrays
- Allocating storage for arrays
- Accessing arrays
- Assigning data to arrays
- Copying data into arrays
- Scanning arrays
- Comparing arrays

This section mostly describes arrays located in the user data segment or in an extended data segment. “Read-only Arrays” at the end of the section briefly discusses arrays located in a user code segment.

Section 8, “Using Structures,” shows how structures can simulate multidimensional arrays, arrays of arrays, or arrays of structures.

Declaring Arrays

Before processing an array, you must declare it and store data in it. The declaration associates an identifier with a memory address. It also tells the compiler how much storage to allocate for the array and the storage area in which to allocate it.

To declare an array, specify:

- A *data type*
- An *identifier*, usually preceded by an indirection symbol (. or .EXT)
- Lower and upper *bounds*—the indexes of the first and last array elements, specified as INT constant expressions in the range $-32,768$ through $32,767$, separated by a colon and enclosed in brackets as follows. The upper bound must be equal to or larger than the lower bound.

```
[0:5]                                !Six elements
```

Here are examples of array bounds you can declare:

```
STRING a_array[0:2];                !Three-element array
INT b_array[0:19];                  !Twenty-element array
UNSIGNED(1) flags[0:15];           !Array of 16 one-bit elements
```

Specify the data type and identifier using the type and identifier formats described in Section 5, “Using Expressions.”

The preceding arrays are all direct arrays; that is, declared without an indirection symbol. Arrays of any data type can be direct. The compiler allocates storage for direct arrays in the primary areas of the global, local, or sublocal storage areas.

Using Indirection The global primary storage area is limited to 256 words and the local primary area is limited to 127 words. The global and local secondary areas have no explicit size, and the total of all primary and secondary areas can be as large as the lower 32K-word area of the user data segment. You can minimize the impact on the primary areas by declaring indirect global and local arrays. Global and local arrays of any data type except UNSIGNED can be indirectly addressed.

The sublocal storage area has no secondary area, so all sublocal arrays must be directly addressed.

To declare a standard indirect array, precede the array identifier with a standard indirection symbol (.) as follows:

```
INT .array_x[0:1];           !Declare standard indirect array
```

To declare an extended indirect array, precede the array identifier with an extended indirection symbol (.EXT) as follows:

```
INT .EXT array_y[0:1];      !Declare extended indirect array
```

Specifying Data Types The data type determines the kind of values the array can contain. The data type also determines the storage unit the compiler allocates for each array element, as follows:

Data Type	Storage Unit
STRING	Byte
INT	Word
INT(32) or REAL	Doubleword
REAL(64) or FIXED	Quadrupleword
UNSIGNED	Sequence of 1, 2, 4, or 8 bits

“Arrays by Data Type” in this section gives more information on each data type.

Initializing Arrays You can initialize most arrays when you declare them. You cannot initialize UNSIGNED or local extended indirect arrays. It is recommended that you initialize arrays with values that are appropriate for the data type of the array.

To initialize an array, include an assignment operator (:=) and a *constant list* or a *constant* in the array declaration. For example, you can initialize an array with a constant list as follows:

```
INT array[0:1] := ["A", "B"];
```

More information on array initializations is given in the following subsections.

Initializing Arrays With Constant Lists

A constant list can include the following elements:

- Numbers

```
INT .numbers[0:5] := [1,2,3,4,5,6];
```

- Character strings of up to 127 characters on one line

```
INT(32) .words[0:3] := ["cats", "dogs", "bats", "cows"];
STRING .buffer[0:102] := [ "A constant list can consist ",
                          "of several character string constants ",
                          "one to a line, separated by commas." ];
```

- Repetition factors—INT constants by which to repeat constant lists

```
INT zeros[0:9] := 10 * [0]; !Repetition factor of 10
```

You can nest constant lists that include repetition factors. The following example expands to [1,1,0,0,1,1,0,0,1,1,0,0]:

```
INT digits[0:11] := [3 * [2 * [1], 2 * [0]]];
```

- LITERALS

```
LITERAL len = 80;
STRING .buffer[0:len - 1] := len * [" "];
```

If you specify fewer initialization values than the number of elements (and the values are appropriate for the data type of the array), the values of uninitialized elements are undefined:

```
STRING bean[0:9] := [1,2,3,4]; !Values in BEAN[4:9]
                          ! are undefined
```

Initializing Arrays With Constants

You can initialize an array with a numeric constant or a character string constant:

```
INT some_array[0:3] := -1;      !Values in SOME_ARRAY[1:3]
                              ! are undefined

INT any_array[0:1] := "abcd"; !Store one character per byte
```

Global, Local, and Sublocal Initializations

You can initialize an array declared at any level except for extended indirect arrays declared at the local level:

```

INT(32) .a[0:1] := [5D, 7D];      !Global array can be
                                  ! initialized

PROC my_procedure;
  BEGIN
    STRING .b[0:1] := ["A","B"]; !Local array can be
                                  ! initialized

    FIXED .EXT c[0:3];           !Local extended indirect
                                  ! array cannot be initialized

    SUBPROC my_subproc;
      BEGIN
        INT d[0:2] := ["Hello!"]; !Sublocal array can be
        !Lots of code              ! initialized
      END;
    END;
  END;

```

Arrays by Data Type The following subsections give information about arrays by data type. For information about the appropriate range and format of values for each data type, see Section 6, "Using Simple Variables."

STRING Arrays

For STRING arrays, the compiler allocates one byte for each element. The compiler always starts the zeroth element of a STRING array on a word boundary. In the diagram, question marks denote undefined values:

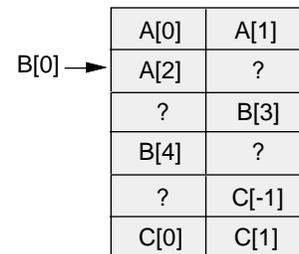
```

STRING a[0:2];

STRING b[3:4];

STRING c[-1:1];

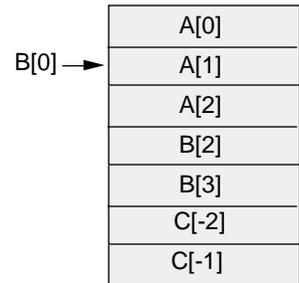
```



INT Arrays

For INT arrays, the compiler allocates a word for each element:

```
INT a[0:2];
INT b[2:3];
INT c[-2:-1];
```

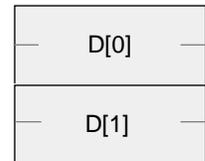


362

INT(32) Arrays

For INT(32) arrays, the compiler allocates a doubleword for each element:

```
INT(32) d[0:1];
```

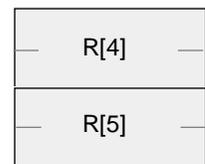


363

REAL Arrays

For REAL arrays, the compiler allocates a doubleword for each element:

```
REAL r[4:5];
```

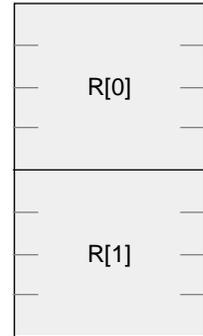


364

REAL(64) Arrays

For REAL(64) arrays, the compiler allocates a quadrupleword for each element:

```
REAL(64) r[0:1];
```

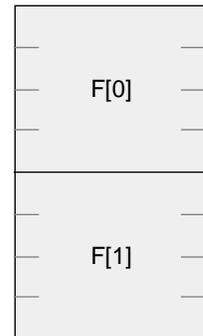


365

FIXED Arrays

For FIXED arrays, the compiler allocates a quadrupleword for each element:

```
FIXED f[0:1];
```



366

When you declare a FIXED array, you can specify the implied *fpoint* of values you store in the array elements. The *fpoint* is an integer in the range -19 through 19, enclosed in parentheses, following the FIXED keyword. The default *fpoint* is 0 (no decimal places). Here is an example of a FIXED array with an *fpoint* of 5:

```
FIXED(5) array[0:2];
```

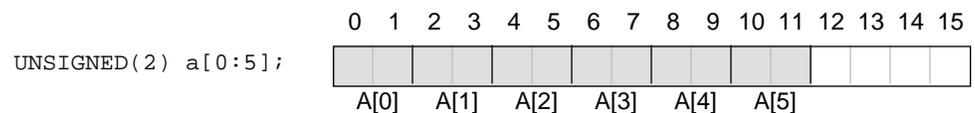
If you declare a FIXED(*) array, values stored in the array are not scaled and are treated as having an *fpoint* of 0.

UNSIGNED Arrays

When you declare an UNSIGNED array, you must specify as part of the data type a value of 1, 2, 4, or 8 that specifies the *width*, in bits, of the elements in the array. Here is an example of an UNSIGNED array that has eight 4-bit elements:

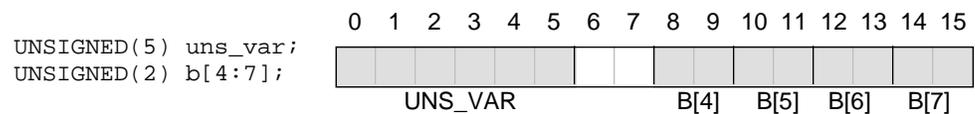
```
UNSIGNED(4) array[0:7];
```

The compiler packs allocation of UNSIGNED array elements in sequential words. A word can contain up to sixteen 1-bit elements, eight 2-bit elements, four 4-bit elements, or two 8-bit elements in successive bit fields. For example, if you declare an array as having six 2-bit elements, the compiler packs allocation of all six 2-bit elements in the same word:



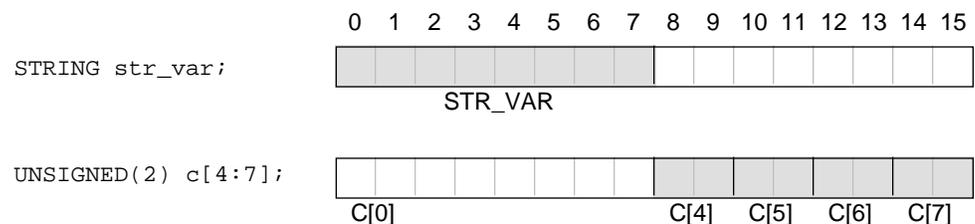
367

The compiler always allocates the zeroth element of an UNSIGNED array at a word boundary. For example, if you declare an UNSIGNED simple variable followed by an UNSIGNED array having bounds of [4:7], the compiler allocates the array in the same word as the simple variable, with the zeroth array element at bit [0]:



368

If you declare a STRING simple variable followed by an UNSIGNED array, the compiler allocates the zeroth array element starting at bit [0] of the next word:



369

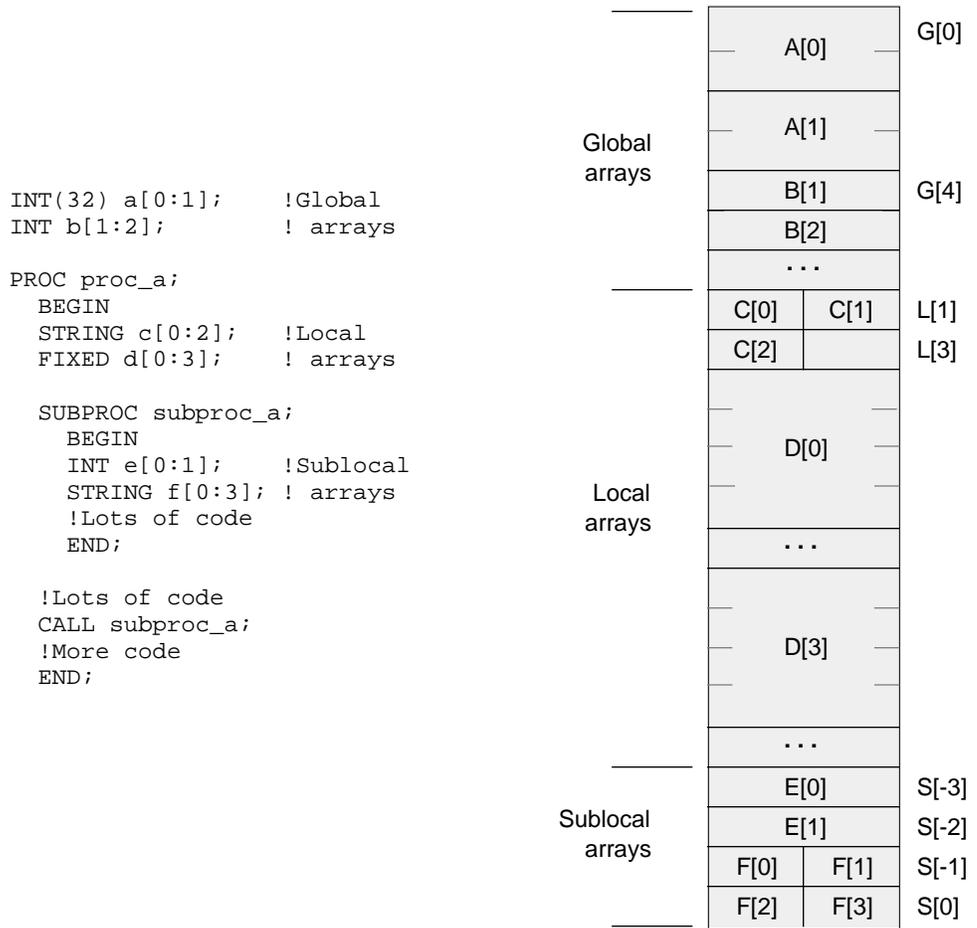
Allocating Arrays The compiler allocates storage for an array in a particular storage area based on declaration characteristics such as:

- The global, local, or sublocal level
- The direct, standard indirect, or extended indirect addressing mode

Allocating Direct Arrays

The compiler allocates storage for directly addressed arrays in the global, local, or sublocal primary areas of the user data segment as shown in Figure 7-1.

Figure 7-1. Allocating Direct Arrays



Allocating Indirect Arrays

You can declare global or local indirect arrays. Sublocal arrays cannot be indirectly addressed.

For each standard indirect array, the compiler allocates space as follows:

1. It allocates a word of storage in the global (or local) primary area of the user data segment for an implicit standard pointer.
2. It then allocates storage for each array in the global (or local) secondary area.
3. Finally, it initializes each implicit pointer (provided in step 1) with the 16-bit address of the array. For a `STRING` array, the pointer contains a byte address. For any other array, the pointer contains a word address.

For each extended indirect array, the compiler allocates space as follows:

1. It allocates a doubleword of storage in the global (or local) primary area of the user data segment for an implicit extended pointer.
2. It then allocates storage for each extended array in an automatic extended data segment.
3. Finally, it initializes each implicit pointer (provided in step 1) with the 32-bit byte address of the array. The address is always an even-byte address.

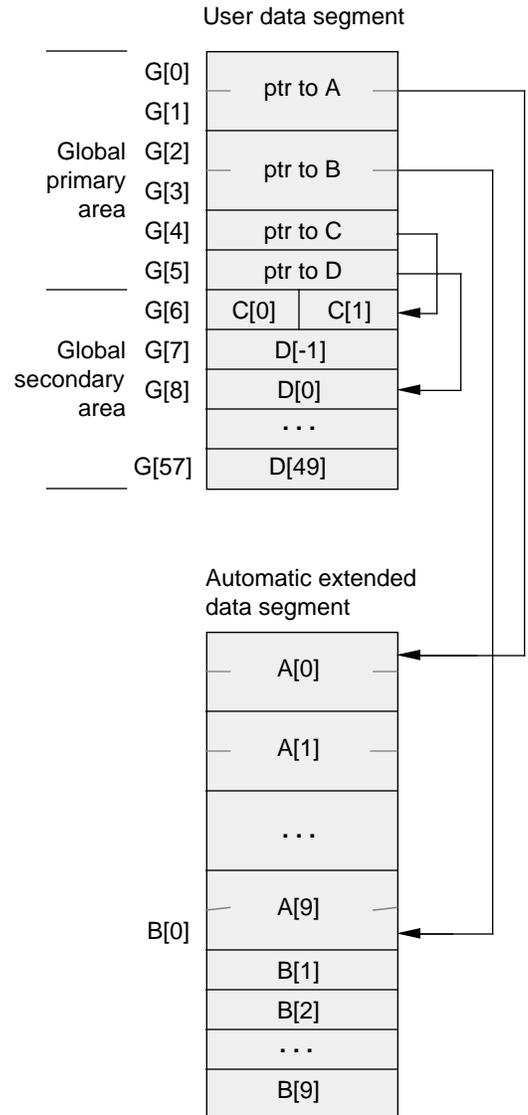
If you declare arrays within `BLOCK` declarations, however, the compiler allocates storage anywhere within the list of data blocks, as described in Section 14, "Compiling Programs."

Figure 7-2 shows storage allocation for global indirect arrays.

Figure 7-2. Allocating Indirect Arrays

```

INT(32) .EXT a[0:9];
INT .EXT b[1:9];
STRING .c[0:1];
INT .d[-1:49];
    
```



Addressability of Arrays The zeroth element of an array must always be addressable. If it is not addressable, the compiler issues an address range violation error.

Addressability in the User Data Segment

The zeroth element of a direct array must fit within the lower 32K-word area of the user data segment, even if the zeroth element is not allocated.

The global area has G-plus addressing. If a global array is located at G[0], its lower bound must be a zero or negative value. Avoid the following practice:



The sublocal area has S-minus addressing. If a sublocal array is located at S[0], its upper bound must be a zero or larger value. Avoid the following practice:



Addressability in the Extended Segment

The zeroth element of an extended indirect array must reside within the automatic extended data segment, even if the zeroth element is not allocated. If an extended indirect array is located at the beginning of the extended data segment, its lower bound must be a zero or negative value. Avoid the following practice:



Accessing Arrays After you declare an array, you can access its elements by using the array identifier in statements, regardless of addressing mode. For example, you can declare a direct array, a standard indirect array, and an extended indirect array, and then access each by its identifier:

```
INT dir_array[0:2];      !Declare direct array
INT .std_array[0:2];    !Declare standard indirect array
INT .EXT ext_array[0:2]; !Declare extended indirect array

dir_array[2] := 5;      !Access third element of each
std_array[2] := 5;      ! array by using its identifier in
ext_array[2] := 5;      ! an assignment statement
```

Indexing Arrays To access an array element, you append an *index* such as [5] to the array identifier. You can access element [0] of any array (except type UNSIGNED) by specifying the array identifier with no index. Thus, the references BUFFER and BUFFER[0] are equivalent. To access an UNSIGNED array, however, you must always append an index:

```
UNSIGNED(8) uns_array[0:2];
uns_array[0] := 0;      !UNS_ARRAY requires index
```

Index Values

The index value represents the array element you want to access, relative to the zeroth element. For example, to access the third element, specify an index of [2]:

```
INT array[0:2];
array[2] := 5;          !Access third element of array
```

For the index value, use a signed arithmetic expression:

- For standard addressing, use a signed INT expression, which has a range of -32,768 through 32,767.
- For extended addressing, use either a signed INT expression or an INT(32) expression, which has a range of -2,147,483,648 through 2,147,483,647.

You can use constants and LITERALS as index values:

```
LITERAL index = 5;           !Declare LITERAL
INT table[0:9];             !Declare array

table[index] := "AB";       !Access sixth element of array
```

You can use variables as index values:

```
INT .EXT b[0:10];           !Declare array
INT .c[0:9];               !Declare array
INT(32) x;                 !Declare variables X, Y, and
INT y;                     ! Z to use for indexes
INT z;

!Code to manipulate indexes and initialize arrays

b[x] := c[y-z];            !Access array element
```

Assigning Data to Array Elements

You can assign data to one array element at a time. For each array element, use a separate assignment statement:

```
STRING .an_array[0:4];     !Declare AN_ARRAY

an_array[1] := "Z";        !Assign "Z" to second
                           ! element of AN_ARRAY
```

You cannot use constant lists in assignment statements (as you can in declarations) to assign values to multiple array elements. For example, the following initialization of THIS_ARRAY is equivalent to the three assignment statements applied to THAT_ARRAY:

```
INT .this_array[0:2] := ["ABCDEF"];
                           !Constant list initializes
                           ! all elements of THIS_ARRAY

INT .that_array[0:2];

that_array[0] := "AB";     !Assignment statements assign
that_array[1] := "CD";     ! values to elements of
that_array[2] := "EF";     ! THAT_ARRAY, one at a time
```

However, you can copy data to multiple array elements by using a move statement, described in “Copying Data Into Arrays” later in this section.

Assigning the Address of Arrays

You can assign the address of array elements to other variables. For example, to assign the address of ARRAY[1] to VAR, prefix the array identifier with the @ operator in an assignment statement:

```
INT .array[0:2];           !Declare array
INT var;                   !Declare simple variable

var := @array[1];         !Assign address of ARRAY[1]
                           ! to VAR
```

Copying Data Into Arrays

To copy data to multiple array elements, use the move statement. You can, for example, copy:

- A constant list into an array
- Data between arrays
- Data within an array

Copying a Constant List Into an Array

To copy a constant list into an array, specify the *destination* array and the *constant list* in a move statement. You can copy the source data from left to right or from right to left.

Copying Left to Right

To start copying from the leftmost item of the source data, use the left-to-right move operator (':='). For example, you can start copying from the leftmost character in a source character string such as "A ... Z." In this case, you copy "A" into element [0] of the destination array, then "B" into element [1], and so on through element [25]:

```
STRING .alpha_array[0:25];           !Declare 26-element array
alpha_array[0] ':=' ["ABCDEFGHIJKLMNOPQRSTUVWXYZ"];
                                   !Copy "A" through "Z" into
                                   ! ALPHA_ARRAY[0] through [25]
```

Copying Right to Left

To start copying from the rightmost item of the source data, use the right-to-left move operator ('=:'). For example, you can start copying from the rightmost constant of a constant list such as [1, 2, 3, 4]. In this case, you copy the constant 4 into element [3] of the destination array, the constant 3 into element [2], and so on through element [0]:

```
INT num_array[0:3];                 !Declare 4-element array
num_array[3] '=: ' [1, 2, 3, 4]; !Copy 4 through 1 into
                                   ! NUM_ARRAY[3] through [0]
```

Using Repetition Factors

To repeat the same value in consecutive elements of the destination array, specify a repetition factor followed by a multiplication operator (*) and the value to repeat. For example, you can copy a zero into all elements of the destination array:

```
LITERAL len = 100;                 !Specify repetition factor
INT .an_array[0:len - 1];         !Declare 100-element array
an_array[0] ':=' len * [0];       !Copy a zero into all
                                   ! elements of AN_ARRAY
```

Copying a Byte Constant Into a STRING Array

To copy a single byte constant into an element of the destination array, enclose the constant in brackets in the move statement. The destination array must be a STRING array or have a byte address:

```
STRING x[0:8];                !Declare STRING array X
x[0] := ["A"];                !Copy a single byte;
                               ! puts "A" in X[0]
```

If you do not enclose the constant in brackets, you copy a word, doubleword, or quadrupleword depending on the size of the constant. The following example repeats the preceding example, substituting an unbracketed constant in the move statement:

```
STRING x[0:8];                !Declare STRING array X
x[0] := "A";                  !Copy a word; put %0 in
                               ! X[0] and "A" in X[1]
```

Copying Data Between Arrays

To copy data from one array to another, specify the *destination* and *source* arrays in the move statement and include the FOR clause. In the FOR clause, specify a *count* value—an INT arithmetic expression that specifies the number of elements, bytes, or words you want to copy.

Copying Bytes

To copy bytes regardless of source data type, specify the BYTES keyword in the FOR clause of the move statement. BYTES copies the number of bytes specified by the *count* value. If both *source* and *destination* have word addresses, however, BYTES generates a word copy for $(count + 1) / 2$ words.

For example, you can copy bytes instead of words from an INT array as follows:

```
LITERAL length = 70;          !Number of array elements
INT .new_array[0:length - 1]; !Destination array
INT .old_array[0:length - 1]; !Source array
INT file_number;              !File number
INT byte_count;               !Count value (number of
                               ! bytes to copy)

!Lots of code here

CALL READ (file_number, old_array, byte_count);
new_array[0] := old_array[0] FOR byte_count BYTES;
                               !Copy bytes from OLD_ARRAY
                               ! to NEW_ARRAY
```

Copying Words

To copy words regardless of source data type, specify the **WORDS** keyword in the **FOR** clause. **WORDS** generates a word copy for the number of words specified by the *count* value.

For example, to copy words instead of doublewords from an **INT(32)** source array, multiply **LENGTH** by 2 and include the **WORDS** keyword:

```
LITERAL length = 12;           !Count value (number of
                                ! words to copy)
INT(32) .new_array[0:length - 1];!Destination
INT(32) .old_array[0:length - 1];!Source

!Some code here to put values in OLD_ARRAY

new_array[0] ':=' old_array[0] FOR 2 * length WORDS;
                                !Copy 24 words from
                                ! OLD_ARRAY to NEW_ARRAY
```

Copying Elements

To copy elements based on the data type of the source array, you can specify the **ELEMENTS** keyword. For example, you can copy doubleword values from an **INT(32)** source array into the destination array as follows:

```
LITERAL length = 12;           !Count value (number of
                                ! elements to copy)
INT(32) .new_array[0:length - 1];!Destination
INT(32) .old_array[0:length - 1];!Source

!Some code here to put values in OLD_ARRAY

new_array[0] ':=' old_array[0] FOR length ELEMENTS;
                                !Copy 12 doublewords from
                                ! OLD_ARRAY to NEW_ARRAY
```

When you copy array elements, the **ELEMENTS** keyword is optional but provides clearer source code. When you copy structure occurrences, the **ELEMENTS** keyword is required, as described in Section 8, “Using Structures.”

Copying Data Within an Array To copy data within an array, specify the same array for the destination and source arrays and include the FOR clause in the move statement.

For example, you can free element [0] of a 12-element array by specifying the following move statement. This move statement first copies element [10] into element [11], then element [9] into element [10], and so forth. Finally, it copies element [0] into element [1], thereby freeing element [0] for new data:

```
LITERAL upper_bound = 11;           !Upper bound of array
FIXED .buffer[0:upper_bound];      !12-element array

!Some code here to put values in
! BUFFER[0] through BUFFER[10]

buffer[upper_bound] '='
  buffer[upper_bound - 1] FOR upper_bound;
                                     !Start copy with BUFFER[10]
                                     ! into BUFFER[11]
```

Using the Next Address The next address (also known as *next-addr*) is the memory location immediately following the last item copied. The next address is returned by the move statement. You can use the next address for various purposes, such as the starting location for a new group of values.

First declare a simple pointer and then use its identifier (prefixed by @) in the next-address clause in a move statement. Here is an example of the next-address clause:

```
-> @next_addr_ptr
```

For example, you can copy spaces into the first five elements of an array, and then use the next address as the destination for copying dashes into the next five elements:

```
LITERAL len = 10;                   !Length of array
STRING .array[0:len - 1];           !Destination array
STRING .next_addr_ptr;              !Simple pointer for
                                     ! the next address

array[0] ':=' 5 * [" "] -> @next_addr_ptr;
                                     !Complete first copy
                                     ! and capture next address
next_addr_ptr ':=' 5 * ["-"];       !Use next address as start
                                     ! of second copy operation
```

The compiler does a standard move and returns a 16-bit next address if:

- Both arrays have standard byte addresses
- Both arrays have standard word addresses

The compiler does an extended move and returns a 32-bit next address if:

- One of the two arrays has a standard byte address and the other has a standard word address
- Either array has an extended address

STRING arrays and arrays pointed to by STRING pointers are byte addressed. All other arrays are word addressed.

Copying Bytes Into INT Arrays

To copy data from a byte-addressed source array into a word-addressed destination array, declare an extended STRING simple pointer and use it in the next-address clause of the move statement:

```
STRING  byte_array[0:9];           !Byte-addressed source array
INT     word_array[0:4];           !Word-addressed destination
                                           ! array
STRING  .EXT next_addr;           !STRING simple pointer for
                                           ! the next-address clause

!Some code here
word_array[0] ':=' byte_array[0] FOR 3 BYTES -> @next_addr;
                                           !Copy three bytes into
                                           ! WORD_ARRAY
```

When the copy operation is complete, the next-address pointer (NEXT_ADDR) points to the right byte of WORD_ARRAY[1], not the left byte.

Concatenating Copy Operations

You can concatenate any number of move sources in a single move statement by using the ampersand operator (&).

The following move statement concatenates six move sources. It copies three string constants and data from three arrays into LINE_ARRAY:

```
LITERAL line_len = 63;             !Length of destination array
LITERAL date_len = 11;            !Length of source array 1
LITERAL id_len = 11;             !Length of source array 2
LITERAL dept_len = 3;            !Length of source array 3

STRING .line_array[0:line_len - 1]; !Destination array
STRING .date_array[0:date_len - 1] := "Feb 1, 1992";
STRING .id_num[0:id_len - 1] := "854-70-1950";
STRING .dept_num[0:dept_len - 1] := "107";

line_array ':=' "    DATE: " & date_array FOR date_len BYTES
                & "    ID NUMBER: " & id_num FOR id_len BYTES
                & "    DEPARTMENT: " & dept_num FOR dept_len BYTES;
```

When the preceding example executes, `LINE_ARRAY` contains the following:

```
DATE: Feb 1, 1992    ID NUMBER: 854-70-1950    DEPARTMENT: 107
```

Initializing a Large Array Quickly

To initialize a large array quickly, you can concatenate two copy operations in a move statement. The first copy operation copies two spaces into element [0], and the second copies the spaces from element [0] into the remaining elements:

```
LITERAL length = 100;           !Length of array
INT .array[0:length - 1];      !Destination array

array[0] := " " & array[0] FOR (length - 1);
                                !Initialize array to blanks
```

In the preceding example, make sure the value in the FOR clause is a positive number. The move statement treats the value as an unsigned integer. It treats a small negative number (such as -1) as a large positive number (in this case, 65,535).

Scanning Arrays

You can use scan statements to scan arrays for a test character. You can apply scan statements to any array (except `UNSIGNED` arrays) located in the lower 32K-word area of the user data segment or no more than 32K words away in the user code segment. You can only scan bytes.

The `SCAN` statement scans an array from left to right. The `RSCAN` statement scans an array from right to left. Both scan statements return the next address, described in "Using the Next Address" earlier in this section.

Delimiting the Scan Area

Unless you delimit the scan area, a scan operation might continue to the 32K-word boundary if:

- A `SCAN UNTIL` operation does not find a zero or the test character
- A `SCAN WHILE` operation does not find a zero or a character other than the test character

When you declare the array to scan, you can use zeros to delimit the start and end of the scan area. Here is an example for delimiting the scan area with zeros:

```
INT .buffer[-1:20] := [0," John James Jones ",0];
```

Here is another example for delimiting the scan area with zeros:

```
LITERAL stopper = 0;
STRING an_array[0:9];

!Fill array from some source
an_array[0] := stopper;
an_array[9] := stopper;
```

Determining What Stopped the Scan To determine what stopped the scan, test `$CARRY` in an IF statement immediately after the `SCAN` or `RSCAN` statement. If `$CARRY` is true after a `SCAN UNTIL`, the test character did not occur. If `$CARRY` is true after `SCAN WHILE`, a character other than the test character did not occur. Here are examples for using `$CARRY`:

```
IF $CARRY THEN ... ;           !If test character not found
IF NOT $CARRY THEN ... ;      !If test character found
```

To determine the number of multibyte elements processed, divide (next address '-' byte address of the array) by the number of bytes per element, using unsigned arithmetic.

Scanning Bytes in Word-Aligned Arrays Operating system procedures require that procedure parameters use INT arrays, which are word aligned. To scan bytes in a word-aligned array, convert the word address of the array to a byte address by using the unsigned left-shift operation (`'<<' 1`).

The following example converts the word address of an INT array to a byte address. The assignment statement stores the resulting byte address into a STRING pointer. The `SCAN` statement then scans the bytes in the array until it finds a comma:

```
INT .words[-1:3] := [0,"Doe, J",0];
                                !Declare INT array (WORDS)

STRING .byte_ptr := @words[0] '<<' 1;
                                !Declare BYTE_PTR; initialize
                                ! with byte address of WORDS[0]

SCAN byte_ptr[0] UNTIL ","; !Scan bytes in WORDS
```

Multipart Scan Example These declarations apply to a series of scan statement examples that follow:

```
INT .int_array[-1:9] := [0,"  Smith, Maurice  ",0];
                                !INT_ARRAY

STRING .sptr := @int_array[0] '<<' 1;
                                !STRING pointer to INT_ARRAY[0]

STRING .start_last_name,      !
    .end_last_name,          !
    .start_first_name,       !STRING pointers for next address
    .end_first_name,         !
    .comma;                  !

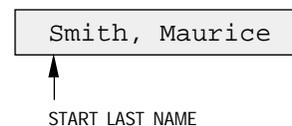
INT offset,                  !
    length;                  !INT variables
```

In the diagrams shown with the following examples, an arrow points to the character that stopped the scan.

Scanning WHILE

SCAN WHILE searches until it finds a byte character other than the test character or a zero. The following example scans left to right while spaces occur, starting from the zeroth element of INT_ARRAY. The scan stops at the beginning of the last name and stores that address in the next-address pointer START_LAST_NAME. An IF statement then checks the carry bit to see what stopped the scan. If a character (rather than a zero) stopped the scan, the program calls a string-handling procedure:

```
SCAN sptr[0] WHILE " " -> @start_last_name;
IF NOT $CARRY THEN
  CALL string_handler;
```

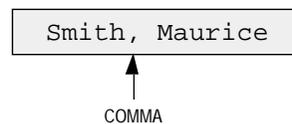


375

Scanning UNTIL

SCAN UNTIL searches until it finds the test character or a zero. The following example scans INT_ARRAY left to right until it finds a comma or a zero, starting from the address stored in START_LAST_NAME by the previous scan. If any character but a comma stops the scan, the program calls an error-printing procedure:

```
SCAN start_last_name UNTIL "," -> @comma;
IF $CARRY THEN
  CALL invalid_input;
```

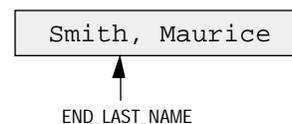


376

Scanning Right to Left

The following RSCAN example finds the end of the last name. It scans INT_ARRAY right to left for a character other than a space or a zero, starting from the location preceding the comma. Because no space separates the end of the last name from the comma, the scan starts and stops at the same location:

```
RSCAN comma[-1] WHILE " " -> @end_last_name;
```

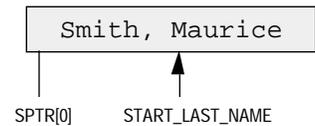


377

Computing the Offset of a Character

The following SCAN WHILE example finds the offset of the first name from the beginning of the array. It scans left to right from the address following the comma, looking for a character other than a space or a zero. It stops at the beginning of the first name and stores that address in the next-address pointer `START_FIRST_NAME`. The assignment statement then computes the offset and assigns it to `OFFSET`:

```
SCAN comma[1] WHILE " " -> @start_first_name;
offset := @start_first_name '-' @sptr;
```

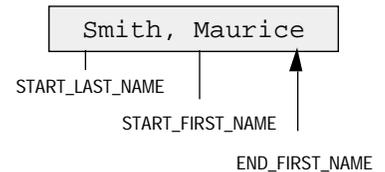


378

Computing the Length of a Character String

The following SCAN UNTIL example finds the length of the name contained in the array. It scans left to right from the address stored in `START_FIRST_NAME` by the preceding scan, looking for a space or a zero. It stores in `END_FIRST_NAME` the address where the space occurs. The assignment statement then computes the length of the entire name and assigns it to `LENGTH`:

```
SCAN start_first_name UNTIL " " -> @end_first_name;
length := @end_first_name '-' @start_last_name;
```



379

Comparing Arrays You can compare two arrays, or compare an array to a constant list, by using a group comparison expression in a statement. Group comparison expressions are described in Section 13, “Using Special Expressions.” Here are some examples.

You can compare an array to a constant list:

```
STRING an_array[0:3];                !Declare array

!Some code here
IF an_array[0] = ["ABCD"] THEN ... ; !Compare array to
! constant list
```

You can compare two arrays:

```
INT in_array[0:8];                   !Declare array
INT out_array[0:8];                  !Declare array

!Some code here
IF in_array = out_array FOR 9 ELEMENTS THEN ... ;
!Compare the arrays
```

You can use the next address in a group comparison expression:

```
STRING .sp;                          !Declare next-address
! pointer
STRING .a[0:1] := "AB";               !Declare array
STRING .b[0:1] := "AC";               !Declare array

IF b <> a FOR 2 BYTES -> @SP THEN ... ; !SP points to B[1]
```

Using Standard Functions With Arrays You can use the following standard functions to get certain information about arrays, such as the number of elements in an array:

Standard Function	Effect
\$BITLENGTH	Returns the length, in bits, of one array element
\$LEN	Returns the length, in bytes, of one array element (1 for STRING arrays, 2 for INT arrays, and so forth)
\$OCCURS	Returns the number of elements in an array
\$TYPE	Returns a value that denotes the data type of an array

For example, you can find the total length of an array as follows:

```
INT array_length;
INT array[0:2];

array_length := $LEN (array) * $OCCURS (array);
```

Using Read-Only Arrays

A read-only array is an array you cannot modify. When you declare a read-only array, the compiler allocates storage for the array in a user code segment.

Declaring Read-Only Arrays

Read-only array declarations differ from other array declarations in a number of ways:

- You must specify the read-only array symbol (= 'P') following the array identifier.
- You must initialize read-only arrays when you declare them, because you cannot assign values to read-only arrays by using assignment statements later in your program.
- You cannot declare read-only arrays of data type UNSIGNED because you cannot initialize UNSIGNED arrays.
- You cannot declare indirect read-only arrays, because code segments have no primary or secondary storage areas.
- You can omit the array bounds. The default lower bound is 0; the default upper bound is the number of elements initialized minus one.

For example, you can declare and initialize two read-only arrays, PROMPT and ERROR, using default bounds:

```
STRING prompt = 'P' := ["Enter a character: ", 0];
INT    error  = 'P' := ["Incorrect input"];
```

Numeric constants in the constant list should be appropriate for the data type of the array.

The system uses the program counter (P register) to access read-only arrays. The P register contains the address of the next instruction to be executed in the current code segment.

If you declare a read-only array in a procedure declared with the RESIDENT procedure attribute, the array is also resident in main memory. For more information on the RESIDENT attribute of procedures, see the *TAL Reference Manual*.

Accessing Read-Only Arrays

You can access read-only arrays in the same way as you access any other array, except that:

- You cannot modify a read-only array; that is, you cannot specify a read-only array on the left side of an assignment or move operator.
- You cannot specify a read-only array on the left side of a group comparison expression.
- In a SCAN or RSCAN statement, you cannot use *next-address* to read the last character of a string. You can use *next-address* to compute the length of the string.

A procedure can access any global read-only array located in the same 32K-word area of the code segment.

A procedure located in the upper 32K-word area of the code segment can access global STRING read-only arrays located in the lower 32K-word area only by using extended pointers. Here is an example:

```
PROC q (sp, len);
    STRING .EXT sp;
    INT len;
    BEGIN
        !Code to print sp[0:len - 1]
    END;

PROC p;
    BEGIN
        STRING s = 'P' := "Hello";
        STRING .EXT sp := $XADR(s);
        LITERAL LEN = 5;
        CALL q (sp, len);
    END;
```

A procedure can pass the data of a read-only array only by reference to a procedure located in the same code segment.

You can copy data from a read-only array into a user data segment array as follows:

```
STRING message = 'P' := ["** LOAD MAG TAPE #00144"];
STRING .array[0:22];

array ':= ' message FOR 23;
```

8 Using Structures

A structure is a collectively stored set of data items that you can access individually or as a group. Structures contain structure items (fields) such as simple variables, arrays, simple pointers, structure pointers, and nested structures (called substructures). The structure items can be of different data types.

Structures usually contain related data items such as the fields of a file record. For example, in an inventory control application, a structure can contain an item number, unit price, and quantity.

Structures can simulate multidimensional arrays, arrays of arrays, or arrays of structures.

This section describes:

- Kinds of structures—definition, template, and referral
- Declaring and allocating definition structures
- Declaring template structures
- Declaring and allocating referral structures
- Declaring and allocating structure items
- Accessing structure items
- Assigning data to structure items
- Copying structure data

Kinds of Structures A structure declaration associates an identifier with any of three kinds of structures. Table 8-1 lists the kinds of structures you can declare.

Table 8-1. Kinds of Structures

Structure	Description
Definition	Describes a structure layout and allocates storage for it
Template	Describes a structure layout but allocates no storage for it
Referral	Allocates storage for a structure whose layout is the same as the layout of a previously declared structure

Structure Layout The structure layout is a BEGIN-END construct that contains declarations of structure items. Table 8-2 lists structure items.

Table 8-2. Structure Items

Structure Item	Description
Simple variable	A single-element variable
Array	A variable that contains multiple elements of the same data type
Substructure	A structure nested within a structure (to a maximum of 64 levels)
Filler byte	A place-holding byte
Filler bit	A place-holding bit
Simple pointer	A variable that contains the memory address, usually of a simple variable or array, which you can access with this simple pointer
Structure pointer	A variable that contains the memory address of a structure, which you can access with this structure pointer
Redefinition	A new identifier and sometimes a new description for a substructure, simple variable, array, or pointer declared in the same structure

You can nest substructures within structures up to 64 levels deep. That is, you can declare a substructure within a substructure within a substructure, and so on, for up to 64 levels. The structure and each substructure has a BEGIN-END level depending on the level of nesting.

The following rules apply to all structure items:

- You can declare the same identifier in different structures and substructures, but you cannot repeat an identifier at the same BEGIN-END level.
- You cannot initialize a structure item when you declare it. After you have declared it, however, you can assign a value to it by using an assignment or move statement.

The following subsections describe how to declare definition, template, referral structures, and structure items.

Declaring Definition Structures

A definition structure describes a structure layout and allocates storage for it. To declare a single occurrence of a definition structure, specify:

- The keyword `STRUCT`
- The structure *identifier*, usually preceded by an indirection symbol (`.` or `.EXT`)
- A semicolon
- The structure *layout* (enclosed in a `BEGIN-END` construct)

You can, for example, declare a definition structure named `INVENTORY` like this:

```
STRUCT .inventory;           !Declare definition structure
  BEGIN                     !Begin structure layout
  INT item;
  FIXED(2) price;
  INT quantity;
  END;                       !End structure layout
```

Specifying Structure Occurrences

A definition structure that contains multiple occurrences is also known as an array of structures. For multiple occurrences, specify the lower and upper *bounds* in the structure declaration. These bounds represent the indexes of the first and last structure occurrences you want allocated. The bounds must be `INT` constant expressions in the range `-32,768` through `32,767`, separated by a colon and enclosed in brackets. The default bounds are `[0:0]` (one structure occurrence).

For example, to declare an array of definition structures that consists of four occurrences of the structure, specify structure bounds such as `[0:3]`:

```
STRUCT .inventory[0:3];     !Declare definition structure
  BEGIN                     !Begin structure layout
  INT item;
  FIXED(2) price;
  INT quantity;
  END;                       !End structure layout
```

The size of one occurrence of a structure must not exceed `32,767` bytes. In the preceding example, the size of each structure occurrence is `12` bytes. The size of the entire structure, including all four occurrences, is `48` bytes.

Using Indirection

You should use indirection for most global and local structures, because storage areas for direct global and local variables are limited. You access indirect structures by *identifier* as you do direct structures.

Do not use indirection for sublocal structures, because sublocal storage has no secondary area.

To declare a standard indirect structure, precede the structure identifier with the standard indirection symbol (.):

```
STRUCT .std_structure;           !Declare standard indirect
BEGIN                           ! definition structure
  INT a;                         ! (global or local scope)
  INT b;
END;
```

For very large structures, you should use extended indirection. When you declare one or more extended indirect structures (or arrays), the compiler allocates the automatic extended data segment. If you also must allocate an explicit extended data segment, follow the instructions given in Appendix B, “Managing Addressing.”

To declare an extended indirect structure, precede the structure identifier with the extended indirection symbol (.EXT):

```
STRUCT .EXT ext_structure;      !Declare extended indirect
BEGIN                           ! definition structure
  INT a;                         ! (global or local scope)
  INT b;
END;
```

Allocating Definition Structures

The compiler allocates storage for direct and indirect structures as shown for arrays in Figures 7-1 and 7-2 in Section 7, “Using Arrays.” That information is summarized here for structures.

At the global and local levels, you can declare direct or indirect structures. At the sublocal level, you can declare direct structures only.

Direct Structures

For each directly addressed structure, the compiler allocates space in the global, local, and sublocal primary areas of the user data segment.

Standard Indirect Structures

For each standard indirect structure, the compiler allocates space as follows:

1. It allocates a word of storage in the global (or local) primary area of the user data segment for an implicit standard structure pointer.
2. Next, it allocates storage for the structure in the global (or local) secondary area of the user data segment.
3. Finally, it initializes the implicit pointer (provided in Step 1) with the 16-bit word address of the zeroth structure occurrence.

Extended Indirect Structures

For each extended indirect structure, the compiler allocates space as follows:

1. It allocates two words of storage in the global (or local) primary area of the user data segment for the implicit extended structure pointer.
2. Next, it allocates storage for the structure in the automatic extended data segment.
3. Finally, it initializes the implicit pointer (provided in Step 1) with the 32-bit byte address of the zeroth structure occurrence.

Word Boundaries

The compiler starts the allocation of each structure occurrence on a word boundary, even if each occurrence contains an odd number of bytes.

For example, the compiler starts structure occurrences A[0] and A[1] on a word boundary as follows. (Slashes in the diagram represent compiler-allocated pad bytes.)



380

Amount of Allocation

The compiler determines the amount of storage to allocate for the structure data from the size and number of structure occurrences. The compiler allocates storage only for the structure occurrences specified in the bounds of the declaration.

For example, if you declare simple variables A and B, and then declare structure S with bounds of [1:2], the compiler allocates storage for A, B, S[1], and S[2] as follows. In this case, the zeroth occurrence of S is located at the address of simple variable A:



381

Note When your code later refers to a structure occurrence, the compiler does no bounds checking. If you refer to a structure occurrence outside the bounds specified in the declaration, you access data at an address outside of the structure.

Addressability of Structures The zeroth occurrence of a structure must be addressable. If it is not addressable, the compiler issues an address range violation error. (In the following diagrams, parentheses enclose the zeroth occurrence to indicate that it is not addressable.)

Addressability in the User Data Segment

The zeroth occurrence of a direct structure must fit within the lower 32K-word area of the user data segment, even if the zeroth occurrence is not allocated.

The global area has G-plus addressing. If a global structure is located at G[0], its lower bound must be a 0 or negative value. Avoid the following practice:



382

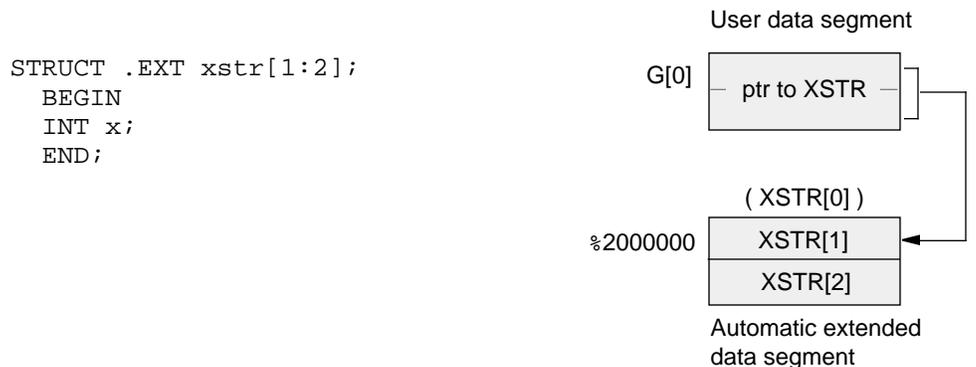
The sublocal area has S-minus addressing. If a sublocal structure is located at S[0], its upper bound must be a 0 or larger value. Avoid the following practice:



383

Addressability in the Extended Segment

The zeroth occurrence of an extended indirect structure must reside within the extended data segment, even if the zeroth occurrence is not allocated. If an extended indirect structure is located at the beginning of the extended data segment, the lower bound must be a 0 or negative value. Avoid the following practice:



384

Declaring Template Structures

(A template structure declares a structure layout but allocates no storage for it. You use the template in subsequent structure, substructure, or structure pointer declarations.)

To declare a template structure, specify:

- The keyword `STRUCT`
- The structure *identifier* (with no indirection symbol)
- An asterisk enclosed in parentheses
- A semicolon
- The structure *layout* (enclosed in a BEGIN-END construct)

For example, you can declare a template structure named `STOCK` like this:

```
STRUCT stock (*);           !Declare template structure
  BEGIN                     !Begin structure layout
    INT item;
    FIXED(2) price;
    INT quantity;
  END;                       !End structure layout
```

A template structure has meaning only when you refer to it in the subsequent declaration of a referral structure (described next), a referral substructure, or a structure pointer. The subsequent declaration allocates space for a structure whose layout is the same as the template layout.

Declaring Referral Structures

A referral structure allocates storage for a structure whose layout is the same as that of a specified structure or structure pointer.

To declare a single occurrence (copy) of a referral structure, specify:

- The keyword `STRUCT`
- The structure *identifier*, usually preceded by an indirection symbol (`.` or `.EXT`)
- A *referral* that provides the structure layout—enclose the identifier of an existing definition structure, template structure, or structure pointer in parentheses

For example, you can declare referral structure `NEW_STRUCT` to use the layout of `OLD_STRUCT`:

```
STRUCT .new_struct (old_struct); !Declare referral structure
```

Specifying Structure Occurrences

A referral structure that contains multiple occurrences is an array of structures. To indicate multiple occurrences, specify the lower and upper *bounds* in the structure declaration. These bounds represent the indexes of the first and last structure occurrences you want allocated. Specify the bounds as `INT` constant expressions in the range `-32,768` through `32,767`, separated by a colon and enclosed in brackets. The default bounds are `[0:0]` (one structure occurrence).

For example, to declare an array of referral structures that consists of 50 occurrences of the structure, specify structure bounds such as `[0:49]`. The following example declares:

- A template structure named `RECORD`
- A referral structure named `CUSTOMER` that uses the layout of `RECORD` for 50 occurrences:

```
STRUCT record (*); !Declare template structure
BEGIN
  STRING name[0:19];
  STRING addr[0:29];
  INT acct;
  END;

STRUCT .customer (record) [0:49];
!Declare referral structure
```

The size of one occurrence of a structure must not exceed 32,767 bytes. In the preceding example, the size of each structure occurrence is 52 bytes. The size of the entire structure, including all 50 occurrences, is 2,600 bytes.

Using Indirection

You should use indirection for most global and local structures, because storage areas for direct global and local variables are limited. You access indirect structures by identifier as you do direct structures.

Do not use indirection for sublocal structures, because sublocal storage has no secondary area.

To declare a standard indirect structure, precede the structure identifier with the standard indirection symbol (`.`), as shown in the preceding example.

For very large structures, you should use extended indirection. When you declare one or more extended indirect structures (or arrays), the compiler allocates the automatic extended data segment. If you also must allocate an explicit extended data segment, follow the instructions given in Appendix B, “Managing Addressing.”

To declare an extended indirect structure, precede the structure identifier with the extended indirection symbol (`.EXT`):

```
STRUCT record (*);           !Declare template structure
  BEGIN
  STRING name[0:19];
  STRING addr[0:29];
  INT acct;
  END;

STRUCT .EXT customer (record) [0:49];
                                !Declare extended indirect
                                ! referral structure
```

Allocating Referral Structures

The compiler allocates storage for each referral structure based on the following characteristics:

- The addressing mode and number of occurrences specified in the new declaration
- The layout of the previous declaration

In all other ways, allocation of referral structures is the same as for definition structures, as described earlier in this section.

Addressability of Structures

The zeroth occurrence of a structure must always be addressable, as described for definition structures.

Declaring Simple Variables and Arrays in Structures

You declare simple variables and arrays inside and outside a structure in the same way, except that inside a structure:

- You cannot initialize simple variables or arrays.
- You cannot declare indirect arrays or read-only arrays.
- You can specify bounds of `[n:n-1]`; the array is addressable but uses no memory.

For example, you can declare simple variables and arrays in a structure like this:

```
STRUCT .record;             !Declare definition structure
  BEGIN
  STRING name[0:19];        !Declare array
  STRING addr[0:29];        !Declare array
  INT acct;                 !Declare simple variable
  END;
```

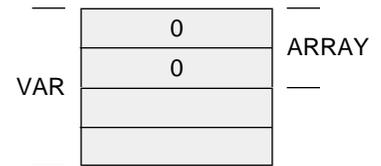
A structure that contains arrays is also known as an array of arrays.

Declaring Arrays That Use No Memory

If you declare within a structure an array that has bounds of $[n:n-1]$, the compiler places the identifier of the array in the symbol table but allocates no storage for the array. You can then apply the array's data type to subsequent items in the same structure.

For example, suppose you declare within a structure an INT(32) array that has bounds of $[0:-1]$, followed by a FIXED variable. If you then assign 0 to the first element of the INT(32) array, you set the two high-order words of the FIXED variable to 0:

```
STRUCT x;
  BEGIN
    INT(32) array[0:-1];
    FIXED(0) var;
  END;
x.array[0] := 0D;
```



385

Allocating Simple Variables and Arrays in Structures

The data type of simple variables and arrays declared within a structure determines the storage unit that the compiler allocates for the variable or array:

Data Type	Storage Unit
STRING	Byte
INT	Word
INT(32) or REAL	Doubleword
REAL(64) or FIXED	Quadrupleword
UNSIGNED	Bit sequence of specified width

Alignment of Simple Variables and Arrays

The compiler aligns and pads simple variables and arrays in structures as follows:

- STRING items are byte aligned.
- All other items are word aligned.
- If a word-aligned item follows a STRING item that ends on an odd byte, the compiler allocates a pad byte after the STRING item.

This example shows how the compiler allocates STRING simple variables (VAR1 and VAR2) and STRING arrays (A, B, and C) on byte boundaries. The compiler allocates a pad byte following array C because VAR4 must be word aligned.

```
STRUCT .padding;
BEGIN
  STRING var1;
  STRING var2;
  INT var3;
  STRING a[0:2];
  STRING b[0:1];
  STRING c[0:3];
  INT var4;
END;
```

VAR1	VAR2
VAR3	
A[0]	A[1]
A[2]	B[0]
B[1]	C[0]
C[1]	C[2]
C[3]	///
VAR4	

386

Allocating UNSIGNED Structure Items

The compiler packs the bits of UNSIGNED simple variables and arrays declared inside a structure in the same way as those declared outside a structure.

For example, the compiler allocates two bits for Y on a word boundary within structure Z and then then allocates four bits for X in the same word unit. The compiler also allocates ten pad bits following X because V must be word aligned.

```
STRUCT .EXT z;
BEGIN
  UNSIGNED(2) y;
  UNSIGNED(4) x;
  INT v;
END;
```

Y	X	///
V		

387

Declaring Substructures

A substructure is a structure embedded within another structure or substructure. In general, substructures have the following characteristics:

- They must be directly addressed.
- They have byte addresses, not word addresses.
- They can be nested to a maximum of 64 levels.
- They can have bounds of $[n: n-1]$. Such substructures are addressable but use no memory.

You can declare definition or referral substructures.

Declaring Definition Substructures

A definition substructure declares a layout and allocates storage for it. To declare a definition substructure, specify:

- The keyword `STRUCT`
- The substructure *identifier* (with no indirection symbol)
- Optional substructure *bounds*—the default bounds are `[0:0]` (one occurrence)
- A semicolon
- The substructure *layout* (the same BEGIN-END construct as for structures)

The substructure layout can contain declarations for simple variables, arrays, substructures, filler bits, filler bytes, redefinitions, simple pointers, and structure pointers. The size of one substructure occurrence is the size of the layout, either in odd or even bytes. The total layout for one occurrence of the encompassing structure must not exceed 32,767 bytes.

For example, within definition structure D, you can declare definition substructure DB. The length of DB is two bytes; the length of D is six bytes:

```
STRUCT .EXT d; !Declare structure D
BEGIN
STRING da;
STRUCT db; !Declare substructure DB
BEGIN
STRING db1;
STRING db2;
END; !End DB
!Implicit byte filler
INT dc;
END; !End D
```

DA	DB1
DB2	///
DC	

Declaring Multidimensional Arrays

You can nest substructures in a structure to simulate a multidimensional array.

The following structure simulates a two-dimensional array. The structure represents two warehouses. The two substructures represent 50 items and ten employees in each warehouse. The substructures are both nested at the second level but contain different kinds of records:

```

LITERAL last = 49;                !Declare number of last item
STRUCT .warehouse[0:1];          !Declare structure WAREHOUSE
  BEGIN
  STRUCT inventory[0:last];       !Declare substructure
    BEGIN                          ! INVENTORY
    INT item_number;
    INT price;
    INT on_hand;
    END;                            !End INVENTORY
  STRUCT employee[0:9];           !Declare substructure
    BEGIN                          ! EMPLOYEE
    STRING name[0:31];
    STRING telephone[0:6];
    END;                            !End EMPLOYEE
  END;                              !End WAREHOUSE

```

The following structure simulates a five-dimensional array. The structure represents a corporation that contains three branches. Each branch contains four divisions. Each division contains up to six departments. Each department contains up to six groups. Each group contains up to 20 employees.

```

STRUCT .corp;                    !Declare structure CORP
  BEGIN
  STRUCT branch[0:2];            !Declare substructure BRANCH
    BEGIN
    STRUCT div[0:3];             !Declare substructure DIV
      BEGIN
      STRUCT dept[0:5];          !Declare substructure DEPT
        BEGIN
        STRUCT group[0:5];       !Declare substructure GROUP
          BEGIN
          STRUCT employee[0:19]; !Declare substructure
            BEGIN                ! EMPLOYEE
            STRING name[0:31];
            STRING telephone[0:6];
            END;                  !End EMPLOYEE
          END;                    !End GROUP
        END;                      !End DEPT
      END;                        !End DIV
    END;                          !End BRANCH
  END;                            !End CORP

```

Allocating Definition Substructures

The compiler allocates storage for each substructure when it allocates storage for the encompassing structure. The compiler aligns a definition substructure on a byte or word boundary.

Byte Alignment. A definition substructure is byte aligned if the first item it contains begins on a byte boundary.

In the following example, definition substructure SUB follows a STRING item (X); the first item in SUB is also a STRING item (AA). Thus, each occurrence of SUB begins on a byte boundary. After the last occurrence of SUB, the compiler allocates a pad byte, because the next variable, Y, is an INT variable and must begin on a word boundary:

```
STRUCT struct_one;
BEGIN
STRING x;
STRUCT sub[0:2];    !Byte-aligned SUB
BEGIN
STRING aa;
INT b;
STRING c;
END;                !End SUB
INT y;
END;
```

X	AA
B	
C	AA
B	
C	AA
B	
C	///
Y	

389

Word Alignment. A definition substructure is word aligned if the first item it contains is word aligned, including arrays that have bounds of [n:n-1]. If a word-aligned substructure has more than one occurrence and contains an odd number of bytes, the compiler allocates a pad byte after each occurrence.

In the following example, definition substructure SUB starts with INT item A_A. The compiler starts each occurrence of SUB on a word boundary, allocating a pad byte in each unused byte:

```
STRUCT struct_two;
BEGIN
STRING x;
STRUCT sub [0:1]; !Word-aligned SUB
BEGIN
INT a_a;
INT b;
STRING c;
END;                !End SUB
END;
```

—	X	///
	A_A	
SUB[0]	B	
—	C	///
	A_A	
SUB[1]	B	
—	C	///

390

Declaring Referral Substructures A referral substructure uses the layout of a previously declared structure. To declare a referral substructure, specify:

- The keyword `STRUCT`
- The substructure *identifier* (with no indirection symbol)
- A *referral* that provides a layout—enclose in parentheses the identifier of an existing structure (except the encompassing structure) or structure pointer
- Optional substructure *bounds*—the default bounds are `[0:0]` (one occurrence)

In the following example, referral substructure `REF_SUB` uses the body of structure `STRUCT_TWO` from the preceding example:

```
STRUCT .EXT struct_three;
BEGIN
  INT a;
  STRUCT ref_sub (struct_two) [0:2];    !Declare REF_SUB
END;
```

Allocating Referral Substructures

The compiler allocates storage for a referral substructure when it allocates storage for the encompassing structure. The compiler allocates the storage based on:

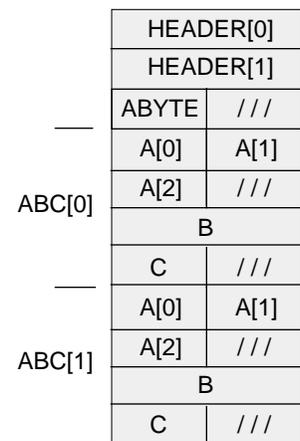
- The addressing mode and number of occurrences specified in the new declaration
- The layout of the previous declaration

A referral substructure always begins on a word boundary. If the substructure contains an odd number of bytes, the compiler appends a pad byte to each occurrence of the substructure.

The following example shows how the compiler aligns referral substructure `ABC` on a word boundary, allocating a pad byte in each unused byte:

```
STRUCT template (*);
BEGIN
  STRING a[0:2];
  INT b;
  STRING c;
END;

STRUCT .indirect_structure;
BEGIN
  INT header[0:1];
  STRING abyte;
  STRUCT abc (template) [0:1];
  !Referral substructure
END;
```



Declaring Fillers You can declare filler bytes or bits to allocate place holder space within a structure. You cannot access filler locations.

You can use filler items to allocate space within a structure when the structure layout must match a structure layout defined by another program. The new structure declaration need only include data items used by your program. You can use filler items for the unused data.

You can use filler declarations to produce clearer source code. For example, you can:

- Document pad bytes or bits that would otherwise be inserted by the compiler
- Provide place holders for unused space in the structure

The compiler allocates space for each byte or bit you specify in a filler declaration. If the alignment of the next data item requires additional pad bytes or bits, the compiler allocates those also.

Declaring Filler Bytes You declare filler bytes within a structure by specifying the *number* of filler bytes as a constant expression in the range 0 through 32,767 as follows:

```
FILLER 5;
```

You can use filler byte declarations to document unused bytes in a structure:

```
LITERAL last = 11;           !Last occurrence
STRUCT .x[1:last];
BEGIN
  STRING byte[0:2];
  FILLER 1;                   !Document word-alignment pad byte
  INT word1;
  INT word2;
  INT(32) integer32;
  FILLER 30;                  !Place holder for unused space
END;
```

Declaring Filler Bits You declare filler bits within a structure by specifying the *number* of filler bits as a constant expression in the range 0 through 255 as follows:

```
BIT_FILLER 5;
```

You can use filler bit declarations to document unused bits in a structure:

```
STRUCT .flags;
BEGIN
  UNSIGNED(1) flag1;
  UNSIGNED(1) flag2;
  UNSIGNED(2) state;         !State = 0, 1, 2, or 3
  BIT_FILLER 12;
END;
```

Declaring Simple Pointers in Structures

You can declare simple pointers within a structure. A simple pointer is a variable in which you store the memory address, usually of a simple variable or array, which you can access with this pointer. The compiler allocates space for the pointer but not for the data to which the pointer points.

Simple pointers can be standard or extended:

- Standard (16-bit) pointers can access addresses in the current user data segment.
- Extended (32-bit) pointers can access addresses in any segment, normally the automatic extended data segment.

To declare a simple pointer inside a structure, specify:

- Any *data type* except UNSIGNED
- The simple pointer *identifier*, preceded by an indirection symbol (. or .EXT)

For example, you can declare STD_POINTER and EXT_POINTER inside MY_STRUCT:

```
STRUCT my_struct;
  BEGIN
    FIXED .std_pointer;           !Standard simple pointer
    STRING .EXT ext_pointer;     !Extended simple pointer
  END;
```

The data type determines how much data a simple pointer can access at a time, as listed in Table 8-3.

Table 8-3. Data Accessed by Simple Pointers

Data Type	Accessed Data
STRING	Byte
INT	Word
INT(32)	Doubleword
REAL	Doubleword
REAL(64)	Quadrupleword
FIXED	Quadrupleword

Addresses Simple Pointers Can Contain The addressing mode and data type of a simple pointer determines the kind of address the pointer can contain, as described in Table 8-4.

Table 8-4. Addresses in Simple Pointers

Addressing Mode	Data Type	Kind of Addresses
Standard	STRING	16-bit byte address in the lower 32K-word area of the user data segment.
Standard	Any except STRING	16-bit word address anywhere in the user data segment.
Extended	STRING	32-bit byte address, normally in the automatic extended data segment.
Extended	Any except STRING	32-bit even-byte address, normally in the automatic extended data segment. (If you specify an odd-byte address, the results are undefined.)

Allocating Simple Pointers in Structures The compiler allocates storage for simple pointers declared within structures when it allocates the encompassing structure. It allocates one word for each standard pointer and one doubleword for each extended pointer. The storage area depends on the scope and addressing mode of the encompassing structure.

The compiler does not allocate space for the data to which the pointer points. You can store addresses of previously declared items in pointers as described in “Assigning Addresses to Pointers in Structures” later in this section. Otherwise, you must manage the allocation of the data to which the pointer points, as described in Appendix B, “Managing Addressing.”

Declaring Structure Pointers in Structures

You can declare structure pointers within a structure or substructure. A structure pointer is a variable in which you store the memory address of a structure, which you can access with this structure pointer. The compiler allocates space for the pointer but not for the data to which the pointer points.

Structure pointers can be standard or extended:

- Standard (16-bit) pointers can access addresses in the current user data segment.
- Extended (32-bit) pointers can access addresses in any segment, normally the automatic extended data segment.

To declare a structure pointer, specify:

- `STRING` or `INT` attribute as described in Table 8-5
- The structure pointer *identifier*, preceded by an indirection symbol (`.` or `.EXT`)
- A *referral* that provides the structure layout—enclose in parentheses the identifier of an existing structure or structure pointer or of the encompassing structure

The following example declares `STRUCT_A` and `STRUCT_B`. `STRUCT_B` contains a declaration for `STRUCT_PTR`, whose layout is the same as the layout of `STRUCT_A`:

```
STRUCT struct_a;                !Declare STRUCT_A
  BEGIN
  INT a;
  INT b;
  END;

STRUCT struct_b;                !Declare STRUCT_B
  BEGIN
  INT .EXT struct_ptr (struct_a); !Declare STRUCT_PTR
  STRING a;
  END;
```

Addresses Structure Pointers Can Contain The addressing mode and `STRING` or `INT` attribute of a structure pointer determine the kind of addresses the pointer can contain, as described in Table 8-5.

Table 8-5. Addresses in Structure Pointers

Addressing Mode	STRING or INT Attribute	Kind of Addresses
Standard	STRING *	16-bit byte address of a substructure, <code>STRING</code> simple variable, or <code>STRING</code> array declared in a structure located in the lower 32K-word area of the user data segment
Standard	INT **	16-bit word address of any structure data item located anywhere in the user data segment
Extended	STRING *	32-bit byte address of any structure data item located in any segment, normally the automatic extended data segment
Extended	INT **	32-bit byte address of any structure data item located in any segment, normally the automatic extended data segment

* If the pointer is the *source* in a move statement or group comparison expression that omits a *count-unit*, the *count-unit* is BYTES.

** If the pointer is the *source* in a move statement or group comparison expression that omits a *count-unit*, the *count-unit* is WORDS.

Allocating Structure Pointers in Structures The compiler allocates storage for structure pointers declared within structures when it allocates the encompassing structure. It allocates one word for each standard pointer and one doubleword for each extended pointer. The storage area depends on the scope and addressing mode of the encompassing structure.

The compiler does not allocate space for the data to which the pointer points. You can store addresses of previously declared items in pointers as described in “Assigning Addresses to Pointers in Structures” later in this section. Otherwise, you must manage the allocation of the data to which the pointer points, as described in Appendix B, “Managing Addressing.”

Declaring Redefinitions

A redefinition declares a new identifier and sometimes a new description for a previously declared item in the same structure. The new item or the previous item can be a simple variable, array, pointer, or substructure.

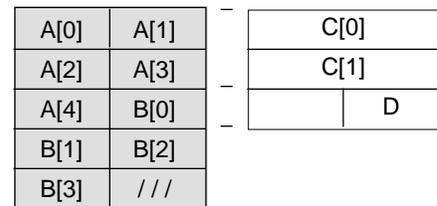
The new item must be at the same BEGIN-END level in a structure as the previous item. The new item must also be of the same length or shorter than the previous item.

The subsections that follow describe how you declare and access redefinitions. Some examples include diagrams that show how a redefinition relates to the previous item. In each diagram:

- The shaded box represents the previous item (the allocated item).
- The unshaded box represents the new item (the redefinition).

Unless otherwise noted, the diagrams refer to memory locations in the primary area of the user data segment. Here is an example diagram:

```
STRUCT array_redefinition;
BEGIN
  STRING a[0:4];
  STRING b[0:3];
  INT c[0:1] = a;
  STRING d = b;
END;
```



393

Simple Variables or Arrays as Redefinitions

To declare a new simple variable or array that redefines a previous item within the same structure, specify:

- Any *data type* (except UNSIGNED)
- The *identifier* of the new simple variable or array
- For an array, its *bounds*—if you omit the bounds, the default bounds are [0:0] (one element)
- An equal sign (=)
- The identifier of a *previous item* at the same BEGIN-END level of a structure—the previous item can be a simple variable, array, pointer, or substructure

For example, you can declare NEW_VAR to redefine OLD_VAR as follows:

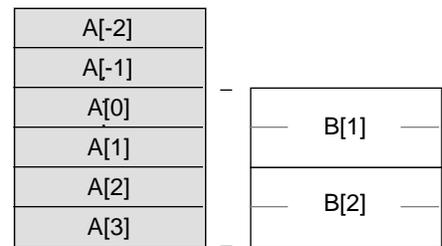
```
STRUCT simple_variable_redefinition;
BEGIN
  INT old_var;
  STRING new_var = old_var;           !Redefinition
END;
```

You can declare DW_ADDR to redefine DW_MEM as follows:

```
STRUCT dw_template (*);
BEGIN
  INT(32) .dw_mem;
  INT dw_addr = dw_mem;           !Redefinition
END;
```

Even if the lower bound of the new array is not zero, the new specified lower bound is always associated with the zeroth element of the previous array. In the following example, new INT(32) array B[1:2] redefines previous INT array A[0:3]:

```
STRUCT .array_redefinition;
BEGIN
  INT a[-2:3];
  INT(32) b[1:2] = a;
END;
```



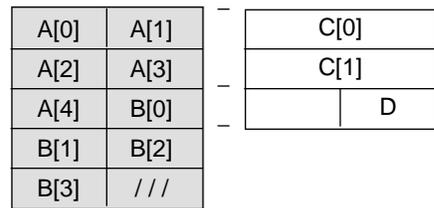
392

Data Type Restrictions

The new item can be any data type except UNSIGNED.

You can redefine the data type of a STRING array only if the array is aligned on a word boundary. For example, you can redefine the data type of STRING array A, because A is aligned on a word boundary. You cannot redefine the data type of STRING array B because B is aligned on a byte boundary, but you can declare another STRING item (such as D) to redefine B:

```
STRUCT array_redefinition;
BEGIN
  STRING a[0:4];
  STRING b[0:3];
  INT c[0:1] = a;
  STRING d = b;
END;
```



393

Byte and Word Addressing

In a redefinition, the new variable and the previous (nonpointer) variable both must have a byte address or both must have a word address. If the previous variable is a pointer, the data it points to must be word or byte addressed to match the new variable.

- Definition Substructures as Redefinitions** To declare a definition substructure that redefines a previously declared item within the same structure, specify:
- The keyword `STRUCT`
 - The *identifier* of the new substructure
 - Optional *bounds*—if you omit the bounds, the default bounds are `[0:0]` (one occurrence)
 - An equal sign (`=`)
 - The identifier of a *previous item* at the same BEGIN-END level of the encompassing structure—the previous item can be a simple variable, array, pointer, or substructure
 - A semicolon
 - The substructure *layout* (the same BEGIN-END construct as for structures)

If the previous item is a substructure and you omit the bounds or if either bound is 0, the new substructure and the previous substructure occupy the same space and have the same offset from the beginning of the structure.

For example, you can declare new substructure `INITIALS` to redefine substructure `WHOLE_NAME` as follows:

```
STRUCT .name_record;  
  BEGIN  
    STRUCT whole_name;    !Declare WHOLE_NAME  
      BEGIN  
        STRING first_name[0:10];  
        STRING middle_name[0:10];  
        STRING last_name[0:15];  
      END;  
    STRUCT initials = whole_name;  
      BEGIN                !Redefine WHOLE_NAME as INITIALS  
        STRING first_initial;  
        FILLER 10;  
        STRING middle_initial;  
        FILLER 10;  
        STRING last_initial;  
        FILLER 15;  
      END;  
    END;  
  END;
```

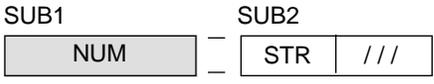
You can declare new substructure SS to redefine array A and then declare simple variable V to redefine substructure SS as follows:

```
STRUCT .st;
BEGIN
  INT a[0:1];           !Declare array A
  STRUCT ss = a;       !Redefine A as substructure SS
  BEGIN
    INT x;
    INT y;
  END;
  INT(32) v = ss;      !Redefine SS as V
END;
```

Size of Substructure Redefinitions

The new substructure must be of the same size or smaller than the previous item:

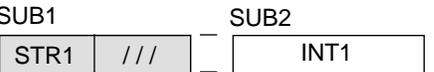
```
STRUCT str;
BEGIN
  STRUCT sub1;         !Declare SUB1
  BEGIN
    INT num;
  END;
  STRUCT sub2 = sub1; !Redefine SUB1 as SUB2
  BEGIN
    STRING str;
  END;
END;
```



394

If the new substructure is larger than the previous item, the compiler issues a warning:

```
STRUCT str2;
BEGIN
  STRUCT sub1;         !Declare SUB1
  BEGIN
    STRING str1;
  END;
  STRUCT sub2 = sub1; !Redefine SUB1 as SUB2, which is
  BEGIN               ! larger; compiler issues warning
    INT int1;
  END;
END;
```



395

Alignment of Substructure Redefinitions

The new substructure must have the same byte or word alignment as the previous substructure. That is, if the previous substructure starts on an odd byte, the first item in the new substructure must be a STRING item.

The following substructures (B and C) both begin on an odd-byte boundary:

```

STRUCT a;
  BEGIN
    STRING x;
    STRUCT b;          !B starts on odd byte
      BEGIN
        STRING y;
      END;
    STRUCT c = b;      !Redefine B as C, also on odd byte
      BEGIN
        STRING z;
      END;
    END;

```



396

Referral Substructures as Redefinitions

To declare a referral substructure that redefines a previously declared item within the same structure, specify:

- The keyword `STRUCT`
- The *identifier* of the new substructure
- A *referral* that provides a layout—enclose in parentheses the identifier of an existing structure (except the encompassing structure) or structure pointer
- Optional *bounds*—if you omit the bounds, the default bounds are [0:0] (one occurrence)
- An equal sign (=)
- The identifier of a *previous item* at the same BEGIN-END level of the structure—the previous item can be a simple variable, array, pointer, or substructure
- A semicolon

If the previous item is a substructure and you omit the bounds or if either bound is 0, the new substructure and the previous substructure occupy the same space and have the same offset from the beginning of the structure.

For example, you can declare a referral substructure redefinition that uses a template structure layout:

```

STRUCT temp(*);                !Declare template structure
  BEGIN
  STRING a[0:2];
  INT    b;
  STRING c;
  END;

STRUCT .ind_struct;           !Declare definition structure
  BEGIN
  INT    header[0:1];
  STRING abyte;
  STRUCT abc(temp) [0:1];     !Declare ABC
  STRUCT xyz(temp) [0:1] = abc;
                                !Redefine ABC as XYZ
  END;

```

Simple Pointers as Redefinitions

To declare a simple pointer that redefines a previously declared item within the same structure, specify:

- Any *data type* except UNSIGNED
- The *identifier* of the new pointer, preceded by an indirection symbol (. or .EXT)
- An equal sign (=)
- The identifier of a *previous item* at the same BEGIN-END level of the structure—a simple variable, array, pointer, or substructure

For example, you can declare new simple pointer EXT_POINTER to redefine simple variable VAR as follows:

```

STRUCT my_struct;
  BEGIN
  STRING var[0:5];
  STRING .EXT ext_pointer = var;           !Redefinition
  END;

```

Structure Pointers as Redefinitions

To declare a structure pointer that redefines a previously declared item within the same structure, specify:

- STRING or INT attribute, as described in Table 8-5 earlier in this section
- The *identifier* of the new pointer, preceded by an indirection symbol (. or .EXT)
- A *referral* that provides the structure layout—enclose in parentheses the identifier of an existing structure or structure pointer or of the encompassing structure
- An equal sign (=)
- The identifier of a *previous item* at the same BEGIN-END level of the structure—a simple variable, array, pointer, or substructure

For example, you can declare new standard and extended structure pointers to redefine simple variables as follows:

```
STRUCT record;
BEGIN
  FIXED(0) data;
  INT std_link_addr;
  INT .std_link(record) = std_link_addr;      !Redefinition
  INT ext_link_addr;
  INT .EXT ext_link(record) = ext_link_addr; !Redefinition
END;
```

Accessing Structure Items

You access an item in a definition or referral structure by using the item's fully qualified identifier in a statement. Structure items you can access are simple variables, arrays, substructures, pointers, and redefinitions. For example, you can:

- Assign a value to a structure item by using an assignment statement
- Copy substructure values within a structure by using a move statement

Qualifying Identifiers

To access a structure item, you must specify its fully qualified identifier and indexes if any. A fully qualified identifier includes all levels of nesting. For example, the fully qualified identifier for ARRAY_Z declared in SUBSTRUCT_Y in STRUCT_X is:

```
struct_x.substruct_y[0].array_z[0]
```

The following example shows how nesting affects qualified identifiers. The two structures both contain three substructures. In both cases, SUB_3 contains ITEM_X:

```
STRUCT .struct_a;          STRUCT .struct_b;
BEGIN                      BEGIN
  STRUCT sub_1;           STRUCT sub_1;
  BEGIN                  BEGIN
    INT a;                STRUCT sub_2;
    END; !End SUB_1       BEGIN
  STRUCT sub_2;           STRUCT sub_3;
  BEGIN                  BEGIN
    INT b;                INT a;
    END; !End SUB_2       INT b;
  STRUCT sub_3;           INT item_x;
  BEGIN                  END; !End SUB_3
    INT item_x;           END; !End SUB_2
  END; !End SUB_3         END; !End SUB_1
  END; !End STRUCT_A     END; !End STRUCT_A
```

For STRUCT_A, the fully qualified identifier for ITEM_X is:

```
struct_a.sub_3.item_x
```

For STRUCT_B, the fully qualified identifier for ITEM_X is:

```
struct_b.sub_1.sub_2.sub_3.item_x
```

You can use a DEFINE declaration to associate alternate names for structure items, as described in Section 5, "LITERALS and DEFINES," in the *TAL Reference Manual*.

Indexing Structures You can access a particular structure item by appending *indexes* (enclosed in brackets) to the various levels in the qualified identifier of the structure item. For example, you can access an array element in a particular structure and substructure occurrence by appending indexes as follows:

```
my_struct[1].my_substruct[0].my_array[4]
```

Each index represents an offset from the zeroth occurrence of a structure or substructure and the zeroth element of the array, respectively, regardless of the declared lower bounds.

The indexed item determines the size of the index offset, as listed in Table 8-6:

Table 8-6. Indexing Structures

Indexed Item	Data Type	Size of Index Offset
Structure or structure pointer	Not applicable	Total bytes in one structure occurrence
Substructure	Not applicable	Total bytes in one substructure occurrence
Simple variable or array	STRING	Byte
Simple variable or array	INT	Word
Simple variable or array	INT(32) or REAL	Doubleword
Simple variable or array	REAL(64) or FIXED	Quadrupleword

Indexing Standard Indirect Structures

The index for standard indirect structures must be a signed INT arithmetic expression in the range $-32,768$ through $32,767$. The offset of a structure item is from the zeroth structure occurrence (not the current structure occurrence). Here are examples of signed INT arithmetic expressions:

```
25
index
index + 2
index - 1
9 * index
```

The following example shows how you can index a standard indirect structure:

```
STRUCT .std_struct[0:99]; !Standard indirect structure
BEGIN
  INT var;                !Simple variable
  STRING array[0:25];     !Array
  STRUCT substr[0:9];     !Substructure
  BEGIN
    STRING array[0:25];   !Array
  END;
END;

INT index := 5;          !Simple variable

PROC x MAIN;            !Declare procedure X
BEGIN
  std_struct[index].var := 35;
                          !Access STD_STRUCTURE[5].VAR

  std_struct[index+2].array[0] := "A";
                          !Access STD_STRUCTURE[7].ARRAY[0]

  std_struct[index+2].array[25] := "Z";
                          !Access STD_STRUCTURE[7].ARRAY[25]

  std_struct[9*index].substr[index-1].array[index-5] := "a";
                          !Access
                          ! STD_STRUCTURE[45].SUBSTR[4].ARRAY[0]
END;                    !End procedure X
```

Indexing Extended Indirect Structures

The index for extended indirect structures must be a signed INT or INT(32) arithmetic expression, depending on the size of the offset of the structure item you want to access. The offset of a structure item is from the zeroth structure occurrence (not the current structure occurrence).

C-Series System. If you are writing a program to run on a C-series system, you can determine whether to use an INT index or an INT(32) index (which is slower) as follows:

1. Compute the lower and upper byte or word offsets of the structure item whose extended indirect structure is being indexed. (A byte-addressed structure item is at a byte offset. A word-addressed structure item is at a word offset.)
2. If the offsets are inside the signed INT range (–32,768 through 32,767), use a signed INT index. Usually, offsets are within the signed INT range.
3. If the offsets are outside the signed INT range, use an INT(32) index. To convert an INT index to an INT(32) index, use \$DBL, a standard function.

Whenever you increase the structure size or number of occurrences, you must repeat the preceding sequence.

To access a structure item whose offset is inside the signed INT range, you can use an INT index as follows:

```

STRUCT .EXT xstruct[0:9];           !Declare extended indirect
  BEGIN                             ! structure; upper byte offset
  STRING array[0:9];                ! is within INT range
  END;

INT index;                           !Declare INT index
STRING var;

PROC my_proc MAIN;
  BEGIN
  !Code to initialize INDEX
  var := xstruct[index].array[0];
  END;                                !Generate correct offset

```

In the preceding example, if the default NOINHIBITXX directive is in effect, the compiler generates efficient addressing (by using XX instructions described in the *System Description Manual* for your system).

Conversely, INHIBITXX suppresses efficient addressing of extended indirect declarations located between G[0] and G[63] of the user data segment—except when the extended indirect declarations are declared in a BLOCK declaration with the AT (0) or BELOW (64) option. (The BLOCK declaration is described in Section 14, “Compiling Programs.”)

To access a structure item whose offset is outside the signed INT range (-32678 through 32,767), you must use an INT(32) index. To convert an INT index to an INT(32) index, you can use the \$DBL standard function:

```

STRUCT .EXT xstruct[0:9999];
BEGIN
  STRING array[0:9];           !Upper byte offset > 32,767;
  END;                          ! INT(32) index required

INT index;

PROC my_proc MAIN;
BEGIN
  !Some code here to initialize INDEX

  xstruct[$DBL(index)].array[0] := 1;
                                     !Generate correct offset
  END;                               ! because INDEX is an INT(32)
                                     ! expression

```

In the preceding example, the upper-byte offset of ARRAY is larger than 32,767, computed as follows:

$$9999 * 10 + 9 = 99999$$

↑ ↑ ↑ ↑
 Size of offset to ARRAY[9]
 Upper bound of array
 Size of structure in bytes
 Upper bound of structure

334

D-Series System. If you are writing a program to run on a D-series system and need to index into extended indirect structures, you can either:

- Determine when to use a signed INT or INT(32) index as described for C-series programs in the preceding subsection.
- Use the INT32INDEX directive, which is easier and safer, albeit slightly less efficient.

INT32INDEX generates INT(32) indexes from INT indexes and computes the correct offset for the indexed structure item. INT32INDEX overrides the INHIBITXX or NOINHIBITXX directive, whichever is in effect.

NOINT32INDEX, the default, generates incorrect offsets for structure items whose offsets are outside the signed INT range. NOINT32INDEX does not override INHIBITXX or NOINHIBITXX.

Specify INT32INDEX or NOINT32INDEX immediately before the declarations to which it applies. The specified directive then applies to those declarations throughout the compilation. The following D-series example shows how INT32INDEX generates correct offsets and NOINT32INDEX generates incorrect offsets:

```
?INT32INDEX                                !Assign INT32INDEX attribute
                                           ! to subsequent declaration

STRUCT .EXT xstruct[0:9999];
  BEGIN                                    !XSTRUCT has INT32INDEX
  STRING array[0:9];                      ! attribute
  END;

INT index;

PROC my_proc MAIN;
  BEGIN
  ?NOINT32INDEX                            !Assign NOINT32INDEX attribute to
                                           ! subsequent declaration

  STRUCT .EXT xstruct2 (xstruct) := @xstruct[0];
                                           !XSTRUCT2 has NOINT32INDEX
                                           ! attribute

  xstruct[index].array[0]:= 1;
                                           !Generate correct offset even
                                           ! when offset is greater than
                                           ! 32,767, because XSTRUCT
                                           ! has INT32INDEX attribute

  xstruct2[index].array[0] := 1;
                                           !Generate incorrect offset if
                                           ! offset is greater than 32,767,
                                           ! because XSTRUCT2 has
                                           ! NOINT32INDEX attribute

  END;
```

Standard Addressing Contrasted With Extended Indexing

The allowed offset of a standard indirect structure item is greater than that of an extended indirect structure item. The following example declares a standard indirect structure item accessed by an index that is greater than that allowed for an extended indirect structure item:

```
LITERAL ub = 32759;

STRUCT .t[0:ub];           !Standard indirect structure
BEGIN
  STRING x, y;
END;

PROC m MAIN;
BEGIN
  INT index;
  FOR index := 0 TO ub DO
    t[index].x := t[index].y := 0;
  END;
```

If you change the preceding standard indirect structure to an extended indirect structure, you must also change the index that is applied to the structure item to an INT(32) index:

```
LITERAL ub = 32759;

STRUCT .EXT t[0:ub];      !Extended indirect structure
BEGIN
  STRING x, y;
END;

PROC m MAIN;
BEGIN
  INT index;
  FOR index := 0 TO ub DO
    BEGIN
      t[index].x := t[index].y := 0;
      !Compiler generates incorrect code

      t[$DBL(index)].x := t[$DBL(index)].y := 0;
      !Compiler generates correct code
    END;
  END;
```

Assigning Values to Structure Items You assign a value to a structure item by using its fully qualified identifier in an assignment statement. For example, the assignment statement for assigning an expression to simple variable VAR declared in SUBSTRUCT_A in STRUCT_B is:

```
struct_b.substruct_a.var := any_expression;
```

Here are examples. You can assign a value to VAR3 in DEF_STRUCT:

```
STRUCT .def_struct;                !Declare definition structure
BEGIN
  FIXED var1;
  STRING var2;
  INT var3;
END;

PROC a MAIN;
BEGIN
  def_struct.var3 := 45;           !Assign 45 to DEF_STRUCT.VAR3
END;
```

You can assign a value to BEAN[2] in REF_STRUCT:

```
STRUCT template_struct (*);       !Declare template structure
BEGIN
  REAL deal;
  STRING bean[0:2];
END;

STRUCT .ref_struct (template_struct);
                                   !Declare referral structure

PROC b MAIN;
BEGIN
  ref_struct.bean[2] := 92;       !Assign 92 to
                                   ! REF_STRUCT.BEAN[2]
END;
```

You can assign a value to ARRAY[5] in SUBST[3] in STRUCT:

```
STRUCT .struc;                    !Declare definition structure
BEGIN
  INT foo;
  STRUCT subst[0:99];
  BEGIN
    REAL var;
    INT array[0:9];
  END;
END;

PROC c MAIN;
BEGIN
  struc.subst[3].array[5] := 8;  !Assign 8 to
                                   ! STRUC.SUBST[3].ARRAY[5]
END;
```

Assigning Addresses to Pointers in Structures

You can assign to pointers the kinds of addresses listed in Tables 8-4 and 8-5 earlier in this section. To assign an address to a pointer within a structure, specify the fully qualified pointer identifier in an assignment statement. Prefix the structure identifier with @. For example, the assignment statement to assign an address to PTR_X declared in SUBSTRUCT_A in STRUCT_B is:

```
@struct_b.substruct_a.ptr_x := arith_expression;
```

In the preceding example, @ applies to PTR_X, the most qualified item. On the left side of the assignment operator, @ changes the address contained in the pointer, not the value of the item to which the pointer points.

You can also prefix @ to a variable on the right side of the assignment operator. If the variable is a pointer, @ returns the address contained in the pointer. If the variable is not a pointer, @ returns the address of the variable itself.

The following example shows @ used on both sides of the assignment operator. This example assigns the address of ARRAY to STD_PTR within a structure. Also, the \$XADR function converts the standard address of ARRAY to an extended address, which is then assigned to an extended simple pointer:

```
INT .array[0:99];           !Declare ARRAY

STRUCT .st;                !Declare ST
BEGIN
  INT .std_ptr;            !Declare STD_PTR
  INT .EXT ext_ptr;       !Declare EXT_PTR
END;

PROC e MAIN;
BEGIN
  @st.std_ptr := @array[0]; !Assign standard address of
                          ! ARRAY[0] to ST.STD_PTR

  @st.ext_ptr := $XADR(array[0]);
                          !Assign extended address of
                          ! ARRAY[0] to ST.EXT_PTR

END;
```

The following example assigns the address of a structure to structure pointers declared in another structure:

```

STRUCT .s1;                                !Declare S1
BEGIN
  INT var1;
  INT var2;
END;

STRUCT .s2;                                !Declare S2
BEGIN
  INT .std_ptr (s1);                       !Declare STD_PTR
  INT .EXT ext_ptr (s1);                   !Declare EXT_PTR
END;

PROC g MAIN;
BEGIN
  @s2.std_ptr := @s1[0];                   !Assign standard address
                                              ! of S1 to S2.STD_PTR

  @s2.ext_ptr := $XADR(s1);                !Assign extended address
                                              ! of S1 to S2.EXT_PTR
END;

```

Accessing Data Through Pointers in Structures

After you declare a pointer inside a structure and assign an address to it, you can use assignment statements to access the data to which the pointer points:

```

INT .array[0:99];                           !Declare ARRAY
STRUCT .st;                                  !Declare ST
BEGIN
  INT .std_ptr;                              !Declare STD_PTR
  INT .EXT ext_ptr;                          !Declare EXT_PTR
END;

PROC h MAIN;
BEGIN
  @st.std_ptr := @array[0];                  !Assign word address of
                                              ! ARRAY[0] to St.STD_PTR

  @st.ext_ptr := $XADR(array[1]);           !Assign extended address of
                                              ! ARRAY[1] to ST.EXT_PTR

  array[2] := st.std_ptr;                   !Assign content of
                                              ! ARRAY[0] to ARRAY[2]

  st.ext_ptr := array[3];                   !Assign content of ARRAY[3]
                                              ! to ARRAY[1]
END;

```

The last two statements in the following example access S3 to which the structure pointers declared in S2 point:

```
INT .a[0:99];                !Declare array A
STRUCT .s3;                  !Declare S3
  BEGIN
    INT b[0:99];            !Declare array B
  END;
STRUCT .s2;                  !Declare S2
  BEGIN
    INT .std_ptr (s3);      !Declare STD_PTR
    INT .EXT ext_ptr (s3); !Declare EXT_PTR
  END;

PROC i MAIN;
  BEGIN
    @s2.std_ptr := @s3;     !Assign standard address
                             ! of S3 to S2.STD_PTR

    @s2.ext_ptr := $XADR(s3); !Assign extended address
                             ! of S3 to S2.EXT_PTR

    a[1] := s2.std_ptr.b[1]; !Assign content of
                             ! S3.B[1] to A[1]

    s2.ext_ptr.b[2] := a[2]; !Assign content of A[2]
                             ! to S3.B[2]
  END;
```

The following example shows how you can access structure items by using structure pointers declared in structures:

```

STRUCT template (*);           !Declare template structure
BEGIN
  STRING b[0:2];
  INT    e;
END;

STRUCT .link_list;           !Declare definition structure
BEGIN
  INT .fwd_ptr (link_list); !Declare structure pointer and
                           ! redefine simple variable
  STRING .EXT ptr_to_ext_item(template);
                           !Declare structure pointer
  INT(32) .b;                !Declare simple pointer
  STRUCT item(template);    !Declare referral substructure
END;

INT .new_item := %100000;    !Declare simple pointer to
                           ! first list item

PROC m MAIN;
BEGIN
  @link_list.fwd_ptr := @new_item;
                           !Put address into first forward
                           ! pointer
  @new_item := @new_item '+' ($LEN(link_list) + 1 ) / 2;
  @link_list.fwd_ptr.fwd_ptr := @new_item;
                           !Put address into second forward
                           ! pointer
END;

```

In the preceding example:

- @LINK_LIST.FWD_PTR refers to the content of the first forward standard simple pointer.
- @LINK_LIST.FWD_PTR.PTR_TO_EXT_ITEM refers to the content of the extended simple pointer in the second LINK_LIST.
- @LINK_LIST.FWD_PTR.PTR_TO_EXT_ITEM.B refers to the address of B in the second instance of LINK_LIST in extended memory.

Copying Data in Structures You can copy data to structures by using a move statement. For example, you can copy:

- Structure occurrences between structures
- Structure occurrences within a structure
- Substructure occurrences between structures
- Structure items

To start copying from the lower occurrence, use the left-to-right move operator (':='). To start copying from the upper occurrence, use the right-to-left move operator ('=:').

Copying Structure Occurrences Between Structures

To copy structure occurrences from one structure to another, specify in a move statement:

- A *destination* structure and a *source* structure
- The FOR clause including the ELEMENTS qualifier

For example, you can copy three occurrences of a source structure to a destination structure as follows:

```
LITERAL copies = 3;                !Number of occurrences
STRUCT .s_struct[0:copies - 1];    !Source structure
  BEGIN
  INT a;
  INT b;
  INT c;
  END;

STRUCT .d_struct (s_struct) [0:copies - 1];
                                     !Destination structure

PROC j;
  BEGIN
  d_struct ':= ' s_struct FOR copies ELEMENTS;
                                     !Move statement copies three
  END;                                ! structure occurrences
```

If you do not specify ELEMENTS, the equivalent move statement is:

```
d_struct ':= ' s_struct FOR copies
                                     * (($LEN(s_struct) + 1) '>>' 1) WORDS;
```

which is the code that the compiler generates in this case. The standard function \$LEN returns the length in bytes of one occurrence of S_STRUCTURE.

Copying Structure Occurrences Within a Structure

To copy occurrences within a structure, specify in a move statement:

- The same structure for *destination* and *source*
- The FOR clause including the ELEMENTS qualifier

For example, you can copy the data in each occurrence of a structure one occurrence to the right, beginning with occurrence [8], thus freeing occurrence [0] for new data.

```
LITERAL last = 9;           !Last occurrence

STRUCT t_struct(*);       !Template structure
  BEGIN
  INT i;
  INT j;
  END;

STRUCT .s_struct (t_struct) [0:last];
                                !Source and destination structure

PROC k;
  BEGIN
  s_struct[last] ':=' s_struct[last-1] FOR last ELEMENTS;
                                !Move nine structure occurrences
  END;
```

Copying Substructure Occurrences Between Structures

To copy occurrences of a substructure between structures, specify in a move statement:

- The fully qualified identifiers of the destination and source substructures
- The FOR clause including the ELEMENTS qualifier

You can copy three substructure occurrences from one structure to another as follows:

```
LITERAL copies = 3;       !Number of occurrences

STRUCT .s_struct;
  BEGIN
  STRUCT s_sub[0:copies - 1];
    BEGIN
    !Source S_SUB is in S_STRUCT
    INT a;
    INT b;
    END;
  END;

STRUCT .d_struct (s_struct);
                                !Destination S_SUB is in D_STRUCT

PROC m;
  BEGIN
  d_struct.s_sub ':=' s_struct.s_sub FOR copies ELEMENTS;
                                !Byte move of three
  END;                                ! substructure occurrences
```

Copying Structure Items

You can use a move statement to copy structure items within and between structures and substructures. Structure items you can copy are simple variables, arrays, and pointers declared within structures or substructures.

For example, to copy an array from one structure to another structure, specify in a move statement:

- The fully qualified identifiers of the destination and source arrays
- The FOR clause with or without the BYTES or WORDS qualifier

The FOR clause copies the specified number of bytes, words, doublewords, or quadruplewords depending on the data type of the source array. To copy bytes or words regardless of source data type, include the BYTES or WORDS qualifier in the FOR clause.

The following example shows how you can copy an array from one structure to another, first in quadrupleword units, then in word units:

```
STRUCT .s_struct;
  BEGIN
    FIXED array[0:2];      !Source ARRAY is in S_STRUCTURE
  END;

STRUCT .d_struct (s_struct);
                                !Destination ARRAY is in D_STRUCTURE

PROC m;
  BEGIN
    d_struct.array := s_struct.array FOR 3;
                                !Copy three quadruplewords as
                                ! dictated by FIXED data type

    d_struct.array := s_struct.array FOR 6 WORDS;
                                !Copy six words as dictated by
  END;                                ! the WORDS qualifier
```

Copying Structure Occurrences Using Structure Pointers

You can copy structure occurrences using structure pointers:

```
LITERAL copies := 3;

STRUCT a_struct;          !Definition structure
  BEGIN
    INT a;
    STRING b;
  END;

STRUCT b_struct (a_struct) [0:copies - 1];
                          !Referral structure

INT .EXT ptr0(a_struct) := $XADR (a_struct);
                          !Assign address of A_STRUCT to
                          ! extended INT pointer PTR0

STRING .EXT ptr1(a_struct) := $XADR (b_struct);
                          !Assign address of B_STRUCT to
                          ! extended STRING pointer PTR1

PROC n;
  BEGIN
    ptr1 ':=' ptr0 FOR copies ELEMENTS;
                          !Word move from A_STRUCT to B_STRUCT

    ptr0 ':=' ptr1 FOR copies ELEMENTS;
                          !Byte move from B_STRUCT to A_STRUCT
  END;
```

Using Standard Functions With Structures

You can use the following standard functions with structures. These functions return information such as the length of a structure occurrence or the offset of a structure item within a structure:

Standard Function	Effect
\$BITLENGTH	Returns the length, in bits, of one occurrence of a structure or substructure
\$BITOFFSET	Returns an item's offset, in bits, from the zeroth occurrence of the encompassing structure
\$LEN	Returns the length, in bytes, of one occurrence of an item
\$OFFSET	Returns an item's offset, in bytes, from the zeroth occurrence of the encompassing structure
\$OCCURS	Returns the number of occurrences of a structure, substructure, or array, but not of a template structure
\$TYPE	Returns the type of an item

The following example reads structured data from a disk file. In the FOR statement, \$OCCURS returns 6 (the number of occurrences in JOB_DATA), and \$LEN returns 24 (the length in bytes of one occurrence of JOB_DATA):

```

INT record_num;                                !Number of records

STRUCT emp_data(*);                            !Template structure
BEGIN
  INT number;
  INT dept;
  STRING ssn[0:11];
  FIXED(2) salary;
END;

PROC p MAIN;                                   !Main procedure
BEGIN
  INT diskfile, num_read;
  STRUCT .job_data (emp_data) [0:5]; !Referral structure
  !Some code
  FOR record_num := 0 TO $OCCURS (job_data) - 1 DO
                                                    !FOR statement

      CALL READ(diskfile,                          !CALL statement
                job_data[record_num],             !Buffer
                $LEN (job_data),                  !Maximum bytes to read
                num_read);                          !Count of bytes read

  !More code
END;

```

In the preceding example, the FOR statement calls the READ system procedure once for each occurrence of structure JOB_DATA. For information on the READ procedure, see the *Guardian Procedure Calls Reference Manual*.

9 Using Pointers

This section describes:

- How to declare and initialize pointers
- How to assign addresses to pointers
- How to access data with pointers
- How the compiler allocates storage for pointers

You can declare the following kinds of pointers:

- Simple pointer—a variable into which you can store a memory address, usually of a simple variable or array, which you can access with this simple pointer.
- Structure pointer—a variable into which you can store the memory address of a structure which you can access with this structure pointer.

Pointers can be standard or extended:

- Standard (16-bit) pointers can access data only in the user data segment.
- Extended (32-bit) pointers can access data in any segment, normally the automatic extended data segment.

Other information on pointers appears in the TAL manuals as follows:

Information	Manual	Section/Appendix
Pointers declared inside structures	<i>TAL Programmer's Guide</i> <i>TAL Reference Manual</i>	8, "Using Structures" 8, "Structures"
Pointer access to the upper 32K-word area of the user data segment, to the user code segment, or to an explicit (user-allocated) extended data segment	<i>TAL Programmer's Guide</i>	B, "Managing Addressing"
Implicit pointers (those generated by the compiler when you declare indirect arrays and structures)	<i>TAL Programmer's Guide</i>	7, "Using Arrays" 8, "Using Structures"
Dereferencing (formerly known as temporary pointers)	<i>TAL Programmer's Guide</i>	5, "Using Expressions"

Using Simple Pointers A simple pointer is a variable into which you must store a memory address, usually of a simple variable or an array element. When you refer to a simple pointer identifier in expressions, you access the item whose address is stored in the simple pointer.

Before accessing data through a simple pointer, you must declare the pointer and store a memory address in it. You can store an address by initializing the pointer when you declare it or by assigning an address to it later in an assignment statement.

Declaring Simple Pointers To declare a simple pointer, specify:

- Any *data type* except UNSIGNED
- An *identifier*, preceded by an indirection symbol (. or .EXT)

To declare a standard simple pointer, use the standard indirection symbol (.):

```
INT .my_ptr;
```

To declare an extended simple pointer, use the extended indirection symbol (.EXT):

```
INT .EXT my_xptr;
```

Extended pointer declarations should precede other global or local declarations. The compiler emits more efficient machine code if it can allocate extended pointers between G[0] and G[63] or between L[0] and L[63].

Specifying a Data Type When you declare a simple pointer, you can specify any of the following data types. The data type determines how much data a simple pointer can access at a time, as listed in Table 9-1.

Table 9-1. Data Accessed by Simple Pointers

Data Type	Accessed Data
STRING	Byte
INT	Word
INT(32)	Doubleword
REAL	Doubleword
REAL(64)	Quadrupleword
FIXED	Quadrupleword

Initializing Simple Pointers To initialize a pointer in the declaration, specify the assignment operator after the pointer identifier, followed by an initialization expression that represents an address. The addressing mode and data type determines the kind of address the simple pointer can contain, as described in Table 9-2.

Table 9-2. Addresses in Simple Pointers

Addressing Mode	Data Type	Kind of Addresses
Standard	STRING	16-bit byte address in the lower 32K-word area of the user data segment.
Standard	Any except STRING	16-bit word address anywhere in the user data segment.
Extended	STRING	32-bit byte address, normally in the automatic extended data segment.
Extended	Any except STRING	32-bit even-byte address, normally in the automatic extended data segment. (If you specify an odd-byte address, results are undefined.)

Initializing Global Simple Pointers At the global level, you can only initialize pointers with constant expressions. Constant expressions can contain the following operands:

- Numeric constants
- LITERALS
- Return values of standard functions whose arguments are constant expressions
- Addresses of previously declared variables obtained by using the @ operator

Standard Pointers

To initialize a standard pointer with the address of a variable, prefix @ to the variable name on the right side of the assignment operator in the declaration. You can specify the address to store in a pointer by using any of the following expressions:

Expression	Operation
<i>@identifier</i>	Accesses address of variable
<i>@identifier'<<' 1</i>	Converts word address to byte address
<i>@identifier'>>' 1</i>	Converts byte address to word address
<i>@identifier[index]</i>	Accesses address of variable indicated by <i>index</i>

The following table shows the kinds of global variables to which you can apply the @ operator:

Variable	@ <i>identifier</i> ?
Direct array	Yes
Standard indirect array	Yes
Extended indirect array	No
Direct structure	Yes
Standard indirect structure	Yes
Extended indirect structure	No
Simple pointer	No
Structure pointer	No

For example, you can initialize a global standard simple pointer with the address of an array element:

```
STRING .array[0:3];      !Declare ARRAY
STRING .string_ptr := @array[3];
                        !Declare STRING_PTR; initialize
                        ! it with address of ARRAY[3]
```

You can use a STRING simple pointer for byte access to a word-addressed array. To convert the word address to a byte address, use a left-shift operation ('<<' 1):

```
INT .word_array[0:39];  !Declare WORD_ARRAY
STRING .byte_ptr := @word_array[0] '<<' 1;
                    !Declare BYTE_PTR; initialize
                    ! it with converted byte
                    ! address of WORD_ARRAY
```

You can use an INT simple pointer for word access to a byte-addressed array. To convert the byte address to a word address when you initialize the pointer, use a right-shift operation ('>>' 1). (Only even byte addresses can be converted to correct word addresses):

```
STRING .byte_array[0:4]; !Declare BYTE_ARRAY
INT .word_ptr := @byte_array[0] '>>' 1;
                !Declare WORD_PTR; initialize
                ! it with converted word
                ! address of BYTE_ARRAY
```

Extended Pointers

You can initialize global extended pointers with extended addresses converted from standard addresses by the \$DBL, \$UDBL, and \$DBLL standard functions. These functions compute the initialized value if their arguments are constant expressions or expressions that follow the rules applied to standard global pointers. Here is an example of the \$UDBL function:

```
INT a[0:2];

INT .EXT ptr := $UDBL(@a[0] '<<' 1);
```

In the preceding example, the word address returned by \$UDBL must be converted to a byte address by shifting it left by one bit, because extended addresses are always byte addresses.

Pointers in BLOCK Declarations

A data declaration and any declaration that refers to that declaration must appear in the same BLOCK declaration. For example, the following data declarations must appear in the same BLOCK declaration:

```
BLOCK my_globals;                !BLOCK declaration
  STRING .EXT array[0:5];
  STRING .EXT ptr_a := @array[0];
END BLOCK;                        !End BLOCK declaration
```

Initializing Local or Sublocal Simple Pointers

At the local or sublocal level, you can initialize simple pointers with any arithmetic expression, including those shown previously for global pointers. In other words, you can use initialization expressions that contain variables, constants, and LITERALS.

You can initialize a local or sublocal standard simple pointer with the content of an array element:

```
INT array[0:1] := [%100000, %110000];    !Declare ARRAY

INT .int_ptr1 := array[0];               !Declare INT_PTR1; initialize
                                           ! it with %100000

INT .int_ptr2 := array[1];               !Declare INT_PTR2; initialize
                                           ! it with %110000
```

You can initialize a local or sublocal standard simple pointer with the address of a structure item:

```
STRUCT .x;                               !Declare structure
  BEGIN
  INT i;
  END;

INT .ip := @x.i;                          !Declare IP; initialize it
                                           ! with address of structure
                                           ! item
```

You can initialize a local or sublocal extended simple pointer with the address of an extended indirect array:

```
INT .EXT ext_array[0:99];      !Declare EXT_ARRAY
INT .EXT ext_ptr := @ext_array[5];
                                !Declare EXT_PTR; initialize it
                                ! with address of EXT_ARRAY[5]
```

You can initialize a local or sublocal extended simple pointer with the 32-bit byte address returned by the \$XADR standard function for an array that has a 16-bit byte address:

```
STRING byte_array[0:1];      !Declare BYTE_ARRAY to have
                                ! 16-bit byte address
INT .EXT ext_ptr := $XADR(byte_array[0]);
                                !Declare EXT_PTR; initialize it
                                ! with converted 32-bit byte
                                ! address of BYTE_ARRAY[0]
```

You can initialize a local or sublocal extended simple pointer with the 32-bit byte address returned by \$XADR for an array that has a 16-bit word address:

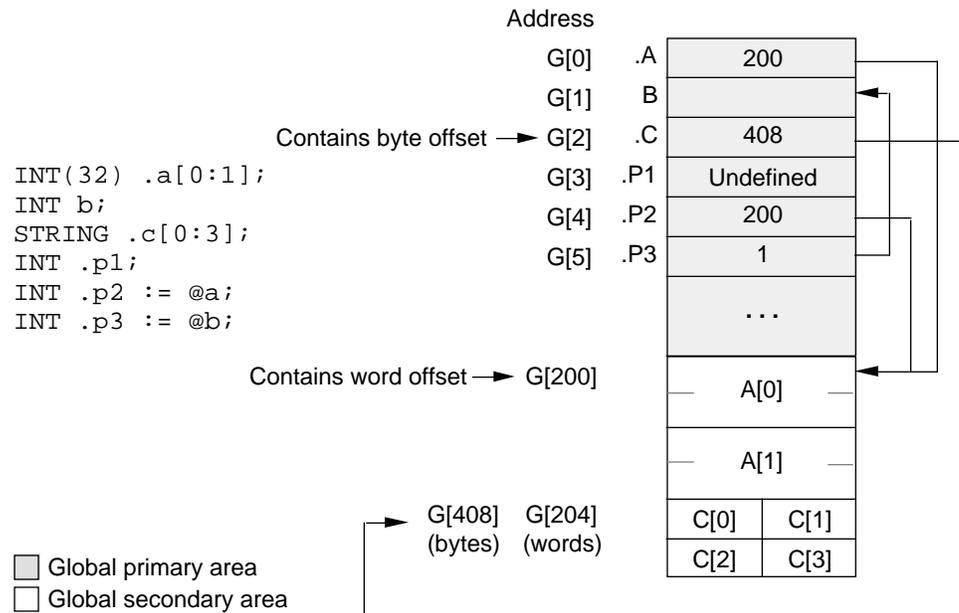
```
INT word_array[0:1];        !Declare WORD_ARRAY to have
                                ! 16-bit word address
STRING .EXT ext_ptr := $XADR(word_array[0]);
                                !Declare EXT_PTR; initialize it
                                ! with converted 32-bit byte
                                ! address of WORD_ARRAY[0]
```

Allocating Simple Pointers The compiler allocates a word of storage for each standard pointer and a doubleword for each extended pointer.

The compiler does not allocate space for the data to which the pointer points. You normally store the addresses of previously declared items in pointers.

Figure 9-1 shows example pointer declarations and the storage allocation that results. (If you use BLOCK declarations, however, the compiler allocates storage as described in Section 14, “Compiling Programs.”)

Figure 9-1. Allocating Simple Pointers



397

Assigning Addresses to Simple Pointers

Once you have declared a simple pointer, you can store a memory address in it by using an assignment statement (even if you initialized the pointer with an address when you declared it).

To assign an address to a pointer, specify the pointer identifier (prefixed by @), followed by the assignment operator and an arithmetic expression that represents a memory address. On the left side of the assignment operator, @ changes the address contained in the pointer, not the value of the variable to which the pointer points.

You can also prefix @ to a variable on the right side of the assignment operator. If the variable is a pointer, @ returns the address contained in the pointer. If the variable is not a pointer, @ returns the address of the variable itself.

Assigning Addresses of Simple Variables or Arrays

To assign the address of a simple variable or array to a standard simple pointer, place the variable or array identifier (prefixed by @) on the right side of the assignment operator:

```

STRING .bytes[0:3];           !Declare indirect array BYTES
STRING .s_ptr;               !Declare simple pointer S_PTR
INT i := 3;                  !Declare simple variable I

@s_ptr := @bytes[i];         !Assign address of
                             ! BYTES[3] to S_PTR

```

You can assign the address of an INT simple variable or array to standard simple pointers of different types. The FIXED simple pointer lets you view four words at a time; the INT(32) simple pointer lets you view two words at a time:

```

INT .array[0:99];           !Declare INT array
FIXED .quad_ptr;           !Declare FIXED simple pointer
INT(32) .dbl_ptr;          !Declare INT(32) simple pointer

@quad_ptr := @array[0];    !Assign address of INT array
                             ! to FIXED simple pointer

@dbl_ptr := @array[0];     !Assign address of INT array
                             ! to INT(32) simple pointer

```

Assigning Converted Addresses

You can convert a word address to a byte address and assign it to a STRING simple pointer. You then have byte access to the word item:

```

STRING .s_ptr;             !Declare STRING simple pointer
INT .word[0:5];           !Declare INT array

@s_ptr := @word[3] '<<' 1; !Assign converted byte address
                             ! of WORD[3] to S_PTR

```

To convert a standard (16-bit) address to an extended (32-bit) address and assign it to an extended simple pointer, use the \$XADR standard function:

```

INT .EXT ext_ptr;         !Declare extended simple pointer
STRING s_array[0:1];     !Declare STRING array

@ext_ptr := $XADR(s_array[0]);
                             !Assign converted 32-bit address
                             ! of S_ARRAY to EXT_PTR

```

Assigning the Content of Simple Pointers

To assign an address contained in a simple pointer (for instance, PTR1) to another simple pointer (for instance, PTR2), specify @PTR1 on the right side of the assignment operator:

```

INT array[0:99];           !Declare ARRAY
INT .ptr1 := @array;      !Declare and initialize PTR1
INT .ptr2;                !Declare PTR2

@ptr2 := @ptr1;          !Assign content of PTR1 to PTR2

```

Accessing Data With Simple Pointers

You access the data item to which a simple pointer points by using the pointer identifier in statements. When you use the pointer identifier in statements, omit the @ operator.

You can use standard and extended simple pointers in any statement. For example, to assign a value, say 45, to an array element, you can store the array address in a simple pointer and then assign 45 to the pointer (without prefixing it with @):

```

INT .addr[0:2] := [1,2,3]; !Declare and initialize
                        ! array ADDR
INT .sp := @addr[0];      !Declare and initialize
                        ! standard simple pointer SP
                        ! with address of ADDR[0]
sp := 45;                !Assign 45 to ADDR[0]; ADDR
                        ! now contains [45,2,3]

```

Accessing Data in Scans and Moves

Here are guidelines for using simple pointers in scan and move statements:

- An array that is the object of a SCAN or RSCAN statement must be located in the lower 32K-word area of the user data segment.
- An extended simple pointer cannot be the object of a SCAN or RSCAN statement.
- In a move statement, the destination pointer must point to a variable large enough to accommodate the source data you want to copy.

The following example shows pointers used in assignment, move, IF, and SCAN statements:

```

INT var_a;                !Declare variables VAR_A and
INT var_b;                ! VAR_B
INT .ptr;                 !Declare standard simple
                           ! pointer PTR
INT .EXT ptr_a;          !Declare extended simple
INT .EXT ptr_b;          ! pointers PTR_A and PTR_B
!Some code to initialize PTR_A and PTR_B

                           !Assignment statements:
var_a := ptr_a;          !Assign to VAR_A the value of
                           ! item pointed to by PTR_A
ptr_a := var_a;          !Assign value of VAR_A to PTR_A
ptr_a := ptr_b;          !Assign to PTR_A the value of
                           ! item pointed to by PTR_B

                           !Move statements:
var_a ':=' ptr_a FOR 2 WORDS; !Copy 2 words starting at
                           ! address in PTR_A (modify
                           ! both VAR_A and VAR_B)

ptr_a ':=' var_a FOR 2 WORDS; !Copy 2 words starting at
                           ! address of VAR_A (copy
                           ! content of VAR_A and VAR_B
                           ! into location pointed to
                           ! by PTR_A)

ptr_a ':=' ptr_b FOR 10 WORDS; !Copy 10 words starting at
                           ! address contained in PTR_B

                           !IF and SCAN statements:
IF var_a = ptr_a FOR 2 WORDS THEN
    SCAN ptr WHILE " ";    !If contents of VAR_A and VAR_B
                           ! match data pointed to by
                           ! PTR_A, scan area starting at
                           ! address contained in PTR
                           ! while spaces occur

```

Indexing Simple Pointers You can access data by appending an *index* (enclosed in brackets) to the identifier of a simple pointer as follows:

```
ptr[2]
```

For a standard simple pointer, the index must be a signed INT arithmetic expression. For an extended simple pointer, the index can be either:

- A signed INT arithmetic expression (-32,768 through 32,767)
- A signed INT(32) arithmetic expression (-2,147,483,648 through 2,147,483,647)

The index represents an element offset from the address stored in the simple pointer; that is, from the address of a simple variable, array, or structure item. The element offset yielded by an index depends on the data type of the simple pointer:

Data Type	Element Offset
STRING	Byte
INT	Word
INT(32) or REAL	Doubleword
REAL(64) or FIXED	Quadrupleword

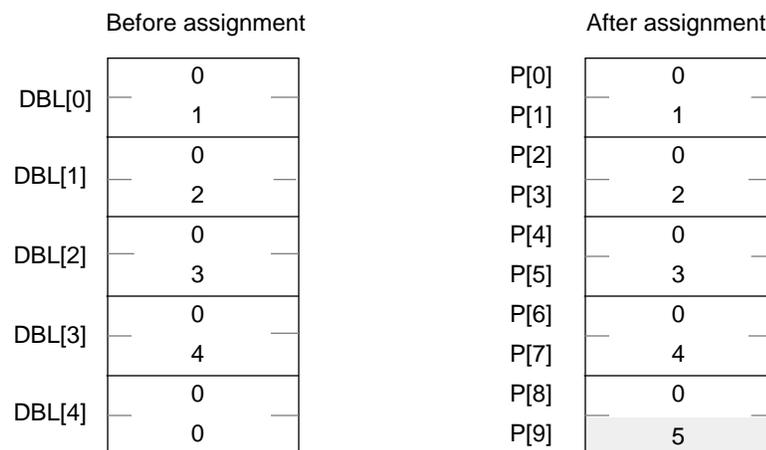
For example, you can initialize an INT simple pointer with the address of an INT(32) array. You can then append an index to the simple pointer and assign a value to the last word of the array:

```
PROC z MAIN;
BEGIN
  INT(32) dbl[0:4] := [1D, 2D, 3D, 4D, 0D];
  INT .p := @dbl;           !View DBL as single words

  p[9] := 5;                !Last word of DBL is at a 9-word
  END;                       ! offset from P[0]
```

Figure 9-2 shows how the compiler allocates a doubleword for each element of the INT(32) array declared in the preceding example. The figure shows how INT pointer P can access each word of each element and how the assignment to P[9] changes the value stored in the last word of the array to 5.

Figure 9-2. Indexing Simple Pointers



Using Structure Pointers

A structure pointer is a variable that you associate with a particular structure layout. You must store the memory address of a structure in the structure pointer before you access the data to which it points. You can store an address by initializing the pointer when you declare it or by assigning an address to it later in an assignment statement. When you refer to a structure pointer identifier in expressions, you access the structure whose address is stored in the structure pointer.

Declaring Structure Pointers

To declare a structure pointer, specify:

- `STRING` or `INT` attribute, as described in Table 9-3
- An *identifier*, preceded by an indirection symbol (`.` or `.EXT`)
- A *referral* that associates the structure pointer with a structure layout—enclose in parentheses the identifier of one of the following:
 - An existing definition structure, template structure, or referral structure
 - An existing structure pointer
 - The encompassing structure if this structure pointer is a structure item

To declare a standard structure pointer, use the standard indirection symbol (`.`) and enclose a referral in parentheses:

```
INT .struct_ptr (prev_struct);
```

To declare an extended structure pointer, use the extended indirection symbol (`.EXT`) and enclose a referral in parentheses:

```
INT .EXT xstruct_ptr (prev_struct);
```

Place extended pointer declarations preceding other global or local declarations. The compiler emits more efficient machine code if it can store extended pointers between `G[0]` and `G[63]` or between `L[0]` and `L[63]`.

Initializing Structure Pointers To initialize a pointer when you declare it, specify the assignment operator after the identifier, followed by an initialization expression.

The addressing mode and STRING or INT attribute of the structure pointer determine the kind of addresses the pointer can contain, as described in Table 9-3.

Table 9-3. Addresses in Structure Pointers

Addressing Mode	STRING or INT Attribute	Kind of Addresses
Standard	STRING *	16-bit byte address of a substructure, STRING simple variable, or STRING array declared in a structure located in the lower 32K-word area of the user data segment
Standard	INT **	16-bit word address of any structure item located anywhere in the user data segment
Extended	STRING *	32-bit byte address of any structure item located in any segment, normally in the automatic extended data segment
Extended	INT **	32-bit byte address of any structure item located in any segment, normally in the automatic extended data segment

* If the pointer is the *source* in a move statement or group comparison expression that omits a *count-unit*, the *count-unit* is BYTES.

** If the pointer is the *source* in a move statement or group comparison expression that omits a *count-unit*, the *count-unit* is WORDS.

Initializing Global Structure Pointers At the global level, you can initialize structure pointers only with constant expressions, as described in “Initializing Global Simple Pointers” earlier in this section.

Standard Pointers

You can initialize a standard structure pointer with the address of a structure occurrence:

```
STRUCT struct_a[0:2];           !Declare STRUCT_A
  BEGIN
  INT var;
  END;

INT .struct_ptr (struct_a) := @struct_a[1];
                             !Declare STRUCT_PTR; initialize
                             ! it with address of
                             ! STRUCT_A[1]
```

You can initialize standard structure pointers with the addresses of structure items by using:

- The `$OFFSET` standard function, which returns an item's byte offset from the zeroth occurrence of the encompassing structure
- An unsigned operator such as `+` to append an offset in the range 0 through 65,535 to a standard pointer
- Bit-shift operations (`>> 1` or `<< 1`) to convert a byte address to a word address or vice versa

For example, the following initializations store addresses of structure items `SUBSTRUC` and `VAR_A` in standard structure pointers:

```

STRUCT .struc;                !Declare STRUC
  BEGIN
  STRUCT substruc;          !Declare SUBSTRUC
    BEGIN
    INT var_a;              !Declare VAR_A
    INT var_b;
    END;
  END;

INT .word_ptr_a (struc) := @struc '+'
                        $OFFSET(struc.substruc) '>>' 1;
                        !Declare WORD_PTR_A; initialize
                        ! it with converted word address
                        ! of STRUC.SUBSTRUC

INT .word_ptr_b (struc) := @struc '+'
                        $OFFSET(struc.substruc.var_a) '>>' 1;
                        !Declare WORD_PTR_B; initialize
                        ! it with converted word address
                        ! of STRUC.SUBSTRUC.VAR_A

STRING .byte_ptr (struc) := @struc '<<' 1 '+'
                        $OFFSET(struc.substruc);
                        !Declare BYTE_PTR; initialize
                        ! it with converted byte
                        ! address of STRUC.SUBSTRUC

```

A standard `STRING` structure pointer can access the following structure items only—a substructure, a `STRING` simple variable, or a `STRING` array—located in the lower 32K-word area of the user data segment. The last declaration in the preceding example shows a `STRING` structure pointer initialized with the converted byte address of a substructure.

Here is another way to access a STRING item in a structure. You can convert the word address of the structure to a byte address when you initialize the STRING structure pointer and then access the STRING item in a statement:

```
STRUCT .astruct[0:1];
BEGIN
  STRING s1;
  STRING s2;
  STRING s3;
END;

STRING .ptr (astruct) := @astruct[1] '<<' 1;
                                !Declare STRING PTR; initialize
                                ! it with converted byte
                                ! address of ASTRUCT[1]

ptr.s2 := %4;                    !Access STRING structure item
```

Initializing Local or Sublocal Structure Pointers

You can initialize a local or sublocal standard structure pointer with the address of a standard indirect structure:

```
STRUCT .std_struct[0:2];        !Declare STD_STRUCTURE
BEGIN
  INT array1[0:7];
END;

INT .std_ptr (std_struct) := @std_struct[0];
                                !Declare STD_PTR; initialize it
                                ! with address of STD_STRUCTURE[0]
```

You can initialize a local or sublocal STRING structure pointer with the address of a substructure:

```
STRUCT name_def(*);
BEGIN
  STRING first[0:3];
  STRING last[0:3];
END;

STRUCT .record;
BEGIN
  STRUCT name (name_def);        !Declare substructure
  INT age;
END;

STRING .my_name (name_def) := @record.name;
                                !Declare STRING structure
                                ! pointer; initialize it with
                                ! address of substructure

my_name ' :=' ["Don Good"];
```

You can initialize a local or sublocal extended structure pointer with the address of an extended indirect structure:

```
STRUCT .EXT ext_struct[0:2]; !Declare EXT_STRUCTURE
BEGIN
  INT array1[0:7];
END;

INT .EXT ext_ptr (ext_struct) := @ext_struct[0];
                                !Declare EXT_PTR; initialize it
                                ! with address of EXT_STRUCTURE[0]
```

You can initialize an extended structure pointer with the 32-bit byte address returned by the \$XADR standard function for a structure that has a 16-bit word address:

```
STRUCT std_struct[0:2];      !Declare STD_STRUCTURE
BEGIN
  INT array1[0:7];
END;

INT .EXT ext_ptr (std_struct) := $XADR(std_struct[0]);
                                !Declare EXT_PTR; initialize it
                                ! with converted 32-bit byte
                                ! address of STD_STRUCTURE[0]
```

Allocating Structure Pointers

The compiler allocates a word of storage for each standard pointer and a doubleword for each extended pointer, as shown in Figure 9-1 earlier in this section.

The compiler does not allocate space for the data to which the pointer points. You normally store the addresses of previously declared structures in structure pointers.

Assigning Addresses to Structure Pointers

Once you have declared a structure pointer, you can store a memory address in it by using an assignment statement (even if you initialized the pointer with an address when you declared it).

To assign an address to a pointer, prefix @ to the pointer identifier on the left side of the assignment operator, and specify an arithmetic expression that represents a memory address. On the left side of the assignment operator, @ changes the content of the pointer (that is, the address contained in the pointer), not the value of the item to which the pointer points.

You can also prefix @ to a variable identifier on the right side of the assignment operator. If the variable is a pointer, @ returns the address contained in the pointer. If the variable is not a pointer, @ returns the address of the variable.

Address of Structure Occurrence

To assign the address of a standard structure occurrence to a structure pointer, place the structure identifier, prefixed with @, on the right of the assignment operator:

```
STRUCT struct_a[0:2];           !Declare STRUCT_A
BEGIN
  INT array1[0:7];
END;

PROC any MAIN;
BEGIN
  INT .struct_ptr (struct_a);   !Declare STRUCT_PTR
  @struct_ptr := @struct_a[2]; !Assign address of STRUCT_A[2]
                                ! to STRUCT_PTR
END;
```

Converting Addresses

To convert a standard (16-bit) address to an extended (32-bit) address and assign it to an extended structure pointer, use the \$XADR standard function:

```
STRUCT struct_a[0:2];           !Declare STRUCT_A
BEGIN
  INT array1[0:7];
END;

PROC any MAIN;
BEGIN
  INT .EXT xstruct_ptr (struct_a);
                                !Declare XSTRUCT_PTR
  @xstruct_ptr := $XADR(struct_a[2]);
                                !Assign address of STRUCT_A[2]
                                ! to XSTRUCT_PTR
END;
```

Assigning the Content of a Structure Pointer

You can assign an address contained in a structure pointer to another structure pointer by using the @ operator on the right side of the assignment operator:

```
PROC any;
BEGIN

STRUCT struct_a[0:2];           !Declare STRUCT_A
BEGIN
  INT array1[0:7];
END;

INT .ptr_a (struct_a) := @struct_a[0];
                                !Declare PTR_A; initialize it
                                ! with address of STRUCT_A[0]

INT .ptr_b (struct_a);         !Declare PTR_B;

@ptr_b := @ptr_a;              !Assign content of PTR_A
                                ! to PTR_B
END;
```

Accessing Data With Structure Pointers

You access the data item to which a pointer points by using the pointer identifier, without @, in statements.

In a move, SCAN, or RSCAN statement, you normally use the unqualified identifier of a structure pointer. (An extended pointer cannot be the object of a SCAN or RSCAN statement.)

Assigning Values to Structure Items

In an assignment statement, you can specify the qualified identifier of a structure item suffixed to a standard or extended structure pointer. The following example shows the qualified identifier of structure item NAME, which is declared in substructure CUSTOMER, which is declared in substructure RECORDS, which is declared in a structure that is pointed to by STRUCT_PTR:

```
struct_ptr.records.customer.name
```

The following example assigns values to items in a structure, pointed to by STRUCT_PTR.INT2 and STRUCT_PTR.STR1:

```
STRUCT .astruct[0:2];           !Declare ASTRUCT
BEGIN
  INT    int1;
  INT    int2;
  INT    int3;
  STRING str1;
END;

INT struct_ptr (astruct) := @astruct[0];
                                !Declare STRUCT_PTR; initialize
                                ! it with address of ASTRUCT[0]

struct_ptr.int2 := %14;         !Assign values to ASTRUCT[0]
struct_ptr.str1 := %3;          ! items INT2 and STR1
```

Copying Structure Occurrences

To copy structure occurrences from one location to another, use unqualified identifiers of structure pointers such as STRUCT_PTR1 and STRUCT_PTR2 in move statements:

```

STRUCT .old_record[0:2];          !Declare OLD_RECORD
  BEGIN
  STRING name[0:31];
  INT salary;
  END;

STRUCT .new_record (old_record) [0:2];
                                     !Declare NEW_RECORD

INT .struct_ptr1 (old_record) := @old_record;
                                     !Declare STRUCT_PTR1;
                                     ! initialize it with
                                     ! address of OLD_RECORD

INT .struct_ptr2 (new_record) := @new_record;
                                     !Declare STRUCT_PTR2;
                                     ! initialize it with
                                     ! address of NEW_RECORD

struct_ptr2 ':= ' struct_ptr1 FOR 3 ELEMENTS;
                                     !Move statement copies data
                                     ! from OLD_RECORD to NEW_RECORD

```

Accessing STRING Items in Structures

A STRING structure pointer can access the following structure items only—a substructure, a STRING simple variable, or a STRING array—located in the lower 32K-word area of the user data segment.

To enable a STRING structure pointer to access a STRING structure item, you must convert the word address of the structure to a byte address before assigning the address to the pointer:

```

STRUCT .astruct[0:1];
  BEGIN
  STRING s1;
  STRING s2;
  STRING s3;
  END;

STRING .ptr (astruct) := @astruct[1] '<<' 1;
                                     !Initialize PTR with converted
                                     ! byte address of ASTRUCT[1]

ptr.s2 := %4;                         !Access STRING structure item

```

In the following example, a STRING structure pointer accesses a substructure in a structure:

```

STRUCT name_def(*);
  BEGIN
    STRING first[0:3];
    STRING last[0:3];
  END;

STRUCT .record;
  BEGIN
    STRUCT name (name_def);      !Declare substructure
    INT age;
  END;

STRING .my_name(name_def) := @record.name;
                                !Only allowed for local or
                                ! sublocal structure pointer

my_name ' := ' ["DON GOOD"];

```

Indexing Structure Pointers To access a structure item in a particular structure occurrence, append an index (enclosed in brackets) to the identifier of a structure pointer. For example, you can access item B in the fourth occurrence of a structure by indexing as follows:

```

STRUCT .low[0:5];                !Declare structure LOW
  BEGIN                          ! to have six occurrences
    INT a;
    INT b;
  END;

INT struct_ptr (low) := @low; !Declare STRUCT_PTR; initialize
                                ! it with address of LOW[0]

struct_ptr[3].b := 45;          !Access item B in LOW[3]

```

You can access a nested structure item by appending indexes to the various levels in the qualified identifier of the structure item. For example, you can access an array element in a particular structure and substructure occurrence as follows:

```

struct_ptr[1].my_substruct[0].my_array[4]

```

Each index represents an offset from the zeroth element of an array or the zeroth occurrence of a structure or substructure, regardless of the declared lower bound.

For a byte-addressed structure item, the index represents a byte offset. For a word-addressed structure item, the index represents a word offset.

The indexed item determines the size of the index offset, as listed in Table 9-4:

Table 9-4. Indexing Structure Pointers

Indexed Item	Data Type	Size of Index Offset
Structure or structure pointer	Not applicable	Total bytes in one structure occurrence
Substructure	Not applicable	Total bytes in one substructure occurrence
Simple variable or array	STRING	Byte
Simple variable or array	INT	Word
Simple variable or array	INT(32) or REAL	Doubleword
Simple variable or array	REAL(64) or FIXED	Quadrupleword

Indexing Standard Structure Pointers

The index for standard structure pointers must be a signed INT arithmetic expression in the range -32,768 through 32,767. Examples of signed INT arithmetic expressions are:

```
25
index
index + 2
index - 1
9 * index
```

The following example shows how you can index a standard structure pointer:

```

STRUCT .std_struct[0:99]; !Standard indirect structure
BEGIN
  INT var;                !Simple variable
  STRING array[0:25];    !Array
  STRUCT substr[0:9];    !Substructure
  BEGIN
    STRING array[0:25];  !Array
  END;
END;

INT index := 5;          !Simple variable

PROC x MAIN;            !Declare procedure X
BEGIN
  INT struct_ptr (std_struct) := @std_struct;
                                !Declare standard structure pointer

  struct_ptr[index].var := 35;
                                !Access STD_STRUCTURE[5].VAR

  struct_ptr[index+2].array[0] := "A";
                                !Access STD_STRUCTURE[7].ARRAY[0]

  struct_ptr[index+2].array[25] := "Z";
                                !Access STD_STRUCTURE[7].ARRAY[25]

  struct_ptr[9*index].substr[index-1].array[index-5] := "a";
                                !Access
                                ! STD_STRUCTURE[45].SUBSTR[4].ARRAY[0]

END;                      !End procedure X

```

Indexing Extended Structure Pointers

The index for extended structure pointers must be a signed INT or INT(32) arithmetic expression, depending on the size of the offset of the structure item you want to access. The offset of a structure item is from the zeroth structure occurrence (not the current structure occurrence).

C-Series System. If you are writing a program to run on a C-series system, you can determine whether to use an INT index or an INT(32) index (which is slower) as follows:

1. Compute the lower and upper byte or word offsets of the structure item whose structure pointer is being indexed. (A byte-addressed structure item is at a byte offset. A word-addressed structure item is at a word offset.)
2. If the offsets are inside the signed INT range (-32,768 through 32,767), use an INT index. Usually, offsets are within the signed INT range.
3. If the offsets are outside the signed INT range, use an INT(32) index. To convert an INT index to an INT(32) index, use the \$DBL function.

Whenever you increase the structure size or number of occurrences, you must repeat the preceding sequence of steps.

To access a structure item whose offset is within the signed INT range, you can use an INT index as follows:

```
!Default NOINHIBITXX in effect

STRUCT .EXT xstruct[0:9];      !Declare extended indirect
  BEGIN                        ! structure; upper byte offset
  STRING array[0:9];          ! is within INT range
  END;

INT index;                     !Declare INT index
STRING var;

PROC my_proc MAIN;
  BEGIN
  INT .EXT xstruct_ptr (xstruct) := @xstruct[0];
  !Code to initialize INDEX
  var := xstruct_ptr[index].array[0];
  END;                          !Generate correct offset
```

In the preceding example, NOINHIBITXX generates efficient addressing for extended indirect declarations (by using XX instructions described in the *System Description Manual* for your system).

Conversely, INHIBITXX suppresses efficient addressing of extended indirect declarations located between G[0] and G[63] of the user data segment—except when the extended indirect declarations are declared in a BLOCK declaration with the AT (0) or BELOW (64) option. (The BLOCK declaration is described in Section 14, “Compiling Programs.”)

To access a structure item whose offset is outside the signed INT range (–32678 through 32,767), you must use an INT(32) index. To convert an INT index to an INT(32) index, you can use the \$DBL function:

```
STRUCT .EXT xstruct[0:9999];
  BEGIN
  STRING array[0:9];          !Upper byte offset > 32,767;
  END;                          ! INT(32) index required

INT index;

PROC my_proc MAIN;
  BEGIN
  INT .EXT xstruct_ptr (xstruct) := @xstruct[0];
  !Some code here to initialize INDEX

  xstruct_ptr[$DBL(index)].array[0] := 1;
  !Generate correct offset
  END;                          ! because INDEX is an INT(32)
  ! expression
```

In the preceding example, the upper-byte offset of ARRAY is larger than 32,767, computed as follows:

$$9999 * 10 + 9 = 99999$$

Size of offset to ARRAY[9]
Upper bound of array
Size of structure in bytes
Upper bound of structure

334

D-Series System. If you are writing a program to run on a D-series system and need to index into extended indirect structures, you can either:

- Determine when to use a signed INT or INT(32) index as described for C-series programs in the preceding subsection.
- Use the INT32INDEX directive, which is easier and safer, albeit slightly less efficient.

INT32INDEX generates INT(32) indexes from INT indexes and computes the correct offset for the indexed structure item. INT32INDEX overrides the INHIBITXX or NOINHIBITXX directive, whichever is in effect.

NOINT32INDEX, the default, generates incorrect offsets for structure items whose offsets are outside the signed INT range. NOINT32INDEX does not override INHIBITXX or NOINHIBITXX.

Specify INT32INDEX or NOINT32INDEX immediately before the declarations to which it applies. The specified directive then applies to those declarations throughout the compilation. The following D-series example shows how INT32INDEX generates correct offsets and NOINT32INDEX generates incorrect offsets:

```

!The default NOINT32INDEX is in effect.

STRUCT .EXT xstruct[0:9999];    !XSTRUCT has NOINT32INDEX
  BEGIN                          ! attribute
  STRING array[0:9];
  END;

INT index;

PROC my_proc MAIN;
  BEGIN
?INT32INDEX                      !Assign INT32INDEX
                                  ! attribute to subsequent
                                  ! declaration

  INT .EXT xstruct_ptr (xstruct) := @xstruct[0];
                                  !XSTRUCT_PTR has INT32INDEX
                                  ! attribute

  !Some code here
  xstruct_ptr[index].array[0]:= 1;
                                  !Generate correct offset even
                                  ! when offset is > 32,767,
                                  ! because XSTRUCT_PTR has
                                  ! INT32INDEX attribute

  xstruct2[index].array[0] := 1;
                                  !Generate incorrect offset
                                  ! if offset is > 32,767,
                                  ! because XSTRUCT2 has
                                  ! NOINT32INDEX attribute

  END;

```

10 Using Equivalenced Variables

Equivalencing lets you declare more than one identifier and description for a location in a primary storage area. This section describes how to equivalence a new variable to a previously declared variable. You can declare:

- Equivalenced simple variables
- Equivalenced simple pointers
- Equivalenced structures
- Equivalenced structure pointers

The new and previous variables can have different data types and byte-addressing and word-addressing attributes. You can, for example, refer to an INT(32) variable as two separate words or four separate bytes.

Other kinds of equivalencing are described in the TAL manual set as follows:

Information	Manual	Section
Redefinitions (equivalencing within structures)	<i>TAL Programmer's Guide</i> <i>TAL Reference Manual</i>	8, "Using Structures" 8, "Structures" (syntax)
Base-address equivalencing	<i>TAL Reference Manual</i>	10, "Equivalenced Variables"
'SG'-equivalencing	<i>TAL Reference Manual</i>	15, "Privileged Procedures"

Example Diagrams

The diagrams in this section show the relationship of an equivalenced variable to the previous variable. Unless otherwise noted, the diagrams refer to memory locations in the primary area of the user data segment. Here is a sample diagram:

- The shaded box is the previous (allocated) variable
- The unshaded box is the equivalenced (new) variable

```
INT word1;
INT word2 = word1;
```



300

Variables You Can Equivalence

You can equivalence any variable in the first column of Table 10-1 to any variable in the second column. (You cannot equivalence an array to another variable.)

Table 10-1. Equivalenced Variables

Equivalenced (New) Variable	Previous Variable
Simple variable	Simple variable
Simple pointer	Simple pointer
Structure	Structure
Structure pointer	Structure pointer
	Array
	Equivalenced variable

For compatibility with future software platforms, however, equivalence indirect structures only to indirect structures or indirect arrays.

Equivalencing Simple Variables

You can equivalence a new simple variable to a previously declared variable as listed in Table 10-1 earlier in this section.

Declaring Equivalenced Simple Variables

To declare an equivalenced simple variable, specify:

- Any *data type* but UNSIGNED
- The *identifier* of the new simple variable
- An equal sign (=)
- The identifier of the *previous variable*

For portability to future software platforms, declare equivalenced variables that fit within the previous variable.

Equivalencing INT Simple Variables

For example, you can equivalence an INT variable (WORD2) to a previous INT variable (WORD1):

```
INT word1;
INT word2 = word1;
```



300

Equivalencing STRING Simple Variables

You can equivalence a STRING variable (S2) to a previous STRING variable (S1):

```
STRING s1 := "A";
STRING s2 = s1;
```

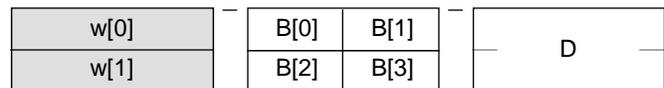


301

Equivalencing Mixed Simple Variables

The data type of the new variable can differ from the data type of the previous variable. For example, you can equivalence STRING and INT(32) variables to a previous INT array. (The middle box looks like an array, but it is an equivalenced STRING simple variable shown with indexes.)

```
INT w[0:1];
STRING b = w[0];
INT(32) d = b;
```

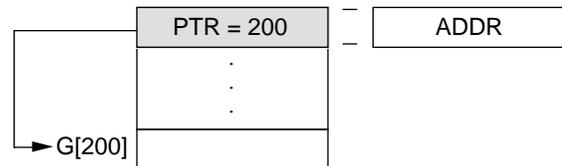


302

Equivalencing Simple Variables to Simple Pointers

If you equivalence a simple variable to a simple pointer, the simple variable is equivalenced to the location occupied by the simple pointer, not to the location whose address is stored in the pointer:

```
INT .ptr := 200;
INT addr = ptr;
```



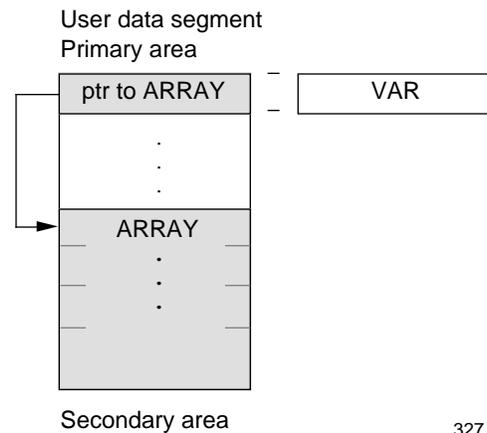
332

Avoid Equivalencing Simple Variables to Indirect Variables

If you equivalence a simple variable to an indirect variable, the simple variable is equivalenced to the location occupied by the implicit pointer, not to the location of the data whose address is stored in the implicit pointer.

Therefore, avoid equivalencing a simple variable to a standard indirect array:

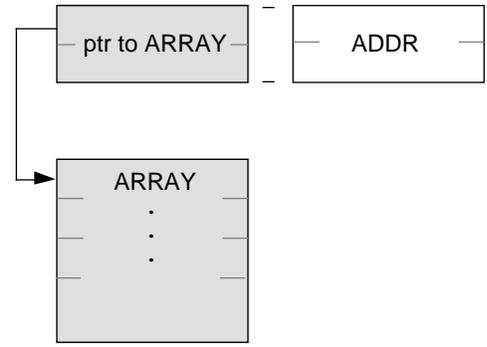
```
INT .array[0:9];
INT var = array;
```



327

Also avoid equivalencing a simple variable to an extended indirect array:

```
INT .EXT array[0:999];
INT(32) addr = array;
```



309

Avoid Odd-Byte Equivalencing

If you equivalencing a `STRING` variable to an odd-byte array element, the compiler converts the odd-byte index to the previous word boundary as shown in the diagram and issues a warning.

Avoid the following practice:

```
STRING a[0:1];
STRING b = a[1];
```



303

Avoid Equivalenced Arrays

You cannot equivalencing arrays to other variables. Avoid the following practice:

```
INT a[0:5];
INT b;
INT c[0:5] = a;           !Error
INT d[0:5] = b;           !Error
```

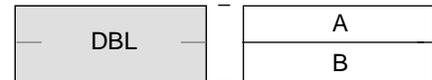
Accessing Equivalenced Simple Variables

Once you have declared an equivalenced variable, you access it in the same way as you access any other variable, by specifying its identifier in a statement.

Accessing Words in Doublewords

You can declare two INT variables, each equivalent to one of the two words of an INT(32) variable. You can then access the location either as an INT variable or as an INT(32) variable:

```
INT(32) dbl;
INT a = dbl;
INT b = a[1];
```



```
a := 0;      !Access first word of DBL
dbl := -1D; !Access DBL as a doubleword
```

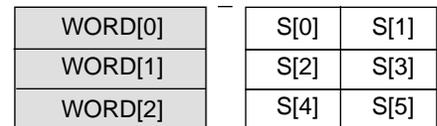
304

Accessing Bytes in Words

You can equivalence a STRING variable to an INT array, and then access bytes in the INT array by indexing the STRING variable:

```
INT word[0:2];
STRING s = word;

s[3] := 0;
IF s[4] > 2 THEN ...;
```



305

Equivalencing Simple Pointers

You can equivalence a new simple pointer to a previously declared variable as listed in Table 10-1 earlier in this section.

Declaring Equivalenced Simple Pointers

To declare an equivalenced simple pointer, specify:

- Any *data type* except UNSIGNED
- The *identifier* of the new simple pointer, preceded by an indirection symbol
- An equal sign (=)
- The identifier of the *previous variable*

For portability to future software platforms, declare equivalenced variables that fit within the previous variable.

Matching Byte or Word Addressing

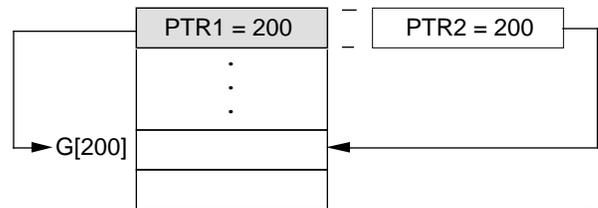
If the previous variable is a pointer, an indirect array, or an indirect structure, the previous pointer and the new pointer must both contain either:

- A standard byte address
- A standard word address
- An extended address

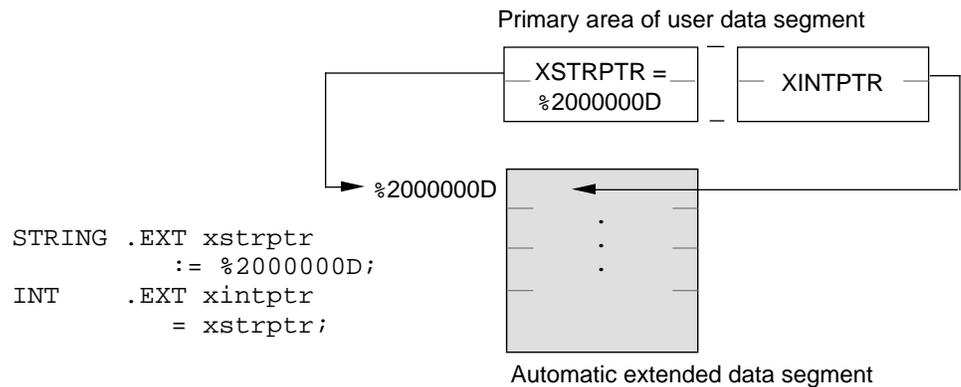
Otherwise, the pointers will point to different locations, even if they both contain the same value. That is, a standard STRING or extended pointer normally points to a byte address, and a standard pointer of any other data type normally points to a word address.

When you equivalence standard pointers, ensure that the byte or word addressing match. For example, INT and INT(32) standard pointers both contain a word address:

```
INT .ptr1 := 200;
INT(32) .ptr2 = ptr1;
```



When you equivalence extended pointers, you need not worry about byte and word addressing mismatches, because extended pointers always point to byte addresses. Thus, you can equivalence an extended INT pointer to an extended STRING pointer:



399

Avoid Mixing Byte and Word Addressing

If you equivalence a STRING standard pointer to an INT standard pointer, the INT pointer points to a word address and the STRING pointer points to a byte address.

Avoid the following practice:

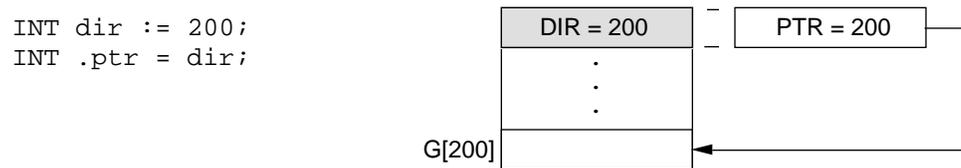


308

Equivalencing Simple Pointers to Direct Variables

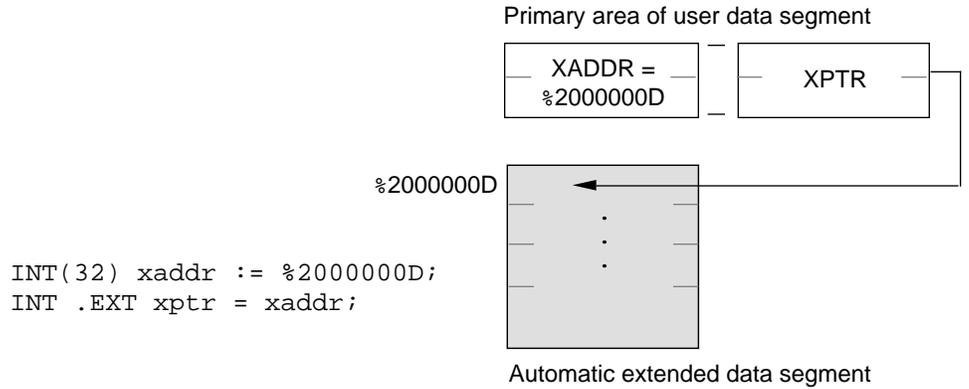
You can equivalence a simple pointer to a direct variable. The content of the simple variable becomes the address of the data to which the pointer points.

You can equivalence a standard pointer to a simple variable:



306

You can equivalence an extended pointer to an INT(32) simple variable:



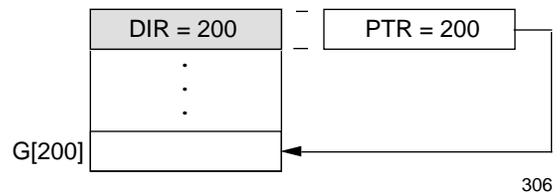
417

Accessing Equivalenced Simple Pointers

After you declare an equivalenced pointer, you can access it by specifying in a statement the identifier of the pointer or of the variable to which the pointer is equivalenced. Suppose you equivalence a simple pointer to a simple variable as follows:

```

INT dir := 200;
INT .ptr = dir;
    
```

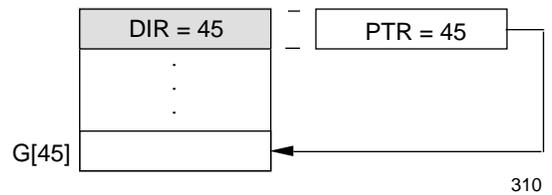


306

If you assign a value to the simple variable in the preceding example, you change the content of both the simple variable and the simple pointer:

```

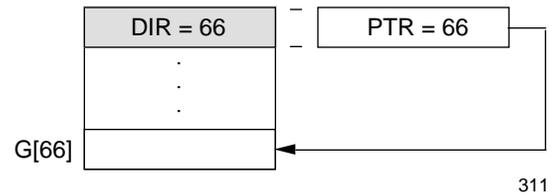
dir := 45;
    
```



310

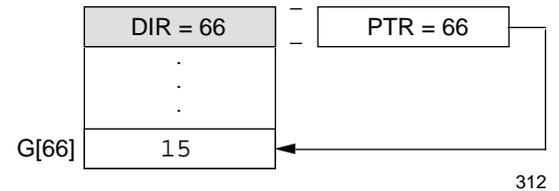
If you prefix @ to the pointer identifier in the preceding example and assign a value to the pointer, you change the content of both the direct variable and the simple pointer:

```
@ptr := 66;
```



If you assign a value to the simple pointer without the @ operator, you change the content of only the variable to which the simple pointer points. Location `G[66]` now contains 15:

```
ptr := 15;
```



Equivalencing Structures

You can equivalence a new definition or referral structure to a previously declared variable as listed in Table 10-1 earlier in this section.

If you want the new structure layout to occupy the same location as the previous variable, be sure that you match the addressing mode of the new structure and of the previous variable as follows:

New Structure	Previous Variable
Direct structure	Simple variable Direct structure Direct array
Standard indirect structure	Standard indirect structure Standard indirect array Standard structure pointer *
Extended indirect structure	Extended indirect structure Extended indirect array Extended structure pointer *

* If the previous variable is a pointer, the new structure is really a pointer.

Definition structures and referral structures are described separately in the following subsections.

Equivalencing Definition Structures

To declare an equivalenced definition structure, specify:

- The keyword **STRUCT**
- The *identifier* of the new structure, often preceded by an indirection symbol
- An equal sign (=)
- The identifier of the *previous variable*
- The *layout* of the new structure (enclosed in a BEGIN-END construct)

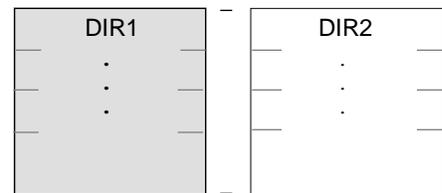
For portability to future software platforms, declare equivalenced variables that fit within the previous variable.

Equivalencing Direct Structures

You can declare a directly addressed definition structure equivalent to another directly addressed structure:

```
STRUCT dir1;
  BEGIN
  STRING name[0:20];
  STRING addr[0:50];
  END;

STRUCT dir2 = dir1;
  BEGIN
  STRING name[0:30];
  STRING addr[0:40];
  END;
```



313

Equivalencing Standard Indirect Structures

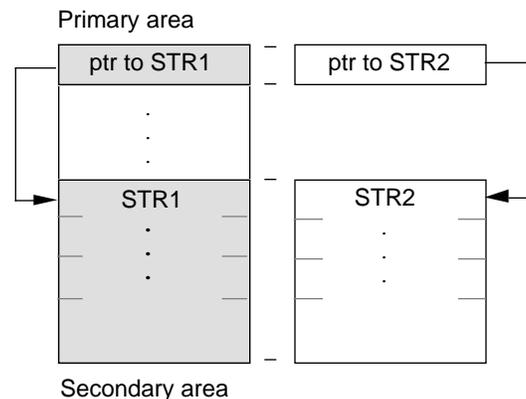
You can equivalence a standard indirect definition structure to another standard indirect structure.

For example, you can equivalence standard indirect structure STR2 equivalent to standard indirect structure STR1:

```
STRUCT temp(*);
  BEGIN
  STRING name[0:20];
  STRING addr[0:50];
  END;

STRUCT .str1 (temp);

STRUCT .str2 = str1;
  BEGIN
  STRING name[0:30];
  STRING addr[0:40];
  END;
```



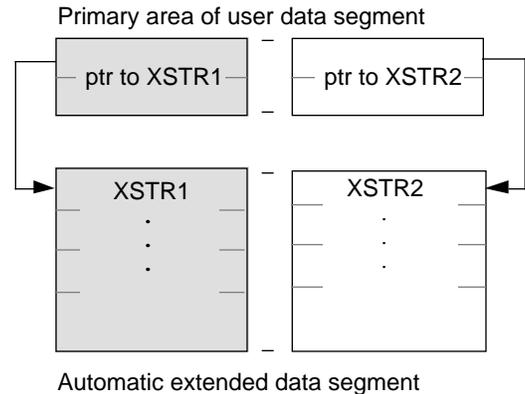
314

Equivalencing Extended Indirect Structures

You can equivalence an extended indirect definition structure to another extended indirect structure:

```
STRUCT .EXT xstr1;
BEGIN
STRING old_name[0:20];
STRING old_addr[0:50];
END;

STRUCT .EXT xstr2 = xstr1;
BEGIN
STRING new_name[0:30];
STRING new_addr[0:40];
END;
```



315

Equivalencing Structures to Simple Variables

You can equivalence a structure to a simple variable. For example, you can declare a two-word structure to an INT(32) simple variable:

```
INT(32) var;

STRUCT astruct = var;
BEGIN
INT high;
INT low;
END;
```



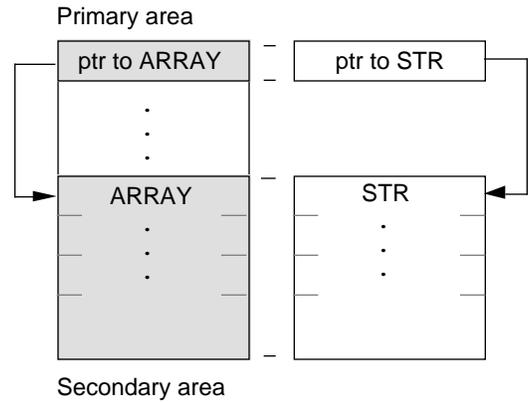
400

Equivalencing Structures to Arrays

You can equivalence a structure to an array. The following example equivalences a standard indirect structure to a standard indirect array:

```
STRING .array[0:127];

STRUCT .str = array;
BEGIN
  STRING name[0:63];
  STRING addr[0:63];
END;
```

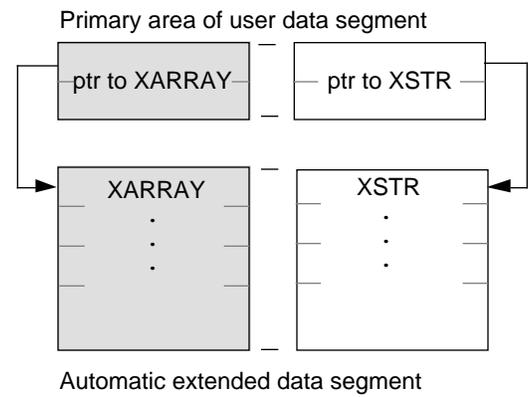


318

You can equivalence an extended indirect structure to an extended indirect array:

```
INT .EXT xarray[0:127];

STRUCT .EXT xstr = xarray;
BEGIN
  STRING name[0:30];
  STRING addr[0:40];
END;
```



401

Equivalencing Structures to Structure Pointers

If you equivalence an indirect structure to a structure pointer, the indirect structure behaves exactly like the structure pointer:

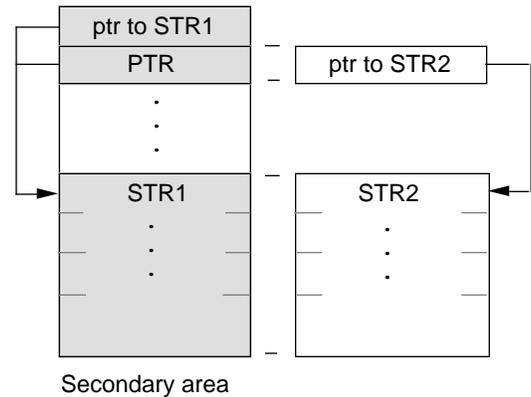
```

STRUCT .str1;
BEGIN
  INT a[0:3];
  INT b[0:9];
END;

INT .ptr (str1) := @str1;

STRUCT .str2 = ptr;
BEGIN
  STRING a2[0:7];
  STRING b2[0:19];
END;

```



317

Avoid Mixing Addressing Modes

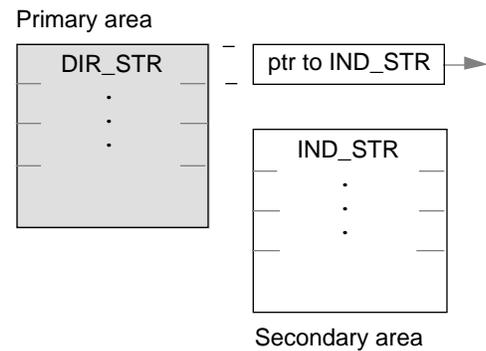
Do not equivalence an indirect structure to a direct variable, because the implicit pointer to the indirect structure cannot point to the structure data. Avoid the following practice:

```

STRUCT dir_str;
BEGIN
  STRING a[0:19];
  STRING b[0:49];
END;

STRUCT .ind_str = dir_str;
BEGIN
  INT a[0:9];
  INT b[0:24];
END;

```

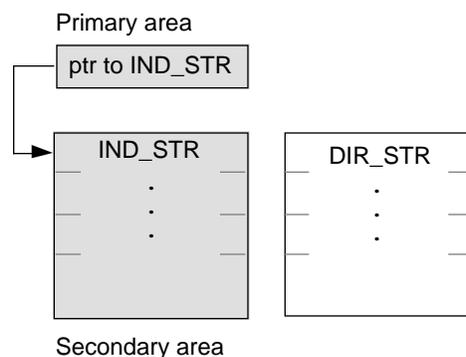


319

Avoid equivalencing a direct structure to an indirect variable, because the direct structure is not equivalenced to the indirect structure. Avoid the following practice:

```
STRUCT .ind_str;
BEGIN
  STRING a[0:19];
  STRING b[0:49];
END;

STRUCT dir_str = ind_str;
BEGIN
  INT a[0:9];
  INT b[0:24];
END;
```



402

Accessing Equivalenced Definition Structures

You access equivalenced definition structures as you do any other structure. That is, you qualify the structure name with the appropriate structure items, as described in Section 8, “Using Structures.”

Equivalencing Referral Structures

To declare an equivalenced referral structure, specify:

- The keyword **STRUCT**
- The *identifier* of the new structure, often preceded by an indirection symbol
- A *referral* that associates the new structure with a structure layout—enclose in parentheses the identifier of a previously declared structure or structure pointer
- An equal sign (=)
- The identifier of the *previous variable*

For portability to future software platforms, declare equivalenced variables that fit within the previous variable.

All other equivalencing guidelines described for equivalenced definition structures apply to equivalenced referral structures.

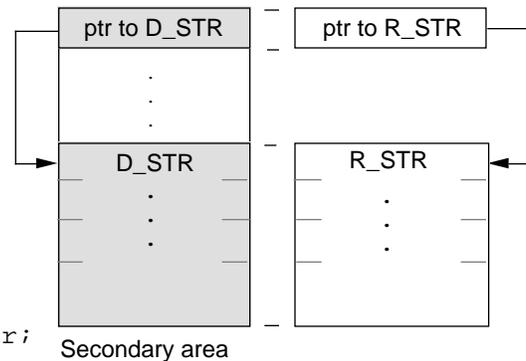
Equivalencing Referral Structures to Definition Structures

You can equivalence a referral structure to a previous definition structure. In the following example, both are standard indirect structures:

```
STRUCT .d_str;
  BEGIN
  STRING name[0:19];
  STRING address [0:49];
  END;
```

```
STRUCT tmp (*);
  BEGIN
  INT name[0:9];
  INT address[0:24];
  END;
```

```
STRUCT .r_str (tmp) = d_str;
```



316

Accessing Equivalenced Referral Structures

You access equivalenced definition structures as you do any other structure. That is, you qualify the structure name with the appropriate structure items, as described in Section 8, "Using Structures."

Equivalencing Structure Pointers

You can equivalence a new structure pointer to a previously declared variable as listed in Table 10-1 earlier in this section.

Be sure that you match the addressing mode of the new structure pointer and of the previous variable as follows:

New Structure Pointer	Previous Variable
Standard structure pointer	Simple variable
	Direct array
	Standard indirect structure
	Standard indirect array
	Standard structure pointer
Extended structure pointer	Simple variable
	Direct array
	Extended indirect structure
	Extended indirect array
	Extended structure pointer

Declaring Equivalenced Structure Pointers

To declare an equivalenced structure pointer, specify:

- `STRING` or `INT` attribute (as described in Table 9-3 in Section 9)
- The *identifier* of the new structure pointer, preceded by an indirection symbol
- A *referral* that provides a structure layout—enclose the identifier of a previously declared structure or structure pointer in parentheses
- An equal sign (=)
- The identifier of the *previous variable*

For portability to future software platforms, declare equivalenced variables that fit within the previous variable.

Equivalencing Structure Pointers to Structure Pointers

You can declare a structure pointer equivalent to another structure pointer as follows:

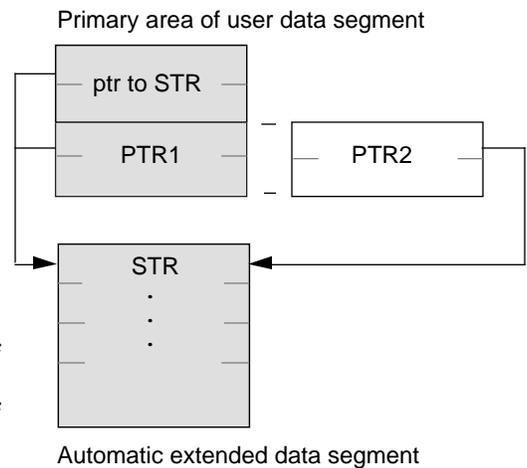
```

STRUCT temp (*);
BEGIN
  STRING s[0:71];
END;

STRUCT .EXT str;
BEGIN
  STRING name[0:20];
  STRING addr[0:50];
END;

INT .EXT ptr1 (str) := @str;

INT .EXT ptr2 (temp) = ptr1;
    
```

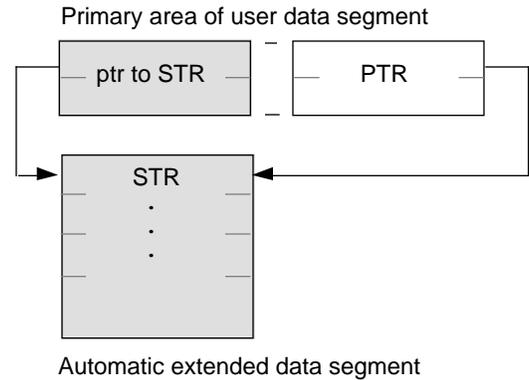


Equivalencing Structure Pointers to Structures

Equivalencing an extended structure pointer to an extended indirect structure is the same as equivalencing an indirect structure to another indirect structure.

```
STRUCT .EXT str;
BEGIN
  STRING name[0:20];
  STRING addr[0:50];
END;

INT .EXT ptr (str) = str;
```



321

Equivalencing Structure Pointers to Simple Variables or Arrays

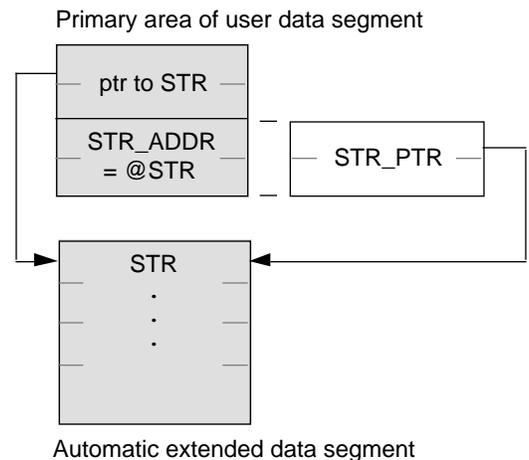
Before you equivalencing a structure pointer to a simple variable or an element of a direct array, make sure that the simple variable or array element contains the address of a structure:

```
STRUCT temp (*);
BEGIN
  STRING s[0:71];
END;

STRUCT .EXT str;
BEGIN
  STRING name[0:20];
  STRING addr[0:50];
END;

INT(32) str_addr := @str;

INT .EXT str_ptr (temp)
  = str_addr;
```



403

Matching Byte or Word Addressing

All guidelines for matching byte or word addressing given earlier in this section apply to equivalenced structure pointers.

Avoid Mixing Addressing Modes

All guidelines for avoiding mixed address modes given earlier in this section apply to equivalenced structure pointers.

Accessing Data Through Equivalenced Structure Pointers

To access a structure through a structure pointer, qualify the name of the pointer with the appropriate structure item names, as was described in Section 9, "Using Pointers."

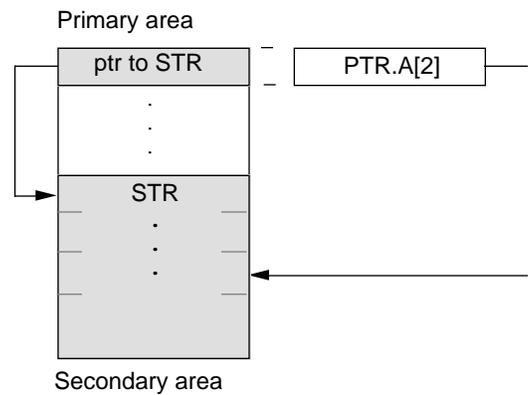
This example accesses item A[2] in structure STR through structure pointer PTR:

```

STRUCT .str;
BEGIN
  INT a[0:3];
  INT b[0:9];
END;

INT .ptr (str) = str;

ptr.a[2] := %14;
!Assign value to
! STR.A[2]
    
```



322

Avoiding an Access Error

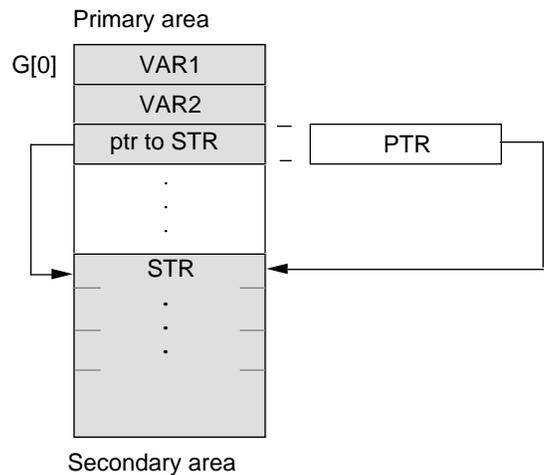
As previously discussed, you can equivalence a structure pointer to an indirect structure. Both the structure pointer and the implicit pointer to the indirect structure then point to the data of the structure.

```

INT var1;
INT var2;

STRUCT .str;
BEGIN
  INT a[0:3];
  INT b[0:9];
END;

INT .ptr (str) = str;
    
```



323

However, if you assign an address to the structure pointer in the preceding example, both the structure pointer and the implicit pointer point to the new address, rather than to the structure data. You can no longer access the structure data.

Avoid the last statement in the following example:

```

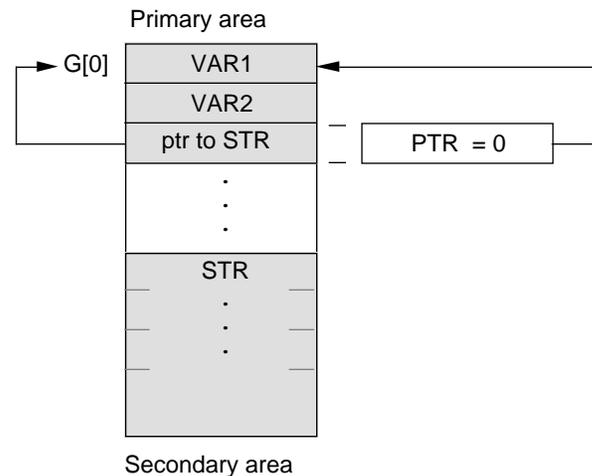
INT var1;
INT var2;

STRUCT .str;
BEGIN
  INT a[0:3];
  INT b[0:9];
END;

INT .ptr (str) = str;

@ptr := 0;

```



418

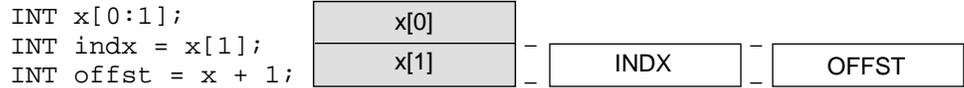
Using Indexes or Offsets

When you declare equivalenced variables, you can append an *index* or an *offset* to the previously declared variable. Table 10-2 compares indexes and offsets.

Table 10-2. Indexes and Offsets

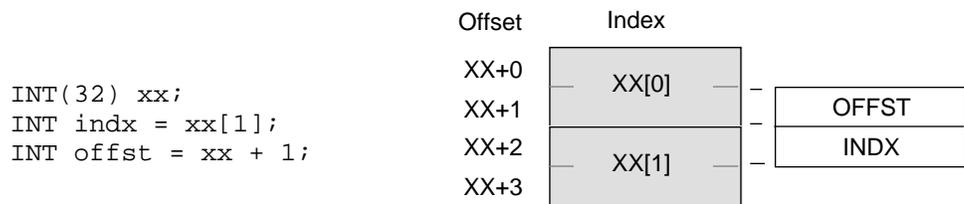
	Specified As:	Used With	Represents
Index	INT constant (enclosed in brackets)	Direct previous variable	An element offset from the location of the previous variable (which cannot be a pointer, including an implicit pointer). The element size depends on the data type of the previous variable. The indexed location must begin on a word boundary.
Offset	INT constant (preceded by a plus (+) or minus (-))	Direct or indirect previous variable	A word offset from either: –The location of a direct previous variable. –The location of the pointer of an indirect previous variable, not from the location of the data pointed to.

For direct INT previous variables, indexes and offsets both have word offsets:



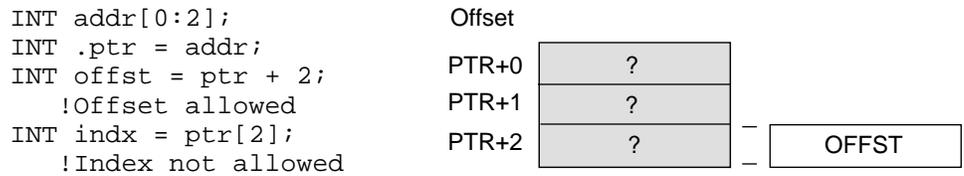
324

For direct previous variables other than INT, indexes have element offsets and offsets have word offsets:



325

You can equivalence a variable to an offset simple pointer but not to an indexed simple pointer:



326

**Emulating Pascal
Variant Parts**

You can simulate variant parts of a Pascal record type by using equivalenced structures as follows. (For information on record type variant parts, see the *Pascal Reference Manual*.)

```
LITERAL triangle, rectangle, circle;

STRUCT geometry (*);
  BEGIN
  REAL x, y;
  REAL area;
  INT shape;                                ! 0 = TRIANGLE,
                                           ! 1 = RECTANGLE,
                                           ! 2 = CIRCLE

  STRUCT triangle_info;                    !If SHAPE = TRIANGLE
  BEGIN
  REAL sidel, side2, side3;
  REAL(64) angle1, angle2, angle3;
  END;

  STRUCT rectangle_info = triangle_info;
                                           !If SHAPE = RECTANGLE

  BEGIN
  REAL sidel, side2;
  END;

  STRUCT circle_info = triangle_info;
                                           !If SHAPE = CIRCLE

  BEGIN
  REAL diameter;
  END;

END;
```

11 Using Procedures

Procedures are program units that can contain the executable portions of a TAL program and that are callable from anywhere in the program. Procedures let you segment a program into discrete parts that each perform a particular operation such as I/O or error-handling.

A procedure can call other procedures, one at a time, or it can call another procedure that in turn calls another, and so on. It can pass parameters (arguments). It can contain subprocedures, which are callable from various points within the same procedure.

A function is a procedure or subprocedure that returns a value to the caller. A function is also known as a typed procedure or a typed subprocedure.

This section discusses:

- How to declare and call procedures and subprocedures
- How to declare and pass parameters
- How the compiler allocates storage for procedures and parameters
- How the compiler provides parameter masks
- How to declare and use labels
- How to get the addresses of procedures and subprocedures

Procedures and Code Segments

The procedures in a program are located in one or more code segments within the user code space. A procedure can call a procedure that is located:

- In the same (or current) segment
- In another segment of the same code space
- In the system code segment
- In a system library segment
- In a user library segment

The starting address of a procedure is known as the entry point. The operating system records entry points in the following system tables:

System Table	Content of Table	Location of Table
Procedure Entry Point (PEP) table	Entry points of all procedures located in the current code segment	At the beginning of the current code segment
External Entry Point (XEP) table	Entry points of procedures located in other code segments	Near the end of the current code segment

When a procedure calls another procedure in the current segment, control passes to the called procedure through the segment's PEP table.

When a procedure calls a procedure in another code segment, control passes out of the current segment through its XEP table to the called procedure through the other segment's PEP table.

You can declare procedures that are up to 32K words in size, minus the size of either:

- The PEP table (for procedures in the lower 32K-word area of the code segment)
- The XEP table (for procedures in the upper 32K-word area of the code segment)

Declaring Procedures To declare a procedure in its simplest form, specify:

- The keyword `PROC`
- The procedure *identifier*, followed by a semicolon
- The procedure *body*—a BEGIN-END construct that can include data declarations, subprocedure declarations, and statements

Here is an example of a procedure declaration:

```
PROC my_proc;                                !Declare procedure MY_PROC
  BEGIN
  INT var;                                    !Declare local data
  !Declare subprocedures, if any
  var := 99;                                  !Specify local statements
  CALL some_proc;
  END;                                         !End MY_PROC
```

Calling Procedures When you run your program, a procedure is activated by a `CALL` statement in another procedure or subprocedure (the caller).

For example, a caller can call the preceding procedure (`MY_PROC`) as follows:

```
PROC caller_proc;                            !Declare CALLER_PROC
  BEGIN
  !Lots of code
  CALL my_proc;                               !Call MY_PROC
  !More code
  END;                                         !End CALLER_PROC
```

When the called procedure finishes executing, control returns to the statement following the `CALL` statement in the caller.

Declaring Parameters in Procedures You can include formal parameters in the procedure declaration. Callers can then pass corresponding actual parameters for use in the called procedure.

Specifying a Formal Parameter List To include formal parameters in a procedure declaration, specify a comma-separated list (enclosed in parentheses) of up to 32 formal parameters.

```
PROC proc_a (param1, param2); !Declare PROC_A; include
                               ! formal parameter list
```

Declaring Formal Parameters After specifying the formal parameter list, declare each formal parameter by specifying:

- A *parameter type* (for example, STRING, INT, STRUCT, or PROC). A procedure can have any number of parameter words.
- The *identifier* of the parameter.
- If the parameter is a reference parameter, precede the identifier with an indirection symbol (. or .EXT). The absence of an indirection symbol denotes a value parameter. Table 11-1 describes value and reference parameters.

Table 11-1. Value and Reference Parameters

Formal Parameter	Description	Indirection Symbol	Passed Value
Value	The caller passes a value. The called procedure cannot change the original value in the caller's scope.	No	Simple variable, constant expression, or procedure
Reference	The caller passes the address of a value. The called procedure can change the original value in the caller's scope.	Yes	Address of simple variable, array, or structure

For example, you can declare simple variables as value and reference formal parameters. The compiler treats the value parameter as if it were a simple variable. The compiler treats the reference parameter as if it were a simple pointer:

```
PROC proc_x (val_param, ref_param); !Include parameter list
    INT val_param;                 !Declare value parameter
    INT .ref_param;                 !Declare reference parameter
BEGIN
    ref_param := val_param + ref_param;
                                   !Manipulate parameters
END;
```

For more information on parameter specifications, see Table 11-3 later in this section.

Passing Actual Parameters The formal parameter declarations in the called procedure dictates how the calling procedure declares the actual parameters. For example, if the called procedure declares a simple variable as a formal reference parameter, the calling procedure must declare a simple pointer as the actual parameter.

The calling procedure passes actual parameters by specifying an actual parameter list in a CALL statement. In the following example, PROC_Y calls PROC_X (declared in the previous example). PROC_Y passes VAR1 by value and VAR2 by reference as dictated by the formal parameter declarations in PROC_X:

```
PROC proc_y;                                !Declare PROC_Y
BEGIN
  INT var1 := 50;                            !Declare simple variable
  INT .var2 := 20;                          !Declare simple pointer
  CALL proc_x (var1, var2);                 !Call PROC_X and pass
  END;                                       !actual parameters to it
```

When a CALL statement occurs, the compiler assigns each actual parameter to a formal parameter in order. More information on parameters is provided in “Using Parameters” later in this section.

Declaring Data in Procedures

Data items you declare before the first procedure declaration have global scope. You can access global data from any procedure in the program.

Data items you declare within a procedure have local scope. You can access local data only from within the same procedure.

A local data item can have the same identifier as a global data item. In this case, however, you cannot access the global data item from within that procedure. For example, if you declare global variables A and B and local variables A and B, the procedure can access local variables A and B but not global variables A and B. If you declare a variable named C only at the global level, the procedure can access C:

```
INT a := 9;                                !Declare global A, B, and C
INT b := 3;
INT c;

PROC a_proc MAIN;                          !Declare A_PROC
BEGIN
  INT a := 4;                               !Declare local A and B
  INT b := 1;
  c := a + b;                               !Access local A and B
  END;                                       !End A_PROC
```

Allocating Local Variables When a procedure is activated, the compiler allocates storage for the procedure's variables in the procedure's private local storage area, which consists of a primary and a secondary storage area.

Local Primary Area

The primary area for each procedure can store up to 127 words of the following kinds of local variables:

- Directly addressed simple variables, arrays, and structures
- Pointers you declare
- Implicit pointers (those the compiler provides when you declare indirect arrays or indirect structures)

When a procedure is activated, the compiler allocates storage at the current top of the data stack for each direct local variable and each pointer. The addresses of the variables are at an increasingly higher offset from L[1].

Local Secondary Area

The local secondary area begins immediately after the last pointer or direct variable of the procedure's local primary area. The local secondary area of a procedure has no explicit size, but the total of all global and local primary and secondary areas cannot exceed the lower 32K-word area of the user data segment.

For each standard indirect array or structure you declare in the procedure:

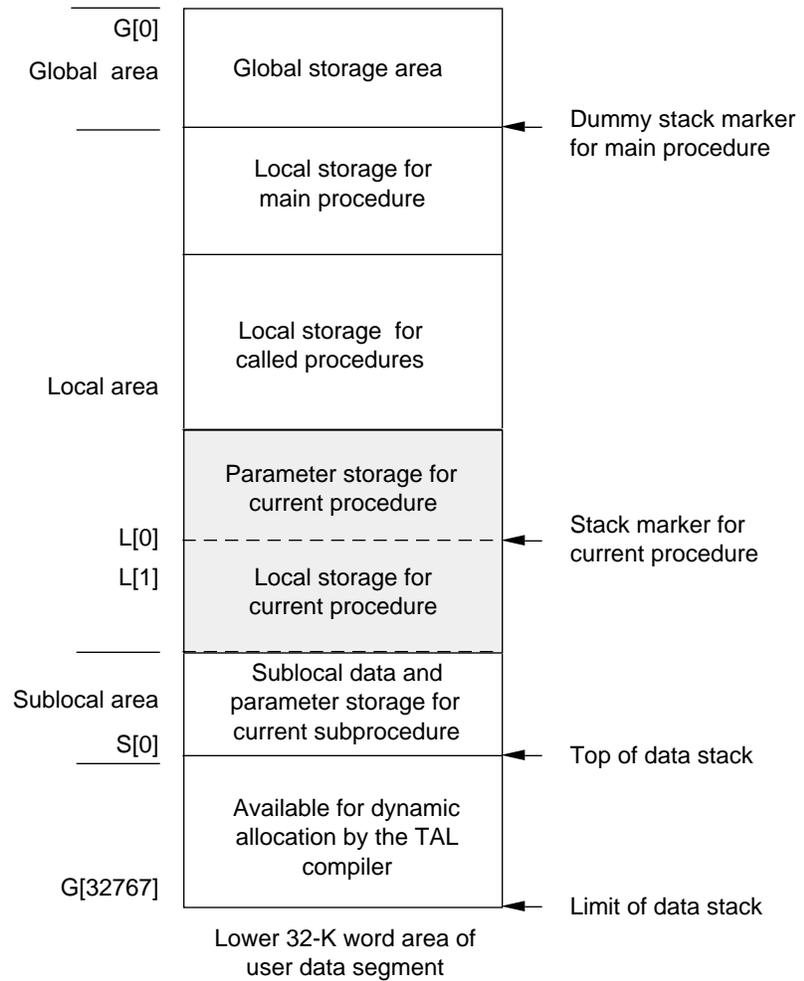
1. The compiler provides an implicit standard pointer and allocates a word of storage for the pointer in the procedure's local primary area.
2. The compiler allocates storage for the array or structure in the procedure's secondary area, which begins immediately following the procedure's last direct item.

If you declare indirect arrays or indirect structures within BLOCK declarations, however, the compiler allocates such data blocks anywhere in the procedure's secondary data blocks, as described in Section 14, "Compiling Programs."

3. The compiler initializes the implicit pointer (provided in step 1) with the address of the array or structure in the procedure's secondary area.
 - For a STRING array, the pointer contains the byte address of the array.
 - For any other array, the pointer contains the word address of the array.
 - For a structure, the pointer contains the word address of the structure.

Allocating Parameters and Variables Figure 11-1 shows how the compiler allocates storage for the called procedure's parameters and variables in the procedure's local primary and secondary storage areas.

Figure 11-1. Procedure Data Allocation



 Data allocation for current procedure

Returning From a Procedure

A called procedure returns control to the caller when a RETURN statement is executed or the last END keyword is reached. (In a nonfunction procedure, a RETURN statement is optional.) In the following example, MY_PROC returns control when A is less than B, which invokes the RETURN statement, or when the last END is reached:

```
PROC my_proc;
  BEGIN
  INT a, b;
  !Lots of code
  IF a < b THEN
    RETURN;                !Return to caller if A < B
  !Lots more code
  END;                    !Last END keyword
```

Using Procedure Options

By specifying procedure declaration options, you can declare:

- The MAIN procedure
- Functions
- FORWARD procedures
- EXTERNAL procedures
- VARIABLE procedures
- EXTENSIBLE procedures

Other procedure declaration options include:

- Public name and LANGUAGE options, described in Section 17, “Mixed-Language Programming.”
- INTERRUPT, RESIDENT, CALLABLE, and PRIV options, described in the *TAL Reference Manual*. Most of these options are used only by system procedures.

Declaring the MAIN Procedure

An executable program requires a MAIN procedure. It is the first procedure executed when you run the program (although it often appears last.) The MAIN procedure cannot have parameters. If more than one MAIN procedure appears in a program, the compiler issues a warning and uses the first one it encounters as the MAIN procedure. When the MAIN procedure completes execution, it passes control to the PROCESS_STOP_ system procedure, rather than executing an EXIT instruction.

To declare the MAIN procedure, include the MAIN keyword following the procedure identifier:

```
PROC mymain MAIN;                !Declare MAIN procedure
  BEGIN
  !Lots of code
  CALL some_procedure;
  END;
```

Declaring Functions A function is a procedure or subprocedure that returns a result to the caller. You declare a function as you would a procedure, plus you specify:

- The *data type* of the result value to be returned by the function
- A RETURN statement that returns the result (and optionally a condition code)

For example, you can declare a function that has two formal parameters, multiplies them, and returns an INT result to the caller:

```
INT PROC mult_function (var1, var2); !INT return type
    INT var1, var2;           !Declare formal parameters
BEGIN
RETURN var1 * var2;         !RETURN statement returns result
END;                         ! and control to caller
```

Callers normally invoke functions by using the function identifiers in expressions. For example, an assignment statement in a procedure can invoke the preceding MULT_FUNCTION, passing two actual parameters to it:

```
PROC caller;                !Declare CALLER
BEGIN
    INT num1 := 5,
        num2 := 3,
        answer;
    answer := mult_function (num1, num2);
                                !Assignment statement invokes
END;                           ! MULT_FUNCTION
```

For information on returning condition codes, see the RETURN statement in Section 12, “Controlling Program Flow.”

Declaring FORWARD Procedures A FORWARD procedure declaration lets you call a procedure before you declare the procedure. You can then declare the procedure anywhere in the same compilation unit.

To declare a FORWARD procedure, specify the FORWARD keyword in place of the procedure body. For example, you can declare PROC1 as a FORWARD procedure, declare PROC2 which calls PROC1, and finally declare the body of PROC1:

```
PROC proc1 (param1, param2); !Declare PROC1 as a FORWARD
    INT .param1, param2;     ! procedure
    FORWARD;

PROC proc2 MAIN;            !Declare PROC2
BEGIN
    INT i1 := 1;
    CALL proc1 (i1, 2);      !Call PROC1
END;

PROC proc1 (param1, param2); !Declare the real PROC1
    INT .param1, param2;
BEGIN
    param1 := param1 - param2;
END;
```

Declaring EXTERNAL Procedures An EXTERNAL procedure declaration lets the current compilation unit call a procedure that is declared in another compilation unit. The other compilation unit can be a system library, a user library, or a user file.

To declare an EXTERNAL procedure, specify the EXTERNAL keyword in place of the procedure body:

```
PROC PROCESS_DEBUG_; EXTERNAL; !Declare EXTERNAL procedure
PROC issue_warning (param);
    INT param;
    EXTERNAL;                !Declare EXTERNAL procedure

PROC a MAIN;
    BEGIN
    INT x, y, z;
    !Manipulate X, Y, and Z
    If x = 5 THEN CALL issue_warning (5);
                                !Call EXTERNAL procedure
    CALL PROCESS_DEBUG_;        !Call EXTERNAL procedure
    END;
```

Declaring VARIABLE Procedures For a VARIABLE procedure, the compiler treats all formal parameters as if they were optional, even if some are required by the procedure's code. If you add new parameters to a VARIABLE procedure, all procedures that call it must be recompiled.

To declare a VARIABLE procedure, specify:

- The *data type* of the return value (optional)
- The keyword PROC
- The procedure *identifier*
- The formal *parameter list*, enclosed in parentheses
- The keyword VARIABLE, followed by a semicolon
- The formal *parameter declarations*, each followed by a semicolon
- The procedure *body*—a BEGIN-END construct that can include data declarations, subprocedure declarations, and statements

Following is an example of a VARIABLE procedure declaration:

```
PROC v (a, b) VARIABLE;        !Declare VARIABLE procedure
    INT a;                    !Declare formal parameters
    INT b;
    BEGIN
    !Lots of code
    END;
```

Checking for Actual Parameters

Each VARIABLE procedure must check for the presence or absence of actual parameters that are required by the procedure's code. The procedure can use the \$PARAM standard function to check for required or optional parameters.

For example, the following VARIABLE procedure returns control to the caller if either required parameter is absent. Also the procedure provides a default 0 to use if the optional parameter is absent.

```
PROC errmsg (msg, count, errnum) VARIABLE;
    INT .msg;                !Required by your code
    INT count;              !Required by your code
    INT errnum;             !Optional
BEGIN
IF NOT $PARAM (msg) OR      !Check for required parameters
    NOT $PARAM (count) THEN
    RETURN;                 !Return to caller if either
                            ! required parameter is absent
IF NOT $PARAM (errnum) THEN !Check for optional parameter
    errnum := 0;           !Use 0 if optional parameter
                            ! is absent

    !Process the error
END;
```

Declaring EXTENSIBLE Procedures

An EXTENSIBLE procedure lets you add new formal parameters to it without recompiling callers unless the callers use the new parameters. The compiler treats all parameters of an EXTENSIBLE procedure as if they were optional, even if some are required by the procedure's code.

To declare an EXTENSIBLE procedure, specify:

- The *data type* of the return value (optional)
- The keyword PROC
- The procedure *identifier*
- The formal *parameter list*, enclosed in parentheses
- The keyword EXTENSIBLE, followed by a semicolon
- The formal *parameter declarations*, each followed by a semicolon
- The procedure *body*—a BEGIN-END construct that can include local data declarations, subprocedure declarations and statements

Following is an example of an EXTENSIBLE procedure declaration:

```
PROC x (a, b, c) EXTENSIBLE;    !Declare EXTENSIBLE procedure
    INT a;
    INT(32) b;
    FIXED c;
BEGIN
    !Process the parameter values
END;
```

Checking for Actual Parameters

Each EXTENSIBLE procedure must check for the presence or absence of actual parameters that are required by the procedure's code. The procedure can use \$PARAM to check for required or optional parameters.

In the following example, the EXTENSIBLE procedure returns control to the caller if either required parameter is absent. Also the procedure provides a default 0 to use if the optional parameter is absent.

```
PROC errmsg (msg, count, errnum) EXTENSIBLE;
  INT .msg;                !Required by your code
  INT count;               !Required by your code
  INT errnum;              !Optional
BEGIN
  IF NOT $PARAM (msg) OR   !Check for required parameters
    NOT $PARAM (count) THEN
    RETURN;                !Return to caller if either
                          ! required parameter is absent
  IF NOT $PARAM (errnum) THEN !Check for optional parameter
    errnum := 0;           !Use 0 if optional parameter
                          ! is absent

  !Process the error
END;
```

Passing Parameters to VARIABLE or EXTENSIBLE Procedures

When you call a VARIABLE or EXTENSIBLE procedure, you can omit parameters indicated as being optional in the called procedure. You can pass or omit such parameters unconditionally or conditionally.

Passing Parameters Unconditionally

To pass parameters or parameter pairs unconditionally, specify the parameter name in the CALL statement parameter list. To omit parameters or parameter pairs unconditionally, use a place-holding comma for each omitted parameter or parameter pair up to the last specified parameter.

Comments within a CALL statement can help you keep track of which actual parameters you are omitting. Here is an example:

```
PROC some_proc (index, error_num, length, limit, total)
  EXTENSIBLE;
  INT index, error_num, length, limit, .total;
BEGIN
  !Lots of code
END;

PROC caller_proc;
BEGIN
  INT total;
  !Some code
  CALL some_proc (0, !error_num!, !length!, 40, total);
                          !Comments within CALL statement
                          ! identify omitted parameters

  !More code
END;
```

Passing Parameters Conditionally

As of the D20 release, you can pass a parameter or parameter pair to a VARIABLE or EXTENSIBLE procedure based on a condition at execution time by using the \$OPTIONAL function. \$OPTIONAL is evaluated each time the encompassing CALL statement is executed:

- If the conditional expression is true, the actual parameter is passed.
- If the conditional expression is false, the actual parameter is not passed.

In the following example, parameter pair S:I is passed because I equals 1, which is less than 9. Parameter J is not passed because J equals 1, which is not greater than 2.

```
PROC p1 (str:len, b) EXTENSIBLE;
    STRING .str;
    INT len;
    INT b;
BEGIN
    !Lots of code
END;

PROC p2;
BEGIN
    STRING .s[0:79];
    INT i:= 1;
    INT j:= 1;
    CALL p1 ($OPTIONAL (i < 9, s:i),    !Pass S:I if I < 9.
            $OPTIONAL (j > 2, j) );    !Pass J if J > 2.
END;
```

You can use \$OPTIONAL and \$PARAM when one procedure provides a front-end interface for another procedure that does the actual work:

```
PROC p1 (i, j) EXTENSIBLE;
    INT .i;
    INT .j;
BEGIN
    !Lots of code
END;

PROC p2 (p, q) EXTENSIBLE;
    INT .p;
    INT .q;
BEGIN
    !Lots of code
    CALL p1 ($OPTIONAL ($PARAM (p), p ),
            $OPTIONAL ($PARAM (q), q ));
    !Lots of code
END;
```

A called procedure cannot distinguish between a parameter that is passed conditionally and one that is passed unconditionally. The execution time is shorter, however, when you pass or omit a parameter unconditionally.

To make your code portable to future software platforms, use \$OPTIONAL for each optional parameter when calling a VARIABLE or EXTENSIBLE procedure.

Converting VARIABLE Procedures to EXTENSIBLE

You can convert a VARIABLE procedure into an EXTENSIBLE procedure. When you do so, the compiler converts the VARIABLE parameter mask into an EXTENSIBLE parameter mask. Parameter masks are described later in this section.

Converting a VARIABLE procedure to EXTENSIBLE is the only way to add parameters to the procedure without recompiling all its callers. You can add new parameters at the time you convert the procedure or later.

You can convert any VARIABLE procedure that meets the following criteria:

- It has at least one parameter.
- It has at most 16 words of parameters.
- All parameters are one word long except the last, which can be a word or longer.

The size of a formal reference parameter is one word if the address mode is standard; the size is two words if the address mode is extended.

To convert an existing VARIABLE procedure to EXTENSIBLE, redeclare the procedure and add the following information:

- Any new formal parameters
- The keyword EXTENSIBLE
- The number of formal parameters in the VARIABLE procedure, specified as an INT *value* in the range 1 through 15 and enclosed in parentheses

The following example converts a VARIABLE procedure and adds a new formal parameter. The value 3 in parentheses specifies that the procedure had three formal parameters before the procedure was converted from VARIABLE to EXTENSIBLE:

```
PROC errmsg (msg, count, errnum, new_param) EXTENSIBLE (3);
                                     !Add NEW_PARAM to parameter list
    INT .msg;
    INT count;                         !Redeclare existing parameters
    INT errnum;
    INT new_param;                     !Declare NEW_PARAM
BEGIN
!Do something
END;
```

Comparing Procedures and Subprocedures

Procedures and subprocedures are program units that can contain executable parts of your program. A subprocedure is declared inside a procedure. A procedure can contain any number of subprocedures, but a subprocedure cannot contain another subprocedure. Neither procedures or subprocedures can contain another procedure.

You use procedures for operations needed throughout a program; procedures are callable from anywhere in the program. You use subprocedures for operations needed within a procedure; subprocedures are callable only from within the encompassing procedure.

Procedures and subprocedures can declare formal parameters and receive data from other procedures and subprocedures. The same procedure or subprocedure can process different sets of data. The system allocates a private data area for each activation of a procedure or subprocedure and deallocates that area when control returns to the caller.

When a procedure or subprocedure calls a procedure (or when a subprocedure calls a subprocedure), the system saves the environment of the caller and restores it when the called procedure or subprocedure completes execution.

When a procedure calls a subprocedure, the caller's environment remains in place while the subprocedure executes.

Table 11-2 compares the characteristics of procedures and subprocedures.

Table 11-2. Procedures and Subprocedures

Characteristic	Procedure	Subprocedure
Can have formal parameters	Yes, except MAIN procedure	Yes
Can be a function and return a value	Yes	Yes
Can be recursive; it can call itself	Yes	Yes
Private primary storage	127 words local storage	32 words sublocal storage
Private secondary storage	Yes	No
Scope of procedure or subprocedure	Global	Local
Scope of data	Local	Sublocal
Can refer to which level of variables	Global or local	Global, local, or sublocal
Attributes	MAIN VARIABLE EXTENSIBLE RESIDENT CALLABLE PRIV INTERRUPT LANGUAGE (D-series system)	VARIABLE
Other options	FORWARD EXTERNAL	FORWARD

Declaring and Calling Subprocedures

To declare a subprocedure in its simplest form, specify:

- The keyword SUBPROC
- The *identifier* of the subprocedure, followed by a semicolon
- A subprocedure *body*—a BEGIN-END construct that can contain sublocal data declarations and statements

The following example declares MY_SUBPROC within MY_PROC. A CALL statement in MY_PROC then calls MY_SUBPROC:

```
PROC my_proc;                !Declare MY_PROC
  BEGIN
    !Declare local data here

    SUBPROC my_subproc;      !Declare MY_SUBPROC
      BEGIN
        !Declare sublocal data here
        !Specify sublocal statements here
      END;                    !End MY_SUBPROC

    !Specify local statements here
    CALL my_subproc;         !Call MY_SUBPROC
  END;                        !End MY_PROC
```

You can declare FORWARD, VARIABLE, or function subprocedures in the same way as described for procedures (but inside a procedure).

Including Formal Parameters

Subprocedures have a 32-word storage area for all sublocal data including variables, temporary results, parameters, and parameter mask if any. In the following example, MAIN_PROC contains subprocedures SUB1 and SUB2. MAIN_PROC calls SUB2. SUB2 calls SUB1 and passes parameters to it:

```
PROC main_proc MAIN;        !Declare MAIN_PROC
  BEGIN
    INT c := 0;

    SUBPROC sub1 (param1);   !Declare SUB1
      INT param1;           !Declare formal parameter
      BEGIN
        INT a := 5;
        INT b := 2;
        param1 := a + b + c;
      END;                  !End of SUB1

    SUBPROC sub2;            !Declare SUB2
      BEGIN
        INT var := 89;
        CALL sub1 (var);     !Call SUB1
      END;                  !End of SUB2

    CALL sub2;
  END;                      !End of MAIN_PROC
```

Sublocal Variables Because each subprocedure has only a 32-word storage area for variables, declare large arrays or structures at the global or local level and make them indirect. In subprocedures, declare only pointers and directly addressed variables as follows:

Valid Sublocal Variables	Example
Simple variables	<code>INT var;</code>
Arrays	<code>INT array[0:5];</code>
Read-only arrays	<code>INT ro_array = 'P' := [0,1,2,3,4,5];</code>
Structures	<code>STRUCT struct_a; BEGIN INT a, b, c; END;</code>
Simple pointers	<code>INT .simple_ptr;</code>
Structure pointers	<code>STRING .struct_ptr (struct_a);</code>

If you specify an indirection symbol (. or .EXT) when you declare sublocal arrays and structures, the compiler allocates direct arrays and structures and issues a warning.

Invalid Sublocal Variables	Example
Indirect arrays	<code>INT .EXT ext_array[0:5];</code>
Indirect structures	<code>STRUCT .EXT ext_struct; BEGIN INT a, b, c; END;</code>

Visibility of Identifiers Sublocal statements can normally refer to sublocal identifiers in the subprocedure, local identifiers in the procedure, and global identifiers. If, however, you declare the same identifiers (such as A and B) at the sublocal, local, and global levels, the subprocedure can access only the sublocal identifiers (A and B):

```

INT a := 9;                                !Declare global A and B
INT b := 3;

PROC a_proc MAIN;                          !Declare A_PROC
BEGIN
  INT a := 4;                              !Declare local A and B
  INT b := 1;
  INT c;

  SUBPROC a_subproc (param);               !Declare A_SUBPROC
  INT param;
  BEGIN
    INT a := 5;                            !Declare sublocal A and B
    INT b := 2;
    c := a + b + param;                    !Access sublocal A and B
  END;                                     !End A_SUBPROC

  a := a + b;                              !Access local A and B
  CALL a_subproc (a);
END;                                       !End A_PROC

```

Sublocal Storage Limitations Because sublocal storage area is so limited, the way you use sublocal variables is as important as how many you declare.

When a subprocedure is activated, the compiler allocates the subprocedure's parameters, variables, and temporary results of expressions in the subprocedure's private storage area. The compiler allocates each sublocal item at location S[0], pushing previously allocated sublocal items to increasingly negative offsets from S[0]. If you attempt to access a sublocal item that is pushed beyond location S[-31], the compiler issues an error.

To avoid problems, use just a few sublocal parameters and variables and avoid complex expressions in subprocedures. For example, avoid the following practice:

```
PROC main_proc MAIN;
  BEGIN
    SUBPROC sub_oops (f);          !Declare SUB_OOPS
      INT .f;
      BEGIN
        INT i[0:24];              !Use 25 sublocal words
        f := 1;
        f := (f * (f *
          (f + 9) + 8) + 7) + 6) + 5) + 4) + 3) + 2) + 1);
      END;                          !Too many temporary values
    CALL sub_oops;
  END;
```

Sublocal Parameter Storage Limitations

The order in which a subprocedure passes parameters could result in a range violation error. The error occurs when an actual parameter of the subprocedure refers to a sublocal variable located beyond S[-31] in the caller's sublocal storage area.

The following example shows two subprocedure calls. The first call passes actual parameters in reverse order, causing an error:

```
PROC my_proc MAIN;
BEGIN
  !Lots of local declarations
  SUBPROC sub1 (a,b,c,d,e,f,g,h,i,j,k,l,m,n,o,p,q,r,s);
    INT          a,b,c,d,e,f,g,h,i,j,k,l,m,n,o,p,q,r,s;
    BEGIN
    END;
  SUBPROC sub2;
    BEGIN
    INT          A,B,C,D,E,F,G,H,I,J,K,L,M,N,O,P,Q,R,S;
    CALL sub1   (S,R,Q,P,O,N,M,L,K,J,I,H,G,F,E,D,C,B,A);
                                     !Cause error
    CALL sub1   (A,B,C,D,E,F,G,H,I,J,K,L,M,N,O,P,Q,R,S);
                                     !No error
    END;   !Actual parameters are shown in italics to
          !distinguish them from variables in the text.
  !Lots of code
END;
```

In effect, here is what happens: When SUB2 is activated, the compiler allocates each variable at location S[0] in the order declared (from A through S) and pushes preceding variables to increasingly negative offsets from S[0]. Part 1 of Figure 11-2 shows the sublocal allocation immediately after activation of SUB2.

The first time SUB2 calls SUB1, SUB2 passes parameters in reverse order from the order of the variables. The compiler allocates each actual parameter at S[0], in the order specified (from S through A) and pushes preceding variables and parameters away from S[0]. Each parameter (when allocated at S[0]), must access the corresponding variable to receive its value. For example, parameter S must receive the value stored in variable S, and so on. When parameter D is allocated at S[0], variable D is allocated at S[-31] and is still accessible. When parameter C is allocated at S[0], however, variable C is pushed beyond S[-31], is not accessible, and an error results. Part 2 of Figure 11-2 shows sublocal allocation at the point at which the error occurs.

The second time SUB2 calls SUB1, SUB2 passes parameters in the same order as the variables, so each variable is accessible when the corresponding parameter is allocated at S[0]. Part 3 of Figure 11-2, shows that every variable is accessible when the corresponding parameter is allocated at S[0].

Figure 11-2. Sublocal Data Storage Limitations

1. At activation of SUB2:

S[-31]	
S[-30]	
S[-29]	
S[-28]	
S[-27]	
S[-26]	
S[-25]	
S[-24]	
S[-23]	
S[-22]	
S[-21]	
S[-20]	
S[-19]	
S[-18]	A
S[-17]	B
S[-16]	C
S[-15]	D
S[-14]	E
S[-13]	F
S[-12]	G
S[-11]	H
S[-10]	I
S[-9]	J
S[-8]	K
S[-7]	L
S[-6]	M
S[-5]	N
S[-4]	O
S[-3]	P
S[-2]	Q
S[-1]	R
S[0]	S

2. First call to SUB1:

S[-31]	E
S[-30]	F
S[-29]	G
S[-28]	H
S[-27]	I
S[-26]	J
S[-25]	K
S[-24]	L
S[-23]	M
S[-22]	N
S[-21]	O
S[-20]	P
S[-19]	Q
S[-18]	R
S[-17]	S
S[-16]	S
S[-15]	R
S[-14]	Q
S[-13]	P
S[-12]	O
S[-11]	N
S[-10]	M
S[-9]	L
S[-8]	K
S[-7]	J
S[-6]	I
S[-5]	H
S[-4]	G
S[-3]	F
S[-2]	E
S[-1]	D
S[0]	C

3. Second call to SUB1:

S[-31]	G
S[-30]	H
S[-29]	I
S[-28]	J
S[-27]	K
S[-26]	L
S[-25]	M
S[-24]	N
S[-23]	O
S[-22]	P
S[-21]	Q
S[-20]	R
S[-19]	S
S[-18]	A
S[-17]	B
S[-16]	C
S[-15]	D
S[-14]	E
S[-13]	F
S[-12]	G
S[-11]	H
S[-10]	I
S[-9]	J
S[-8]	K
S[-7]	L
S[-6]	M
S[-5]	N
S[-4]	O
S[-3]	P
S[-2]	Q
S[-1]	R
S[0]	S

 Variables

 Actual parameters

Using Parameters This subsection gives additional information about declaring, passing, and allocating parameters of procedures and subprocedures.

For portability to future software platforms, treat formal parameters as spatially unrelated in memory storage.

Declaring Formal Parameters Table 11-3 summarizes the formal parameter characteristics you can declare depending on the kind of actual parameter your procedure expects.

Table 11-3. Formal Parameter Specification

Actual Parameter	Formal Parameter Characteristics			
	Formal Parameter	Parameter Type	Indirection	Referral
Simple variable	Value or reference	STRING * INT INT(32) REAL REAL(64) FIXED(<i>n</i>) FIXED(*)	Value, no; reference, yes	No
Simple variable	Value	UNSIGNED	No	No
Array or simple pointer	Reference	STRING INT INT(32) REAL REAL(64) FIXED(<i>n</i>)	Yes	No
Definition structure, referral structure, or structure pointer	Reference	INT or STRING	Yes	Yes
Definition structure,* referral structure, or structure pointer	Reference	STRUCT	Yes	Yes
Constant expression ** (including @ <i>identifier</i>)	Value	STRING INT INT(32) UNSIGNED REAL REAL(64) FIXED(<i>n</i>)	No	No
Procedure	Value	PROC PROC(32) ***	No	No

* These features are not supported in future software platforms.

** The data type of the expression and of the formal parameter must match, except that you can mix the STRING, INT, and UNSIGNED (1–16) data types, and you can mix the INT(32) and UNSIGNED(17–31) data types.

*** PROC(32) is a D-series feature.

Value parameters are described next, followed by reference parameters.

Using Value Parameters If a procedure or subprocedure declares a formal parameter as a value parameter, callers in effect pass a copy of the actual parameter. The called procedure or subprocedure can modify the parameter as if it were a local or sublocal variable but cannot modify the original variable in the caller's scope.

The compiler allocates storage for parameters in the parameter area of the called procedure or subprocedure. For value parameters, the kind of parameter and the parameter type determine the amount of space that is allocated. Table 11-4 summarizes the amount of storage the compiler allocates for formal value parameters:

Table 11-4. Value Parameter Storage Allocation

Formal Parameter	Parameter Type	Allocation	Actual Parameter
Simple variable or constant expression	STRING INT * UNSIGNED(1-16) *	Word	Simple variable or constant expression
Simple variable or constant expression	INT(32) ** REAL UNSIGNED(17-31)	Doubleword	Simple variable or constant expression
Simple variable	REAL(64) FIXED(*) FIXED(<i>n</i>)	Quadrupleword	Simple variable
Constant expression	REAL(64) FIXED(<i>n</i>)	Quadrupleword	Constant expression
16-bit procedure address	PROC—or its alias PROC(16)	Word	Procedure with 16-bit address
32-bit procedure address	PROC(32)	Doubleword	Procedure with 32-bit address

* An INT or UNSIGNED(16) actual parameter can be a standard address.

** An INT(32) actual parameter can be an extended address.

Simple Variables as Value Parameters

When you declare a simple variable as a formal value parameter, specify its *parameter type* and *identifier* but omit any indirection symbol:

```
PROC my_proc (a, b, c);
    INT a, b, c;           !Declare value parameters
    BEGIN
    !Lots of code
    END;
```

Passing INT, INT(32), REAL, and REAL(64) simple variables as value parameters is straightforward. Passing STRING, FIXED, and UNSIGNED simple variables is described in the following subsections.

STRING Value Parameters. Declare byte value parameters as INT simple variables where possible because:

- Passing STRING parameters by value is not portable to future software platforms.
- The system places an actual STRING value in the right byte of a word as if the value were an INT expression.

If you do declare and pass STRING value parameters, you can use the following techniques for accessing the STRING parameter value:

- The called procedure can declare a DEFINE that indexes to the parameter value:

```
PROC p (s);
  STRING s;
  BEGIN
    DEFINE str = s[1]#;          !Declare DEFINE STR
    IF str = "A" THEN ... ;     !Use STR instead of S
  END;
```

- The called procedure can left shift the parameter value by eight bits:

```
PROC p (s);
  STRING s;
  BEGIN
    s := s '<<' 8;              !Eight-bit left shift of value
    !Lots of code
  END;
```

- The caller can left shift the parameter value in the actual parameter list by eight bits:

```
CALL proc1 (byte '<<' 8);
```

FIXED Value Parameters. If a FIXED actual parameter has a different *fpoint* than the formal parameter, the system applies the *fpoint* of the formal parameter to the actual parameter.

If, however, you specify parameter type FIXED(*), the called procedure treats the actual parameter as having an *fpoint* of 0. That is, the system copies the content of the actual parameter to the formal parameter without an *fpoint*.

UNSIGNED Value Parameters. You can pass UNSIGNED parameters only as value parameters.

Procedures as Value Parameters

You can specify procedures (but not subprocedures) as PROC or PROC(32) formal parameters.

PROC Value Parameters. Specify a procedure as a formal PROC parameter if the actual parameter is the address of:

- A C small-memory-model routine
- A FORTRAN routine compiled with the NOEXTENDEDREF directive
- A TAL procedure or function procedure

For each actual PROC parameter, the compiler allocates a word of storage in the parameter area of the called procedure. The actual PROC parameter is a 16-bit address that contains the PEP and map information of the passed procedure.

The following example contains the following procedures:

- GREATER_THAN, which is passed as an actual PROC parameter
- BUBBLE_SORT, which declares the formal PROC parameter
- SORT, which calls BUBBLE_SORT and passes GREATER_THAN to it

```
LITERAL s = 10;
INT .a[0:s - 1] := [10,9,8,7,6,5,4,3,2,1];

INT PROC greater_than (a, b); !Declare actual PROC parameter
    INT a, b;
    BEGIN
    IF a > b THEN RETURN -1
    ELSE RETURN 0;
    END;

PROC bubble_sort (array, size, compare_function);
    INT .array;
    INT size;
    INT PROC compare_function; !Declare formal PROC parameter
    BEGIN
    INT i, j, temp, limit;
    limit := size - 1;
    FOR i := 0 TO limit -1 DO
        FOR j := i + 1 TO limit DO
            IF compare_function (array[i], array[j]) THEN
                BEGIN
                temp := array[i];
                array[i] := array[j];
                array[j] := temp;
                END;
        END;
    END;

PROC sort MAIN;
    BEGIN
    CALL bubble_sort (a, s, greater_than);
    END;
```

PROC(32) Value Parameters. Specify a procedure as a formal PROC(32) parameter if the actual parameter is the address of:

- A C large-memory-model routine
- A FORTRAN routine compiled with the EXTENDEDREF directive
- A Pascal routine

For each actual PROC(32) parameter, the compiler allocates a doubleword of storage in the parameter area of the called procedure. The actual PROC(32) parameter is a 32 bit address (the high-order word contains the PEP and map information of the passed routine; the low-order word must contain a zero or a trap results). For more information, see Section 17, "Mixed-Language Programming."

Procedures as Parameters That Have Parameters. If a procedure declared as a formal parameter has formal parameters of its own, you must ensure that its callers pass the necessary actual parameters. The compiler does not provide this check. Callers must prefix the identifier of each actual reference parameter with either:

- The @ operator, which returns a standard address, either the address contained in a pointer or the address of a nonpointer item
- The \$XADR standard function, which returns an extended address for a variable that has a standard address

The following example shows use of both @ and \$XADR:

```

PROC a (f);                                !Declare procedure to
  STRING .EXT f;                            ! pass as actual parameter
  BEGIN
  !Lots of code
  END;

PROC hi (p);                                !Declare procedure to call
  PROC p;                                    !Declare PROC formal parameter
  BEGIN
  STRING .s[0:7] := "Hi there";
  CALL p ($XADR (s));                        !Use $XADR
  END;

PROC bye (p);                                !Declare procedure to call
  PROC p;                                    !Declare PROC formal parameter
  BEGIN
  STRING .EXT s[0:6] := "Bye bye";
  CALL p (@s);                               !Use @ operator
  END;

PROC m MAIN;                                !Declare caller procedure
  BEGIN
  CALL hi (a);                               !Call procedures HI and BYE
  CALL bye (a);                              ! and pass procedure A to both
  END;

```

VARIABLE or EXTENSIBLE Procedures as Parameters. When you call a VARIABLE or EXTENSIBLE procedure declared as a formal parameter, include an actual parameter list as usual. For each omitted actual parameter, however, you must specify an empty comma for each word of the parameter. For example, to omit a FIXED parameter, specify four empty commas, one for each word of the omitted quadrupleword.

In the actual parameter list, you must also include a parameter mask that indicates which parameters are passed and which are omitted.

In the following example, the actual parameter list in the call to PARAM_PROC includes:

- Two empty commas, one for each word of omitted INT(32) parameter named P2.
- A parameter mask (%B101, where 1 denotes a passed parameter and 0 denotes an omitted parameter)

```

PROC var_proc (p1, p2, p3) VARIABLE;      !Declare VAR_PROC
      INT(32) p1, p2, p3;
BEGIN
  !Lots of code
END;

PROC ext_proc (p1, p2, p3) EXTENSIBLE; !Declare EXT_PROC
      INT(32) p1, p2, p3;
BEGIN
  !Lots of code
END;

PROC normal_proc (param_proc);           !Declare NORMAL_PROC
      PROC param_proc;
BEGIN
  CALL param_proc (
      0d,      !Call PARAM_PROC
                !Pass parameter for p1
      , ,      !Omit parameter for p2 (two words)
      -1d,     !Pass parameter for p3
      %B101);  !Pass parameter mask
END;

PROC m MAIN;
BEGIN
  CALL normal_proc (var_proc);
  CALL normal_proc (ext_proc);
END;

```

“Parameter Masks” later in this section describes the format of VARIABLE and EXTENSIBLE parameter masks. Section 17, “Mixed-Language Programming” describes the use of procedures as parameters in other languages.

Addresses and Pointer Contents as Value Parameters

You can pass the address of a variable or the content of a pointer as an actual value parameter. The called procedure can then assign the address to a pointer.

In the actual parameter list, prefix the identifier of the variable or pointer with the @ operator:

```

INT .EXT array1[0:9];
INT .EXT array2[0:9];
INT .EXT ptr := @array2[9];

PROC move1 (x, y);
    INT(32) x;
    INT(32) y;
    BEGIN
        INT .EXT xptr := x;
        INT .EXT yptr := y;
        INT i;
        FOR i := 0 TO 9 DO
            yptr[i] := xptr[i];
        END;
    END;

PROC m1 MAIN;
    BEGIN
        CALL move1 (@array1, @ptr); !Pass address of ARRAY1
        END;                          ! and content of PTR

```

The following example is equivalent to the preceding example:

```

INT .EXT array1[0:9];
INT .EXT array2[0:9];
INT .EXT ptr := @array2[9];

PROC move2 (x, y);
    INT .EXT x;
    INT .EXT y;
    BEGIN
        INT .EXT xptr := @x;
        INT .EXT yptr := @y;
        INT i;
        FOR i := 0 TO 9 DO
            yptr[i] := xptr[i];
        END;
    END;

PROC m2 MAIN;
    BEGIN
        CALL move2 (array1, ptr);
    END;

```

Index Register Contents as Value Parameters

When you call a procedure or subprocedure, you can pass the content of an index register as a value parameter. If the called procedure or subprocedure expects a reference parameter, the compiler issues a warning. When the compiler encounters the invocation, it saves the content of the index registers, evaluates actual parameters, and branches to the called procedure or subprocedure. On return to the caller, the compiler restores the saved register content.

Passing the content of an index register as a parameter, however, is not portable to future software platforms. Also, if you accelerate your program for TNS/R systems, the Accelerator requires that you provide additional information, as described in the *Accelerator Manual*.

In support of existing programs, here are the steps for passing the content of an index register:

1. Specify a USE statement to reserve an index register and give it a *name*.
2. Assign a *value* to the index register.
3. Specify a CALL statement to pass the index identifier as a value parameter.
4. Specify a DROP statement to free the index register.

Here is an example:

```
PROC some_proc (f);
    INT f;
    BEGIN
        !Lots of code
    END;

PROC m MAIN;
    BEGIN
        USE x;                !Reserve and name an index register
        x := 1;              !Assign a value to the index register
        CALL some_proc (x);  !Pass the index register content
        DROP x;              !Free the index register
    END;
```

In the CALL statement, do not modify the content of an index register. If you do so, the index register contains the original (unmodified) content when control returns to the caller, and the compiler emits an error message. Avoid the following practice:

```
PROC some_proc (f);
    INT f;
    BEGIN
    !Lots of code
    END;

PROC m MAIN;
    BEGIN
    USE x;
    x := 1;                !X contains 1
    !Lots of code
    CALL some_proc (x := x + 2);
                                !Pass X; change its value to 3
                                !Upon return, X still contains 1

    DROP x;
    END;
```

If you must modify the content of an index register, assign a new value to the index register before listing the index register in the CALL statement. Here is an example:

```
PROC some_proc (f);
    INT f;
    BEGIN
    !Lots of code
    END;

PROC m MAIN;
    BEGIN
    USE x;
    x := 1;                !X contains 1
    !Lots of code
    x := x + 2;           !Assign new value to X
    CALL some_proc (x);  !Pass X
                                !Upon return, X contains 3

    DROP x;
    END;
```

Using Reference Parameters A reference parameter is a formal parameter for which the caller passes the address of a value. The called procedure can access and modify the original value in the caller's scope. Except in the case of definition structures, you declare formal reference parameters as if they were simple pointers or structure pointers (by using an indirection symbol). When you list reference parameters in the actual parameter list, however, omit any indirection symbol.

The compiler allocates storage for parameters in the parameter area of the called procedure or subprocedure. For reference parameters, the standard or extended addressing mode determines the amount of space that is allocated. Table 11-5 summarizes the amount of storage the compiler allocates for formal reference parameters:

Table 11-5. Reference Parameter Storage Allocation

Formal Parameter	Parameter Type	Allocation	Actual Parameter
Simple pointer for simple variable	STRING INT INT(32) REAL REAL(64) FIXED(<i>n</i>) FIXED(*)	Word (standard indirection) or doubleword (extended indirection)	Address of simple variable
Simple pointer for array	STRING INT INT(32) REAL REAL(64) FIXED(<i>n</i>)	Word (standard indirection) or doubleword (extended indirection)	Address of array
Structure pointer or indirect structure	STRING INT STRUCT	Word (standard indirection) or doubleword (extended indirection)	Address of structure

Simple Variables as Reference Parameters

If a procedure expects a simple variable as an actual reference parameter, declare the formal parameter as a simple pointer, preceding its identifier with an indirection symbol.

Arrays as Reference Parameters

If a procedure expects an array as an actual parameter, declare the formal parameter as a simple pointer, preceding its identifier with an indirection symbol.

The following example declares two formal reference parameters (for array elements) as simple pointers, one standard and the other extended:

```
PROC new_proc (array1, array2, elements_to_move);
  INT .array1;      !Declare simple pointers as formal
  INT .EXT array2; ! parameters for array elements
  INT elements_to_move;
BEGIN
  !Manipulate the parameters
  array1 ':= ' array2 FOR elements_to_move ELEMENTS;
END;
```

Another procedure calls the preceding procedure and passes two arrays as actual parameters. No indirection symbols precede the array identifiers in the CALL statement:

```
PROC main_proc MAIN;
BEGIN
  INT first[0:99];    !Declare arrays to pass
  INT second[0:99];

  CALL new_proc (first, second, 100);
                                !Call NEW_PROC; pass
END;                               ! arrays by reference
```

Passing the Array Size. When you pass an array, you also might need to pass the array size or bounds information. (The declaration of the formal parameter does not provide such information.)

The following example passes array size information:

```
LITERAL array_size = 10;
INT .EXT array[0:array_size - 1];

PROC p (a, s);
  INT .EXT a;
  INT s;
BEGIN
  INT i;
  INT n := s - 1;
  FOR i := 0 TO n DO
    a[i] := 0;
  END;

PROC m MAIN;
BEGIN
  CALL p (array, array_size);    !Pass array size
END;
```

The following example passes array bounds information:

```

LITERAL array_lb = 0;
LITERAL array_ub = 9;
INT .array[array_lb:array_ub];

PROC zero (a, lb, ub);
    INT .a;
    INT lb;
    INT ub;
BEGIN
    INT i;
    FOR i := lb TO ub DO
        a[i] := 0;
    END;

PROC m MAIN;
BEGIN
    CALL zero (array, array_lb, array_ub);
END;
                                !Pass lower and upper array bounds

```

Structures as Reference Parameters

Either definition structures or referral structures can be formal reference parameters. You can treat them as if they were structure pointers. For the address of the structure, the compiler allocates one word for a standard indirect structure and two words for an extended indirect structure.

Definition structures as parameters is described next, followed by referral structures as parameters. For portability to future software platforms, declare referral structures (rather than definition structures) as parameters where possible.

Definition Structures as Parameters. When you declare a definition structure as a formal reference parameter, include an indirection symbol and a structure *layout*:

```

PROC proc_a (def_struct);
    STRUCT .EXT def_struct;
    BEGIN
        INT a;
        INT b;
    END;
BEGIN
    !Process the parameter
END;
                                !Declare PROC_A
                                !Declare definition
                                ! structure as a
                                ! formal parameter

```

Referral Structures as Parameters. When you declare a referral structure as a formal parameter, include an indirection symbol and a *referral* (enclosed in parentheses) that provides the structure layout. The referral can be the identifier of an existing structure or structure pointer.

In the following example, STRUCT_A is the referral in the formal parameter declaration for REF_STRUCT:

```
STRUCT .EXT struct_a[0:99];      !Declare STRUCT_A
BEGIN
  INT a;
  INT b;
END;

PROC proc_a (ref_struct);        !Declare PROC_A;
  STRUCT .EXT ref_struct (struct_a);
BEGIN
  !Process the parameter
END;

PROC m MAIN;
BEGIN
  CALL proc_a (struct_a);
END;
```

Passing the Number of Structure Occurrences. If the structure being passed has more than one occurrence, you might need to pass the number of occurrences. (The formal parameter declaration of the structure does not provide such information.)

```
PROC proc_b (ref_struct, ub);
  INT .EXT ref_struct (struct_a);
  INT ub;
BEGIN
  INT i;
  FOR i := 0 TO ub DO
    ref_struct[i].a := ref_struct[i].b := 0;
  END;
```

FIXED Reference Parameters

If the *fpoint* of an actual parameter does not match the *fpoint* of the formal parameter, the compiler issues a warning. The system then applies the *fpoint* of the formal parameter to the actual parameter.

Pointer Content as Reference Parameters

To pass the content of a pointer, prefix the pointer identifier with @ in the actual parameter list. The called procedure can then change the pointer content in the caller. An example is:

```

INT .array1[0:99];      !Declare ARRAY1
INT .array2[0:99];      !Declare ARRAY2

PROC proc1 (wptr);
  INT .wptr;             !Declare WPTR (formal parameter)
  BEGIN
  wptr := @array2[0];    !Assign address of ARRAY2[0] to WPTR
  END;

PROC proc2 MAIN;
  BEGIN
  INT .my_ptr := @array1[0];
  !Declare MY_PTR; initialize it with address of ARRAY1[0]

  array1[0] := 100;
  array2[0] := 200;
  !MY_PTR = 100 before the following CALL statement
  CALL proc1 (@my_ptr);
  !MY_PTR = 200 after the preceding CALL statement

  !Avoid the following CALL statement:
  CALL proc1 (@array1);
  !ARRAY1 now refers to ARRAY2; previous ARRAY1 value is lost
  END;

```

Initializing Pointers Passed as Parameters

Before passing pointers as parameters, be sure they contain addresses:

```

INT .iptr;              !Declare IPTR
INT ivar := 0;          !Declare and initialize IVAR

PROC p (fiptr);
  INT .fiptr;           !Declare FIPTR
  BEGIN
  fiptr := 2;
  END;

PROC m MAIN;
  BEGIN
  CALL p (iptr);        !Not OK, IPTR is not initialized
  @iptr := @ivar;      !Assign address of IVAR to IPTR;
                       ! IPTR = 0 before the following call
  CALL p (iptr);       !OK, PTR now contains an address;
                       ! IPTR = 2 after the call
  END;

```

Here is another example that shows the need to store addresses in pointers before passing them as reference parameters:

```
STRUCT t (*);
  BEGIN
    INT i;
    !More structure items
  END;

STRUCT .x (t);
STRING .a (t);           !Uninitialized structure pointer
!or INT .a (t);
!or STRING .EXT a (t);
!or INT .EXT a (t);

PROC p (f);
  STRUCT .f (t);         !Formal reference parameter; the
!or STRUCT .EXT f(t);   ! corresponding actual parameter
!or STRING .f (t);     ! need not be declared in same way
!or STRING .EXT f (t);
!or INT .f (t);
!or INT .EXT f (t);
  BEGIN
    f.i := 3;
  END;

PROC m MAIN;
  BEGIN
    CALL p (a);          !Not OK, A is not initialized
    CALL p (x);          !OK, X is initialized
    @a := @x;           !Address of X is assigned to A
    CALL p (a);          !OK, A contains a value
  END;
```

Reference Parameter Address Conversions

When you pass a reference parameter, its addressing mode should match that of the formal parameter. If not, the compiler converts the addressing mode of the actual parameter to match that of the formal parameter. When converting an address, the compiler assumes that a `STRING` pointer contains a byte address and that a pointer of any other type contains a word address. In some conversions, part of the address is lost.

Table 11-6 lists combinations of addressing modes and the kind of address that results in each case.

Table 11-6. Reference Parameter Address Conversions

Formal Parameter's Addressing Mode	Actual Parameter's Addressing Mode	Converted Address of Actual Parameter
Standard byte	Standard word	Standard byte address *
Standard word	Standard byte	Standard word address **
Standard byte	Extended word	Standard byte address in segment 0 ***
Standard byte	Extended byte	Standard byte address in segment 0 ***
Standard word	Extended word	Standard word address in segment 0 ***
Standard word	Extended byte	Standard word address in segment 0 ** ***
Extended	Standard	Extended address
Extended word	Extended byte	No address conversion; the compiler issues a warning

* The most significant bit is lost.

** The left-or-right-byte specifier—bit 15 of a standard byte address, bit 31 of an extended byte address—is lost or absent. The converted address might access the wrong byte in a word. The compiler issues a warning.

*** The segment-number specifier—bits 2 through 14 of an extended address—is lost, so the converted address defaults to segment 0, the current user data segment. The compiler issues a warning.

Parameter Pairs You can include formal parameter pairs when you declare a procedure or subprocedure. A parameter pair consists of two formal parameters connected by a colon that together describe a single data type to some programming languages. For information on using parameter pairs, see Section 17, “Mixed-Language Programming.”

Procedure Parameter Area The system creates a private parameter area for each activation of a procedure. Before control passes to the called procedure, the system stores any actual parameter values in the private parameter area.

For each activation of a procedure, the parameter area provides storage for:

- Up to 32 named parameters with no limit on the number of words of parameters. (The compiler generates code for accessing parameter words beyond L-31.)
- One of the following parameter masks, if present:
 - A one-word or doubleword VARIABLE parameter mask
 - A one-word to eight-word EXTENSIBLE parameter mask, plus a word value that represents the number of parameter words passed, stored in its negative form

Subprocedure Parameter Area The system creates a private parameter area for each activation of a subprocedure. Before control passes to the called subprocedure, the system stores any actual parameter values in the private parameter area.

For each activation of a subprocedure, the parameter area provides storage for up to 30 words of parameters, less storage required for sublocal variables and for a word or doubleword parameter mask, if present.

Scope of Formal Parameters The scope of an identifier is its visibility in the program; that is, the part of the program from which the identifier is accessible. Formal parameters have either:

- Local scope if declared in a procedure
- Sublocal scope if declared in a subprocedure

Normally, local statements can access global identifiers, and sublocal statements can access local identifiers in the encompassing procedure and global identifiers.

If the same identifier appears at the global, local, and sublocal levels, however, local statements cannot access the global identifier, and sublocal statements cannot access the local or global identifier:

```
INT a := 4;           !Declare global variables A and B
INT b := 1;

PROC p (a, b);
  INT a;             !Declare local formal parameters
  INT b;             ! A and B
  BEGIN
  INT c;             !Declare local variable C

  SUBPROC sub2 (b);
    INT b;           !Declare sublocal formal parameter B
    BEGIN
    c := b + 5;      !Access local C and sublocal B
    END;             ! (not global or local B)

  a := a + b;        !Access local A and B (not global
                    ! A and B)

  CALL sub2 (a);     !Call subprocedure; pass local A
  END;               ! (not global A)
```

Parameter Masks When a procedure calls a VARIABLE or EXTENSIBLE procedure, the compiler provides a parameter mask and keeps track of which actual parameters are passed to the procedure.

When a procedure calls a VARIABLE or EXTENSIBLE procedure that is declared as a formal parameter, however, the compiler does not provide a parameter mask. The caller must provide the parameter mask as part of the CALL statement. For an example, see “Procedures as Value Parameters” earlier in this section.

The following subsections describe how the compiler formats and allocates VARIABLE and EXTENSIBLE parameter masks for the TNS system. (The format of VARIABLE masks differ from that of EXTENSIBLE masks.)

VARIABLE Parameter Masks

When a VARIABLE procedure is called, the compiler:

1. Allocates storage for each formal parameter in the called procedure’s parameter area
2. Generates one of the following parameter masks:
 - A one-word mask for 16 or fewer formal parameters
 - A doubleword mask for 17 or more parameters
3. Initializes the mask bits to 0
4. Allocates storage for the parameter mask in the called procedure’s parameter area as follows:
 - One-word mask—at location L[-3]
 - Doubleword mask—at location L[-3] for the low-order word and L[-4] for the high-order word
5. Associates each formal parameter with a bit in the parameter mask, right justifying the layout in the mask, so the last parameter corresponds to bit <15> of the low-order word
6. Sets the bit for each passed parameter to 1

VARIABLE Word Parameter Mask

If a VARIABLE procedure has 16 or fewer formal parameters, the compiler generates a one-word mask. It associates each formal parameter with a bit in the mask, right justifying the layout so the last parameter corresponds to bit <15>.

In the following example, PROC_A has six formal parameters. Procedure A_CALLER calls PROC_A and passes three actual parameters to it. Each empty comma in the actual parameter list denotes an omitted parameter, regardless of size:

```
PROC proc_a (p1,p2,p3,p4,p5,p6) VARIABLE;
  INT p1,p2,p3,p4,p5,p6;
  BEGIN
    !Process the parameter values
  END;

PROC a_caller MAIN;
  BEGIN
    INT x, y, z;
    !Process the variables
    CALL proc_a (,x,y,,z);      !Call PROC_A; pass three
                                ! parameters (each empty comma
                                ! denotes an omitted parameter)
  END;
```

Figure 11-3 shows the parameter mask for the preceding example. In this case, bits <0:9> are not used and contain zeros. Bits corresponding to passed parameters each show a 1, while bits corresponding to omitted parameters each show a 0.

Figure 11-3. VARIABLE Word Parameter Mask

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	
L [-3]:	0	0	0	0	0	0	0	0	0	0	0	1	1	0	1	0	
Actual parameters:												X	Y		Z		
Formal parameters:												P1	P2	P3	P4	P5	P6

339

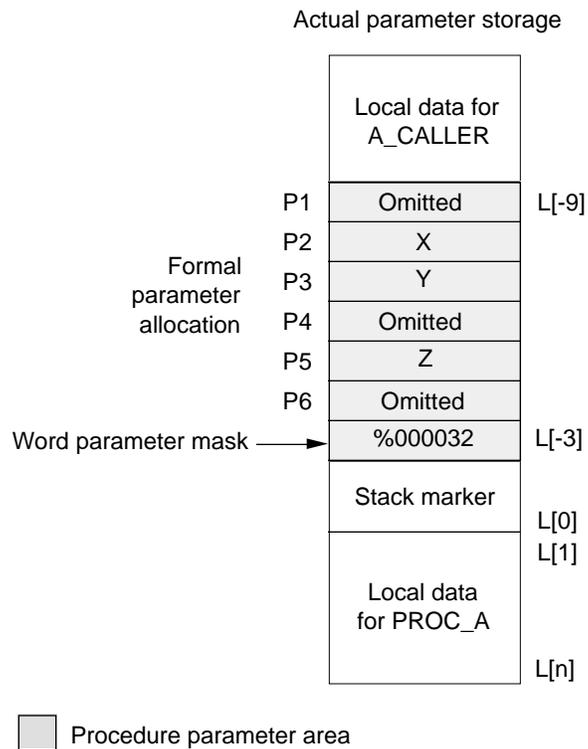
VARIABLE Parameter Area

Figure 11-4 shows the parameter area of PROC_A when called by A_CALLER in the preceding example. The figure shows:

- Storage allocation for the formal parameters P1 through P6 of PROC_A
- Storage of actual parameters X, Y, and Z passed by A_CALLER
- Storage of the parameter mask of PROC_A.

The value of the parameter mask (%000032) represents the bit settings for the actual parameters (shown in Figure 11-3 earlier in this section).

Figure 11-4. Parameter Area of a VARIABLE Procedure



VARIABLE Doubleword Parameter Mask

If a VARIABLE procedure has more than 16 parameters, the compiler generates a doubleword mask. It allocates the high-order word of the mask in location L[-4] and the low-order word of the mask in location L[-3].

The compiler associates each formal parameter with a bit in the mask, right-justifying the layout so the last formal parameter corresponds to bit <15> of the low-order word.

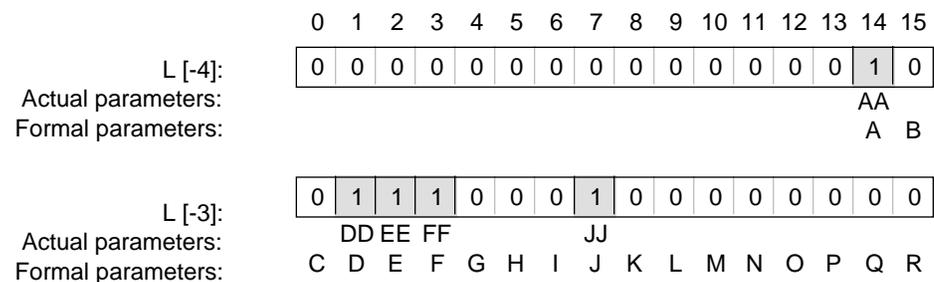
In the following example, PROC_B has 18 formal parameters. Procedure B_CALLER calls PROC_B and passes five parameters to it:

```
PROC proc_b (a,b,c,d,e,f,g,h,i,j,k,l,m,n,o,p,q,r) VARIABLE;
  INT a,b,c,d,e,f,g,h,i,j,k,l,m,n,o,p,q,r;
  BEGIN
    !Process the parameter values
  END;

PROC b_caller;
  BEGIN
    INT aa, dd, ee, ff, jj;
    !Lots of code
    CALL proc_b (aa,!bb!,!cc!,dd,ee,ff,!gg!,!hh!,!ii!,jj);
                                !Pass AA, DD, EE, FF, and JJ
                                ! (comments denote omitted
                                ! parameters)
  END;
```

Figure 11-5 shows the doubleword parameter mask for the preceding example. In this case, bits <0:13> in the high-order word are not used and contain zeros. Bits corresponding to passed parameters each show a 1, while bits corresponding to omitted parameters each show a 0.

Figure 11-5. VARIABLE Doubleword Parameter Mask



340

EXTENSIBLE Parameter Masks When a procedure calls an EXTENSIBLE procedure, the compiler provides a parameter mask and keeps track of which actual parameters are passed to the procedure.

The format of the EXTENSIBLE parameter mask differs from the format of the VARIABLE parameter mask.

When an EXTENSIBLE procedure is called, the compiler:

1. Allocates storage for each formal parameter in the called procedure's parameter area
2. Generates a one-word to eight-word parameter mask, depending on how many words must be allocated to hold the formal parameters
3. Initializes the mask bits to 0
4. Allocates storage for the mask in the called procedure's parameter area as follows:
 - Location L[-4] for the lowest order word of the mask, location L[-5] for the second lowest order word if needed, and so on
 - Location L[-3] for the number of words of parameters in its negative form
6. Associates each word of the formal parameters with a bit in the mask, left justified, so that the highest order word of the first parameter corresponds to bit <0> of the lowest order word of the mask
7. Sets the bits associated with each passed parameter word to 1

EXTENSIBLE Word Parameter Mask

If an EXTENSIBLE procedure has 16 or fewer parameter words, the compiler generates a one-word mask. The compiler associates each word in each formal parameter with a bit in the parameter mask, left justifying the layout so the first parameter word corresponds to bit <0>.

In the following example, PROC_C has seven formal parameters of varying lengths. Procedure C_CALLER calls PROC_C and passes four actual parameters, totaling seven parameter words:

```
PROC proc_c (a,b,c,d,e,f,g) EXTENSIBLE;
  INT      a,d,f,g;
  INT(32) b,e;
  FIXED   c;
BEGIN
  !Code for processing
END;

PROC c_caller;
BEGIN
  INT aa, ff, gg;
  FIXED cc;
  !Code for processing
  CALL proc_c (aa,,cc,,,ff,gg);
END;
```

Figure 11-6 shows the parameter mask for the preceding example. In this case, bits <12:15> are not used and contain zeros. Bits corresponding to passed parameters each show a 1, while bits corresponding to omitted parameters each show a 0.

Figure 11-6. EXTENSIBLE Word Parameter Mask

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
L [-4]:	1	0	0	1	1	1	1	0	0	0	1	1	0	0	0	0
Actual parameters:	AA					CC					FF	GG				
Formal parameters:	A	B			C		D	E		F	G					

341

The compiler stores the number of parameter words passed (in its negative form) at location L[-3]. For the preceding example, the value stored is -7.

EXTENSIBLE Doubleword Parameter Mask

If an EXTENSIBLE procedure has more than 16 (but less than 33) parameter words, the compiler generates a doubleword mask. It associates each word of each formal parameter with a bit in the mask, left justifying the layout so the first parameter word corresponds to bit <0> of the low-order word.

In the following example, PROC_D has 12 formal parameters of varying lengths. Procedure D_CALLER calls PROC_D and passes five actual parameters, totaling nine parameter words.

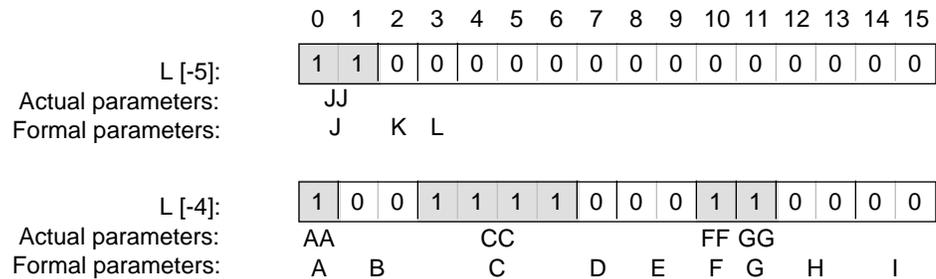
```

PROC proc_d (a,b,c,d,e,f,g,h,i,j,k,l) EXTENSIBLE;
    INT      a,d,f,g,k,l;
    INT(32)  b,e,h,i,j;
    FIXED    c;
BEGIN
    !Do more work
END;

PROC d_caller;
BEGIN
    INT aa, ff, gg;
    FIXED cc;
    INT(32) jj;
    !Do some work
    CALL proc_d (aa,,cc,,,ff,gg,,,jj);
END;
    
```

Figure 11-7 shows the parameter mask settings for the preceding example. In this case, bits <4:15> of the high-order word (L[-5]) are not used and contain zeros. Bits corresponding to passed parameters each show a 1, while bits corresponding to omitted parameters each show a 0.

Figure 11-7. EXTENSIBLE Doubleword Parameter Mask



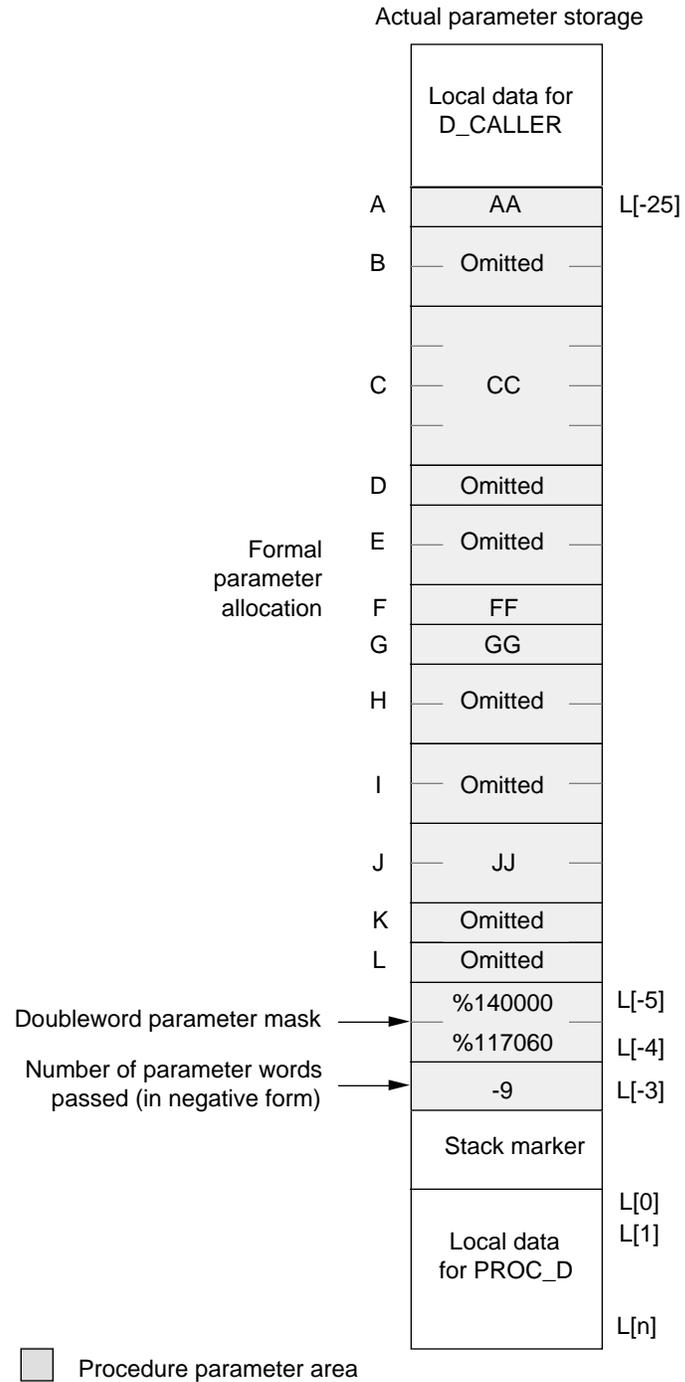
The compiler stores the number of passed parameter words (in its negative form) at location L[-3]. For the preceding example, the value stored is -9.

EXTENSIBLE Parameter Area

Figure 11-8 shows the parameter area of PROC_D when PROC_D is called by D_CALLER in the preceding example. The figure shows:

- Storage allocation for the formal parameters of PROC_D
- Storage of actual parameters passed by D_CALLER
- Storage of the parameter mask of PROC_D
- Storage of the number of parameter words passed

Figure 11-8. Parameter Area of EXTENSIBLE Procedure



Procedure Entry Sequence

When an EXTENSIBLE procedure is activated, the procedure's entry code loads certain values onto the register stack. Table 11-7 shows:

- The values that the entry code loads onto the register stack
- The RP setting that results

Table 11-7. Entry Values Loaded Onto Register Stack

Kind of Procedure	Register Stack	Value Loaded	RP Setting
EXTENSIBLE	R[0]	Number of parameter words expected	0
EXTENSIBLE, converted	R[0]	Number of parameters when procedure was VARIABLE	
	R[1]	Number of parameter words when procedure was VARIABLE	
	R[2]	Number of parameter words now expected	2

The procedure's entry code then executes an ESE instruction, which uses the RP setting to tell the cases apart. ESE sets RP to 7 but does not save the values in R0 through R7.

For a converted VARIABLE procedure, ESE converts the mask format to the EXTENSIBLE format. ESE adds the needed bits and words and initializes them to 0. It does not initialize any extra words on the register stack caused by the stack movement.

For information on the RP setting and ESE instruction, see the *System Description Manual* for your system.

Using Labels Within a procedure, you can declare labels for use within the same procedure. Labels have local or sublocal scope only.

Labels are the only declarable objects that you need not declare before using them. For best program management, however, declare all labels.

Using Local Labels To declare and use local labels:

1. Declare the label *identifier* inside a procedure (in the local declarations) by specifying the keyword LABEL and the label identifier; for example:

```
LABEL loc_label;           !Declare LOC_LABEL
```

2. Place the label identifier and a colon (:) preceding a statement in the same procedure (but not in a subprocedure); for example:

```
loc_label:                !Use the label for
a := 5;                   !this statement
```

3. Reference the label in a GOTO statement located in the same procedure or in any subprocedure contained in that procedure.

You can branch to local labels by using local or sublocal GOTO statements in the same procedure. In the following example, a local GOTO statement references a local label:

```
INT op1, op2, result;     !Declare global data
PROC p;
  BEGIN
    LABEL addr;           !Declare label ADDR
    op1 := 5;
    op2 := 28;
  addr:                   !Use label ADDR for
    result := op1 + op2;  ! this statement
    op1 := op2 * 299;
    !More code
    IF result < 100 THEN
      GOTO addr;         !Branch to label ADDR
    END;
```

In the following example, sublocal GOTO statements reference local labels in the encompassing procedure. Future software platforms require that you declare local labels to which sublocal GOTO statements refer:

```

PROC p;
  BEGIN
    LABEL a;      !Declare label A
    INT i;

    SUBPROC s;
      BEGIN
        !Lots of code
        GOTO a;    !This branch is portable to future
                  ! software platforms; label A is declared
        GOTO b;    !This branch is not portable; label B is
                  ! not declared

      END;

    !Lots of code
    a : i := 0;
    !More code
    b : i := 1;
    !Still more code
  END;

```

When a local label and a sublocal variable in a procedure have the same identifier and that identifier is referenced within the subprocedure, the sublocal variable is accessed instead of the label:

```

INT data;

PROC a;                                !Declare procedure A,
  BEGIN                                  ! which has global scope.
    LABEL a;                             !Declare local label A.

    SUBPROC sp;
      BEGIN
        INT a;                            !Declare sublocal variable A.
        data := @a;                       !Assign address of sublocal A,
                                           ! not of procedure A or
                                           ! local label A.
      END;
    a:
    CALL sp;
  END;

```

Using Sublocal Labels To declare and use sublocal labels:

1. Declare the label *identifier* inside a subprocedure in the sublocal declarations; for example:

```
LABEL sub_label;           !Declare SUB_LABEL
```

2. Place the label identifier and a colon (:) preceding a statement in the same subprocedure:

```
sub_label:                 !Use the label for
a := 5;                    ! this statement
```

3. Reference the label in a GOTO statement located in the same subprocedure.

You can branch to sublocal labels by using GOTO statements in the same subprocedure:

```
INT op1, op2, result;     !Declare global declarations
PROC p;
BEGIN
  !Declare local variables
  LABEL exit;             !Declare local label EXIT
  SUBPROC s;              !Declare subprocedure
  BEGIN
    LABEL addr;          !Declare sublocal label ADDR
    op1 := 5;
    op2 := 28;
  addr:                   !Use label ADDR for
    result := op1 + op2;  ! this statement
    IF result < 0 THEN    !If overflow, exit the
      GOTO exit;         ! subprocedure to label EXIT
    result := op2 * 2;
    !Lots of code
    IF result < 100 THEN  !If result < 100, go to
      GOTO addr;        ! label ADDR
    END;                 !End subprocedure
    !Lots of code
  exit:                   !Use label EXIT for
    CALL s;              ! this statement
    !More code
  END;                   !End procedure
```

Using Undeclared Labels You need not declare labels before using them; however, if a variable and an undeclared label in the same scope have the same identifier and if you use the label before you access the variable, the compiler issues an error message.

No error results from the following example because the assignment to the variable occurs before the label is used:

```
INT data;

PROC p;
  BEGIN
    INT a;           !Declare local variable A

    SUBPROC sp;
      BEGIN
        data := @a;  !Assign address of local variable A
                    ! because sublocal label A is not
                    ! declared and not used yet
        a:          !Use sublocal label A
          data := @a; !Assign address of sublocal label A
        END;
      CALL sp;
    END;
```

Applying @ to a label name is not portable to future software platforms.

Getting Addresses of Procedures and Subprocedures

To get the address of a procedure or subprocedure, prefix its identifier with @. For example, to assign the address of MY_PROC to MY_VAR, specify:

```
my_var := @my_proc;
```

When prefixed to procedure or subprocedure identifiers, @ yields 16-bit addresses as follows:

Item	Address Yielded by @ Operator
Procedure	Procedure entry point (PEP) number of the procedure LORed with code segment information
Subprocedure	Word address of the subprocedure's entry point in the current code segment

The LOR operator performs a bit-wise logical OR operation on INT or STRING values and returns a 16-bit result, as described in Section 5, "Using Expressions."

You can get the PEP number, which is contained in bits <7:15> of the procedure's address, as follows:

```
PROC my_proc MAIN;                !Declare MY_PROC
  BEGIN
  INT pepnum;
  !Some code
  pepnum := @my_proc.<7:15>;      !Assign PEP number of MY_PROC
  !More code
  END;
```

You can get the code location of a subprocedure as follows:

```
PROC my_proc;
  BEGIN
  INT sub_loc;

  SUBPROC my_subproc;             !Declare MY_SUBPROC
  BEGIN
  !Some code
  sub_loc := @my_subproc;         !Assign address of MY_SUBPROC
  !More code
  END;

  END;
```

Applying @ to a procedure name and using PEP or XEP table entries are not portable to future software platforms.

12 Controlling Program Flow

Conditional statements let you provide a choice of paths in your program. These statements contain a condition, which lets the program select which path to take during execution.

Some statements can also be executed repeatedly, applying a set of operations to different data during each iteration. A group of statements that can be executed repeatedly is called a loop. Each loop must contain a termination condition, which lets the program decide whether to continue repeating the loop or to stop after a particular iteration.

Other statements provide unconditional control over program flow. They contain no decision-making conditions but often follow, or are nested in, a conditional statement.

This section discusses conditional and unconditional control statements in the order shown in Table 12-1.

Table 12-1. Program Control Statements

Statement	Type	Operation
IF	Conditional	Conditionally selects one of two possible statements
CASE	Conditional	Selects a set of statements based on a selector value
WHILE	Conditional	Executes a pretest loop while a condition is true
DO	Conditional	Executes a posttest loop until a condition is true
FOR	Conditional	Executes a pretest loop <i>n</i> times
ASSERT	Conditional	Conditionally calls an error-handling procedure
CALL	Unconditional	Calls a procedure or subprocedure
RETURN	Unconditional	Returns from a procedure or subprocedure to the caller; returns a value from a function. As of the D20 release, it can also return a condition code value.
GOTO	Unconditional	Branches to a label within a procedure or subprocedure

IF Statement The IF statement conditionally selects one of two statements. The IF statement tests the condition before selecting a statement. If the condition is true, the THEN clause executes. If the condition is false, the ELSE clause executes, if present.

To specify an IF statement, include:

IF *condition*

Specifies a condition that, if true, causes the THEN clause to execute. If false, the *condition* causes the ELSE clause to execute. If no ELSE clause is present, the statement following the IF statement executes. The *condition* can be either:

A conditional expression

An INT arithmetic expression. If the result of the arithmetic expression is not 0, the *condition* is true. If the result is 0, the *condition* is false.

THEN *statement*

Specifies a statement to execute if the *condition* is true. If you omit the statement in the THEN clause, no action occurs for the THEN clause. The *statement* can be any TAL statement.

ELSE *statement* (optional)

Specifies a statement to execute if the *condition* is false. If the *condition* is false and no ELSE clause is present, the statement following the IF statement executes. The *statement* can be any TAL statement.

For example, the following IF statement calls an error handler if VAR_ITEM contains a nonzero value:

```
INT var_item;
!Some code here
IF var_item <> 0 THEN
    CALL error_handler;
```

The following example is equivalent to the preceding example:

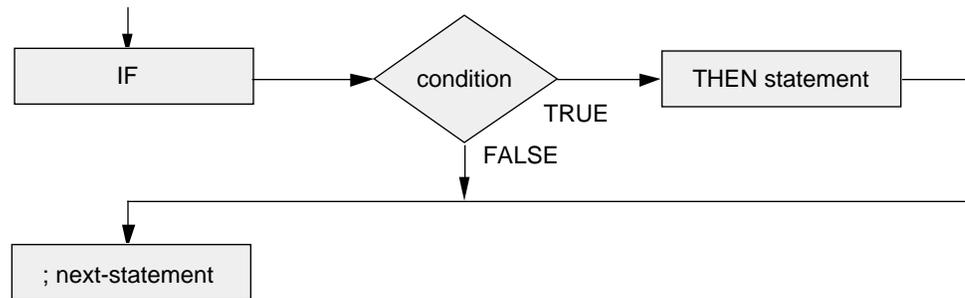
```
IF var_item THEN
    CALL error_handler;
```

The following IF statement compares two arrays. If the arrays are equal, the IF statement assigns a 1 to ITEM_OK. If they are not equal, it assigns 0 to ITEM_OK:

```
INT new_array[0:9];
INT old_array[0:9];
INT item_ok;
!Some code here
IF new_array = old_array FOR 10 WORDS THEN
    item_ok := 1           !No semicolon when followed by ELSE
ELSE
    item_ok := 0;
```

IF Statement Execution The IF-THEN form executes as shown in Figure 12-1.

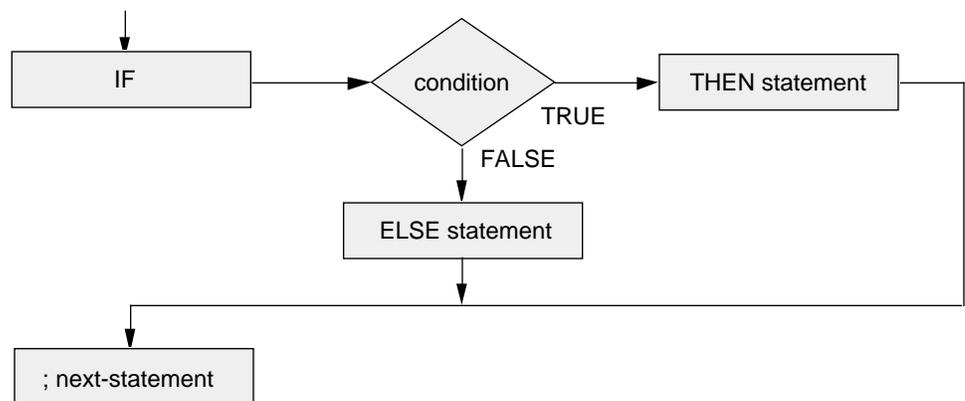
Figure 12-1. IF-THEN Execution



404

The IF-THEN-ELSE form executes as shown in Figure 12-2.

Figure 12-2. IF-THEN-ELSE Execution



405

IF-ELSE Pairing You can nest IF statements to any level. In a nested IF statement, the innermost IF clause pairs with the closest ELSE clause. Formatting can make the IF-ELSE pairing obvious or ambiguous.

The following side-by-side examples are equivalent. In both cases, the ELSE clause belongs to the second IF statement, but the pairing is clearer in the example on the left:

Recommended Format	Unclear Format
<pre>IF expression1 THEN IF expression2 THEN stmt1 ELSE stmt2;</pre>	<pre>IF expression1 THEN IF expression2 THEN stmt1 ELSE stmt2;</pre>

To override the default IF-ELSE pairing, you can use the BEGIN-END construct. For example, if you insert a BEGIN-END pair in the preceding example, the ELSE clause belongs to the first IF clause rather than to the second IF clause:

```
IF expression1 THEN
  BEGIN                                !Begin compound statement
    IF expression2 THEN
      stmt1;
    END                                !End compound statement;
  ELSE                                  ! no semicolon before ELSE
    stmt2;
```

**CASE Statement,
Labeled**

A labeled CASE statement consists of a selector and a series of case alternatives. Each alternative associates one or more case labels with a set of statements. When the selector matches a case label, the associated set of statements executes.

The unlabeled CASE statement is described in the *TAL Reference Manual*.

To specify a labeled CASE statement, include:

- CASE *selector* OF BEGIN

Specifies a value that selects the *case-alternative* to execute. The *selector* must be an INT arithmetic expression; for example:

```
INT i;                                !Declare INT variable I
!Code to initialize I
CASE i OF BEGIN ...                  !Selector is I
```

- case-alternatives*

Each *case-alternative* specifies a choice of *statements* to execute when the *selector* matches a *case-label*. You can specify any number of *case-alternatives*; at least one is required. Each *case-alternative* consists of:

- case-labels* ->

One or more INT constants in any order, separated by commas, followed by ->. To include a *case-label* as a range of constants, specify the lowest and highest values separated by two periods (..):

```
2, 9, 4 .. 7, 11 ->                !Case labels 2, 4, 5, 6, 7,
! and 9 for one alternative
```

- statements*

One or more statements to execute if the *selector* matches any *case-label* in this alternative:

```
2, 9, 4 .. 7 ->                    !Case labels
aa := cc + dd;                      !When I is 2 or 9 or any of
bb := cc - dd;                      ! 4 through 7, execute these
! statements
```

- OTHERWISE -> *statements*

An optional clause that specifies optional *statements* to execute if the *selector* does not match any *case-label* in the CASE statement. If no OTHERWISE clause is present and the *selector* does not match a *case-label*, a run-time error results. So always include the OTHERWISE clause, even if it contains no *statements*.

- END

Specifies the end of the CASE statement.

For example, this labeled CASE statement has four *case-alternatives* and the OTHERWISE clause:

```

INT location;
LITERAL bay_area, los_angeles, hawaii, elsewhere;

PROC area_proc (area_code);           !Declare procedure
  INT area_code;                       !Declare selector as
  BEGIN                                 ! formal parameter
  CASE area_code OF                     !Selector is AREA_CODE
  BEGIN
  408, 415 ->
    location := bay_area;
  213, 818 ->
    location := los_angeles;
  808 ->
    location := hawaii;
  OTHERWISE ->
    location := elsewhere;
  END;                                  !End CASE statement
END;                                    !End AREA_PROC

```

Statement Forms Generated by the Compiler

The compiler generates a branch table form or a conditional test form of the labeled CASE statement, depending on the maximum range and number of case label values:

- The compiler generates a branch table form if the maximum range (between the smallest and largest case labels) is either:
 - Less than 257
 - Between 257 and 2048 inclusive and the approximate number of words of code for the branch table form (maximum range + 12) is no greater than the approximate number of words of code for the conditional test form (number of labels * 13 + 2)
- In all other cases, the compiler generates a conditional test form. If the statement has more than 63 case labels, the compiler issues an error.

The branch table form is much faster than the conditional test form.

To improve the efficiency of the conditional test form, you can order the alternatives from most common to least common.

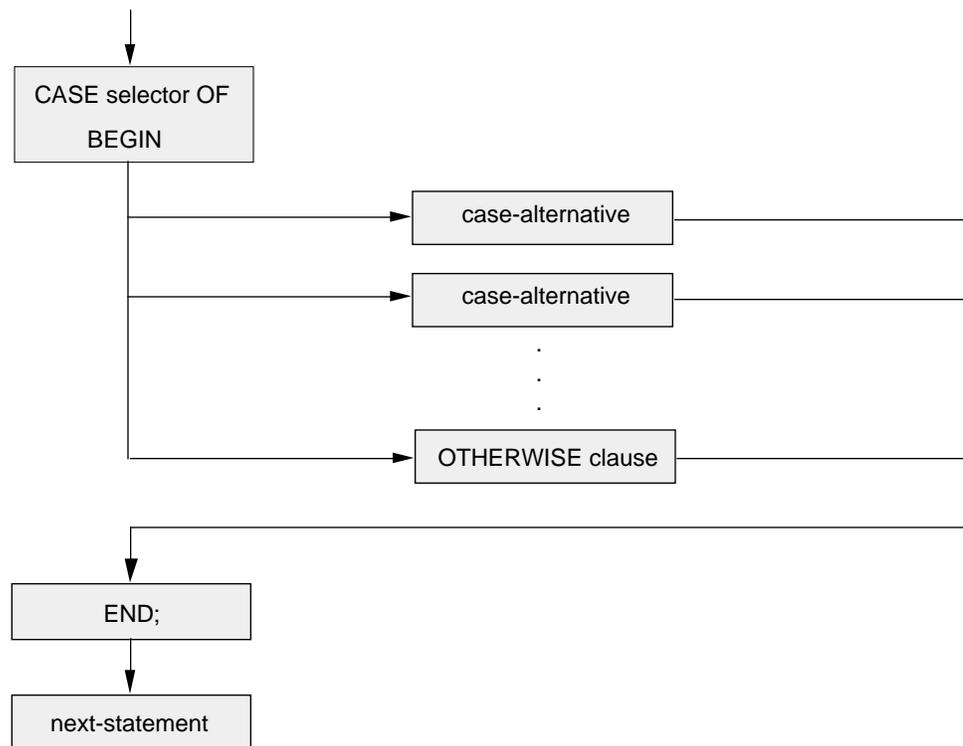
Directives and CASE Statements If a CASE statement contains many GOTO statements, you can use the OPTIMIZE directive to reduce the size of the object code. Use OPTIMIZE level 1 or 2 as follows:

- In a program under development, use level 1.
- In a stable program, use level 2, which optimizes code across statement boundaries, so debugging is more difficult.

The CHECK directive does not affect the labeled CASE statement (as it does the unlabeled CASE statement).

Labeled CASE Statement Execution Figure 12-3 shows how the labeled CASE statement executes.

Figure 12-3. Labeled CASE Statement Execution



WHILE Statement The WHILE statement is a repeating loop that executes a statement while a specified condition is true.

To specify a WHILE statement, include:

WHILE *condition*

Specifies a condition that, if true, causes the loop to execute. If the *condition* is false, the loop does not execute. The *condition* can be either:

A conditional expression

An INT arithmetic expression. If the result of the arithmetic expression is not 0, the *condition* is true. If the result is 0, the *condition* is false.

DO *statement*

Specifies a statement to execute while the *condition* is true. The *statement* can be any TAL statement.

For example, this WHILE loop continues while ITEM is less than LEN:

```
LITERAL len = 100;
INT .array[0:len - 1];
INT item := 0;

WHILE item < len DO          !WHILE statement
  BEGIN
    array[item] := 0;
    item := item + 1;
  END;
  !ITEM equals LEN at this point
```

The WHILE statement tests the *condition* before each iteration of the loop. If the *condition* is false before the first iteration, the loop never executes. If the *condition* is always true, the loop executes indefinitely unless a statement in the loop causes an exit. In the following example, a GOTO statement branches to a label outside the WHILE statement when the IF condition is true:

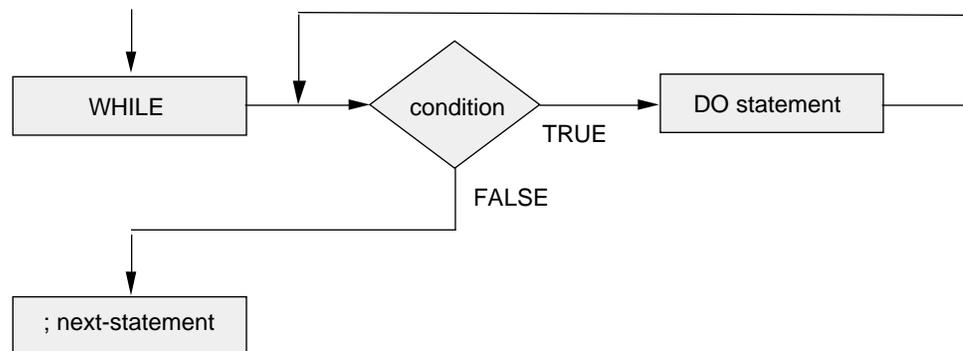
```
WHILE -1 !true! DO
  BEGIN
    !Lots of code
    IF <condition> THEN GOTO exit_loop;
    !More code
  END;
exit_loop:                    !Label
!<statement>
!More code
```

This WHILE loop increments INDEX until a nonalphabetic character occurs:

```
LITERAL len = 255;  
STRING .array[0:len - 1];  
INT index := -1;  
  
WHILE (index < len - 1) AND  
      ($ALPHA(array[index := index + 1]))  
DO ... ;
```

Figure 12-4 shows the action of the WHILE statement.

Figure 12-4. WHILE Statement Execution



407

DO Statement The DO statement is a repeating loop that executes a statement until a specified condition becomes true. If the condition is always false, the loop repeats until a statement in the DO loop causes an exit.

To specify a DO statement, include:

DO statement

Specifies a statement to execute until the *condition* becomes true. The *statement* can be any TAL statement.

UNTIL condition

Specifies a condition that, if false, causes the DO loop to continue. If the *condition* is true, the statement following this DO statement executes. The *condition* can be either:

A conditional expression

An INT arithmetic expression. If the result of the arithmetic expression is not 0, the *condition* is true. If the result is 0, the *condition* is false.

A DO statement always executes at least once because the compiler tests the *condition* at the end of the loop. Unless you have a special reason to use the DO statement, it is safer to use the WHILE statement.

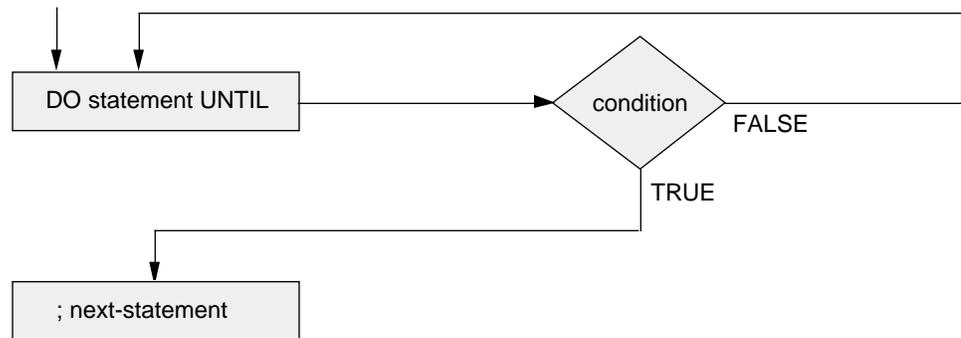
The following DO loop cycles through an array until each element is assigned a 0:

```
LITERAL len = 50;
LITERAL limit = len - 1;
INT i := 0;
STRING .array_a[0:limit];           !Declare array

DO                                   !DO statement
  BEGIN
    array_a[i] := 0;                 !Compound statement to
    i := i+1;                         ! execute in DO loop
  END
UNTIL i > limit;                     !Condition for ending loop
!Rather than I = LEN
```

Figure 12-5 shows the action of the DO statement.

Figure 12-5. DO Statement Execution



408

FOR Statement The FOR statement is a repeating loop that executes a statement while incrementing or decrementing an index automatically. The loop terminates when the index reaches a limit value. If the index is greater than the limit on the first test, the loop never executes.

To specify a FOR statement, include:

FOR *index* :=

Specifies a value that increments or decrements automatically until it reaches a specified value and terminates the loop.

In a standard FOR loop, *index* is the identifier of an INT simple variable, array element, simple pointer, or structure data item.

In an optimized FOR loop, *index* is the identifier of an index register you have reserved by using the USE statement.

initial-value

An INT arithmetic expression (such as 0) that initializes *index*.

limit

An INT arithmetic expression specified as either:

TO *limit*—Increments *index* each time the loop executes until *index* exceeds *limit*

DOWNTO *limit*—Decrements *index* each time the loop executes until *index* is less than *limit*

BY *step*

An optional clause for specifying an INT arithmetic expression by which to increment or decrement *index*. The default value is 1.

DO *statement*

Specifies the statement to execute each time through the loop. The *statement* can be any TAL statement.

For example, this standard FOR loop clears an array by assigning a space to each element in the array:

```
LITERAL len = 100;
LITERAL limit = len - 1;
STRING .array[0:limit];
INT index;

FOR index := 0 TO limit DO           !Use default step of 1;
    array[index] := " ";           ! fill elements with spaces

FOR index := 4 TO limit BY 5 DO
    array[index] := "!";           !Replace every fifth space
                                   ! with exclamation point
```

This standard FOR loop uses the DOWNTO clause to reverse a string from "BAT" to "TAB":

```
LITERAL len = 3;
LITERAL limit = len - 1;
STRING .normal_str[0:limit] := "BAT";
STRING .reversed_str[0:limit];
INT index;

FOR index := limit DOWNTO 0 DO
    reversed_str[limit - index] := normal_str[index];
```

Nesting FOR Loops You can nest FOR loops to any level. The following nested FOR loop treats MULTIPLES as a two-dimensional array. It fills the first row with multiples of 1, the next row with multiples of 2, and so on:

```
INT .multiples[0:10*10-1];
INT row;
INT column;

FOR row := 0 TO 9 DO
    FOR column := 0 TO 9 DO
        multiples [row * 10 + column] := column * (row + 1);
```

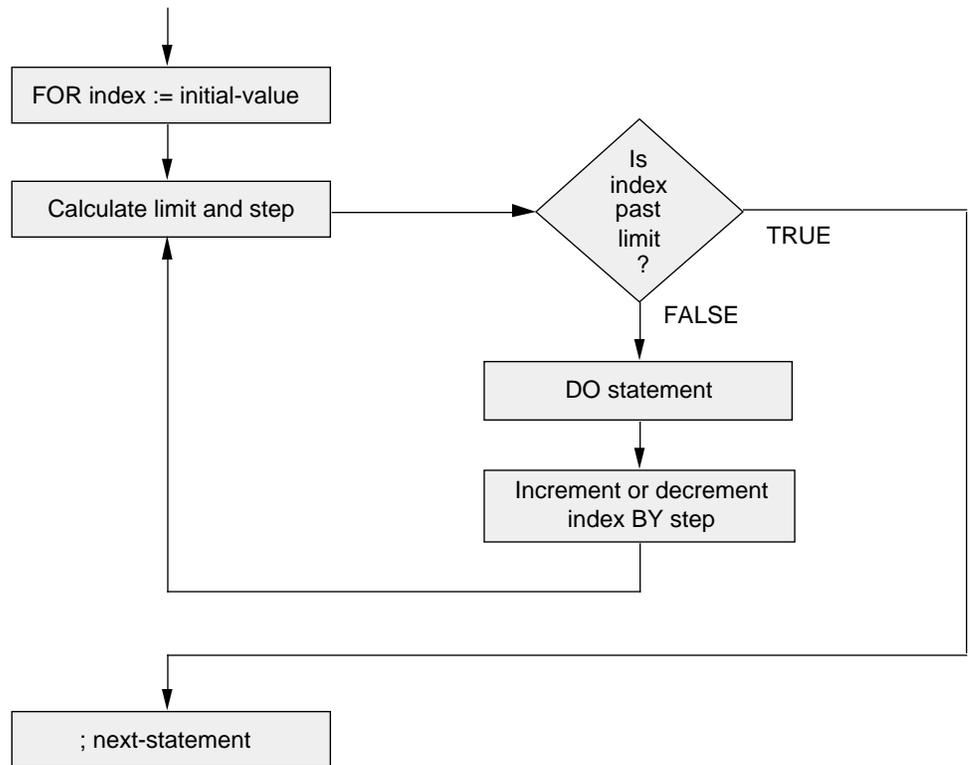
Standard FOR Loops For *index*, standard FOR loops specify an INT variable. Standard FOR loops execute as follows:

- When the looping terminates, *index* is one greater than *limit* if:
 - The *step* value is 1.
 - The TO keyword (not DOWNTO) is used.
 - The *limit* value (not a GOTO statement) terminates the looping.
- limit* and *step* are recomputed at the start of every iteration of the loop.

Standard FOR Loop Execution

Figure 12-6 shows the action of the standard FOR loop.

Figure 12-6. Standard FOR Loop Execution



Optimized FOR Loops For *index*, optimized FOR loops specify a register reserved by a USE statement. Optimized FOR loops execute faster than standard FOR loops; they execute as follows:

- When the looping terminates, *index* is equal to *limit*.
- limit* is calculated only once, at the start of the first iteration of the loop.

You optimize a FOR loop as follows:

1. Before the FOR statement, specify a USE statement to reserve an index register.
2. In the FOR statement:
 - For *index*, use the identifier of the index register.
 - Omit the *step* value (thereby using the default value of 1).
 - For *limit*, use the TO keyword.
3. If you modify the register stack, save and restore it before the end of the looping. (Modifying the register stack, however, is not portable to future software platforms.)
4. After the FOR statement, specify a DROP statement to release the index register. Do not drop the index register during the looping.

The following example contrasts a standard FOR loop with an equivalent optimized FOR loop. Both FOR loops clear the array by assigning a blank space to each element in the array:

```

LITERAL len = 100;
LITERAL limit = len-1;           !Declare ARRAY to use
STRING .array[0:limit];         ! in both FOR loops
INT index;                       !Declare INDEX

FOR index := 0 TO limit DO       !Standard FOR loop;
  array[index] := " ";

USE x;                            !Reserve index register
FOR x := 0 TO limit DO          !Optimized FOR loop
  array[x] := " ";
DROP x;                          !Release index register

```

For more information on the USE and DROP statements, see the *TAL Reference Manual*.

Inclusion of procedure calls in the FOR loop slows down the loop because the compiler must save and restore registers before and after each call.

Optimized FOR Loop Execution

The execution of optimized FOR loops differs from standard FOR loops as follows:

- When the looping terminates, *index* is equal to *limit*.
- limit* is calculated only once—at the start of the first time through the loop.

The following example compares the value of *index* in standard and optimized FOR loops:

```
PROC a MAIN;
  BEGIN
    INT i, j, k;
    FOR i := 1 TO 10 DO k := i;      !Standard FOR loop
    k := i;                          !K is 11

    USE j;
    FOR j := 1 TO 10 DO k := j;    !Optimized FOR loop
    k := j;                          !K is 10

    DROP j;
  END;
```

ASSERT Statement The ASSERT statement conditionally invokes the procedure named in an ASSERTION directive. It is a debugging or error-handling tool.

To specify the ASSERT statement, include:

- ASSERT assert-level*

Specifies an integer in the range 0 through 32,767, followed by a colon. If the *assert-level* is equal to or higher than the assertion level specified in the current ASSERTION directive and if the *condition* is true, the compiler activates the procedure specified in the ASSERTION directive. If the *assert-level* is lower than the assertion level, the procedure is not activated.

- condition*

An expression that tests a program condition and yields a true or false result.

For example, this ASSERT statement specifies an *assert-level* of 7 and the expression \$CARRY, which tests a program condition:

```
ASSERT 7 : $CARRY;
```

\$CARRY is a standard function that tests the state of the carry indicator. If the carry indicator is on, \$CARRY returns a true; if it is off, it returns a false. The carry indicator is set when a scan resulting from a SCAN or RSCAN statement is stopped by a 0. It is also set by some arithmetic operations.

Using ASSERT with ASSERTION

You use the ASSERT statement with the ASSERTION directive as follows:

1. Place an ASSERTION directive in the source code where you want to start debugging. In the directive, specify an *assertion-level* and an error-handling procedure such as the D-series PROCESS_DEBUG_ or the C-series DEBUG system procedure. The *assertion-level* is an integer in the range 0 through 32,767:

```
?ASSERTION 5, PROCESS_DEBUG_ !Assertion-level is 5
```

2. Place an ASSERT statement at places where you want to invoke the error-handling procedure when an error occurs. In the statement, specify an *assert-level* that is equal to or higher than the *assertion-level* and specify an expression such as \$CARRY that tests a program condition:

```
ASSERT 10 : $CARRY; !Assert-level is 10
```

3. During program execution, if an *assert-level* is equal to or higher than the current *assertion-level* and *condition* is true, the compiler activates the error-handling procedure.
4. After you debug the program, you can nullify all or some of the ASSERT statements by specifying an ASSERTION directive with a higher *assertion-level* than the ASSERT statements you want to nullify:

```
?ASSERTION 11, PROCESS_DEBUG_
!Assertion-level nullifies assert-level 10 and below
```

The following example activates `PROCESS_DEBUG_` when an out-of-range condition occurs:

```
?SOURCE $SYSTEM.SYSTEM.EXTDECS (PROCESS_DEBUG_)
?ASSERTION 5, PROCESS_DEBUG_
    !Assertion-level 5 activates all ASSERT conditions
SCAN array WHILE " " -> @pointer;
ASSERT 10 : $CARRY;
!Lots of code
ASSERT 10 : $CARRY;
!More code
ASSERT 20 : $OVERFLOW;
    !$OVERFLOW function tests for arithmetic overflow
```

Nullifying ASSERT Statements To make it easier to nullify all ASSERT statements that cover a particular condition, set all such ASSERT statements to the same *assert-level*. In the previous example, if you specify an ASSERTION directive with an *assertion-level* of 11, you nullify the two ASSERT statements that are set at 10. If you specify an ASSERTION directive set to 30, you nullify all the ASSERT statements.

CALL Statement You use the CALL statement to call a procedure, subprocedure, or entry-point identifier, and optionally pass parameters to it.

To specify a CALL statement, you can include:

CALL identifier

Specifies the identifier of a procedure, subprocedure, or entry-point identifier. As of the D20 release, the CALL keyword is optional.

parameter-list

Specifies an optional list, enclosed in parentheses, of one or more comma-separated actual parameters that you want to pass to the called procedure or subprocedure. The actual parameters in the list must correspond to the formal parameters of the called procedure or subprocedure.

Calling Procedures and Subprocedures To call a procedure or subprocedure that has no formal parameters, simply include the identifier of the procedure or subprocedure in the CALL statement:

```
CALL error_handler;
```

When a called procedure or subprocedure finishes executing, control returns to the statement following the CALL statement that invoked the procedure or subprocedure.

To call a procedure or subprocedure that has formal parameters, include the identifier of the procedure or subprocedure and a list of actual parameters in the CALL statement:

```
CALL compute_tax (item, rate, result);
```

For VARIABLE or EXTENSIBLE procedures, some or all of the formal parameters are optional. When you call such a procedure, you can omit some or all of the optional parameters:

Omitting Parameters Unconditionally

To omit some of the parameters or parameter pairs, use a place-holding comma for each omitted parameter or parameter pair up to the last specified parameter or parameter pair. Here are examples of omitted parameters:

```
CALL extensible_proc (num, , char, , , eof);
```

```
CALL extensible_proc ( , , , , , x);
```

To omit all parameters, you can specify an empty parameter list or you can omit the parameter list altogether:

```
CALL extensible_proc ( );
```

```
CALL extensible_proc;
```

In addition to place-holding commas, you can include comments to keep track of omitted parameters:

```
CALL extensible_proc (num, !name!, char, !val!, !size!, eof);
```

Omitting Parameters Conditionally

To omit a parameter or parameter pair conditionally, use the \$OPTIONAL standard function (D20 release or later) as described in Section 11, "Using Procedures."

Calling Functions Callers usually invoke functions by using the function identifier in expressions, as described in Section 11, "Using Procedures." A caller can also invoke functions by using a CALL statement, in which case the caller ignores the returned value.

Calling Procedures Declared as Formal Parameters Callers can call procedures declared as formal parameters. If the called procedure is a VARIABLE or EXTENSIBLE procedure, the caller must provide the appropriate parameter mask. For more information, see Section 11, "Using Procedures."

Passing Parameter Pairs You can pass parameter pairs in the CALL statement, as described in Section 17, "Mixed-Language Programming."

RETURN Statement The RETURN statement returns control to the caller. If the invoked procedure or subprocedure is a function, RETURN must return a result. As of the D20 release, RETURN can also return a program-specified condition code (CC).

The form of the RETURN statement depends on whether the return is from a function or from a procedure or subprocedure that is not a function.

Returning From Functions A function should contain a RETURN statement and must return a *result-expression* to the caller. The *result-expression* can be any arithmetic or conditional expression of the same data type as specified in the function declaration. (A function can be a procedure or a subprocedure.)

If a function lacks a RETURN statement, the compiler issues a warning but the compilation can complete and the resulting object file can be run. After the function executes, it returns a zero.

The following function returns the *result-expression* 20 in a RETURN statement:

```
INT PROC a;
  BEGIN
    !Lots of code
    RETURN 20;                !Return a result from a function
  END;
```

The following function contains two RETURN statements nested in an IF statement:

```
INT PROC other (nuff, more); !Declare function with type INT
  INT nuff;
  INT more;
  BEGIN
    IF nuff < more THEN      !IF statement
      RETURN nuff * more    !Return either of two
    ELSE                    ! results based on the
      RETURN 0;            ! condition NUFF < MORE
  END;
```

Returning a Condition Code

As of the D20 release, if *result-expression* is any type except FIXED or REAL(64), a function can return a *cc-expression* and a result-expression. *cc-expression* is an INT expression whose numeric value specifies the condition code value to return to the caller. If the *cc-expression* is:

Less than 0	Set the condition code to less than (<)
Equal to 0	Set the condition code to equal (=)
Greater than 0	Set the condition code to greater than (>)

The following function returns a result and a condition code to inform its caller that the returned value is less than, equal to, or greater than some maximum value:

```
INT PROC p (i);
  INT i;
  BEGIN
    RETURN i, i - max_val;    !Return a value and a
  END;                       ! condition code
```

If you call a function, rather than invoking it in an expression, you can test the returned condition code:

```
INT PROC p1 (i);
  INT i;
  BEGIN
    RETURN i;
  END;

INT PROC p2 (i);
  INT i;
  BEGIN
    INT j := i + 1;
    RETURN i, j;
  END;

CALL p1 (i);
IF < THEN ... ;                !Test the condition code
CALL p2 (i);
IF < THEN ... ;                !Test the condition code
```

The following procedure returns a condition code that indicates whether an add operation overflows:

```

PROC p (s, x, y);
  INT .s, x, y;
  BEGIN
  INT cc_result;
  INT i;
  i := x + y;
  IF $OVERFLOW THEN cc_result := 1
                    ELSE cc_result := 0;
  s := i;

  RETURN , cc_result;      !If overflow, condition code
  END;                    ! is >; otherwise, it is =

```

Returning From Nonfunction Procedures

In procedures and subprocedures that are not functions, a RETURN statement is optional. If you include a RETURN statement, it cannot return a *result*.

The following procedure returns control to the caller when A is less than B:

```

PROC something;
  BEGIN
  INT a,
      b;
  !Manipulate A and B
  IF a < b THEN
    RETURN;                !Return to caller
  !Lots more code
  END;

```

To return a condition code value, a nonfunction procedure or subprocedure must use a RETURN statement that includes *cc-expression*.

In a main procedure, a RETURN statement stops execution of the program and returns control to the operating system.

GOTO Statement The GOTO statement unconditionally transfers program control to the statement that is preceded by the specified label.

To specify a GOTO statement, include a label identifier after the GOTO keyword; for example:

```
GOTO label_one;
```

Local Scope A local GOTO statement can refer only to a local label in the same procedure. A local GOTO statement cannot refer to a label in a subprocedure or in any other procedure.

In the following example, a local GOTO statement branches to a local label:

```
PROC p
  BEGIN
    LABEL calc_a;           !Declare local label
    INT a;
    INT b := 5;

  calc_a :                 !Place label at local statement
    a := b * 2;
    !Lots of code
    GOTO calc_a;           !Local branch to local label
  END;
```

Sublocal Scope A sublocal GOTO statement can refer to a label in the same subprocedure or in the encompassing procedure. A sublocal GOTO statement cannot refer to a label in another subprocedure.

In the following example, a sublocal GOTO statement branches to a local label:

```
PROC p;
  BEGIN
    LABEL a;               !Declare local label
    INT i;

    SUBPROC s;
      BEGIN
        !Lots of code
        GOTO a;           !Sublocal branch to local label
      END;

    !Lots of code
  a :                     !Place label at local statement
    i := 0;
    !More code
  END;
```

Usage Guidelines GOTO statements can make a program harder to understand and maintain, but they are useful when you need to:

- Branch to a common exit or common error-handling code after some condition is met
- Exit from the middle of a multidimensional search or loop after some condition is met
- Maximize program performance

For most other circumstances, use conditional control statements such as labeled CASE, DO, IF, and WHILE.

13 Using Special Expressions

Special expressions let you perform specialized arithmetic or conditional operations. This section describes the special expressions listed in Table 13-1.

Table 13-1. Special Expressions

Expression Form	Kind of Expression	Description
Assignment	Arithmetic	Assigns the value of an expression to a variable
CASE	Arithmetic	Selects one of several expressions
IF	Arithmetic	Conditionally selects one of two expressions
Group comparison	Conditional	Does unsigned comparison of two sets of data

The result of an expression can be of any data type except `STRING` or `UNSIGNED`. The compiler determines the data type of an expression from the data type of the operands in the expression. All operands in an expression must have the same data type, with the following exceptions:

- An `INT` expression can include `STRING`, `INT`, and `UNSIGNED(1–16)` operands. The system treats `STRING` and `UNSIGNED(1–16)` operands as if they were 16-bit values. The system:
 - Puts a `STRING` operand in the right byte of a word and sets the left byte to 0.
 - Puts an `UNSIGNED(1–16)` operand in the right bits of a word and sets the unused left bits to 0, with no sign extension. For example, for an `UNSIGNED(2)` operand, the system fills the 14 leftmost bits of the word with zeros.
- An `INT(32)` expression can include `INT(32)` and `UNSIGNED(17–31)` operands. The system treats `UNSIGNED(17–31)` operands as if they were 32-bit values. It places an `UNSIGNED(17–31)` operand in the right bits of a doubleword and sets the unused left bits to 0, with no sign extension. For example, for an `UNSIGNED(29)` operand, the system fills the three leftmost bits of the doubleword with zeros.

In all other cases, if the data types do not match, use type transfer functions (described in the *TAL Reference Manual*) to make them match.

Assignment Expression

The assignment expression assigns the value of an expression to a variable.

To use an assignment expression, specify:

- The identifier of a variable
- An assignment operator (:=)
- An *expression* that represents a value of the same data type as the variable. The result of the *expression* becomes the result of the assignment expression. The *expression* can be either:
 - An arithmetic expression
 - A conditional expression (excluding a relational operator with no operands), the result of which has data type INT.

For example, you can increment a variable by specifying an assignment expression in an IF statement. As long as $A + 1$ is not 0 in the following example, the condition is true and the THEN clause executes:

```
IF (a := a + 1) THEN ... ;
```

You can use an assignment expression as an index. In the following example, A is incremented and accesses the next array element:

```
IF array[a := a + 1] <> 0 THEN ... ;
```

You can use an assignment expression in a relational form. The following example assigns the value of B to A, then checks for equality with 0:

```
IF (a := b) = 0 THEN ... ;
```

You can use assignment expressions to assign a value to multiple variables:

```
a := b := c := d := 0;
```

CASE Expression The CASE expression selects one of several expressions. You can nest CASE expressions. To use a CASE expression, specify:

CASE *selector* OF

Specifies an INT arithmetic expression that selects the expression to evaluate.

BEGIN *expressions*;

Specifies a choice of one or more *expressions*. Separate successive *expressions* with semicolons. Each *expression* must be either:

An INT arithmetic expression

A conditional expression (excluding a relational operator with no operands), the result of which has data type INT.

The compiler numbers each BEGIN *expression* consecutively, starting with 0. When the *selector* matches the compiler-assigned number of a BEGIN *expression*, the *expression* is evaluated. The result of *expression* becomes the result of the overall CASE expression.

OTHERWISE *expression*; (optional)

Specifies an expression to evaluate if the *selector* does not select one of the BEGIN *expressions*. The OTHERWISE *expression* and the BEGIN *expressions* must have all the same data type. If you omit the OTHERWISE clause and an out-of-range case occurs, results are unpredictable.

END

Specifies the end of the CASE expression.

For example, you can use a CASE expression to select the value resulting from one of several *expressions* and assign it to variable X. The *expression* selected depends on the value of *selector* A:

```
INT x, a, b, c, d;
!Code to initialize variables

x := CASE a OF
    BEGIN
        b;           !If A is 0, assign value of B to X.
        c;           !If A is 1, assign value of C to X.
        d;           !If A is 2, assign value of D to X.
        OTHERWISE -1; !If A is any other value,
    END;           ! assign -1 to X.
```

The CASE expression resembles the unlabeled CASE statement except that the CASE expression selects an *expression*, while the unlabeled CASE statement selects a *statement*. (The unlabeled CASE statement is described in the *TAL Reference Manual*.)

IF Expression The IF expression conditionally selects one of two expressions, usually for assignment to a variable. To use an IF expression, specify:

IF *condition*

Specifies a condition that, if true, causes the result of the THEN *expression* to become the result of the overall IF expression. If the *condition* is false, the result of the ELSE *expression* becomes the result of the overall IF expression. The *condition* is either:

A conditional expression

An INT arithmetic expression. If the result of this expression is not 0, the *condition* is true. If the result is a 0, the *condition* is false.

THEN *expression*

Specifies an expression to evaluate if the *condition* is true. The *expression* is either:

An INT arithmetic expression

A conditional expression (excluding a relational operator with no operands), the result of which has data type INT.

ELSE *expression* (optional)

Specifies the expression to evaluate if the *condition* is false. The *expression* is either:

An INT arithmetic expression

A conditional expression (excluding a relational operator with no operands), the result of which has data type INT

For example, you can assign either of two arithmetic *expressions* to VAR depending on the *condition* LENGTH > 0:

```
var := IF length > 0 THEN 10 ELSE 20;
```

You can include an IF expression, enclosed in parentheses, inside another expression:

```
var := index +
      (IF index > limit THEN var * 2 ELSE var * 3);
```

You can nest an IF expression within another IF expression:

```
var := IF length < 0 THEN -1
      ELSE IF length = 0 THEN 0
           ELSE 1;
```

The IF expression resembles the IF statement except that:

- The THEN and ELSE clauses are both required in the IF expression.
- The THEN and ELSE clauses contain *expressions*, not *statements*.

Group Comparison Expression

The group comparison expression compares a variable with another variable or with a constant. In general, to use a group comparison expression, specify these items:

- A variable (*var1*) with or without an index
var1 can be a simple variable, array, simple pointer, structure, structure data item, or structure pointer, but not a read-only array.
- A relational operator. All comparisons are unsigned regardless of whether you use a signed or unsigned relational operator. Relational operators are:
 Signed: <, =, >, <=, >=, <>
 Unsigned: '<', '=', '>', '<=', '>=', '<>'
- An item to which to compare *var1*. The item can be one of:
 - A variable (*var2*)—a simple variable, array, read-only array, simple pointer, structure, structure data item, or structure pointer—followed by the FOR clause
 - A *constant*—a number, a character string, or a LITERAL
 - A *constant list*

Group comparison expressions often appear in IF statements:

```
IF var_1 <> 0 FOR n BYTES THEN ... ;
```

Comparing a Variable to a Constant List

To compare an array (but not a read-only array) to a constant list, specify the constant list on the right side of the group comparison expression:

```
STRING array[0:3];
!Some code here
IF array = [ "ABCD" ] THEN ... ;
!Compare ARRAY to constant list
```

Comparing a Variable to a Single Byte

To compare a variable to a single byte, enclose a single constant in brackets ([]) on the right side of the group comparison expression. If the variable has a byte address or is a STRING structure pointer, the system compares a single byte regardless of the size of the constant:

```
STRING var[0:1];
!Some code here
IF var[0] = [5] THEN ... ; !Compare VAR to a single byte
```

In the preceding example, if you do not enclose the constant 5 in brackets (or if VAR has a word address or is an INT structure pointer), the system compares a word, doubleword, or quadrupleword as appropriate for the size of the constant. The following example shows the preceding IF statement with brackets omitted:

```
IF var[0] = 5 THEN ... ; !Compare VAR to two bytes
```

Comparing a Variable to a Variable To compare a variable to another variable, include the FOR clause in the group comparison expression. In the FOR clause, specify a *count* value—a positive INT arithmetic expression that specifies the number of bytes, words, or elements you want to compare.

Comparing Bytes

To compare bytes regardless of the data type of *var2*, specify the BYTES keyword following the *count* value in the FOR clause of the group comparison expression. BYTES compares the number of bytes specified by the *count* value. If both *var1* and *var2* have word addresses, however, BYTES generates a word comparison for $(count + 1) / 2$ words.

For example, you can compare bytes between INT arrays:

```
LITERAL length = 12;                !Number of array
                                     ! elements
INT word_array_one[0:length - 1];    !var1
INT word_array_two[0:length - 1];    !var2
INT byte_count;                      !count value (number
                                     ! of bytes to compare)

!Code to assign values to variables
IF word_array_one = word_array_two
   FOR byte_count BYTES THEN ... ;
```

Comparing Words

To compare words regardless of the data type of *var2*, specify the WORDS keyword following the *count* value in the FOR clause of the group comparison expression. WORDS compares the number of words specified by the *count* value.

For example, to compare words instead of doublewords between INT(32) arrays, multiply LENGTH by 2 and include the WORDS keyword:

```
LITERAL length = 12;                !count value (number
                                     !of words to compare)
INT(32) dblw_array_one[0:length - 1]; !var1
INT(32) dblw_array_two[0:length - 1]; !var2

!Code to assign values to arrays
IF dblw_array_one[0] = dblw_array_two[0]
   FOR 2 * length WORDS THEN ... ;
```

Comparing Elements

When you compare elements between arrays, you can specify the `ELEMENTS` keyword following the *count* value in the `FOR` clause of the group comparison expression. For example, you can compare doubleword elements between `INT(32)` arrays:

```
INT(32) in_array[0:19];
INT(32) out_array[0:19];

!Code to assign values to arrays
IF in_array <> out_array FOR 20 ELEMENTS THEN ... ;
```

When you compare array elements (as in the preceding example), the `ELEMENTS` keyword is optional but provides clearer source code.

When you compare structure or substructure occurrences, you must specify the `ELEMENTS` keyword in the group comparison expression:

```
STRUCT struct_one[0:9];
  BEGIN
    INT a[0:2];
    INT b[0:7];
    STRING c;
  END;

STRUCT struct_two (struct_one)[0:9];

!Code to assign values to structures
IF struct_one = struct_two FOR 10 ELEMENTS THEN ... ;
```

Using the Next Address The next address is the address (returned by the group comparison expression) of the first byte or word in *var1* that does not match the corresponding byte or word in *var2*.

To use the next address, you can declare a simple pointer and then use its identifier (prefixed by `@`) in the next-address clause in a group comparison expression. Here is an example of the next-address clause:

```
-> @next_addr_ptr
```

Here is an example of a group comparison expression that includes the next-address clause:

```
array_one = array_two FOR 100 -> @next_addr_ptr ...
```

The compiler does a standard comparison and returns a 16-bit next address if:

- Both *var1* and *var2* have standard byte addresses
- Both *var1* and *var2* have standard word addresses

The compiler does an extended comparison (which is slightly less efficient) and returns a 32-bit next address if:

- Either *var1* or *var2* has a standard byte address and the other has a standard word address
- Either *var1* or *var2* has an extended address

Variables (including structure data items) are byte addressed or word addressed as follows:

Byte addressed	STRING simple variables STRING arrays Variables to which STRING simple pointers point Variables to which STRING structure pointers point Substructures
Word addressed	INT, INT(32), FIXED, REAL(32), or REAL(64) simple variables INT, INT(32), FIXED, REAL(32), or REAL(64) arrays Variables to which INT, INT(32), FIXED, REAL(32), or REAL(64) simple pointers point Variables to which INT structure pointers point Structures

After an element comparison, the next address might point into the middle of an element, rather than at the beginning of the element.

You can, for example, compare the contents of two arrays and then determine from the next address the first element that does not match:

```

INT .s_array[0:11] := "$SYSTEM SYSTEM MYFILE ",
    .d_array[0:11] := "$SYSTEM USER MYFILE ",
    .ptr,
    n;

IF d_array = s_array FOR 12 -> @ptr THEN ... ;

```

The preceding comparison stops with element [4]; PTR contains the address of D_ARRAY[4] shown below:

```

0 1 2 3 4 5 6 7 8 9 ...
$SYSTEM SYSTEM MYFILE !Content of S_ARRAY
$SYSTEM USER MYFILE !Content of D_ARRAY

```

To determine the number of array elements that matched, subtract the address of D_ARRAY[0] from the address in PTR, using unsigned arithmetic:

```

n := @next_addr_ptr '-' @d_array;
!N gets 4 (fifth element)

```

Testing the Condition Code Setting The system treats the items being compared as unsigned values. After a group comparison, you can test the condition code setting by using the following relational operators (with no operands) in a conditional expression:

```

<          CCL if var1 '<' var2
=          CCE if var1 = var2
>          CCG if var1 '>' var2

```

The following example compares two arrays and then tests the condition code setting to see if the value of the element in D_ARRAY that stopped the comparison is less than the value of the corresponding element in S_ARRAY:

```

INT in_array[0:9];
INT out_array[0:9];

!Code to assign values to arrays
IF d_array = s_array FOR 10 -> ELEMENTS @ptr THEN
  BEGIN                                !They matched
    !Do something
  END
ELSE
  IF < THEN ... ;                      !PTR points to D_ARRAY element
                                         ! that is less than the
                                         ! corresponding S_ARRAY element

```

14 Compiling Programs

When you run the TAL compiler, the input is a source file—a file that contains TAL source text such as data declarations, statements, compiler directives, and comments.

The output from a compilation is an executable or bindable object file that consists of relocatable code and data blocks. You can bind object files with other object files into a new executable or bindable object file.

This section describes:

- The TAL compiler
- Compiling source files
- Binding object files
- Compiling with source lists
- Compiling with search lists
- Compiling with relocatable data blocks
- Compiling with saved global data
- Collecting cross-references

The Compiler The TAL compiler process is integrated with two other processes—BINSERV and SYMSERV. You can govern all three processes by using compiler directives.

The compiler compiles source code, processes compiler directives, and starts BINSERV and SYMSERV for additional processing. The compiler also produces any listings that result from the three processes.

Compiler directives let you select compilation options such as:

- Using conditional compilation (IF directive)
- Saving compiled global declarations for use in later compilations (SAVEGLOBALS, USEGLOBALS, BEGINCOMPILATION, and SEARCH directives)
- Checking the syntax without producing an object file (SYNTAX directive)

BINSERV If compilation is successful and the SYNTAX directive is not in effect, BINSERV:

- Constructs an object file
- Resolves external references by locating pertinent code and data blocks in object files listed in SEARCH directives and binding them into the object file
- Produces binder statistics for inclusion in the compiler listings

You can do further binding by using Binder.

SYMSERV If you compile using the SYMBOLS directive, SYMSERV provides symbol-table information to the object file for use by the Inspect product. If you compile using the CROSSREF directive, SYMSERV generates source-level cross-reference information for your program.

Compiling Source Files

Figure 14-1 shows the source file as input to the compiler and the object file as output from the compiler.

Figure 14-1. Compiling a Source File Into an Object File



410

The source file can include SOURCE directives that read in code from other source files. In effect, you can compile more than one source file into an object file, but the input to the compiler is always a single source file. The source file and the code that is read in from other source files by SOURCE directives together compose a compilation unit.

The compiler accepts information you specify in TACL commands (DEFINE, PARAM, and ASSIGN) if you issue them before you run the compiler. These commands are described in Appendix E, "File Names and TACL Commands."

Running the Compiler

To run the compiler, issue a compilation command at the TACL prompt. For example, you can compile the source file MYSOURCE and have the object code sent to the object file MYOBJECT as follows:

```
TAL /IN mysource/ myobject
```

Following are options you can specify in the compilation command.

IN File Option

In the compilation command, the IN file is the source file. You can specify a file name or a TACL DEFINE as described in Appendix E. In the preceding example, the IN file is MYSOURCE.

The IN file can be an edit-format disk file, a terminal, a magnetic tape unit, or a process. The compiler reads the file as 132-byte records.

If you omit the IN file and the TACL product is in interactive mode, the default file is your home terminal. In noninteractive mode, the default file is the TACL command file. For information about the TACL product, see the *TACL Reference Manual*.

OUT File Option

The OUT file receives the compiler listings. The OUT file can be a disk file (not in edit format), a terminal, a line printer, a spooler location, a magnetic tape unit, or a process.

In an unstructured disk file, each record has 132 characters; partial lines are filled with blanks through column 132. You can specify a file name or a TACL DEFINE name. The file must exist before you specify its name for the OUT file. You can create the file by using the File Utility Program (FUP).

The following example specifies a file named MYLIST as the OUT file:

```
TAL /IN mysource, OUT mylist/ myobject
```

The OUT file is often a spooler location, in this case, \$\$.#LISTS:

```
TAL /IN mysource, OUT $s.#lists/ myobject
```

If you specify OUT with no file, you suppress the listings:

```
TAL /IN mysource, OUT/ myobject
```

If you omit OUT and the TACL product is in interactive mode, the listings go to the home terminal. In noninteractive mode, the listings go to the current TACL OUT file:

```
TAL /IN mysource/ myobject
```

TACL Run Options

You can include one or more TACL run options in the compilation command, such as:

- A process name
- A CPU number
- A priority level
- The NOWAIT option

For example, you can specify CPU 3 and NOWAIT when you run the compiler:

```
TAL /IN mysource, CPU 3, NOWAIT/ myobject
```

Another run option you can specify is the MEM (memory) option, but the compiler always uses 64 pages. For information on all the TACL run options, see the RUN command in the *TACL Reference Manual*.

Target File Option

The target file is the disk file that is to receive the object code. You can specify a file name or a TA CL DEFINE name as described in Appendix E.

Previous examples sent the object code to a disk file named MYOBJECT:

```
TAL /IN mysource/ myobject
```

If you omit the target file, BINSERV creates a file named OBJECT on your current default subvolume. If an existing file has the name OBJECT or the name you specify, BINSERV purges the file before creating the new target file. If the existing file is secured so BINSERV cannot purge it, BINSERV creates a file named ZZBINnnn, where nnnn is a different number each time.

Compiler Directives

You can include one or more compiler directives in the compilation command. Precede the directives with a semicolon and separate them with commas.

You can control the compilation listing. For example, NOMAP suppresses the symbol map, and CROSSREF produces cross-reference listings:

```
TAL /IN mysource/ myobject; NOMAP,CROSSREF
```

You can specify any directive in the compilation command except the following, which can appear only in the source file:

```
ASSERTION  
BEGINCOMPILATION  
DECS  
DUMPCONS  
ENDIF  
IF  
IFNOT  
PAGE  
RP  
SECTION  
SOURCE
```

The following directives can appear only in the compilation command:

```
EXTENDTALHEAP  
SQL with the PAGES option  
SYMBOLPAGES
```

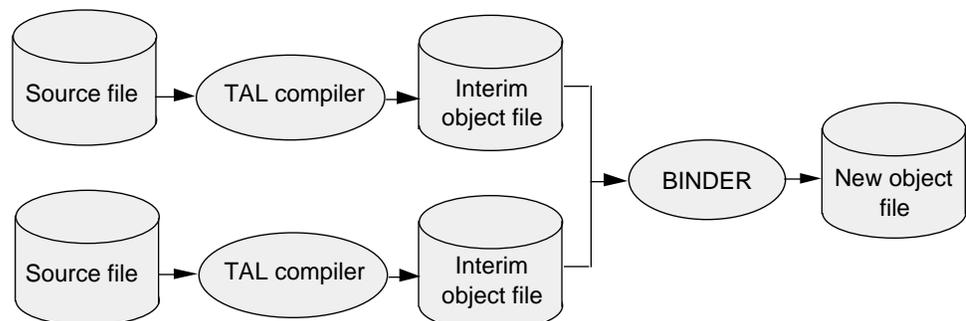
Completion Codes Returned by the Compiler When the compiler compiles a source file, it either completes the compilation normally or stops abnormally. It then returns a process-completion code to the TACL product indicating the status of the compilation. Table 14-1 explains the process-completion code values.

Table 14-1. Completion Codes

Code	Termination	Meaning
0	Normal	The compiler found no errors or unsuppressed warnings in the source file. (Warnings suppressed by the NOWARN directive do not count.) The object file is complete and valid (unless a SYNTAX directive suppressed its creation).
1	Normal	The compiler found at least one unsuppressed warning. (Warnings suppressed by the NOWARN directive do not count.) The object file is complete and valid (unless a SYNTAX directive suppressed its creation).
2	Normal	The compiler found at least one compilation error and did not create an object file.
3	Abnormal	The compiler exhausted an internal resource such as symbol table space or could not access an external resource such as a file. The compiler did not create an object file.
5	Abnormal	The compiler discovered a logic error during internal consistency checking or one of the compiler's server processes terminated abnormally. The compiler did not create an object file.
8	Normal	The compiler could not use the object file name you specified, so it chose the name reported in the summary. The object file is complete and valid.

Binding Object Files You can compile source files into interim object files and then use Binder to bind the interim object files into a new object file, as shown in Figure 14-2.

Figure 14-2. Binding Object Files



Binding can take place:

- During a compilation session
- After a compilation session
- At run time (library binding)

Binding During Compilation During compilation, BINSERV constructs a master search list of object files from SEARCH directives in the source file. After a successful compilation, BINSERV binds into the new object file any procedures from object files listed in the master search list that resolve external references.

You can do further binding on the object file produced by BINSERV by using Binder or the operating system.

Binding After Compilation After compilation, you can bind object files interactively by using Binder as described in the *Binder Manual*. For example, you can build a target file from separate object files, display the content of object files, reorder target-file code blocks, produce optional load maps and cross-reference listings, specify a user run-time library, and modify the content of named global data blocks and code blocks in the target file.

Binding at Run Time You can build a library of procedures to share at run time among applications or to extend an application's code space. At run time, the operating system binds the library file to the program file. You store the run-time library in a separate file, and then associate the library file with your object file by using any of the following methods:

- A LIBRARY directive in the source file
- The Binder SET LIBRARY command, described in the *Binder Manual*
- The TACL RUN LIB command, described in the *TACL Reference Manual*

The LIBRARY directive lets you specify a user library to search before searching the system library for satisfying external references. LIBRARY can appear anywhere on the compilation command or in the source code:

```
!Lots of code
?LIBRARY mylib
!More code
```

**Using Directives
in the Source File**

Compiler directives let you:

- Specify input source code
- Control listings, generate the object code, and build the object file
- Perform conditional compilation

A directive line in your source file can contain any number of compiler directives. You must start each new directive line and any continuation lines with a question mark (?) in column 1.

The following directive line contains one directive:

```
?NOLIST
```

The following directive lines contain multiple directives:

```
?NOLIST, NOCODE, INSPECT, SYMBOLS, NOMAP, NOLMAP, GMAP  
?CROSSREF, INNERLIST
```

The following directive line shows a continuation line for the argument list of a directive:

```
?SEARCH (file1, file2, file3, file4,  
?file5, file6)
```

If the list of arguments in a directive continues on subsequent lines, you must specify the leading parenthesis of the argument list on the same line as the directive name:

```
?NOLIST, SOURCE $SYSTEM.SYSTEM.EXTDECS (  
? PROCESS_GETINFO_  
? PROCESS_STOP_)
```

Using Directive Stacks Several directives have a compile-time directive stack on which you can push and pop directive settings. Directives that have directive stacks are:

```
CHECK
CODE
DEFEXPAND
ICODE
INNERLIST
INT32INDEX
LIST
MAP
```

Each directive stack is 32 levels deep. The compiler initially sets all levels of each directive stack to the off state.

Pushing Directive Settings When you push the current directive setting onto a directive stack, the current directive setting of the source file remains unchanged until you specify a new directive setting.

To push a directive setting onto a directive stack, specify the directive name prefixed by PUSH. For example, to push the current setting of the LIST directive onto the LIST directive stack, specify PUSHLIST. The other values in the directive stack move down one level. If a value is pushed off the bottom of the directive stack, that value is lost.

Popping Directive Settings To restore the top value from a directive stack as the current setting of the source file, specify the directive name prefixed by POP. For example, to restore the top value off the LIST directive stack, specify POPLIST. The remaining values in the directive stack move up one level, and the vacated level at the bottom of the stack is set to the off state.

Directive Stack Example In the following example:

1. LIST is the default setting for the source file.
2. PUSHLIST pushes the LIST directive setting onto the LIST directive stack.
3. NOLIST suppresses listing of sourced-in procedures.
4. POPLIST pops the top value off the LIST directive stack and restores LIST as the current setting for the remainder of the source file:

```
!LIST is the default setting for the source file
?PUSHLIST, NOLIST, SOURCE $SYSTEM.SYSTEM.EXTDECS (
? PROCESS_GETINFO_, FILE_OPEN_, WRITEREADX, READX)
?POPLIST
```

File Names as Directive Arguments

The following directives accept names of disk files as arguments:

ERRORFILE
LIBRARY
SAVEGLOBALS
SEARCH
SOURCE
USEGLOBALS

A **disk file name** consists of four parts, with each part separated by periods:

- A D-series node name or a C-series system name
- A volume name
- A subvolume name
- A file ID

Here is an example of a file name:

```
\mynode.$myvol.mysubvol.myfileid
```

Partial File Names

You can omit any part of the file name except the file ID. If you specify a partial file name, the compiler uses the current default node (system), volume, and subvolume as needed.

For the SEARCH, SOURCE, and USEGLOBALS directives, the compiler can also use the node (system), volume, and subvolume specified in TACL ASSIGN SSV (Search SubVolume) commands.

Logical File Names

The following directives accept a logical file name in place of a file name:

ERRORFILE
SAVEGLOBALS
SEARCH
SOURCE
USEGLOBALS

A logical file name is a TACL DEFINE name or a TACL ASSIGN name.

Appendix E gives more information on specifying disk file names, including those specified in TACL DEFINE and ASSIGN commands.

Compiling With SOURCE Directives

You can specify a SOURCE directive in a source file to read in source code from other source files. In the SOURCE directive, specify the source file name, followed by an optional list of one or more section names enclosed in parentheses. If you omit the list of section names, the compiler reads in the entire file.

Section Names

If you specify SOURCE with no section names, the compiler processes the specified source file until an end of file occurs. The compiler treats any SECTION directives in the source file as comments.

If you specify SOURCE with section names, the compiler processes the source file until it reads all the specified sections. A section begins with a SECTION directive and ends with another SECTION directive or the end of the file, whichever comes first.

The compiler reads the sections in order of appearance in the source file, not in the order specified in the SOURCE directive. If you want the compiler to read sections in a particular order, use a separate SOURCE directive for each section and place the SOURCE directives in the desired order.

Nesting Levels

You can nest SOURCE directives to a maximum of seven levels, not counting the original outermost source file. For example, the deepest nesting allowed is as follows:

1. The MAIN file F sources in file F1.
2. File F1 sources in file F2.
3. File F2 sources in file F3.
4. File F3 sources in file F4.
5. File F4 sources in file F5.
6. File F5 sources in file F6.
7. File F6 sources in file F7.

Effect of Other Directives

If LIST and NOSUPPRESS are in effect after a SOURCE directive completes execution, the compiler prints a line identifying the source file to which it reverts and begins reading at the line following the SOURCE directive.

You can precede SOURCE with NOLIST to suppress the listings of procedures to be read in. Place NOLIST and SOURCE on the same line, because the line containing NOLIST is not suppressed:

```
?PUSHLIST, NOLIST, SOURCE $src.current.routines
!Suppress listings; read in external declarations of routines
?POPLIST
```

If USEGLOBALS is in effect, the compiler ignores all SOURCE directives until it encounters BEGINCOMPILATION. For more information on how these directives interact, see “Compiling With Saved Global Data” later in this section.

**Including System
Procedure Declarations**

You can use SOURCE directives to read in external declarations of system procedures from the EXTDECS files. In these files, the procedure name and the corresponding section name are the same. EXTDECS0 contains the current release version of system procedures, for example, the D20 version.

In the following D-series example, a SOURCE directive specifies the current version of system procedures. A NOLIST directive suppresses the listings for the system procedures. Place NOLIST and SOURCE on the same line, because the line containing the NOLIST directive is not suppressed:

```
?PUSHLIST, NOLIST, SOURCE $SYSTEM.SYSTEM.EXTDECS0 (
?  PROCESS_DEBUG_, PROCESS_STOP_)
!Suppress listings; read in external declarations of
! current system procedures
?POPLIST
```

A D-series procedure in the same source file can then call the procedures listed in the preceding SOURCE directive:

```
PROC a MAIN;
  BEGIN
  INT x, y, z, error;
  !Code for manipulating x, y, and z
  If x = 5 THEN CALL PROCESS_STOP_;
  CALL PROCESS_DEBUG_;           !Call procedures listed
  END;                           ! in SOURCE directive
```

To convert the two preceding examples to C-series examples, change the procedure names PROCESS_STOP_ and PROCESS_DEBUG_ to STOP and DEBUG, respectively.

Compiling With Search Lists

You can share data and procedures between object files by specifying search lists of object file names for resolving unsatisfied external references and validating parameter lists at the end of the compilation session.

To create search lists of object files, use the SEARCH directive:

```
?SEARCH (file1, file2, file3)
```

Creating the Master Search List

The compiler sends the search list from each SEARCH directive to BINSERV, the compile-time binder process. BINSERV appends the file names, in the order specified, to the master search list for the current source file.

For example, if you specify the following SEARCH directives, the master search list is in the order FILE2, FILE1, FILE3, and FILE4:

```
?SEARCH (file2, file1)
!Lots of code
?SEARCH (file3, file4)
```

Clearing the Master Search List

You can clear the current master search list at any point in the source file. BINSERV uses only the files that remain on the search list at the end of compilation to resolve external references.

To clear the master search list at any point, specify a SEARCH directive with no file names. For example, suppose you specify two search lists that comprise the master search list. You can then clear the master search list as follows:

```
?SEARCH (file1, file2)           !Specify search list
?SEARCH (file3)                 !Add FILE3 to search list
?SEARCH                          !Clear master search list
                                ! of FILE1, FILE2, FILE3
!Lots of code
?SEARCH (file4, file5)          !Specify new search list;
                                ! master search list is
                                ! now FILE4 and FILE5
```

Searching the Master Search List BINSERV searches the object files in the order in which they appear in the master search list. If a procedure or entry-point name that resolves an external reference appears in more than one file, BINSERV uses only the first occurrence. Thus, the order in which you specify the files could be important.

Binding the Master Search List If the compilation is successful, BINSERV binds the new object file by using procedures from object files in the master search list to resolve any unsatisfied references in your program. If procedures from object files in the search list contain references to other external procedures or to data blocks, BINSERV tries to resolve those from object files in the master search list.

This example shows SEARCH directives for external procedures:

```
?SEARCH partx           !Object file containing PROC_X
PROC proc_x;
  EXTERNAL;

?SEARCH party           !Object file containing PROC_Y
PROC proc_y;
  EXTERNAL;

PROC proc_z;
  BEGIN
  CALL proc_x;
  CALL proc_y;
  END;
```

Retrieving Global Initializations You can use SEARCH to retrieve global initialization values and template structure declarations as described in “Compiling With Saved Global Data” later in this section.

Compiling With Relocatable Data Blocks

When you compile modules of a program separately or bind TAL code with code written in other languages, the binding process might relocate some of your global data. All global data to be shared with compilation units written in other languages must be relocatable.

Declaring Relocatable Global Data

You can declare blocked and unblocked relocatable global data (variables, LITERALS, and DEFINES).

Blocked global data declarations are those appearing within BLOCK declarations. BLOCK declarations let you group global data declarations into named or private blocks. Named blocks are shareable among all compilation units in a program. The private block is private to the current compilation unit. If you include a BLOCK declaration in a compilation unit, you must assign an identifier to the compilation unit by using a NAME declaration.

Unblocked global data declarations are those appearing outside a BLOCK declaration. Such declarations are also relocatable and shareable among all compilation units in a program.

If present in a compilation unit, global declarations must appear in the following order:

1. NAME declaration
2. Unblocked global data declarations
3. BLOCK declarations
4. PROC declarations

Naming Compilation units

To assign an identifier to a compilation unit, specify the NAME declaration as the first declaration in the compilation unit. (If no BLOCK declaration appears in the compilation unit, you need not include the NAME declaration.) In the NAME declaration, specify an identifier that is unique among all BLOCK and NAME declarations in the target file. The following example assigns the identifier INPUT_MODULE to the current compilation unit.

```
NAME input_module;           !Name the compilation unit
```

Declaring Named Data Blocks

A named data block is a global data block that is shareable among all compilation units in a program. You can include any number of named data blocks in a compilation unit. To declare a named data block, specify an identifier in the BLOCK declaration that is unique among all BLOCK and NAME declarations in the target file. The following declaration assigns the identifier GLOBALS to the named data block:

```
BLOCK globals;              !Declare named data block
  INT .vol_array[0:7];      !Declare global data
  INT .out_array[0:34];
  DEFINE xaddr = INT(32)#;
END BLOCK;
```

As of the D20 release, a variable declared in a named data block can have the same name as the data block. Modules written in TAL can share global variables with modules written in C by placing each shared variable in its own block and giving the variable and the block the same name. Here is an example of a variable named the same as the data block:

```
BLOCK c_var;
  INT c_var;
END BLOCK;
```

Declaring Private Data Blocks

A private data block is a global data block that is shareable only among the procedures within a compilation unit. You can include only one private data block in a compilation unit. The private data block inherits the identifier you specify in the NAME declaration. To declare a private global data block, specify the PRIVATE option of the BLOCK declaration:

```
BLOCK PRIVATE;                !Declare private global data block
  INT term_num;                !Declare global data
  LITERAL msg_buf = 79;
END BLOCK;
```

Specifying the Data Block Location

You can use the AT and BELOW clauses to control where Binder locates a block. For example:

- AT (0)—to detect the use of uninitialized pointers
- BELOW (64)]—to use XX (extended, indexed) machine instructions
- BELOW (256)—to use directly addressed global data

For example, you can specify where to allocate data blocks as follows:

```
BLOCK ext_indexed_stuff BELOW (64);    !Specify location
  INT .EXT symbol_table[0:32760];
  INT .EXT error_table[0:16380];
END BLOCK;
```

The following limitations apply to the AT and BELOW clauses:

- Using the AT[0] option might cause conflicts if you:
 - Share data with compilation units written in other languages
 - Run your program in the CRE
 - Use 0D as nil for pointers
- Some of the AT and BELOW clauses are not portable to future software platforms.

Declaring Unblocked Data

Place all unblocked global declarations (those not contained in BLOCK declarations) before the first BLOCK declaration. Unblocked declarations are relocatable and shareable among all compilation units in a program.

Here is an example of unblocked data declarations:

```
INT a;
INT .b[0:9];
INT .EXT c[0:14];
LITERAL limit = 32;
```

The compiler places unblocked data declarations in implicit primary data blocks. As of the D20 release, the compiler creates implicit data blocks as follows:

- A data block named #GLOBAL for all unblocked declarations except template structures. A compilation unit can have only one #GLOBAL block.
- A data block for each unblocked template structure declaration. The data block for a given template structure is given the template name prefixed with an ampersand (&).

You can bind object files compiled with and without template blocks with no loss of information. You can use Binder commands to replace the #GLOBAL and template blocks in the target file.

Referencing Declarations

A referral structure and the structure layout to which it refers can appear in different data blocks. The structure layout must appear first.

In all other cases, a data declaration and any data to which it refers must appear in the same data block. The following declarations, for example, must appear in the same data block:

```
INT var;                                !Declare VAR
INT .ptr := @var;                       !Declare PTR by referring to VAR
```

Allocating Global Data Blocks

When you compile a program, the compiler constructs relocatable blocks of code and data that are bound into the object file. The compiler:

- Allocates each read-only array in its own data block in the user code segment in which the array is referenced
- Allocates all other variables in relocatable global data blocks in the user data segment (except LITERALS and DEFINES, which require no storage space)

In the user data segment, the compiler creates and names global data blocks that are primary, secondary, or extended.

Primary Relocatable Data Blocks

The compiler creates and names primary data blocks as follows:

Primary Block	Compiler-Assigned Name
Implicit	#GLOBAL
Implicit	& <i>template-name</i> (for each unblocked template structure)
Named	The identifier specified in the BLOCK declaration
Private	The identifier specified in the NAME declaration

The compiler allocates the following variables in primary data blocks:

- Directly addressed variables
- Standard or extended pointers (including those you declare and those the compiler provides when you declare indirect arrays and structures)

The compiler associates the symbol information for the allocated variables with that data block. The compiler also associates the symbol information for any LITERALS, DEFINES, or read-only arrays declared in that data block, but allocates 0 words of storage for such declarations. If a global data block contains only LITERALS, DEFINES, or read-only arrays, the compiler creates a primary data block and associates their symbol information with the data block, but allocates 0 words of storage for the data block.

Size of Combined Primary Blocks. After a binding session, the combined primary global data blocks in the resulting object file must not exceed 256 words.

Secondary Relocatable Data Blocks

The compiler creates and names secondary data blocks as follows:

Secondary Block	Compiler-Assigned Name
Implicit	.#GLOBAL
Named	The identifier specified in the BLOCK declaration, prefixed with a dot (.)
Private	The identifier specified in the NAME declaration, prefixed with a dot (.)

Secondary data blocks contain the data of standard indirect arrays and standard indirect structures.

Extended Relocatable Data Blocks

The compiler names the extended data blocks as follows:

Extended Block	Compiler-Assigned Name
Implicit	\$#GLOBAL
Named	The identifier specified in the BLOCK declaration, prefixed with \$
Private	The identifier specified in the NAME declaration, prefixed with \$

Extended data blocks contain the data of extended indirect arrays and extended indirect structures.

Example of Data Blocks Created by the Compiler

Table 14-2 shows the primary, secondary, and extended data blocks the compiler creates from the example global data declarations, including the names (shown in boldface) that the compiler gives them.

Table 14-2. Data Blocks Created by the Compiler

		Data Blocks Created by the TAL Compiler		
	Example Declaration	Primary Data Block	Secondary Data Block	Extended Data Block
Implicit block	<pre>INT a; INT .b[0:9]; INT .EXT c[0:14]; INT(32) .d; LITERAL lmt = 32;</pre>	<p>#GLOBAL contains: Variable A (1 word) Pointer for B (1 word) Pointer for C (2 words) Simple pointer D (1 word) LITERAL (0 words)</p>	<p>.#GLOBAL contains: Data for B (10 words)</p>	<p>\$\$GLOBAL contains: Data for C (15 words)</p>
Named block	<pre>BLOCK myglobals; INT g; INT .h[0:9]; INT .EXT k[0:14]; LITERAL one = 1; END BLOCK;</pre>	<p>MYGLOBALS contains: Variable I (1 word) Pointer for J (1 word) Pointer for K (2 words) LITERAL (0 words)</p>	<p>.MYGLOBALS contains: Data for J (10 words)</p>	<p>\$\$MYGLOBALS contains: Data for K (15 words)</p>
Private block	<pre>NAME mysource; BLOCK PRIVATE; INT x; INT .y[0:9]; INT .EXT z[0:14]; DEFINE xaddr = INT(32)#; INT ro_array = 'P' := [0,1]; END BLOCK;</pre>	<p>MYSOURCE contains: Variable X (1 word) Pointer for Y (1 word) Pointer for Z (2 words) DEFINE (0 words) Read-only array (0 words)</p>	<p>.MYSOURCE contains: Data for Y (10 words)</p>	<p>\$\$MYSOURCE contains: Data for Z (15 words)</p>

Address Assignments

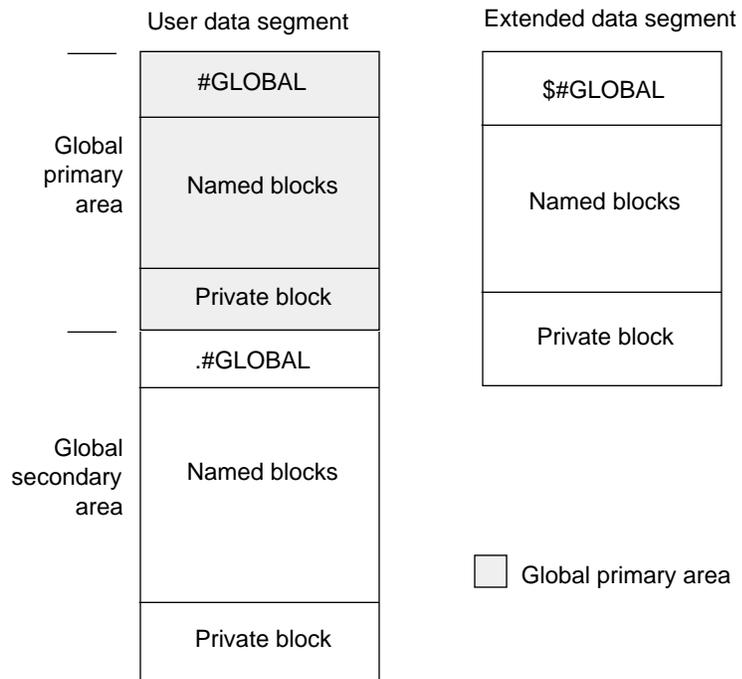
The compiler assigns each direct variable and each pointer an offset from the beginning of the encompassing global data block. Within the data block, it allocates storage for each data declaration according to its data type and size.

Binder uses the address of the data block and the offset within the block to construct addresses for indirect data in the secondary and extended storage areas.

Allocation Example

Figure 14-3 shows the global storage allocation resulting from binding object files that contain BLOCK declarations. You can rearrange the primary block by using Binder commands. Secondary blocks must always follow the primary blocks.

Figure 14-3. Allocating Global Data Blocks



Sharing Global Data Blocks Because the length of any shared data block must match in all compilation units, it is recommended that you declare all shareable global data in one source file. You can then share that global data block with other source files as follows:

1. In the source file that declares the data block, specify the SECTION directive at the beginning of the data block to assign a section name to the data block. The SECTION directive remains in effect until another SECTION directive or the end of the source file occurs:

```

NAME calc_unit;

?SECTION unblocked_globals           !Name first section
  LITERAL true   = -1,               !Implicit data block
         false  =  0;
  STRING read_only_array = 'P' := [ " ", "COBOL",
                                     "FORTRAN", "PASCAL", "TAL"];

?SECTION default                     !Name second section
  BLOCK default_vol;                !Declare named block
  INT .vol_array [0:7],
     .out_array [0:34];
  END BLOCK;

?SECTION msglits                     !Name third section
  BLOCK msg_literals;               !Declare named block
  LITERAL
    msg_eof   = 0,
    msg_open  = 1,
    msg_read  = 2;
  END BLOCK;                          !End msglits section

?SECTION end_of_data_sections

```

2. In each source file that needs to include the sections, specify the file name and the section names in a SOURCE directive:

```

NAME input_file;

?SOURCE calcsrc(unblocked_globals) !Specify implicit block
?SOURCE calcsrc(default)           !Specify named block

```

3. If you then change any declaration within a data block that has a section name, you must recompile all source files that include SOURCE directives listing the changed data block.

Directives for Relocatable Data The RELOCATE and INHIBITXX directives help you make sure that your global data is relocatable. These directives do not affect local and sublocal variables.

RELOCATE Directive

RELOCATE instructs the compiler and BINSERV to issue warnings if references to nonrelocatable data occur. RELOCATE can appear anywhere on the compilation command or in the source file. It is effective for the source code that follows it.

The following RELOCATE example shows base-address equivalencing, which declares nonrelocatable data because it locates variables relative to the global, local, or sublocal base address.

```
?RELOCATE
INT i = 'G' + 22;           !Nonrelocatable global data;
                           ! base-address equivalencing

!Some code
i := 25;                   !Compiler emits warning
```

For more information on base-address equivalencing and the RELOCATE directive, see the *TAL Reference Manual*.

INHIBITXX Directive

INHIBITXX suppresses efficient addressing for extended pointers, extended indirect array elements, and extended indirect structure items located within the first 64 words of primary global storage.

INHIBITXX has no effect on data declared in BLOCK declarations with the AT (0) or BELOW (64) option. The compiler always generates efficient code for such BLOCK declarations regardless of INHIBITXX.

If the default NOINHIBITXX is in effect, the compiler produces efficient addressing that might become incorrect if binding relocates extended pointers, extended indirect array elements, or extended indirect structure items outside the first 64 words.

The INT32INDEX directive overrides INHIBITXX or NOINHIBITXX.

Specify INHIBITXX or NOINHIBITXX immediately before the global declarations to which it applies. The specified directive then applies to those declarations throughout the compilation. The following example shows how NOINHIBITXX generates efficient addressing, while INHIBITXX suppresses efficient addressing:

```

!Default NOINHIBITXX in effect; assign NOINHIBITXX
! attribute to subsequent declaration.

STRUCT .EXT xstruct[0:9]; !XSTRUCT has NOINHIBITXX
  BEGIN                    ! attribute.
  STRING array[0:9];
  END;

INT index;
STRING var;
?INHIBITXX                !Assign INHIBITXX attribute to
                          ! subsequent declaration.
STRING .EXT xstruct2 (xstruct);
                          !XSTRUCT2 has INHIBITXX attribute.
                          !Preceding declarations are
                          ! allocated in #GLOBAL and
                          ! $#GLOBAL.

PROC my_proc MAIN;
  BEGIN
  @xstruct2 := @xstruct;
  var := xstruct[index].array[0];
                          !Generate efficient addressing
                          ! because XSTRUCT has NOINHIBITXX
                          ! attribute, but if Binder
                          ! relocates #GLOBAL beyond G[63],
                          ! the addressing is incorrect.

  var := xstruct2[index].array[0];
                          !Generate less efficient addressing
                          ! because XSTRUCT2 has INHIBITXX
                          ! attribute; the addressing is
                          ! correct even if Binder relocates
                          ! #GLOBAL.

  END;

```

Compiling With Saved Global Data

During program development or maintenance, you often need to change procedural code or data without changing the global declarations. You can save the global data in a file during a compilation session and then use the saved global data during a subsequent compilation. You can shorten the compile time by not compiling global declarations each time.

Saving Global Data

To save the compiled global data declarations, use the SAVEGLOBALS directive. SAVEGLOBALS causes the data declarations to be stored as follows:

- Identifiers and data characteristics (including data type and kind of variable) in a global declarations file
- Initialization values (including addresses and constant lists) in the object file

Note

Whenever you switch to a different version of the compiler, you must create a new global declarations file by using SAVEGLOBALS. Otherwise, an error message occurs when you compile to retrieve the saved globals. Each version of the compiler expects declarations in a different format. (C30, D10, and D20, for example, are different versions of the compiler.)

Retrieving Global Data

After a SAVEGLOBALS compilation completes successfully, you can retrieve the global data declarations and initializations in a subsequent USEGLOBALS compilation by specifying the following directives:

Directive for Retrieving Global Data	Effect
USEGLOBALS	Retrieves global data declarations; suppresses compilation of text lines and SOURCE directives (but not other directives) until BEGINCOMPILATION appears
SEARCH	Retrieves global initialization values and template structures
BEGINCOMPILATION	Begins compilation of text lines and SOURCE directives

Specify BEGINCOMPILATION between the last global data declaration or SEARCH directive and the first procedure declaration, including EXTERNAL or FORWARD declarations. (You must recompile EXTERNAL or FORWARD procedure declarations in the USEGLOBALS compilation. SAVEGLOBALS does not save such declarations.)

Note

If you specify SAVEGLOBALS and USEGLOBALS in the same compilation, the compiler issues an error message and uses only the first of the two directives.

If you use CROSSREF with USEGLOBALS, the compiler does not pass Inspect and CROSSREF symbols information for global identifiers to SYMSERV.

Saved Global Data Compilation Session The following session shows how you can save and retrieve global data. It also shows how you can check the syntax of global data declarations and how to save and retrieve such declarations.

Creating the Source File

Using an editor, you can create a source file, such as MYPROG, that includes BEGINCOMPILATION and USEGLOBALS.

```

!Source file MYPROG

!If USEGLOBALS is active, the compiler ignores text lines
! and SOURCE directives (but not other directives) until
! BEGINCOMPILATION appears.

?SOURCE globfile
?SOURCE glbfile1 (section1, section2)
?SOURCE moreglbs
    INT ignore_me1;
    INT ignore_me2;

?BEGINCOMPILATION                !Compile code that follows
?PUSHLIST, NOLIST, SOURCE $SYSTEM.SYSTEM.EXTDECS
?POPLIST

PROC my_first_proc;
    BEGIN
    !Lots of code
    END;

PROC my_last_proc;
    BEGIN
    !Lots of code
    END;

```

Saving the Global Data

The following compilation command compiles MYPROG and produces object file MYOBJ. The SAVEGLOBALS directive saves global declarations in file TALSYM and the global initializations in object file MYOBJ:

```
TAL /IN myprog/ myobj; SAVEGLOBALS talsym
```

Retrieving the Saved Global Data

The following compilation command recompiles MYPROG and produces object file NEWOBJ. The USEGLOBALS directive retrieves the saved global declarations from file TALSYM. The SEARCH directive retrieves the global initializations from MYOBJ:

```
TAL /IN myprog/ newobj; USEGLOBALS talsym, SEARCH myobj
```

Checking the Syntax of Global Data

The following compilation command compiles source file MYPROG but produces no object file. The SAVEGLOBALS directive saves global declarations in file TALSVM. The SYNTAX directive checks the syntax of the global declarations:

```
TAL /IN myprog/; SAVEGLOBALS talsvm, SYNTAX
```

You can then recompile and retrieve the saved global declarations (but not the saved global initializations, because no object file was produced in the preceding compilation):

```
TAL /IN myprog/; USEGLOBALS talsvm, SYNTAX
```

Effects of Other Directives When you use the following directives in the SAVEGLOBALS compilation, they affect subsequent USEGLOBALS compilations as follows:

Directive in SAVEGLOBALS Compilation	Effect in Subsequent USEGLOBALS Compilations
SYNTAX	Negates the need for using SEARCH in the USEGLOBALS compilation because no object file was produced by the SAVEGLOBALS compilation
INHIBITXX	Continues to inhibit generation of extended indexed instructions for extended pointers located in the first 64 words of primary global area
INT32INDEX	Continues to generate INT(32) indexes from INT indexes
PRINTSYM	Continues to print symbols in the listing
SYMBOLS	Continues to make symbols available for all data blocks that had symbols during the SAVEGLOBALS compilation

These directives set the corresponding attribute in ensuing variable declarations. The compiler saves such information in the SAVEGLOBALS object file along with all the other information it saves about each variable.

Compiling With Cross-References

The CROSSREF directive either:

- Collects source-level cross-reference information produced during compilation
- Selects the identifier classes for which you want to collect cross-references

The default is NOCROSSREF.

You can specify CROSSREF or NOCROSSREF in the compilation command or any number of times anywhere in the source file.

Note

If you use CROSSREF with USEGLOBALS, the compiler does not pass Inspect and cross-reference symbols information for global identifiers to SYMSERV.

Selecting Classes

You can specify that the compiler collects cross-reference information for one or more of the following classes:

Class	Description
BLOCKS	Named and private data blocks
DEFINES	Named text
LABELS	Statement labels
LITERALS	Named constants
PROCEDURES	Procedures
PROCPARAMS	Procedures that are formal parameters
SUBPROCS	Subprocedures
TEMPLATES	Template structures
UNREF	Unreferenced identifiers
VARIABLES	Simple variables, arrays, definition structures, referral structures, pointers, and equivalenced variables

The default class list includes all classes except UNREF. The CONSTANTS class is available in the stand-alone Crossref product, but not in the CROSSREF directive.

You can make changes to the current class list at any point in the compilation unit. When you specify parameters, CROSSREF and NOCROSSREF only modifies the class list. To start (or stop) the collection of cross-references, you must specify CROSSREF (or NOCROSSREF) without parameters. The compiler collects cross-references for the class list in effect at the end of the compilation.

You add or delete classes from the current class list as follows. When you list more than one class, enclose the list in parentheses.

- To add classes to the current list, specify `CROSSREF` and list the classes you want to add. The following example adds the one missing class to the default list:

```
?CROSSREF UNREF
```

- To delete classes from the current list, specify `NOCROSSREF` and list the classes you want to delete. The following example retains the procedure, subprocedure, block, and template classes by deleting all other classes from the default list:

```
?NOCROSSREF (DEFINES, LABELS, LITERALS, PROCPARAMS, VARIABLES)
```

- To add and delete classes, specify a `CROSSREF` that adds classes and a `NOCROSSREF` that deletes classes:

```
?CROSSREF UNREF, NOCROSSREF LITERALS
```

Collecting Cross-References

You can collect cross-reference information for individual procedures or data blocks. When a `CROSSREF` without parameters appears, it starts collection of cross-references at the beginning of a procedure or data block and remains in effect until a `NOCROSSREF` without parameter appears. `CROSSREF` and `NOCROSSREF` without parameters do not modify the class list.

For each class in effect at the end of the compilation, `CROSSREF` without parameters collects the following information:

- Identifier qualifiers—structure, subprocedure, and procedure identifiers
- Compiler attributes—class and type modifiers
- The name of the host source file
- The type of reference—definition, invocation, parameter, write, or other

To start collecting cross-references, specify `CROSSREF` without parameters. To stop collecting cross-references, specify `NOCROSSREF` without parameters. For example, you can stop the collection for the private data block, and then start the collection for a procedure:

```
?CROSSREF          !Start collecting cross-references
NAME test;
  INT i;

?NOCROSSREF        !Stop cross-references for BLOCK
BLOCK PRIVATE;
  INT j;
  END BLOCK;

?CROSSREF          !Start cross-references for procedure
PROC p MAIN;
  BEGIN
  !Lots of code
  END;
```

You can add and delete classes when you start the collection:

```
?CROSSREF, CROSSREF UNREF, NOCROSSREF VARIABLES
                                !Start collecting cross-references and
                                ! change the class list

NAME test;
  INT i;

?NOCROSSREF                      !Stop cross-references for BLOCK
BLOCK PRIVATE;
  INT j;
  END BLOCK;

?CROSSREF, CROSSREF VARIABLES
                                !Start cross-references for procedure and
                                ! add a class to the class list

PROC p MAIN;
  BEGIN
    !Lots of code
  END;
```

For other cross-reference options, use the stand-alone Crossref product as described in the *Crossref Manual*. For example, stand-alone Crossref can collect cross-references from source files written in one or more languages.

Printing Cross-References

To print the collected cross-references in the compiler listing, LIST and NOSUPPRESS (the defaults) must be in effect at the end of compilation. CROSSREF collects cross-references even if NOLIST is in effect for all or part of the compilation. In the compiler listing, the cross-reference list follows the global map and precedes the load maps.

In the following example, SUPPRESS suppresses part of the cross-reference listing, and NOSUPPRESS resumes the listing for subsequent code:

```
!Default LIST and NOSUPPRESS are in effect.
?CROSSREF                      !Collect (and list) cross-references
PROC p;
  BEGIN
    !Some code
  END;

?SUPPRESS                      !Stop listing cross-references
PROC q;
  BEGIN
    !More code
  END;

?NOSUPPRESS                    !Resume listing cross-references
!More code
!LIST and NOSUPPRESS are in effect at the end of compilation.
```

The following example makes changes to the current class list and turns the collection or listing of cross-references on and off:

```

!Default LIST and NOSUPPRESS are in effect
?CROSSREF, CROSSREF UNREF, NOCROSSREF VARIABLES
                                !Collect (and list) cross-references
NAME test;
  INT i;

?NOCROSSREF                      !Stop collecting cross-references
BLOCK PRIVATE;
  INT j;
  END BLOCK;

?CROSSREF, CROSSREF VARIABLES
                                !Resume collecting (and listing)
                                ! cross-references

PROC p MAIN;
  BEGIN
  !Lots of code
  END;

?SUPPRESS                        !Stop listing cross-references
PROC q;
  BEGIN
  !More code
  END;

?NOSUPPRESS                      !Resume listing cross-references
!Lots more code
!LIST and NOSUPPRESS are in effect at the end of
! compilation.

```

15 Compiler Listing

This section describes the TAL listing and gives brief samples of the information. A TAL listing can consist of:

- Page header
- Banner
- Compiler messages
- Source listing
- Local or sublocal map
- INNERLIST listing
- CODE listing
- ICODE listing
- File map
- Global map
- Cross-reference listings
- LMAP listings
- Compilation statistics

Page Header The header for each page consists of:

- The page number of the listing
- The sequence number for the current source file
- The name of the current source file
- The date and time of compilation in the form *yy-mm-dd hh:mm:ss* (not shown)
- An optional page heading caused by the PAGE directive or by the compiler

In a listing for multiple source files, the header for pages that contain load maps, cross-references, and statistics shows the name and number of the first file. Figure 15-1 shows the format of the header:

Figure 15-1. Page Headers

Page No.	Source File	Source File Name	Optional Heading
Page 1	[1]	\$VOL.PROG1.SOURCE1S	
Page 2	[2]	\$VOL.PROG1.SOURCE2S	MY ROOT SOURCE FILE
Page 3	[2]	\$VOL.PROG1.SOURCE2S	MY ROOT SOURCE FILE
Page 4	[3]	\$SHR.MSGXX.IMGSHRS	INTERPROCESS MESSAGES
Page 59	[1]	\$VOL.PROG1.SOURCE1S	GLOBAL MAP
Page 66	[1]	\$VOL.PROG1.SOURCE1S	LOAD MAPS
Page 70	[1]	\$VOL.PROG1.SOURCE1	BINDER AND COMPILER STATISTICS

Banner The first page of the listing contains a banner, which consists of two lines that list the compiler version and the copyright notice. Figure 15-2 shows a sample banner.

Figure 15-2. Sample Banner

```
TAL - T9250D20 - (01JUN93)
Copyright Tandem Computers Incorporated 1976, 1978, 1981-1983, 1985,
1987-1993
```

Directives in Compilation Commands The line following the banner shows the directives you specified in the compilation command to run the compiler. For example, if you issue the following compilation command:

```
TAL /in mysrc, out mylst/ myobj; FMAP, ICODE
```

the line following the banner is:

```
? FMAP, ICODE
```

The compiler must process the EXTENDTALHEAP, SQL, and SYMBOLPAGES directives before it processes any other directives. On a D-series system, if you specify any of these three directives in the compilation command along with other directives, the compiler splits the command into two lines in the listing. The first line lists EXTENDTALHEAP, SQL, and SYMBOLPAGES, if present. The second line lists the remaining directives specified in the command.

For example, suppose you issue the following compilation command:

```
TAL /in .../ myobj; FMAP, SQL, ICODE, SYMBOLPAGES 4096
```

In a D-series listing, the directives listed in the preceding compilation command appear on two lines as follows:

```
?          SQL,          SYMBOLPAGES 4096
? FMAP,          ICODE
```

In a C-series listing, the directives listed in the preceding compilation command appear on the same line:

```
? FMAP, SQL, ICODE, SYMBOLPAGES 4096
```

After the list of directives specified in the compilation command, the compiler lists the source text if the LIST directive is in effect.

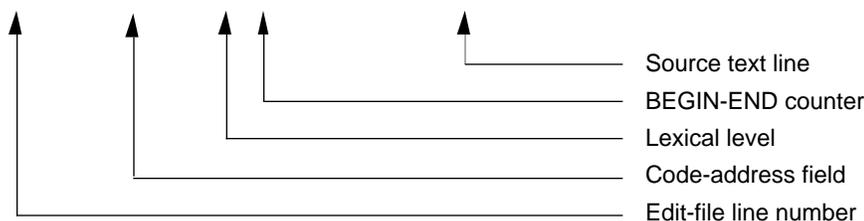
Compiler Messages When the compiler detects unusual conditions, it issues diagnostic messages interleaved with source statements. BINSERV diagnostic messages appear during and after the source listing.

Source Listing If the LIST directive is in effect (the default), the source text is listed line by line. Each line consists of:

- The edit-file line number
- The code-address field
- The lexical (nesting) level of the source text
- The BEGIN-END pair counter
- A text line from the source file

For example, here are two lines from a source listing:

```
31. 000021 1 1 IF length THEN BEGIN
32. 000023 1 2 CALL FILE_OPEN_ (array1, out_file);
```



413

Edit-File Line Number An edit-file line number precedes each line of source text. For text that is included in response to a SOURCE directive, the edit-file line numbers correspond to the file named in the SOURCE directive.

Code-Address Field The code address is a six-digit octal number. Depending on the line of source text, it represents an instruction offset or a secondary global count.

For a line of data declarations, the code-address value is a cumulative count of the amount of secondary global storage allocated for the program. The count is relative to the beginning of the secondary global storage. The beginning address is one greater than the last address assigned to primary global storage.

For a line of instructions, the code-address value is the address of the first instruction generated from the TAL source statement on the line. Normally, the octal value is the offset from the base of the current procedure; the instruction at the base has an offset of zero. Adding the offset to the procedure base address yields the code-relative address of the instruction. The procedure base address is listed in the entry-point load map (described later in this section).

If a procedure or subprocedure has initialized data declarations, the compiler emits code to initialize the data at the start of the procedure or subprocedure. The offset or code address listed for the first instruction is greater than one to allow for the initialization code.

If the ABSLIST directive is in effect, the compiler attempts to list the address for each line relative to location C[0]. The limitations on the use of ABSLIST are given in the directive description in the *TAL Reference Manual*.

Lexical-Level Counter The lexical-level counter is a single-digit value that represents the compiler's interpretation of the current source line. The values have the following meaning:

Value	Lexical Level
0	Global level
1	Procedure level
2	Subprocedure level

BEGIN-END Pair Counter The BEGIN-END pair counter indicates approximately the nesting of data elements (such as structures and substructures) and compound statements (such as IF statements, CASE statements, and CASE expressions). For unlabeled CASE statements, the counter also indicates the case selector.

To count BEGIN keywords and to match each with an END keyword in structure declarations and in instructions that generate code, the compiler increments the counter for each BEGIN and decrements it for each END. The compiler displays the value of the counter for each line of source text, except when it reports CASE selector values.

When listing a CASE statement body, the compiler reports the case selector in the form CE n , where n represents the case selector number. The compiler prints this string in the BEGIN-END pair counter column of the next line it displays when it has recognized the corresponding CASE branch. Because the compiler uses only one pass, however, by the time the compiler can distinguish an unlabeled CASE statement from a labeled one, it is usually too late to print the CE0 tag on the first line of case alternative zero. If case alternative zero constitutes only one line, the CE0 tag does not appear at all.

Figure 15-3 shows part of a sample listing page that illustrates the BEGIN-END pair counter.

Figure 15-3. Source Listing

```

?ICODE, SYMBOLS, SAVEABEND, INSPECT
2. 000000 0 0 NAME mymodule;
3. 000000 0 0
4. 000000 0 0 ?SOURCE outd
Source file: [2] $VOL.PROG1.OUTD 1993-04-22 09:22:45
1. 000000 0 0 !Size declarations
2. 000000 0 0
3. 000000 0 0 BLOCK out_data;
4. 000000 0 0 LITERAL
5. 000000 0 0 outblklen = 1024,
6. 000000 0 0 out_rec_len = 256;
7. 000000 0 0 END BLOCK;
5. 000000 0 0
Source file: [1] $VOL.PROG1.SOURCE1S 1993-05-13 19:18:07
.
.
24. 000000 0 0 PROC myproc;
25. 000000 1 0 BEGIN
26. 000000 1 1 STRING array1[0:7] := [" TPR "];
27. 000004 1 1 INT array2[0:11];
28. 000004 1 1 INT length, error;
.
.
31. 000021 1 1 IF length THEN BEGIN
32. 000023 1 2 CALL FILE_OPEN_ (array1, out_file);
33. 000032 1 2 IF < THEN BEGIN
34. 000033 1 3 CALL file_hndle (out_file, error);
.
.
37. 000051 1 3 END;
38. 000051 1 2 END
39. 000051 1 1 ELSE BEGIN

```

**Conditional
Compilation Listing**

An asterisk (*) in column 10 marks statements not compiled because of a conditional compilation directive (IF or IFNOT).

Local or Sublocal Map If the MAP and LIST directives are in effect (the defaults), a map of local or sublocal identifiers follows the corresponding source listing and gives information on the identifier class of an object, its variable type, and addressing. Table 15-1 lists the column headings and possible values. Only one of the columns named Addressing Mode, Offset, or Value appears in the map.

Table 15-1. Local/Sublocal Map Information

Column Heading	Meaning	Possible Value
Class	The Identifier class of the item. Variable (bytes-in-octal) denotes a structure.	Variable Variable (<i>bytes-in-octal</i>) Subproc Entry Label Define Literal
Type	For a VARIABLE class item, the data type or kind of structure. STRUCT-I denotes an INT structure pointer.	STRING INT INT(32) REAL REAL(64) FIXED STRUCT STRUCT-I SUBSTRUCT TEMPLATE (<i>bytes-in-octal</i>)
Addressing Mode	The direct or indirect addressing mode of the item.	Direct Indirect
Offset	The offset of a SUBPROC, ENTRY, or LABEL, relative to the base of the mapped PROC or SUBPROC. For a nested SUBPROC, the base corresponds to the current map.	%nnnnnn
Value	The value of a LITERAL or the text of a DEFINE truncated at the end of the listing line.	LITERAL value DEFINE text
Relative Address	For data, the base (L+, L-, P+, S-, or X) and the offset from the base in octal	L+ <i>nnn</i> (local variable) L- <i>nnn</i> (local parameter) P+ <i>nnn</i> (read-only array) S- <i>nnn</i> (sublocal parameter or variable) X00 <i>n</i> (index register)

Figure 15-4 shows a local map that corresponds to the following hash procedure:

```

INT PROC compute_hash (name, table_len);
    INT .name;
    INT(32) table_len;
BEGIN
    INT int_table_len := $INT (table_len);
    INT hash_val := 0;
    USE name_index;
    USE name_limit;

    name_limit := name.<8:14>;
    FOR name_index := 0 TO name_limit DO
        hash_val := ((hash_val '<<' 3) LOR
            hash_val.<0:2>) XOR name[name_index];
    DROP name_index;
    DROP name_limit;
    RETURN $UDBL($INT (hash_val '*' 23971)) '\
        int_table_len;
END; !compute_hash

```

Figure 15-4. Local Map

Identifier	Class	Type	Addressing Mode	Relative Address
HASH_VAL	Variable	INT	Direct	L+002
INT_TABLE_LEN	Variable	INT	Direct	L+001
NAME	Variable	INT	Indirect	L-005
TABLE_LEN	Variable	INT(32)	Direct	L-004

INNERLIST Listing If the INNERLIST and LIST directives are in effect, the compiler lists the instruction mnemonics generated for each statement after that statement. If optimization is performed, the compiler first lists the original code and then reports “Optimizer replacing the last *n* instructions” and lists the optimized code.

Figure 15-5 shows a sample INNERLIST listing that corresponds to the previous hash procedure.

Figure 15-5. INNERLIST Listing

```

?INNERLIST
1      000000  0  0 INT PROC compute_hash (name, table_len);
2      000000  1  0      INT .name;
3      000000  1  0      INT(32) table_len;
4      000000  1  0      BEGIN
5      000000  1  1      INT int_table_len := $INT (table_len);
6      000000  1  1      INT hash_val := 0;
7      000000  1  1      USE name_index;
000000  1  LDD      L-004
000001  0  STAR      0
000002  1  LDI      +000
000003  7  PUSH     711
8      000004  1  1      USE name_limit;
9      000004  1  1
10     000004  1  1      name_limit := name.<8:14>;
000004  0  LOAD L-005,I
000005  0  LRS      01
000006  0  ANRI     +177
000007  7  STAR      6
11     000010  1  1      FOR name_index := 0 TO name_limit DO
000010  0  LDI      -001
000011  0  STAR      7
Optimizer replacing the last 2 instructions with next 1
000010  7  LDXI    -001,7
000011  0  LDRA     6
000012  0  BUN      +000
12     000013  1  1      hash_val := ((hash_val '<<' 3) LOR
13     hash_val.<0:2>) XOR name[name_index];
000013  1  LOAD     L+002
000014  1  LLS      03
000015  2  LOAD     L+002
000016  2  LRS      15
000017  1  LOR
000020  2  LOAD L-005,I,7
000021  1  XOR
000022  0  STOR     L+002
000023  7  BOX      -011,7
14     000024  1  1      DROP name_index;
15     000024  1  1      DROP name_limit;
16     000024  1  1      RETURN $UDBL ($INT (hash_val '*' 23971)) '\\'
17     int_table_len;
000024  0  LDI      +000
000025  1  LOAD     L+002
000026  2  LDLI     +135
000027  2  ORRI     243
000030  2  LMPY
000031  1  STAR      1
000032  2  LOAD     L+001
000033  1  LDIV
000034  0  STRP      0
000035  0  EXIT     006
18     000036  1  1      END; !compute_hash

```

CODE Listing If the CODE and LIST directives (the defaults) are in effect, the compiler produces an octal code listing following the local map if one exists.

Figure 15-6 shows a sample CODE listing that corresponds to the previous hash procedure. The octal address in the leftmost column is the offset from the procedure base. (If ABSLIST is in effect, the compiler attempts to list addresses relative to the code segment.) Each octal address is followed by eight words of instructions to the end of the procedure.

Figure 15-6. CODE Listing

Address	Octal	Instruction	Words					
000000	060704	000110	100000	024711	140705	030101	006177	000116
000010	103777	000136	010410	040402	030003	040402	030115	000011
000020	143705	000012	044402	013767	100000	040402	005135	004243
000030	000202	000111	040401	000203	000100	125006		

ICODE Listing If the ICODE and LIST directives are in effect, the compiler produces an instruction code mnemonic listing. Figure 15-7 shows a sample ICODE listing that is equivalent to the CODE sample.

Figure 15-7. ICODE Listing

Line	Address	Instruction	Mnemonics									
9.	000000	1 LDD	L-004	0 STAR	0	1 LDI	+000	7 PUSH	711			
10.	000004	0 LOAD	L-005,I	0 LRS	01	0 ANRI	+177	7 STAR	6			
11.	000010	7 LDXI	-001,7	0 LDRA	6	0 BUN	+010					
13.	000013	1 LOAD	L+002	1 LLS	03	2 LOAD	L+002	2 LRS	15	1 LOR	2 LOAD	
											L-005,I,7	
	000021	1 XOR		0 STOR	L+002	7 BOX	-011,7					
15.	000024	0 LDI	+000	1 LOAD	L+002	2 LDLI	+135	2 ORRI	243	1 LMPY	2 STAR	1
	000032	2 LOAD	L+001	1 LDIV		0 STRP	0	0 EXIT	006			

Global Map If the GMAP, MAP, and LIST directives are in effect, the global map lists all identifiers in the compilation unit. If the NOMAP directive appears at the end of the source file, the compiler suppresses the global map but not the local maps. Figure 15-8 shows sample entries of a global map.

Figure 15-8. Global Map

Identifier	Class	Type	Class-Specific Information
PROCESS_STOP_	PROC		EXTERNAL
ABENDPARAM	DEFINE		OPTIONS.<10:10>
AB_OPENERR	DEFINE		%B00000000001D
ACCESS_JNK	DEFINE		ASSIGN.OPTION1.<05:05>
ACCESS_INFO	VARIABLE	TEMPLATE, 402	
1 INCL_LEN		0, 2 INT	
1 AC		2, 2 INT	
AC_INFO_DEF	DEFINE		BEGIN INT INCL_LEN; INT AC[0:
ADD_	LITERAL	INT	%000021
ALL_FCB	DEFINE		INT.\$1[0:Fsize-1]:= [Fsize, %000
AP_BLOCK	BLOCK		
AP_FILE_OK	PROC	INT	EXTERNAL
BLIST_CTL	VARIABLE, 4	STRUCT	INDIRECT BLST_P=001
COD_PTR	VARIABLE	INT(32)	DIRECT AP_BLOCK+002
COMPRS	VARIABLE	INT	DIRECT AP_BLOCK+011
DIMEN_INFO	VARIABLE	TEMPLATE, 16	
1 NUM		0, 2 INT	
1 DOUCE		2, 2 INT	
1 DIM_T		4, 12 SUBSTRUCT	
2 LOW_C		4, 1 STRING	
2 UP_C		5, 1 STRING	
2 LOW_B		6, 4 INT(32)	
2 UP_B		12, 4 INT(32)	
FILE_GETINFO_	PROC		EXTERNAL
FNAMECOLLAPSE	PROC		EXTERNAL

In the preceding example, the C-series equivalent for the D-series PROCESS_STOP_ procedure is ABEND; for FILE_GETINFO_, it is FILEINFO.

File Name Map

When the FMAP directive is in effect, the compiler prints the file map, starting with the first file it encounters and reporting each file introduced by SOURCE directives and TACL ASSIGN and DEFINE commands. The file map shows the complete name of each file and the date and time when the file was last modified. Figure 15-9 shows the file map format for a multisource file listing.

Figure 15-9. File Name Map

FILE MAP BY ORDINAL			
File No.	Date	Time	Source File
[1]	1992-12-31	15:30:14	\$VOL.PROG1.SOURCE1S
[2]	1993-02-27	12:42:19	\$VOL.PROG1.SOURCE2S
[3]	1993-02-29	2:32:34	\$SHR.MSGXX.MSGSHRS

Cross-Reference Listings To collect cross-reference information, specify the CROSSREF directive without parameters. If LIST and NOSUPPRESS (the defaults) are in effect at the end of the source file, the cross-reference listings follow the global map. These listings are:

- Source-file cross-reference listing (the first page)
- Identifier cross-reference listing (subsequent pages)

Source-File Cross-References Figure 15-10 shows the format of a source-file cross-reference listing. It gives the following information for each source file in the compilation:

- File sequence number in the compilation
- File name from a TAL RUN command or a SOURCE directive
- Name of the source file that contained the SOURCE directive if any
- Edit-file line number of the SOURCE directive if any

Figure 15-10. Source-File Cross-Reference Listing

```
CROSSREF CROSS-REFERENCE PROGRAM-T9622D20 (01JUN93)          SYSTEM \X
Copyright Tandem Computers Incorporated 1982-1986, 1989-1993
```

File No.	Filename		
[1]	\$VOL.PROG1.SOURCE1S		
[2]	\$VOL.PROG1.SOURCE2S	SOURCE1S[1]	0.1
[3]	\$SYSTEM.SYSTEM.GPLDEFS	SOURCE2S[2]	2
[4]	\$VOL.PROG1.SOURCE4S	SOURCE1S[1]	7
[5]	\$SYSTEM.SYSTEM.EXTDECS	SOURCE1S[1]	8

Identifier Cross-References The identifier cross-reference listing gives the following information about each specified identifier class:

- Identifier qualifiers—structure, subprocedure, and procedure identifiers
- Compiler attributes—identifier class and type
- Host source file
- Reference lines—type of references (read, write, declaration, or other)

Identifier Qualifiers

An item declared within a structure, subprocedure, or procedure can have from zero to three levels of qualifiers (listed immediately following the identifier name). Here is an example that shows the ordering of qualifier levels:

```
OF mystruct    OF mysubproc    OF myproc
```

The qualifier field varies according to the following rules:

- If an identifier has no qualifier, it is a global item.

GLOBAL_X

- If an identifier has one qualifier, it is declared in a global structure or in a procedure.

ITEM_A OF GLOB_STRUCT_OR_PROC

- If an identifier has two qualifiers, it is declared in either a structure or subprocedure within a procedure.

ITEM_B OF LOC_STRUCT_OR_SUBPROC OF PROC_P

- If an identifier has three qualifiers, it is declared in a structure within a subprocedure within a procedure.

ITEM_C OF SUBLOC_STRUCT OF SUBPROC_Q OF PROC_P

Compiler Attributes

Compiler attributes are class (as specified in the CROSSREF directive) and type modifiers as listed in Table 15-2.

Table 15-2. Compiler Attributes

Class	Modifiers
BLOCK	None
DEFINE	None
ENTRY	Type
LABEL	None
LITERAL	Type
PROC	Type, EXTERNAL
SUBPROC	Type
TEMPLATE	None
VARIABLE	Type, DIRECT or INDIRECT
UNDEFINED	None

Types that apply to the ENTRY, PROC, SUBPROC, and LITERAL classes are STRING, INT, INT(32), REAL, REAL(64), and FIXED. Type FIXED includes the scale if it is nonzero.

Types that apply to the VARIABLE class are those listed in Table 15-2 plus STRUCT, SUBSTRUCT, STRUCT-I, STRUCT-S and UNSIGNED.

Host Source File

The abbreviated edit-file name of the host source file appears on the same line as the identifier name. The sequence number assigned to the source file appears in brackets. The line number where the declaration starts accompanies the file name. An example is:

```
SOURCE1S[23] 137
```

Reference Lines

Reference lines include an entry for each reference in the compilation. For each reference line except read references, an alphabetic code indicates the type of reference. Codes are D (definition), I (invocation), P (parameter), W (write), and M (other). Refer to the *Crossref Manual* for additional information.

Identifier Cross-Reference Example

The identifier cross-reference pages begin with the format shown in Figure 15-11. The header line (only on the first page of references) lists the total number of symbols referenced and the total number of references.

Figure 15-11. Identifier Cross-Reference Listing

```
152 TOTAL SYMBOLS COLLECTED WITH 61 TOTAL REFERENCES COLLECTED
```

ALLOCATE_CBS	DEFINE	GPLDEFS[3]	15	GPLDEFS[3]	198	
ALLOCATE_FCB	DEFINE	GPLDEFS[3]	27	SOURCE2S[2]	5	
ASSIGN_BLOCKLENGTH	INT LITERAL	GPLDEFS[3]	81	GPLDEFS[3]	81.1	135
DEFAULT_VOL	INT DIRECT VARIABLE	SOURCE4S[4]	2	SOURCE1S[1]	14	W
MESSAGE OF STARTUP	INT INDIRECT VARIABLE	SOURCE1S[1]	12	SOURCE1S[1]	11	D 14
MSG_CLOSE	EXTERNAL PROC	SOURCE4S[4]	10	SOURCE1S[1]	28	I
RUCB	INT INDIRECT VARIABLE	SOURCE2S[2]	5	SOURCE1S[1]	18	P

LMAP Listings

Depending on the LMAP directive option in effect, BINSERV produces one of the following maps:

Directive	Kind of Load Map
LMAP	Same as LMAP ALPHA, the default
LMAP ALPHA	Procedures and data blocks, ordered by name (the default)
LMAP LOC	Procedures and data blocks, ordered by starting address
LMAP XREF	Procedure and data-block cross-references for the object file
LMAP *	Procedures and data blocks, ordered by name and by starting address, plus cross-references for the object file

Entry-Point Load Map The entry-point load map gives information about each procedure entry point. Table 15-3 describes the information in each column of the map.

Table 15-3. Entry-Point Load Map Information

Column	Meaning
SP	Code segment number specifier for the entry point
PEP	Sequence number of the entry point in the Procedure Entry Point (PEP) table
BASE	Base address of the procedure defining the entry point
LIMIT	End address of the procedure defining the entry point
ENTRY	Address of executable code for the entry point
ATTRS	Attributes of the entry point: C (CALLABLE), E (ENTRY), I (INTERRUPT), M (MAIN), P (PRIVILEGED), R (RESIDENT), V (VARIABLE), X (EXTENSIBLE)
NAME	Entry-point name
DATE	Date of compilation
TIME	Timestamp of the compilation
LANGUAGE	Source language of the procedure
SOURCE FILE	File name of the source code for the procedure

Figure 15-12 shows the format of a sample entry-point load map by name.

Figure 15-12. Entry-Point Load Map by Name

```
ENTRY POINT MAP BY NAME
```

SP	PEP	Base	Limit	Entry	Attrs	Name	Date	Time	Language	Source File
00	031	010345	043630	0010420		MY_PROC	11FEB93	18:13	TAL	\$JNK.PRG1.SRCE1S
00	073	032224	032636	032224	V	ANY_PROC	11FEB93	10:29	TAL	\$JNK.PRG1.SRCE2S
00	020	000736	001072	000736	M	MAIN_PROC	11FEB93	13:38	TAL	\$JNK.PRG1.MAIN
00	367	131432	131441	131432	E	SORT_PROC	11FEB93	18:14	TAL	\$JNK.PRG1.SORTS

Data-Block Load Maps BINSERV produces a data-block map and a read-only data-block map for primary and secondary global blocks. The data-block map lists the following kinds of data blocks:

- Named blocks, listed by BLOCK declaration name
- Private blocks, listed by NAME declaration name
- #GLOBAL, .#GLOBAL, and \$#GLOBAL implicit global data blocks
- &template-name implicit global data blocks

The read-only data-block map lists global read-only arrays, listed by name.

Table 15-4 describes the information that the data-block load map and the read-only data-block load map give for each data block.

Table 15-4. Data-Block Load Map Information

Column	Meaning
BASE	Base address of the block
LIMIT	End address of the block (blank if block is empty)
TYPE	Binder data-block type; for TAL code, only the common blocks and the two special blocks, \$EXTENDED#STACK and EXTENDED#STACK#POINTERS, can occur
MODE	Word or byte addressing
NAME	Data-block name
DATE	Date of compilation in the form ddmmyy
TIME	Timestamp for the compilation in the form hh:mm
LANGUAGE	Source language of the block
SOURCE FILE	Edit-file name of the source file containing the block declaration

Figure 15-13 shows the format of a data-block load map by location.

Figure 15-13. Data-Block Load Map by Location

```
DATA BLOCK MAP BY LOCATION
```

Base	Limit	Type	Mode	Name	Date	Time	Language	Source File
000000	000014	COMMON	WORD	GLOBAL_	11FEB93	13:38	TAL	\$VOL.PRG.GLBS
000015	000015	COMMON	WORD	LIB_PUB				

Figure 15-14 shows the read-only data-block map. The leftmost column in this map gives the code segment number specifier for each read-only array.

Figure 15-14. Read-Only Data-Block Load Map by Location

```
READ-ONLY DATA BLOCK MAP BY LOCATION
CODE SPACE 00
```

SP	Base	Limit	Type	Mode	Name	Date	Time	Language	Source File
00	000025	000417	COMMON	WORD	HASH	11FEB93	10:48	TAL	\$VOL.PRG.SRC1S
00	000055	000442	COMMON	WORD	TAB	11FEB93	10:48	TAL	\$VOL.PRG.SRC1S

Compilation Statistics The compiler prints compilation statistics at the end of each compilation. If the SYNTAX directive is in effect or if source errors occur, the compiler does not print any other statistics. Figure 15-15 shows the statistics emitted when source errors stop the compilation.

Figure 15-15. Compiler Statistics

```
PAGE 3      $TRMNL [0]                                BINDER AND COMPILER STATISTICS

TAL - Transaction Application Language-T9250D20 - (01JUN93)
Number of compiler errors = 5
Last compiler error on page # 2 IN PROC C
Number of unsuppressed compiler warnings = 1
Number of warnings suppressed by NOWARN = 0
Last compiler warning on page # 1
Maximum symbol table space used was =          562 bytes
Number of source lines= 22
Compile cpu time = 00:00:45
Total Elapsed time = 00:02:58
```

Object-File Statistics If an object file results from the compilation, the compiler prints the following BINSERV statistics preceding the compiler statistics:

- Name of the constructed object file
- Timestamp of the constructed object file
- Number of words of primary data area
- Number of words of secondary data area
- Number of code pages
- Minimum number of data pages required for data space allocation
- Number of resident pages required for total code space allocation
- Number of extended data pages allocated
- Top of stack location
- Number of code segments
- Number of binder warnings
- Number of binder errors

Figure 15-16 shows sample BINSERV statistics.

Figure 15-16. Object-File Statistics

```
PAGE 91  \SYS.$VOL.SUBV.SRC [1]          BINDER AND COMPILER STATISTICS
BINDER - OBJECT FILE BINDER - T9621D20 - (01JUN93)          SYSTEM \X
Copyright Tandem Computers Incorporated 1982-1993

Object file name is $XVOL.XSUBVOL.OFILE
TIMESTAMP 1993-2-11 16:48:21

      45  Code pages
      64  Data page
      0  Resident code pages
      0  Resident data pages

     144  Top of stack location in words

      1  Binder Warnings
      0  Binder Errors

TAL - Transaction Application Language - T9250D20 - (01JUN93)
Number of compiler errors = 0
Number of unsuppressed compiler warnings = 0
Number of warnings suppressed by NOWARN = 0
Maximum symbol table space used was = 128338 bytes
Number of source lines = 6467
Compile cpu time = 00:01:32
Total Elapsed time - 00:07:47
```

Because the compilation unit includes SEARCH directives that cause previously compiled object code to be bound with the source code, the number of source lines is small compared to the generated code.

If a compilation ends due to a BINSERV error, the compiler prints statistics including the BINSERV banner, the message "No object file created," and the number of BINSERV errors and warnings.

16 Running and Debugging Programs

Running Programs After the compiler produces an executable object file for your program, you can run the program by using the TACL RUN command. This command is summarized here and described fully in the *TACL Reference Manual*.

In the RUN command, specify the name of an executable object file and run options if any. The following example runs an object file named MYPROG with no run options:

```
RUN myprog
```

Specifying Run Options When you run a program, you can include any of the TACL RUN command options. Specify the run options in a comma-separated list, enclosed in slashes (/):

```
RUN myprog /IN myfile, LIB mylib/
```

Following are some commonly used options.

IN File Option

The IN file can, for example, be a terminal, a disk file, or a process. Your program is given the IN file name and can use the file as it wishes. If you omit an IN file, your program uses the default input file (normally the home terminal). For example, you can specify MYFILE as the IN file in the RUN command:

```
RUN myprog /IN myfile/
```

OUT File Option

The OUT file can, for example, be a terminal, a disk file, a printer, a spooler location, or a process. Your program is given the OUT file name and can use the file as it wishes. If you omit an OUT file, the output goes to the default output device (normally the home terminal). For example, you can specify a spooler location as the OUT file in the RUN command:

```
RUN myprog /OUT $s.#host/
```

LIB File Option

You can specify the name of a user library file to satisfy external references in the program. A library file is an object file that contains user-written procedures. If you specify a user library file in the RUN command, the system searches that library file before searching the system library file. For example, you can specify MYLIB as the LIB file in the RUN command:

```
RUN myprog /LIB mylib/
```

Once you specify a user library file, the program uses it for all subsequent runs of the program until you specify another library or LIB with no file name. The latter means that no user library file is used.

MEM Pages Option

You can use the MEM (memory) option in the RUN command to increase the number of memory pages for your program's data. For the MEM value, specify an integer in the range 1 through 64. For example, you can specify 40 memory pages in the RUN command:

```
RUN myprog /MEM 40/
```

If you omit the MEM option or if the MEM value is less than the compile-time or bind-time memory-pages value (described next), the system uses the larger value.

Compile-Time Memory-Pages Value. In your source file or in the compilation command, you can use the DATAPAGES directive to increase the number of memory pages. For example, you can specify 33 memory pages in your source file:

```
?DATAPAGES 33
```

In your source file, you can also increase the number of memory pages by calling the C-series NEWPROCESS procedure or the D-series CREATE_PROCESS_ procedure and passing a memory-pages parameter.

If you do not specify the number of memory pages or if you specify an insufficient value, BINSERV allocates sufficient pages for global data and two times the space needed for local data.

Bind-Time Memory-Pages Value. In Binder, you can use SET command options to set the memory-pages value as described in the *Binder Manual*.

NOWAIT Option

If you use the NOWAIT option, the program runs in NOWAIT mode, and the TACL product does not pause while your program runs. Instead, TACL displays a command input prompt after sending the startup message to the new process. You can specify the NOWAIT option in the RUN command as follows:

```
RUN myprog /LIB mylib, NOWAIT/
```

If you omit the NOWAIT option, the program runs in WAIT mode, and the TACL product pauses while the program runs.

Passing Run-Time Parameters You can pass parameters to a program at run time in the RUN command. The syntax and meaning of the parameters are dictated by the program. For example, you can run the program AVERAGE and pass five parameters to the program as follows:

```
RUN average 8 99 571 28 5
```

You can group several words into a single parameter by enclosing them in quotation marks; for example:

```
RUN mystery "The butler did it."
```

You can include a quotation mark as part of a parameter by using two quotation marks; for example:

```
RUN books ""Raw Deal"" by I. M. Poor"
```

Stopping Programs You can let a program execute until completion or until a run-time error stops the program. You can also stop a program before it completes execution in any of the following ways:

- If the program runs in NOWAIT mode, enter the STOP command at the TACL prompt.
- If the program runs in WAIT mode, press the BREAK key and enter the STOP command at the TACL prompt.
- In a C-series source file, call the STOP system procedure. (STOP ends a process normally and ABEND ends a process abnormally.) Declarations for system procedures are located in the EXTDECS file.
- In a D-series source file written for a language-specific run-time environment outside the CRE, call the PROCESS_STOP_ system procedure.
PROCESS_STOP_ replaces STOP and ABEND; it ends a process normally or abnormally depending on the parameter you specify, as described in the *Guardian Procedure Calls Reference Manual*.
- In a D-series source file written for the CRE, call the CRE_TERMINATOR_ routine, described in the *CRE Programmer's Guide*.

When a program stops, it can return a status message, a completion code value, and additional information to the process that started it (usually a TACL product).

Run-Time Errors Some programming errors or program-usage errors are detected at run time rather than at compile time (for example, arithmetic overflow TRAP#2). The *Guardian Procedure Errors and Messages Manual* lists system run-time diagnostic messages. The *Guardian Programmer's Guide* provides information on error processing and error recovery.

-
- Debugging Programs** You can use the Inspect or Debug product to debug your program. Debug is the default debugger on the system; however, it displays values only by machine address and only in octal or ASCII base. For symbolic debugging, you can use the Inspect product.
- Using the Inspect Product** In high-level Inspect mode, you can display values by variable name or statement number. In low-level Inspect mode, you can display values by machine address. By default, values display in decimal base.
- You can step through your program a statement at a time or you can set breakpoints at points in your program at which you want to suspend execution. Each time your program pauses, you can display values to determine what is happening during execution.
- Requesting the Inspect Product** To request the Inspect product, use the `INSPECT` directive in the compilation. To request the high-level Inspect mode, use the `SYMBOLS` directive in the compilation. `SYMBOLS` saves your program symbols in the object file for use in Inspect sessions.
- You can specify the `INSPECT` and `SYMBOLS` directives in the compilation command or in your source file. The following example shows a directive line in a source file.
- ```
?INSPECT, SYMBOLS
```
- Compiling the Source File** When your source file is completed, you can compile the source file by issuing a compilation command at the TACL prompt:
- ```
TAL /IN mysrc/ myprog
```
- Starting the Inspect Session** You can start the Inspect session and the object file by issuing the `RUND` (debugger) command at the TACL prompt:
- ```
RUND myprog
```
- Your program drops into high-level Inspect mode and suspends program execution before the first instruction in your program executes. While the program is suspended, you can use Inspect commands to request breakpoints, step through the program, display program results, and so on.

**Setting Breakpoints** A breakpoint is a location within your program at which to suspend execution so you can use Inspect commands to check results at that point. Usually, you request at least one breakpoint at the Inspect prompt before the first instruction in your program executes. For example, you can set an unconditional breakpoint at a statement or at an edit line number.

If you set an unconditional breakpoint at a statement, the program suspends execution before the first machine instruction generated for that statement executes. The following example sets a breakpoint at the eighth statement from the beginning of MYPROC:

```
BREAK #myproc + 8 STATEMENTS
```

If you set an unconditional breakpoint at an edit line number, the program suspends execution immediately before that line executes. The following example sets a breakpoint at edit line 21:

```
BREAK #21
```

You can also set conditional and other breakpoints as described in the *Inspect Manual*.

**Stepping Through a Program** At the Inspect prompt, before the first statement executes or when the program pauses at a breakpoint, you can step through the program and execute a single statement at a time.

1. To execute the first statement, enter:

```
STEP
```

2. To repeat the STEP command, press the Return key.

3. To set a temporary breakpoint two statements hence and resume execution, enter:

```
STEP 2 STATEMENTS
```

**Displaying Values** When the program suspends execution at a breakpoint, you can use the DISPLAY command to display the values of variables. You can shorten the command to its first letter. For example, to display the values of variables LENGTH, WIDTH, and DEPTH, use either of the following commands:

```
DISPLAY length, width, depth
```

```
D length, width, depth
```

**Stopping the Inspect Session** To stop the Inspect session and your program, use the STOP command at the Inspect prompt. In high-level mode, enter STOP exactly as shown here:

```
STOP
```

**Sample Inspect Session** This discussion presents a source file and shows the steps for running the object file in an Inspect session.

#### Sample Source File

The sample source file is named MYSRC. Figure 16-1 shows the source code in the sample source file.

---

**Figure 16-1. Sample Source File**

```
!This is a source file named MYSRC.

?INSPECT !Request symbolic debugger
?SYMBOLS !Save symbols in object file
 ! for symbolic debugger
?NOLIST, SOURCE $system.system.extdecs (initializer)
 !Include system procedure
 !without its listing

?LIST

PROC myproc MAIN; !Declare procedure MYPROC
 BEGIN
 INT var1; !Declare variables
 INT var2;
 INT total;

 CALL initializer; !Read the start-up message
 var1 := 5; !Assign value to VAR1
 var2 := 10; !Assign value to VAR2
 total := var1 + var2; !Assign sum to TOTAL
 END; !End MYPROC
```

---

### Compiling the Sample Source File

To compile the sample source file into an object file, specify the source file name (MYSRC) and an object file name (MYPROG):

```
TAL /IN mysrc/ myprog
```

Figure 16-2 shows the beginning of the compiler listing for the sample object file.

**Figure 16-2. Sample Compiler Listing**

```
PAGE 1 [1] $MYVOL.MYSUBV.MYSRC

TAL - T9250D20 - (01JUN93)
Copyright Tandem Computers Incorporated 1976, 1978, 1981-1983, 1985, 1987-1993

 1. 000000 0 0 !This is a source file named MYSRC.
 2. 000000 0 0
 3. 000000 0 0 ?INSPECT !Request symbolic debugger
 4. 000000 0 0 ?SYMBOLS !Request symbols in Object file
 5. 000000 0 0 ! for symbolic debugger
 6. 000000 0 0 ?NOLIST, SOURCE $system.system.extdecs (initializer)
 7. 000000 0 0 !Include system procedure
 8. 000000 0 0 ! without its listing
 9. 000000 0 0 ?LIST
 10. 000000 0 0
 11. 000000 0 0 PROC myproc MAIN; !Declare procedure MYPROC
 12. 000000 1 0 BEGIN
 13. 000000 1 1 INT var1; !Declare variables
 14. 000000 1 1 INT var2;
 15. 000000 1 1 INT total;
 16. 000000 1 1
 17. 000000 1 1 CALL initializer; !Read the start-up message
 18. 000006 1 1 var1 := 5; !Assign value to VAR1
 19. 000010 1 1 var2 := 10; !Assign value to VAR2
 20. 000012 1 1 total := var1 + var2; !Assign sum to TOTAL
 21. 000016 1 1 END; !End MYPROC
```

### Running a Sample Inspect Session

The step numbers in the following Inspect session correspond to the interactive operations shown in Figure 16-3. In the figure, commands you enter are shown in boldface.

1. To run your program in an Inspect session, enter the TACL RUND command at the TACL prompt. Specify the name of your object file and any appropriate run options. For the sample session, specify MYPROG:

```
RUND myprog
```

The Inspect product suspends program execution before the first instruction. An Inspect header message and prompt appears. The prompt consists of the object file name enclosed in hyphens if your program has symbols (or underscores if it has no symbols).

2. You can set breakpoints at points where you want the program to pause. For the sample session, set a breakpoint at edit line 21, which is the END of the program:

```
BREAK #21
```

An Inspect message indicates the number, type, and location of the breakpoint. The breakpoint will suspend execution before edit line 21 executes.

3. To run your program until the breakpoint, enter the RESUME command at the Inspect prompt:

```
RESUME
```

When a breakpoint occurs, the program suspends execution. An Inspect message identifies the breakpoint.

4. You can now display object values or clear and set breakpoints. Normally, you can resume execution or step through the program until it reaches another breakpoint.

For the sample session, display the value of variable TOTAL:

```
DISPLAY total
```

An Inspect message shows the value of TOTAL.

5. Normally, after your program executes correctly, you clear all breakpoints (for example, by issuing the CLEAR \* command). For the sample session, clear the breakpoint by specifying its number:

```
CLEAR 1
```

An Inspect message tells you the breakpoint is cleared.

6. You can now stop the Inspect session and return to the TACL prompt. (In high-level mode, do not abbreviate the STOP command.)

```
STOP
```

---

**Figure 16-3. Running a Sample Inspect Session**

```
1. 25> RUND myprog

INSPECT - Symbolic Debugger - T9673D20 - (01JUN93) . . .
Copyright Tandem Computers Incorporated 1983, 1985-1993
INSPECT
*175,08,111 MYPROG #MYPROC.#6(MYSRC)

2. -MYPROG-BREAK #21

Num Type Subtype Location
 1 Code #MYPROC.#21(MYSRC)

3. -MYPROG-RESUME

INSPECT BREAKPOINT 1: #21
175,08,111 MYPROG #MYPROC.#21(MYSRC)

4. -MYPROG-DISPLAY total

TOTAL = 15

5. -MYPROG-CLEAR *

Breakpoint cleared: 1 Code #MYPROC.#21(MYSRC)

6. -MYPROG-STOP

26>
```

---

If the results of the program are incorrect, correct the source file by using a text editor, recompile the source file, and rerun the object file in an Inspect session.

---

# 17 Mixed-Language Programming

---

This section gives an overview of:

- Mixed-language features provided by TAL
- TAL and C guidelines
- CRE guidelines for TAL programs

---

## Mixed-Language Features of TAL

You can use the following TAL features in mixed-language programs:

- NAME and BLOCK declarations
- Procedure declaration LANGUAGE attribute
- Procedure declaration public name
- PROC and PROC(32) parameter types
- Parameter pairs
- ENV directive
- HEAP directive

## NAME and BLOCK Declarations

All global data to be shared with routines written in other languages must be relocatable. After binding, you should not depend on the data being located at a particular location.

In TAL, you can use BLOCK declarations to group global data declarations into named or private data blocks. If a BLOCK declaration is present, a NAME declaration at the beginning of the compilation unit must name the unit. The identifiers of NAME and BLOCK declarations must be unique among all NAME and BLOCK declarations in all the compilation units in the program. Here is an example of a NAME declaration:

```
NAME input_module; !Name the compilation unit
```

A named data block is shareable among all compilation units in a program. You can declare any number of named data block in a compilation unit. Here is an example of a BLOCK declaration for a named data block (GLOBALS):

```
BLOCK globals; !Declare named global data block
 INT .an_array[0:7];
 INT .another_array[0:34];
 INT(32) total;
 LITERAL msg_buf = 79;
 DEFINE xaddr = INT(32)#;
END BLOCK;
```

A private data block is shareable only among routines within the same compilation unit. To declare a private data block, specify the PRIVATE keyword in place of the data block identifier. You can declare only one private data block in a compilation unit. The private data block inherits the identifier you specify in the NAME declaration for the compilation unit. Here is an example of a BLOCK declaration for a private data block:

```
BLOCK PRIVATE;
 INT average;
 INT total;
END BLOCK;
```

You can specify the location of a named or private data block. For more information, see Section 14, “Compiling Programs.”

**LANGUAGE Attribute** Before calling an external routine, a TAL module must include an EXTERNAL procedure declaration for the external routine. If you are using a D-series TAL compiler, you can use the LANGUAGE attribute in the declaration to specify that the external routine is a C, COBOL85, FORTRAN, or Pascal routine. For example, if the external routine is a C routine, you can specify LANGUAGE C following the routine identifier in the EXTERNAL procedure declaration:

```
PROC c_func !EXTERNAL procedure declaration
 LANGUAGE C; !LANGUAGE attribute
 EXTERNAL; !EXTERNAL option
```

If the C, COBOL85, FORTRAN, or Pascal routine has formal parameters, the LANGUAGE attribute follows the formal parameter list in the EXTERNAL procedure declaration:

```
PROC c_func (a, b, c) !Formal parameter list
 LANGUAGE C; !LANGUAGE attribute
 STRING .a, .b, .c; !Formal parameter declarations
 EXTERNAL; !EXTERNAL option
```

If you are not sure of the language, you can specify LANGUAGE UNSPECIFIED in the EXTERNAL procedure declaration:

```
PROC some_proc !EXTERNAL procedure declaration
 LANGUAGE UNSPECIFIED; !LANGUAGE attribute
 EXTERNAL; !EXTERNAL option
```

Here are guidelines for specifying a EXTERNAL procedure declaration:

- Always include the EXTERNAL keyword if you use the LANGUAGE attribute.
- Specify no more than one LANGUAGE attribute in a declaration.
- Omit the LANGUAGE attribute if the external routine is written in TAL.

**Public Name** Before calling an external routine, a D-series TAL module can include an EXTERNAL procedure declaration that specifies a public name for use in Binder. In particular, specify an external routine identifier as a public name when the identifier does not conform to TAL rules.

In the EXTERNAL procedure declaration, specify an equal sign (=) and the *public name*, enclosed in quotes, following the routine identifier:

```
PROC cobol_proc = "cobol-program-unit" !Public name
 LANGUAGE COBOL; !LANGUAGE attribute
EXTERNAL;
```

The public name must conform to the identifier rules of the language in which the external routine is written. For all languages except C, the TAL compiler upshifts all public names automatically. In the preceding example, the public name conforms to COBOL rules.

If the external routine has formal parameters, the formal parameter list follows the public name:

```
PROC cobol_proc = "cobol-program-unit" !Public name
 (a, b, c) !Parameter list
 LANGUAGE COBOL; !LANGUAGE attribute
 STRING .a, .b, .c; !Parameter
 ! declarations
EXTERNAL;
```

**PROC Parameter Type** Specify a procedure as a formal PROC parameter in a TAL routine that expects one of the following actual parameters:

- A C small-memory-model routine
- A FORTRAN routine compiled with the NOEXTENDEDREF directive
- A TAL routine

#### TAL Receiving PROC Parameters

The following TAL routine declares a formal PROC parameter:

```
PROC tal_proc (param_proc);
 PROC param_proc; !Formal PROC parameter
 BEGIN
 !Lots of code
 END;
```

The following callers can call a TAL routine that declares a formal PROC parameter:

- C small-memory-model routines
- COBOL85 routines
- FORTRAN routines compiled with the NOEXTENDEDREF directive
- TAL routines

When a caller lists an appropriate routine in the calling sequence, the caller's compiler passes the routine's 16-bit address of PEP and map information.

#### TAL Passing PROC Parameters

A TAL routine can pass actual PROC parameters to any of the following routines:

- C small-memory-model routines
- FORTRAN routines compiled with the NOEXTENDEDREF directive
- TAL routines

If the actual PROC parameter is a C or FORTRAN routine, specify an EXTERNAL procedure declaration such as:

```
PROC c_func (param_proc) LANGUAGE C;
 PROC param_proc; !Formal PROC parameter
 EXTERNAL;
```

Additional guidelines are provided later in this section in:

- "TAL Routines as Parameters to C"
- "C Routines as Parameters to TAL"

**PROC(32) Parameter Type** Specify a procedure as a formal PROC(32) formal parameter in a TAL routine that expects one of the following actual parameters:

- A C large-memory-model routine
- A FORTRAN routine compiled with the EXTENDEDREF directive
- A Pascal routine

#### TAL Receiving PROC(32) Parameters

The following TAL routine declares a formal PROC(32) parameter::

```
PROC tal_proc (param_proc32);
 PROC(32) param_proc32; !Formal PROC(32) parameter
 BEGIN
 !Lots of code
 END;
```

The following callers can call a TAL routine that declares a formal PROC(32) parameter:

- C large-memory-model routines
- COBOL85 routines
- FORTRAN routines compiled with the EXTENDEDREF directive
- Pascal routines
- TAL routines

When a caller lists an appropriate routine in the calling sequence, the caller's compiler passes the routine's 32-bit address to the TAL compiler. The high-order word of the address contains PEP and map information; the low-order word contains a zero. If the TAL compiler receives a 16-bit address instead, it converts the address to a 32-bit address and passes the converted address to the called routine.

#### TAL Passing PROC(32) Parameters

A TAL routine can pass actual PROC(32) parameters to any of the following routines:

- C large-memory-model routines
- FORTRAN routines compiled with the EXTENDEDREF directive
- Pascal routines
- TAL routines

For each actual PROC(32) parameter, specify an EXTERNAL procedure declaration such as:

```
PROC c_func (param_proc32) LANGUAGE C;
 PROC(32) param_proc32; !Formal PROC(32) parameter
 EXTERNAL;
```

Additional guidelines and examples are provided later in this section in:

- "TAL Routines as Parameters to C"
- "C Routines as Parameters to TAL"

**Parameter Pairs** A TAL parameter pair consists of two formal parameters (connected by a colon) that together describe a single data type to FORTRAN or Pascal routines. Some D-series system routines require that callers pass actual parameter pairs (as described in the *Guardian Application Conversion Guide*).

Table 17-1 lists the parameter types in other languages that correspond to the TAL parameter pair.

**Table 17-1. Parameter Pair Type Correspondence**

| Language | Type                                   |
|----------|----------------------------------------|
| C        | Not applicable                         |
| COBOL    | Not applicable                         |
| FORTRAN  | CHARACTER * <i>length</i>              |
| Pascal   | FSTRING(*) or FSTRING( <i>length</i> ) |

### Declaring Parameter Pairs

When you declare a TAL routine, you can include a parameter pair by specifying a *string* parameter and a *length* parameter separated by a colon:

```
PROC in_procedure (astring:length) !Parameter pair
 LANGUAGE PASCAL;
 STRING .EXT astring; !Declare string parameter
 INT length; !Declare length parameter
EXTERNAL;
```

The string and length parameters of a parameter pair have the following characteristics:

| Parameter        | Pass By   | Formal Parameter                             | Actual Parameter                                                               |
|------------------|-----------|----------------------------------------------|--------------------------------------------------------------------------------|
| String parameter | Reference | A standard or extended STRING simple pointer | A STRING array or simple pointer declared inside or outside a structure        |
| Length parameter | Value     | A directly addressed INT simple variable     | An INT expression that specifies the length, in bytes, of the string parameter |

If the called routine does not change the length of the string parameter, the length parameter represents the maximum size, the initial size, or the current size of the string parameter.

### Passing Parameter Pairs

The calling routine can pass parameter pairs to the called routine in a CALL statement. For example, the calling routine can declare and pass a parameter pair to IN\_PROCEDURE (declared in the preceding example) as follows:

```
PROC caller;
 BEGIN
 LITERAL length = 5;
 STRING .EXT array[0:length - 1];
 !Some code here
 CALL in_procedure (array:length);
 END;
```

### Modifying the String Length

If the called routine modifies the length of the string parameter, the called routine must also provide an INT reference parameter in which it returns the new length of the string. This parameter can represent the length, in bytes, of the string parameter before and after the routine modifies the length. In that case, it is an input and output parameter. For example, you can declare the current-length parameter as follows:

```
PROC out_procedure (astring:max_length, current_length);
 STRING .EXT astring; !Output or input/output parameter
 INT max_length; !Input parameter
 INT .current_length; !Output or input/output parameter
 BEGIN
 !Code to process ASTRING
 END;
```

The formal *current-length* parameter is a reference parameter, either a standard or extended INT simple pointer. It can be an input-output or output-only parameter:

Input-output parameter

**Input** The actual parameter is an INT simple variable that specifies the length of the string parameter, in bytes, before the called routine processes the string parameter. When you list the simple variable in the calling sequence, the compiler passes the compiler-assigned address of the simple variable.

**Output** The called routine can process the string parameter and return the new length of the string parameter to the calling routine. The called routine must ensure that the initial length and the new length are in the range 0 through the value of the maximum-length parameter. (The compiler does no range checking.)

Output-only parameter—The compiler ignores any initial parameter value.

In the following example, the caller passes a parameter pair and the current-length parameter to the procedure declared in the preceding example:

```
PROC out_proc_caller;
 BEGIN
 INT cur_len; !Declare simple variable CUR_LEN
 LITERAL max_len = 20;
 STRING .name[0:max_len - 1];

 name ' := ' "KENNETH";
 cur_len := 7;
 CALL out_procedure (name:max_len, cur_len);
 END; !Compiler passes address of CUR_LEN
```

### Omitting Actual Parameter Pairs

If you want to omit an optional parameter-pair unconditionally in the actual parameter list, substitute a comma for the omitted parameter pair. You cannot omit half of a parameter pair.

For example, suppose the called routine declares the optional parameter pair VALUE2:VALUE3 as follows:

```
PROC some_procedure (value1, value2:value3, value4)
 EXTENSIBLE;
 INT value1;
 STRING .value2; !First half of optional parameter pair
 INT value3; !Other half of optional parameter pair
 INT .value4;
```

The caller can omit the optional parameter pair from its CALL statement, like this:

```
INT val_1, val_4;

CALL some_procedure (val_1, , val_4);
 !Comma replaces omitted parameter pair
```

As of the D20 release, you can omit an optional parameter-pair conditionally. Use the \$OPTIONAL standard function as described in Section 13, "Using Procedures."

- ENV Directive** In TAL, you use the ENV directive to specify the run-time environment of a D-series object file as described later in this section. The run-time environment is either:
- The CRE, which provides services for mixed-language programs
  - A C, COBOL, FORTRAN, Pascal, or TAL run-time environment outside the CRE
- HEAP Directive** In TAL, you can set the size of the user heap in the CRE, if ENV COMMON is also in effect for the MAIN routine. The user heap, named #HEAP, is a shared CRE resource that all routines in your program can access directly or indirectly as described later in this section.

---

**TAL and C Guidelines** This subsection provides guidelines for writing programs composed of TAL and Tandem C modules. This discussion assumes that you have a working knowledge of TAL and C and are familiar with the contents of the following manuals:

- TAL Programmer's Guide*
- TAL Reference Manual*
- C Reference Manual*

This subsection discusses:

- TAL and C identifiers
- TAL and C data types
- Memory models
- TAL calling C
- C calling TAL
- Sharing data
- Parameters and variables
- Extended data segments

For information on calling TAL routines from another language, see the manual for COBOL85, FORTRAN, or Pascal.

**Using Identifiers** TAL and C identifiers differ as follows:

- TAL and C have independent sets of reserved keywords.
- TAL identifiers can include circumflexes (^); C identifiers cannot.
- The C compiler is case-sensitive; the TAL compiler is not case-sensitive.

To declare variable identifiers that satisfy both compilers:

- Avoid using reserved keywords in either language as identifiers.
- Specify TAL identifiers without circumflexes.
- Specify C identifiers in uppercase.

You can declare TAL-only or C-only routine identifiers and satisfy both compilers by using the public name option in:

- Interface declarations in C
- EXTERNAL procedure declarations in TAL

In Inspect sessions:

- Use uppercase for TAL identifiers
- Use the given case for C identifiers

In Binder sessions, use mode noupshift for lowercase C identifiers.

**Matching Data Types** Use data types that are compatible between languages for:

- Shared global variables
- Formal or actual parameters
- Function return values

Table 17-2 lists compatible TAL and C data types for each TAL addressing mode.

**Table 17-2. Compatible TAL and C Data Types**

| TAL Addressing Mode      | TAL Data Type | C Data Type        | Notes                     |
|--------------------------|---------------|--------------------|---------------------------|
| Direct                   | STRING        | char               |                           |
| Direct                   | INT           | short              | TAL INT signed range only |
| Direct                   | INT(32)       | long               |                           |
| Direct                   | FIXED(0)      | long long          | TAL FIXED(0) only         |
| Direct                   | REAL          | float              |                           |
| Direct                   | REAL(64)      | double             |                           |
| Standard indirect (.)    | STRING        | char *             |                           |
| Standard indirect (.)    | INT           | short *            |                           |
| Standard indirect (.)    | INT(32)       | long *             |                           |
| Standard indirect (.)    | FIXED(0)      | long long *        |                           |
| Standard indirect (.)    | REAL          | float *            |                           |
| Standard indirect (.)    | REAL(64)      | double *           |                           |
| Extended indirect (.EXT) | STRING        | extptr char *      | For extptr, see Note.     |
| Extended indirect (.EXT) | INT           | extptr short *     |                           |
| Extended indirect (.EXT) | INT(32)       | extptr long *      |                           |
| Extended indirect (.EXT) | FIXED(0)      | extptr long long * |                           |
| Extended indirect (.EXT) | REAL          | extptr float *     |                           |
| Extended indirect (.EXT) | REAL(64)      | extptr double *    |                           |

Note: In C, use extptr only in the parameter-type-list of an interface declaration to specify a parameter type that is defined in TAL as an extended pointer.

Incompatibilities between TAL and C data types include the following:

- TAL has no numeric data type that is compatible with C unsigned long.
- TAL UNSIGNED is not compatible with C unsigned short. TAL UNSIGNED(16) can represent signed or unsigned values.

For more information on C and TAL data types, see “Parameters and Variables” later in this section.

**Memory Models** A C program can use the small-memory model or the large-memory model, depending on the amount of data storage required. The large-memory model is recommended and is the default setting. All examples in this subsection illustrate the large-memory model unless otherwise noted.

A TAL program can use any of the following memory combinations, depending on the application's needs:

- The user data segment
- The user data segment and the automatic extended data segment
- The user data segment and one or more explicit extended data segments
- The user data segment, the automatic extended data segment, and one or more explicit extended data segments

The following table describes some aspects of memory usage by C and TAL programs. The rightmost column refers to the upper 32K-word area of the user data segment.

| Language | Memory Model   | Addressing       | Data Storage                                                                                       | Upper 32K-Word Area              |
|----------|----------------|------------------|----------------------------------------------------------------------------------------------------|----------------------------------|
| C        | Small          | 16-bit           | 32K words                                                                                          | Reserved                         |
| C        | Large          | 32-bit           | 127.5 megabytes                                                                                    | Reserved                         |
| TAL      | Not applicable | 16-bit or 32-bit | 64K words (without the CRE), plus 127.5 megabytes in each extended data segment that is allocated. | Reserved only if you use the CRE |

Any TAL module that uses the upper 32K-word area of the user data segment cannot run within a C object file that contains the MAIN routine.

**Calling C Routines From TAL Modules** A TAL module must include an EXTERNAL procedure declaration for each C routine to be called. The following TAL code shows the EXTERNAL procedure declaration for C\_FUNC and a routine that calls C\_FUNC. ARRAY is declared with .EXT, because C\_FUNC uses the large-memory model:

| TAL Code                                                                                                                                                                                                                       | C Code                                                                             |
|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|------------------------------------------------------------------------------------|
| <pre> INT status := 0; STRING .EXT array[0:4];  INT PROC c_func (a)   LANGUAGE C;   STRING .EXT a; EXTERNAL;  PROC example MAIN;   BEGIN     array[2] := "B";     status := c_func (array);     array[1] := "B";   END; </pre> | <pre> short C_FUNC(char *str) {   *str = 'A';   str[2] = 'C';   return 1; } </pre> |

A C-series C module called by a TAL module has limited access to the C run-time library. If the C module needs full access to the C run-time library, you can either:

- Modify the program to run in the CRE as described later in this section.
- Specify a C MAIN routine that calls the original TAL MAIN routine as follows.

In the TAL module, remove the MAIN keyword from the TAL MAIN routine and remove any calls to the INITIALIZER or ARMTRAP system procedure. The TAL module must also meet the requirements of the C run-time environment.

| TAL Code                                                                                                                                                                                         | C Code                                                                                                                                                                                                                                          |
|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <pre> INT status := 0; INT .EXT array[0:4];  INT PROC cfunc (a)   LANGUAGE C;   INT .EXT a; EXTERNAL;  PROC talmain;   BEGIN     array[2] := 2;     status :=       cfunc (array);   END; </pre> | <pre> #include &lt;stdio.h&gt; nolist  tal void TALMAIN ( void );  short CFUNC (short *num) {   printf("num B4=%d\n", *num);   num[0] = 10;   printf("num AF=%d\n", *num);   return 1; }  main () /* C MAIN routine */ {   TALMAIN (); } </pre> |

### Calling TAL Routines From C Modules

A D-series C module has full access to the C run-time library even if the C module does not contain the MAIN routine. A C-series C module that does not contain the MAIN routine cannot fully access the C run-time library.

When you code C modules that call TAL routines:

- Include an interface declaration for each TAL routine to be called.
- If a called TAL routine sets the condition code, include the talh header file.
- If a called routine is a system procedure, include the cextdec header file.

In C, interface declarations are comparable to EXTERNAL procedure declarations in TAL. To specify an interface declaration in C, include:

- The keyword `_tal` (D-series code) or `tal` (C-series code)
- The variable or extensible *attribute*, if any, of the TAL routine
- The *data type* of the return value, if any, of the TAL routine
- A routine *identifier*
- A *public name* if the TAL identifier is not a valid C identifier
- A *parameter-type-list* or, if no parameters, the keyword `void`
- For extended pointers in the parameter-type-list, the keyword `extptr` before the parameter type

The return type value can be any of the following:

| Return Type Value       | Meaning                                                                                                                   |
|-------------------------|---------------------------------------------------------------------------------------------------------------------------|
| <code>void</code>       | The TAL routine does not return a value.                                                                                  |
| <i>fundamental-type</i> | The TAL routine returns a value. Specify one of, or a pointer to one of, the character, integer, or floating-point types. |
| <code>cc_status</code>  | The TAL routine sets the condition-code register to CCL, CCE, or CCG (defined in talh).                                   |

For information on calling TAL routines that both return a value and set a condition code (CC), see the *C Reference Manual*.

Here are examples of interface declarations for calling TAL routines. For D-series code, prefix the `tal` keyword with an underscore):

```
_tal variable short SEGMENT_ALLOCATE_ (short, long,
 short *, short);

_tal variable cc_status SEGMENT_DEALLOCATE_ (short, short);

_tal variable cc_status READ (short, short *, short,
 short *, long);

_tal extensible cc_status READX (short, extptr char *, short,
 short *, long);

_tal void c_name = "tal^name" (short *);
```

After specifying an interface declaration, use the normal C routine call to access the TAL routine.

This example shows a large-memory-model C module that calls a TAL routine:

### C Code

```
#include <stdio.h> nolist

short arr[5]; /*stored in extended segment */
_tal void C_Name = "tal^name" (extptr short *);

void func1 (short *xarr)
{
 C_Name (xarr);
 printf ("xarr[2] after TAL = %d", xarr[2]);
}

main ()
{
 arr[4] = 8;
 func1 (arr);
}
```

### TAL Code

```
PROC tal^name (a);
 INT .EXT a; !32-bit pointer
 BEGIN
 a[2] := 10;
 END;
```

**Sharing Data** You can share global data in the user data segment between the following kinds of TAL and C modules:

- TAL modules that declare global variables having standard indirection (.)
- C small-memory-model modules

You can share global data in the automatic extended data segment between the following kinds of TAL and C modules:

- TAL modules that declare global variables having extended indirection (.EXT)
- C large-memory-model modules

In a large-memory-model C module, you can use the `lowmem` declaration to allocate a C array or structure that can be represented by a 16-bit address if needed in a call to a TAL routine or a system procedure.

Using pointers to share data is easier and safer than trying to match declarations in both languages. Using pointers also eliminates problems associated with where the data is placed.

To share data by using pointers, first decide whether the TAL module or the C module declares the data:

- If the TAL module is to declare the data, follow the guidelines in “Sharing TAL Data With C Using Pointers.”
- If the C module is to declare the data, follow the guidelines in “Sharing C Data With TAL Using Pointers.”

#### Sharing TAL Data With C Using Pointers

To share TAL global data with C modules, follow these steps:

1. In the TAL module, declare the data using C-compatible identifiers, data types, and alignments. (Alignments depend on byte or word addressing and variable layouts as described in “Parameters and Variables” later in this section.)  
When you declare TAL arrays and structures, use indirect addressing.
2. In the C module, declare pointers to the data, using TAL-compatible data types.
3. In the C module, declare a routine to which TAL can pass the addresses of the shared data.
4. In the C routine, initialize the pointers with the addresses sent by the TAL module.
5. Use the pointers in the C module to access the TAL data.

The following example shows how to share TAL data with a large-memory-model C module. The TAL module passes to a C routine the addresses of two TAL arrays. The C routine assigns the array addresses to C pointers.

**C Code**

```

short *c_int_ptr; /* pointer to TAL data */
char *c_char_ptr; /* pointer to TAL data */

short INIT_C_PTRS (short *tal_intptr, char *tal_strptr)
{
 /* called from TAL */
 c_int_ptr = tal_intptr;
 c_char_ptr = tal_strptr;
 return 1;
}

/* Access the TAL arrays by using the pointers */

```

**TAL Code**

```

STRUCT rec (*);
 BEGIN
 INT x;
 STRING tal_str_array[0:9];
 END;

INT .EXT tal_int_array[0:4]; !TAL data to share with C
STRUCT .EXT tal_struct (rec); !TAL data to share with C

INT status := -1;

INT PROC init_c_ptrs (tal_intptr, tal_strptr) LANGUAGE C;
 INT .EXT tal_intptr;
 STRING .EXT tal_strptr;
 EXTERNAL;

PROC tal_main MAIN;
 BEGIN
 status := init_c_ptrs
 (tal_int_array, tal_struct.tal_str_array);
 !Do lots of work
 END;

```

**Sharing C Data With TAL Using Pointers**

To share C global data with TAL modules, follow these steps:

1. In the C module, declare the data using TAL-compatible identifiers, data types, and alignments. (Alignments depend on byte or word addressing and variable layouts as described in “Parameters and Variables” later in this section.)

C arrays and structures are automatically indirect.

2. In the TAL module, declare pointers to the data, using C-compatible data types.
3. In the TAL module, declare a routine to which C can pass the addresses of the shared data.
4. In the TAL routine, initialize the pointers with the addresses sent by C.
5. Use the pointers in the TAL module to access the C data.

This example shows how to share C data with a TAL module. The C module passes the addresses of two C arrays to a TAL routine. The TAL routine assigns the array addresses to TAL pointers.

#### C Code

```
#include <stdio.h> nolist

short arr[5]; /* C data to share with TAL */
char charr[5]; /* C data to share with TAL */

_tal void INIT_TAL_PTRS (extptr short *, extptr char *)
_tal void C_Name = "tal^name" (void);

void example_func(int *x)
{
 printf("x before TAL = %d\n", x[2]);
 C_Name();
 printf("x after TAL = %d\n", x[2]);
}

main ()
{
 INIT_TAL_PTRS (&arr[0], &charr[0]); /* initialize ptrs */
 /* test pointer values */
 arr[0] = 8;
 example_func(arr);
 arr[2] = 18;
 charr[2] = 'B';
}
```

#### TAL Code

```
INT .EXT tal_int_ptr; !Pointer to C data
STRING .EXT tal_char_ptr; !Pointer to C data

PROC init_tal_ptrs (c_addr1, c_addr2); !Called from C
 INT .EXT c_addr1;
 STRING .EXT c_addr2;
 BEGIN
 @tal_int_ptr := @c_addr1;
 @tal_char_ptr := @c_addr2;
 END;

PROC tal^name;
 BEGIN
 tal_int_ptr[0] := 10;
 tal_int_ptr[2] := 20;
 tal_char_ptr[2] := "A";
 END;
```

### Sharing TAL Data With C Using BLOCK Declarations

As of the D20 release, TAL modules can share global data with C modules by declaring each shared variable in its own BLOCK declaration and giving both the variable and the BLOCK the same name. The C modules must also declare each shared variable; the layout of the variable must match in both the TAL and C modules.

In the following example, a TAL module declares a variable within a BLOCK declaration, and the C module declares the equivalent variable:

| TAL Code         | C Code                      |
|------------------|-----------------------------|
| NAME TAL_module; |                             |
| BLOCK fred;      |                             |
| INT .EXT fred;   | int FRED; /*all uppercase*/ |
| END BLOCK;       |                             |

Because the preceding method requires that the layout of the corresponding TAL and C declarations match, it is recommended that you share data by using pointers where possible.

**Parameters and Variables** This subsection gives guidelines for declaring compatible TAL and C variables and parameters. These guidelines supplement those given in “Sharing Data” earlier in this section. The following topics are discussed:

- STRING and char variables
- Arrays
- Structures
- Substructures
- Multidimensional arrays
- Arrays of structures
- Redefinitions and unions
- Pointers
- Enumeration variables
- Bit-field manipulation
- UNSIGNED variables and compacted bit fields
- TAL routines as parameters
- C routines as parameters

When you declare formal reference parameters, remember to use indirection as follows:

- If the caller is a small-memory-model C routine, use standard indirection (.) for the TAL formal parameter.
- If the caller is a large-memory-model C routine, use extended indirection (.EXT) for the TAL formal parameter.

### STRING and char Variables

TAL STRING and C char simple variables each occupies one byte of a word. Following are STRING and char compatibility guidelines:

- Share variables of type TAL STRING and C char by using pointers.
- Declare TAL STRING and C char formal parameters as reference parameters to avoid the following value parameter incompatibility:
  - When you pass a STRING parameter to a C routine, the actual byte value occupies the left byte of the word allocated for the C char formal parameter.
  - When you pass a char parameter to a TAL routine, the actual byte value occupies the right byte of the word allocated for the TAL STRING formal parameters.

For example, if you declare a TAL STRING formal parameter as a value parameter rather than as a reference parameter, the TAL routine can access the C char actual parameter only by explicitly referring to the right byte of the word allocated for the STRING formal parameter:

```
PROC sample (s);
 STRING s; !Declare TAL STRING parameter as a
 BEGIN ! value (not reference) parameter
 STRING dest;
 dest := s[1]; !Refer to right byte of word
END;
```

### Arrays

TAL and C arrays differ as follows:

| Characteristic          | TAL Array                                                                                          | C Array                                                                                           |
|-------------------------|----------------------------------------------------------------------------------------------------|---------------------------------------------------------------------------------------------------|
| Lower bound             | Any integer                                                                                        | Always zero                                                                                       |
| Dimensions              | One dimension                                                                                      | One or more dimensions                                                                            |
| Direct or indirect      | Direct or indirect                                                                                 | Indirect only                                                                                     |
| Byte or word addressing | STRING arrays and extended indirect arrays are byte addressed; all other arrays are word addressed | char arrays and large-memory-model arrays are byte addressed; all other arrays are word addressed |

TAL structures can emulate multidimensional C arrays, as discussed in “Multidimensional Arrays” later in this section.

To declare compatible TAL and C arrays:

- Use data types and alignments that satisfy both compilers.
- Declare TAL arrays that have a lower bound of 0.
- Declare one-dimensional C arrays.
- Declare indirect TAL arrays.

The following are compatible arrays in TAL and C (large-memory model):

| TAL Code                 | C Code            |
|--------------------------|-------------------|
| INT .EXT robin[0:9];     | short robin [10]; |
| INT(32) .EXT gull[0:14]; | long gull [15];   |
| STRING .EXT grebe[0:9];  | char grebe [10];  |

### Structures

All TAL and C structures begin on a word boundary. Following are guidelines for sharing TAL and C structures and passing them as parameters:

- Specify the same layout for corresponding TAL and C structures.
- Specify compatible data types for each item of both structures.
- In TAL, pass structures by reference.
- In C, use the & operator.
- In TAL, a routine cannot return a structure as a return value.

The following TAL and C structures have compatible layouts:

| TAL Code                     | C Code          |
|------------------------------|-----------------|
| STRUCT rec (*);              | struct birdname |
| BEGIN                        | {               |
| INT x;                       | short x;        |
| STRING y[0:2];               | char y[3];      |
| END;                         | } robin[10];    |
| STRUCT .EXT robin(rec)[0:9]; |                 |

The following TAL and C structures have compatible layouts:

| TAL Code         | C Code        |
|------------------|---------------|
| STRUCT rec1 (*); | struct rec1   |
| BEGIN            | {             |
| STRING a, b, c;  | char a, b, c; |
| END;             | };            |

The following TAL and C structures also have compatible layouts:

| TAL Code         | C Code      |
|------------------|-------------|
| STRUCT rec2 (*); | struct rec2 |
| BEGIN            | {           |
| STRING e;        | char e;     |
| INT y;           | short y;    |
| STRING g;        | char g;     |
| END;             | };          |

## Substructures

The TAL compiler allocates alignment of substructures on a byte or word boundary as follows:

- Each definition substructure occurrence is byte aligned if the first item it contains begins on a byte boundary.
- Each definition substructure occurrence is word aligned if the first item it contains begins on a word boundary.
- Each referral substructure occurrence is always word aligned.

C substructures always begin and end on word boundaries.

In this example, TAL referral substructure TSUB and C substructure CSUB have compatible layouts:

| <b>TAL Code</b>     | <b>C Code</b>     |
|---------------------|-------------------|
| STRUCT rec1 (*);    | struct rec1       |
| BEGIN               | {                 |
| STRING a, b, c;     | char a, b, c;     |
| END;                | };                |
| <br>                |                   |
| STRUCT rec3 (*);    | struct rec3       |
| BEGIN               | {                 |
| INT x;              | short x;          |
| STRING var;         | char var;         |
| STRUCT tsub (rec1); | struct rec1 csub; |
| STRING f;           | char f;           |
| END;                | };                |

In this example, TAL definition substructure TSUB1 follows a STRING variable and begins on a byte boundary. The layouts of TSUB1 and CSUB1 are not compatible, so you cannot share the substructures between the two languages:

| <b>TAL Code</b>  | <b>C Code</b> |
|------------------|---------------|
| STRUCT rec4 (*); | struct rec4   |
| BEGIN            | {             |
| INT x;           | short x;      |
| STRING a;        | char a;       |
| STRUCT tsub1;    | struct        |
| BEGIN            | {             |
| STRING b,c,d;    | char b,c,d;   |
| END;             | } csub1;      |
| STRING e;        | char e;       |
| END;             | };            |

If you use the Data Definition Language (DDL) to describe your files, the byte-aligned substructure layout is the only layout DDL cannot generate. (DDL is described in the *Data Definition Language (DDL) Reference Manual*.)

You can ensure compatible layouts between TAL and C substructures as follows:

- Declare TAL referral substructures rather than definition substructures. Referral substructures are always word-aligned.
- If you must declare TAL definition substructures, either:
  - Use FILLER declarations as needed to begin and end TAL definition substructures on word boundaries, as shown in TAL structure REC4 in the example that follows.
  - Declare C structures that emulate the layout of a byte-aligned TAL substructure, as shown in C structure REC5 in the second example that follows.

In TAL structure REC4, each FILLER declaration inserts a pad byte before and after the definition substructure TSUB2 so it begins and ends on a word boundary. Thus, the following TAL and C structures have compatible layouts and can be shared:

| <b>TAL Code</b>                                                                                                                                  | <b>C Code</b>                                                                                        |
|--------------------------------------------------------------------------------------------------------------------------------------------------|------------------------------------------------------------------------------------------------------|
| <pre>STRUCT rec4 (*); BEGIN   INT x;   STRING a;   FILLER 1;   STRUCT tsub2;   BEGIN     STRING b,c,d;   END;   FILLER 1;   STRING e; END;</pre> | <pre>struct rec4 {   short x;   char a;    struct   {     char b,c,d;   } csub2;    char e; };</pre> |

In C structure REC5, three variables (not a substructure) emulate the byte-aligned layout of TAL substructure ST. Thus, the following TAL and C structures have compatible layouts and can be shared:

| <b>TAL Code</b>                                                                                                                                                          | <b>C Code</b>                                                                                                        |
|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------|----------------------------------------------------------------------------------------------------------------------|
| <pre>STRUCT rec5 (*); BEGIN   INT x;   STRING a;   STRUCT st;   BEGIN     STRING b;     STRING c;     STRING d;   END;   STRING e; END;  STRUCT .EXT match (rec5);</pre> | <pre>struct rec5 {   short x;   char a;    char stb;   char stc;   char std;    char e; };  struct rec5 match;</pre> |

### Multidimensional Arrays

In C, you can declare multidimensional arrays. In TAL, you can emulate multidimensional arrays by declaring structures that contain arrays.

Here is an example of multidimensional arrays in TAL and C (large-memory model):

| <b>TAL Code</b>             | <b>C Code</b>       |
|-----------------------------|---------------------|
| STRUCT rec1 (*);            |                     |
| BEGIN                       |                     |
| INT y[0:4];                 | short cma[10][5];   |
| END;                        |                     |
| STRUCT .EXT tma(rec1)[0:9]; |                     |
| !Sample access!             | /* sample access */ |
| tma[8].y[3] := 100;         | cma[8][3] = 100;    |

### Arrays of Structures

If you specify bounds when you declare a TAL structure, you create an array of structures. The following TAL and C arrays of structures are equivalent. Each declaration contains an array of ten structure occurrences:

| <b>TAL Code</b>               | <b>C Code</b>           |
|-------------------------------|-------------------------|
| STRUCT cell (*);              | struct cell             |
| BEGIN                         | {                       |
| INT x;                        | short x;                |
| STRING y;                     | char y;                 |
| END;                          | };                      |
| STRUCT .EXT tcell(cell)[0:9]; | struct cell ccell [10]; |
| PROC honey (c);               | void JOANIE             |
| INT .EXT c (cell);            | (struct cell *);        |
| EXTERNAL;                     |                         |

### Redefinitions and Unions

Variant records are approximated by TAL structure redefinitions and C unions.

A TAL redefinition declares a structure item that uses the same memory location as an existing structure item. The existing structure item can be a simple variable, array, substructure, or pointer that:

- Begins on a word boundary
- Is at the same BEGIN-END level in the structure as the redefinition
- Is the same size or larger than the redefinition

A C union defines a set of variables that can have different data types and whose values alternatively share the same portion of memory. The size of a union is the size of its largest variable; the largest item need not come first. A union always begins on a word boundary.

Here is an example of TAL redefinitions and equivalent C unions:

| <b>TAL Code</b>                                                                                                                                                                                                                                                                     | <b>C Code</b>                                                                                                                                                                                                                                  |
|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <pre>STRUCT mtns (*);   BEGIN     INT(32) tamalpais;     INT diablo = tamalpais;     STRING hamilton = diablo;     FIXED num;     STRUCT cascade = num;       BEGIN         INT ranier;         INT sthelens;         INT adams;         INT hood;       END;     END;   END;</pre> | <pre>struct Mtns { union {   long Tamalpais;   short Diablo;   char Hamilton;} Calif_mtns;   union {   long long Num;   struct {     short Ranier;     short StHelens;     short Adams;     short Hood; } Cascade;   } Northern_mtns; };</pre> |
| <pre>STRUCT c_high (mtns);</pre>                                                                                                                                                                                                                                                    | <pre>struct Mtns High;</pre>                                                                                                                                                                                                                   |

The following identifiers access equivalent structure items in the preceding example:

|         |                        |
|---------|------------------------|
| In TAL: | c_high.diablo          |
| In C:   | High.Calif_mtns.Diablo |

## Pointers

Pointers contain memory addresses of data. You must store an address into a pointer before you use it. In TAL and C pointer declarations, you specify the data type of the data to which the pointer points. You must use pointers when sharing global variables. You can pass pointer contents by value between TAL and C routines.

Differences between TAL and C pointers include the following:

- TAL structure pointers can point to a byte or word address.
- C structure pointers always point to a word address. To pass a C structure pointer to a TAL routine that expects a byte structure pointer, you must explicitly cast the C pointer to type char.
- TAL pointers are dereferenced implicitly.
- C pointers are usually dereferenced explicitly.
- Small-memory-model C routines use 16-bit pointers only.
- Large-memory-model C routines use 32-bit pointers only, even if the pointers refer to the user data segment. In global structure declarations, you must specify lowmem in the storage class of the declaration.
- If a TAL routine expects a 16-bit pointer, the C pointer you pass must refer to an object in user data space.

Here are examples of TAL and C pointers (large-memory model):

| <b>TAL Code</b>                                                                 | <b>C Code</b>                                                      |
|---------------------------------------------------------------------------------|--------------------------------------------------------------------|
| <pre>STRUCT rec (*);   BEGIN     INT d;     INT .p (rec);   END;</pre>          | <pre>struct rec {   short d;   struct rec *p; };</pre>             |
| <pre>BLOCK joe;   INT .EXT joes (rec); END BLOCK;</pre>                         | <pre>struct rec *JOE;</pre>                                        |
| <pre>PROC tonga (p);   INT .EXT p (rec);   BEGIN     !Lots of code   END;</pre> | <pre>void CALEDONIA (struct rec *p) {   /* Lots of code */ }</pre> |

Each language can call the other, passing the address in the pointer by value:

| <b>TAL Code</b>                   | <b>C Code</b>           |
|-----------------------------------|-------------------------|
| <pre>CALL caledonia (joes);</pre> | <pre>TONGA (joe);</pre> |

Here are examples of TAL and C structure pointers (large-memory-model) that implement a linked list:

**TAL Code**

```
STRUCT rec (*);
 BEGIN
 INT x;
 INT .EXT strptr (rec);
 END;

STRUCT .EXT joe (rec);

PROC callme (param1);
 INT .EXT param1 (rec);
EXTERNAL;
```

**C Code**

```
struct rec
{
 short x;
 struct rec *p;
};

struct rec joe;

void f1 (struct rec *);
```

### Enumeration Variables

Using C enumeration constants, you can associate a group of named constant values with an int variable. A C enumeration variable occupies 16 bits of memory. You define all integer operations on them. The C compiler provides no range checking, so an enumeration variable can hold a value not represented in the enumeration.

A C routine can share an enumeration variable with TAL routines. A TAL routine cannot access the enumeration constants, but can declare LITERALS for readability. For example:

#### TAL Code

```
LITERAL no = 0,
 yes = 3,
 maybe = 4;

BLOCK answer;
 INT answer_var;
END BLOCK;
```

#### C Code

```
enum choice {no = 0,
 yes = 3,
 maybe = 4 };

enum choice ANSWER;
```

A C routine can pass enumeration parameters to TAL routines, placing the actual value in a TAL INT variable. For example:

#### TAL Code

```
LITERAL no = 0,
 yes = 3,
 maybe = 4;

PROC tal_proc (n);
 INT n;
 BEGIN
 !Lots of code
 IF n = yes THEN ... ;
 !Lots of code
 END;
```

#### C Code

```
enum choice {no = 0,
 yes = 3,
 maybe = 4 };

enum choice answer;

_tal void TAL_PROC (short);

main ()
{
 answer = yes;
 TAL_PROC (answer);
 /* lots of code */
}
```

### Bit-Field Manipulation

You can manipulate bit fields in both TAL and C.

In TAL, you can use either:

- Built-in bit-extraction and bit-deposit operations
- Bit-wise operators LAND and LOR

In C, you can use either:

- Bit-wise operators & (and) and | (or)
- Defines

The following TAL bit-deposit operation and C bit-wise operation are equivalent:

#### TAL Code

```
INT x := -1;
INT y := 0;

PROC example;
 BEGIN
 y.<0:2> := x.<10:12>;
 END;
```

#### C Code

```
short a = -1;
short b = 0;
short temp = 0;

void example ()
{
 /* you can combine these */
 /* with wider margins */

 temp = a & 070;
 temp = temp << 10;
 b = (b & 017777)|temp;
}
```

Bit extractions and bit deposits are not portable to future software platforms.

### UNSIGNED Variables and Packed Bit Fields

In general, TAL UNSIGNED simple variables in structures are compatible with C unsigned packed bit fields (which only appear in structures). You cannot, however, pass C bit fields as reference parameters to TAL routines.

The following UNSIGNED variables and C unsigned bit fields are compatible:

| TAL Code                 | C Code                 |
|--------------------------|------------------------|
| STRUCT stuffed (*);      | struct stuffed;        |
| BEGIN                    | {                      |
| INT x;                   | int x;                 |
| UNSIGNED(1) a;           | unsigned a : 1;        |
| UNSIGNED(5) b;           | unsigned b : 5;        |
| UNSIGNED(3) c;           | unsigned c : 3;        |
| UNSIGNED(4) d;           | unsigned d : 4;        |
| UNSIGNED(9) e;           | unsigned e : 9;        |
| UNSIGNED(2) f;           | unsigned f : 2;        |
| END;                     | };                     |
| STRUCT packed (stuffed); | struct stuffed PACKED; |

When the WIDE pragma is not specified, the C compiler normally packs adjacent bit fields in a 16-bit word. When the WIDE pragma is specified, the C compiler normally packs adjacent bit fields in a 32-bit word.

TAL UNSIGNED(1-16) and C bit fields of like size are compatible. TAL UNSIGNED(17-31) and C bit fields of like size are compatible.

The TAL compiler always packs adjacent UNSIGNED simple variables in 16-bit words as follows:

- It starts the first UNSIGNED variable on a word boundary.
- It packs each successive UNSIGNED variable in the remaining bits of the same word as the preceding UNSIGNED variable if:
  - The variable contains 1 to 16 bits and fits in the same word
  - The variable contains 17 to 31 bits and fits in the same word plus the next word
- If an UNSIGNED variable does not fit in the same word or doubleword, the compiler starts the variable on the next word boundary.

The operator you use determines whether UNSIGNED values are signed or unsigned:

|                  |                     |
|------------------|---------------------|
| UNSIGNED(3) x;   | !TAL code           |
| UNSIGNED(3) y;   |                     |
| IF x + y ... ;   | !Signed operation   |
| IF x '+' y ... ; | !Unsigned operation |

UNSIGNED arrays that contain 8-bit or 16-bit elements are compatible with C arrays that contain elements of like size. UNSIGNED arrays that contain 1-bit, 2-bit, or 4-bit elements are incompatible with C arrays.

### TAL Routines as Parameters to C

You can call C routines and pass TAL routines as parameters. You can pass any TAL routine except EXTENSIBLE or VARIABLE routines as parameters.

A passed TAL routine can access the routine's local variables and global TAL variables. The passed routine can contain subprocedures, but they cannot be passed as parameters.

If you call a large-memory-module C routine, the EXTERNAL procedure declaration for the C routine must specify the PROC(32) parameter type in the parameter declaration. When you pass the PROC(32) parameter to the C routine, the compiler passes a 32-bit address that contains PEP and map information in the high-order word and a zero in the low-order word.

If you call a small-memory-module C routine, the EXTERNAL procedure declaration for the C routine must specify the PROC parameter type in the parameter declaration. When you pass the PROC parameter to the C routine, the compiler passes a 16-bit address that contains PEP and map information.

In the following example, a large-memory-model C module contains C\_FUNC, which expects a TAL procedure as a parameter. The TAL module contains:

- An EXTERNAL procedure declaration for C\_FUNC
- TAL\_PARAM\_PROC, a routine to be passed as a parameter to C\_FUNC
- TAL\_CALLER, a routine that calls C\_FUNC and passes TAL\_PARAM\_PROC as a parameter

**C Module**

```
/* C function that accepts TAL routine as a parameter */
void C_FUNC (short (*F) (short n))
{
 short j;
 j = (*F)(2);
 /* lots of code */
}
```

**TAL Module**

```
PROC c_func (x) LANGUAGE C; !EXTERNAL procedure declaration
 ! for C routine to be called
 INT PROC(32) x; !Parameter declaration
EXTERNAL;

INT PROC tal_param_proc (f); !Procedure to be passed as a
 INT f; !parameter to C_FUNC
BEGIN
RETURN f;
END;

PROC tal_caller; !Procedure that calls C_FUNC
BEGIN ! and passes TAL_PARAM_PROC
 !Lots of code
CALL c_func (tal_param_proc);
 !Lots of code
END;

PROC m MAIN;
BEGIN
CALL tal_caller;
END;
```

### C Routines as Parameters to TAL

You can call TAL routines and pass C routines as parameters. You can call a TAL entry-point identifier as if it were the routine identifier. C routines cannot be nested.

When a called TAL routine in turn calls a C routine received as a parameter, the TAL routine assumes that all required parameters of the C routine are value parameters. The TAL compiler has no way of checking the number, type, or passing method expected by the C routine. If the C routine requires a reference parameter, the TAL routine must explicitly pass the address by using:

- The @ operator for a small-memory-model parameter
- The \$XADR standard function for a large-memory-model parameter

In the following example, a C large-memory-model module contains C routine C\_PARAM\_FUNC, which is to be passed as a parameter. The TAL module contains:

- An EXTERNAL procedure declaration for C\_PARAM\_FUNC
- TAL\_PROC, which expects C\_PARAM\_FUNC as a parameter
- TAL\_CALLER, which calls TAL\_PROC and passes C\_PARAM\_FUNC as a parameter

#### TAL Module

```

INT i;
STRING .EXT s[0:9];

PROC c_param_func (i, s) !EXTERNAL procedure declaration
 LANGUAGE C; ! for C routine expected as
 INT i; ! a parameter
 STRING .EXT s; !Extended indirection for large-
 EXTERNAL; ! memory-model

PROC tal_proc (x); !TAL routine that expects
 PROC(32) x; ! a large-memory-model C routine
 BEGIN ! as a parameter
 CALL x (i, $XADR (s));
 END;

PROC tal_caller;
 BEGIN
 CALL tal_proc (c_param_func);
 END;

PROC m main;
 BEGIN
 CALL tal_caller;
 END;

```

#### C Module

```

void C_PARAM_FUNC (short i, char * s)
{
 /* C routine to be passed as */
 /* a parameter to TAL_PROC */
}

```

When you pass a large-memory-model C routine as a parameter, the compiler passes a 32-bit address that contains PEP and map information in the high-order word and a zero in the low-order word. When you pass a small-memory-model C routine as a parameter, the compiler passes a 16-bit address that contains PEP and map information.

**Extended Data Segments** In addition to the user data segment, you can store data in:

- The automatic extended data segment
- One or more explicit extended data segments

You should use only the automatic extended data segment if possible. You should not mix the two approaches. If you must use explicit segments and the automatic segment, however, follow guidelines 4 and 11 in “Explicit Extended Data Segments” later in this section.

#### Automatic Extended Data Segment

The TAL compiler allocates the automatic extended data segment when a TAL program declares arrays or structures that have extended indirection.

The C compiler allocates the automatic extended data segment for all large-memory-model modules.

#### Explicit Extended Data Segments

TAL modules and large-memory-model C modules can allocate and deallocate extended data segments explicitly. Since the advent of the automatic extended data segment, however, programs usually need not use explicit extended data segments. The information in this subsection is provided to support existing programs.

To create and use an explicit extended segment, you call system procedures. You can allocate and manage as many extended segments as you need, but you can use only one extended segment at a time. You can access data in an explicit extended segment only by using extended pointers. The compiler allocates memory space for the extended pointers you declare. You must manage allocation of the data yourself.

Here are guidelines for using explicit extended segments:

1. Declare an extended pointer for the base address of the explicit extended segment.
2. To allocate an explicit extended segment and obtain the segment base address, call `SEGMENT_ALLOCATE_`.
3. To make the explicit extended segment the current segment, call `SEGMENT_USE_`.
4. If the automatic extended segment is already in place, `SEGMENT_USE_` returns the automatic extended segment's number. Save this number so you can later access the automatic segment again.
5. C requires special treatment for `SEGMENT_USE_`, which returns a value and sets the condition code. (See the *C Reference Manual*.)

6. You must keep track of addresses stored in extended pointers. When storing addresses into subsequent pointers, you must allow space for preceding data items. (See “Managing Data Allocation in Extended Segments” in Appendix B.)
7. To refer to data in the current segment, call READX or WRITEX .
8. To move data between extended segments, call MOVEX.
9. To manage large blocks of memory, call DEFINEPOOL, GETPOOL, and PUTPOOL.
10. To determine the size of a segment, call SEGMENT\_GETINFO\_.
11. To access data in the automatic extended segment, call SEGMENT\_USE\_ and restore the segment number that you saved at step 4.
12. To delete an explicit segment that you no longer need, call SEGMENT\_DEALLOCATE\_.

For information on using these system procedures, see the *Guardian Programmer's Guide* and the *Guardian Procedure Calls Reference Manual*.

If you do not restore the automatic extended data segment before you manipulate data in it, any of the following actions can result:

- An assignment statement is beyond the segment's memory limit and causes a trap.
- All assignments within range occur in the hardware base and limit registers of the automatic extended segment. Data in the currently active extended segment is overwritten. The error is undetected until you discover the discrepancy at a later time.
- The C code runs until it accesses an invalid address or accesses an inaccessible library routine.
- You get the wrong data from valid addresses in the explicit extended data segment.

In Example 17-1, a large-memory-model C routine calls a TAL routine that manipulates data in an explicit extended segment and then restores the automatic extended segment. When control returns to the C routine, it manipulates data in the restored automatic extended segment:

---

**Example 17-1. D-Series TAL and C Extended Segment Management (Page 1 of 2)**

**TAL Code**

```

INT .EXT array[0:10]; !Allocated in the automatic
 ! extended data segment ID 1024D

INT .EXT arr_ptr;

?PUSHLIST, NOLIST, SOURCE $SYSTEM.SYSTEM.EXTDECS0 (
? PROCESS_DEBUG_, DEFINEPOOL, GETPOOL,
? SEGMENT_ALLOCATE_, SEGMENT_USE_, SEGMENT_DEALLOCATE_)
?POPLIST

PROC might_lose_seg;
BEGIN
 INT status := 0;
 INT old_seg := 0;
 INT new_seg := 100;
 INT(32) seg_len := %2000D; !1024D

 array[0] := 10; !Do work in automatic segment

 status := SEGMENT_ALLOCATE_ (
 new_seg, seg_len, , , , arr_ptr);
 !Allocate an explicit extended data segment;
 !store segment base address in ARR_PTR
 IF status <> 0 THEN CALL PROCESS_DEBUG_;

 Status := SEGMENT_USE_ (new_seg, old_seg, arr_ptr);
 !Make the explicit extended data segment current
 IF status <> 0 THEN CALL PROCESS_DEBUG_;

 !Use DEFINEPOOL, GETPOOL to retrieve a block in the
 ! explicit extended data segment.

 arr_ptr[2] := 10; !Do some work in the segment.

 !When you no longer need the explicit extended data
 ! segment, call SEGMENT_DEALLOCATE_.

 status := SEGMENT_USE_ (old_seg);
 !Restore the automatic extended data segment
 IF status <> 0 THEN CALL PROCESS_DEBUG_;

 END;

```

---

---

**Example 17-1. D-Series TAL and C Extended Segment Management (Page 2 of 2)****C Code**

```
#pragma symbols, inspect, strict

short arr[10];
char sarr[10];
char *s;

tal void MIGHT_LOSE_SEG (void);

main ()
{
 s = &sarr[0];
 *s = 'A';
 arr[0] = 10;
 MIGHT_LOSE_SEG (); /* Call TAL routine, which uses the*/
 /* explicit extended data segment */

 /* next two statements depend on the automatic extended */
 /* data segment being restored after the call to TAL */
 sarr[1] = *s;
 arr[1] = arr[0] + 5;
}
```

---

For a TAL example program that manages an explicit extended data segment, see Appendix B, “Managing Addressing.”

## CRE Guidelines for TAL

The CRE provides routines that support mixed-language programs compiled on D-series compilers. A mixed-language program can consist of C, COBOL85, FORTRAN, Pascal, and TAL routines. By using the CRE, each routine in your program, regardless of language, can:

- Use its run-time library without overwriting the data of another run-time library
- Share data in the CRE user heap
- Access the standard files (standard input, standard output, and standard log)

Without the CRE, only routines written in the language of the MAIN routine can fully access their run-time library. For example, if the MAIN routine is written in TAL, a routine written in another language might not be able to use its own run-time library.

D-series C and Pascal routines run only in the CRE. D-series COBOL85, FORTRAN, and TAL routines can run in the CRE if you specify the ENV COMMON directive. Such programs must also meet CRE, system, and language requirements. The *CRE Programmer's Guide* describes CRE requirements and routines.

This section gives CRE guidelines for TAL programs. It discusses the following:

- General coding guidelines
- Specifying a run-time environment
- Setting the user heap size
- Initializing the CRE
- Terminating programs
- Sharing standard files
- Using \$RECEIVE
- Handling errors in CRE math routines

## General Coding Guidelines

All D-series language products except TAL have run-time libraries that call CRE routines and system routines as needed. TAL routines that meet CRE requirements can call CRE routines (and system procedures) directly.

The following list summarizes some general guidelines for coding D-series TAL source code for the CRE. The subsections that follow this list give more information about some of these guidelines:

- For the MAIN routine, specify the following TAL directives:
  - The ENV directive with the COMMON attribute to request the CRE
  - The HEAP directive if any routine needs the user heap in the CRE
- To run in the CRE, your program needs the TALLIB and CRELIB library files.
- As of the D20 release, your program can manipulate saved startup, PARAM, and ASSIGN messages by using the Saved Messages Utility (SMU) functions provided by the CLULIB library file.
- The first statement of the MAIN routine must call TAL\_CRE\_INITIALIZER\_. This routine initializes the CRE and optionally saves the startup, PARAM, and ASSIGN messages.

- To terminate execution, call `CRE_TERMINATOR_`, which calls `PROCESS_STOP_`. Do not call the `PROCESS_STOP_`, `STOP`, or `ABEND` system procedures.
- For services provided by the CRE, call CRE routines. For example, call CRE routines to:
  - Open, manipulate, and close the standard files
  - Perform math or string operations
- For services not provided by the CRE, call system procedures. For example, call system procedures to:
  - Create processes
  - Manage extended data segments
- Do not call the `ABEND`, `ARMTRAP`, `PROCESS_STOP_`, or `STOP` system procedure, because of probable conflict with `TAL_CRE_INITIALIZER_` and `CRE_TERMINATOR_`.
- Do not call the `INITIALIZER` system procedure or read the startup message from `$RECEIVE` itself, because of probable conflict with `TAL_CRE_INITIALIZER_` and `CRE_TERMINATOR_`.
- If you use sequential I/O (SIO) procedures (such as `WRITE^FILE`, `SET^FILE`, and `READ^FILE`), resolve how to remove any calls to the `INITIALIZER` procedure. The *Guardian Programmer's Guide* describes how you use the SIO and `INITIALIZER` procedures.
- Do not use the `DATAPAGES`, `EXTENDSTACK`, or `STACK` directive. (The compiler automatically allocates 64K words of memory space for the user data segment.)
- Do not write data to areas in the user data segment that are reserved for use by the CRE; that is, do not use:
  - Locations `G[0]` or `G[1]`
  - The upper 32K-word area
- Your program can access `$RECEIVE` by calling either CRE routines only or system procedures only, as described in "Accessing `$RECEIVE`" later in this section.
- When an error occurs in a CRE routine, the CRE returns control to the calling TAL routine without taking any action. The TAL routine must explicitly handle all errors except certain errors detected by the CRE routines.
- If a CRE math routine detects an error, the calling TAL routine must manage the trap enable bit of the environment register to control program behavior.

**Specifying a Run-Time Environment**

To specify the intended run-time environment of a D-series TAL compilation unit, use the ENV directive. To execute in the intended environment, the routines must also meet the requirements of the intended run-time environment.

The attributes of the ENV directive are:

| ENV Attribute | Intended Run-Time Environment                            |
|---------------|----------------------------------------------------------|
| COMMON        | The CRE.                                                 |
| OLD           | A COBOL or FORTRAN run-time environment outside the CRE. |
| NEUTRAL       | None; the program relies primarily on system procedures. |

You can include the ENV directive in the TAL compilation command or in the compilation unit before any declarations. The ENV directive can appear only once during a compilation.

For example, you can specify ENV on a directive line and include the COMMON attribute:

```
?ENV COMMON
```

If you compile without the ENV directive, all routines in the compilation unit have the ENV NEUTRAL attribute.

**ENV COMMON Directive**

An object file can run in the CRE if the MAIN routine has the ENV COMMON attribute and if all routines meet CRE requirements. If ENV COMMON is in effect, all routines in the compilation unit (except routines in object files listed in SEARCH directives) have the ENV COMMON attribute.

SEARCH directives can list object files compiled with any ENV attribute except OLD. Each routine in a SEARCH file retains its original ENV attribute.

When a compilation unit contains a MAIN routine and ENV COMMON is in effect, the compiler allocates special CRE data blocks and initializes certain fields of those data blocks. These data blocks, listed in Table 17-3, are reserved for use by the CRE.

**Table 17-3. CRE Data Blocks**

| Data Block           | Name         | Location                                                                                                                      |
|----------------------|--------------|-------------------------------------------------------------------------------------------------------------------------------|
| Basic control block  | #CRE_GLOBALS | At G[0] and G[1] of the user data segment.                                                                                    |
| Master control block | #MCB         | In the upper 32K-word area of the user data segment.                                                                          |
| CRE heap             | #CRE_HEAP    | Last block in the upper 32K-word area of the user data segment. (This heap differs from the CRE user heap block named #HEAP.) |

Using Binder, you can bind an object file compiled with ENV COMMON to any object file except those compiled with ENV OLD. Each routine in the new object file retains its original ENV attribute.

### **ENV OLD Directive**

An object file compiled with ENV OLD can run in a COBOL85 or FORTRAN run-time environment outside the CRE (if the object file meets the requirements of the run-time environment).

When ENV OLD is in effect, all routines in the compilation unit (except routines in object files listed in SEARCH directives) have the ENV OLD attribute. SEARCH directives can list object files compiled with any ENV attribute except COMMON. Each routine in a SEARCH file retains its original ENV attribute.

Using Binder, you can bind a TAL object file compiled with ENV OLD to any object file (regardless of language) except those compiled with ENV COMMON. Each routine in the new object file retains its original ENV attribute.

A D-series TAL program compiled with ENV OLD can run on a C-series system. To debug the program with the Inspect product, however, your system must be a release level C30.06 or later

### **ENV NEUTRAL Directive**

An object file compiled with ENV NEUTRAL should not rely on any external services except system procedures.

When ENV NEUTRAL is in effect, all routines in the compilation unit (except routines in object files listed in SEARCH directives) have the ENV NEUTRAL attribute. SEARCH directives can list object files compiled with ENV NEUTRAL or with no ENV directive. Each routine in the new object file has the ENV NEUTRAL attribute.

Using Binder, you can bind an object file compiled with ENV NEUTRAL to any object file. Each routine in the new object file retains its original ENV attribute.

### **ENV Directive Not Specified**

An object file compiled without the ENV directive probably does not rely on any external services except system procedures.

When no ENV directive is in effect, all routines in the compilation unit (except routines in object files listed in SEARCH directives) have the ENV NEUTRAL attribute. SEARCH directives can list object files compiled with any ENV attribute. Each routine in a SEARCH file retains its original ENV attribute.

Using Binder, you can bind an object file compiled without the ENV directive to any object file. Each routine in the new object file retains its original ENV attribute.

**The User Heap** The user heap is a shared resource the CRE makes available to your routines, regardless of language. The user heap is in the block named #HEAP, which differs from the CRE-reserved block named #CRE\_HEAP.

Binder allocates memory space for the user heap as follows:

- If a TAL program contains a small-memory model C or Pascal module, the user heap is the last global data block in the lower 32K-word area of the user data segment. (The last data block is located just below the user data stack.)
- If a TAL program contains a large-memory-model C or Pascal module or contains no C or Pascal routines, the user heap is the last data block in the automatic extended data segment.

If the TAL program also uses explicit extended data segments, follow the instructions given in “Explicit Extended Data Segments” earlier in this section.

### Setting the User Heap Size

To set the size of the user heap, specify the HEAP directive in the TAL compilation command or anywhere in the TAL module that contains the MAIN routine. The size specified in the last HEAP directive encountered in the TAL compilation takes effect. The ENV COMMON directive must also be in effect. If a TAL program invokes a routine that needs the user heap, the HEAP directive is required. The following TAL directive line sets a user heap of 5 memory pages:

```
?HEAP 5
```

The heap size is the number of 2048-byte memory pages. Specify an unsigned decimal constant in one of the following ranges:

| Memory Model | Range            | Notes                                                                                                                                                                                                                                 |
|--------------|------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Small        | 0 through 31     | The heap size you can specify depends on your use of the lower 32K-word area of the user data segment.                                                                                                                                |
| Large        | 0 through 32,767 | When a program runs, the CRE increases the heap size as needed up to the maximum size of the automatic extended segment. Do not set an arbitrarily large size because an equal amount of disk space (for swapping) must be available. |

### Accessing the User Heap

Only C and Pascal routines can allocate and return user heap space. To allocate and return user heap space, C and Pascal routines call the following library routines:

| Request           | C Library Routine | Pascal Library Routine |
|-------------------|-------------------|------------------------|
| Obtain heap space | malloc            | NEW                    |
| Return heap space | free              | DISPOSE                |

Only C, Pascal, or TAL routines can access data in the user heap. To access heap data, a C, Pascal, or TAL routine assigns the address returned from malloc or NEW to a pointer and uses the pointer in an expression. All routines that need access to such data must either:

- Use the same data layout
- Call a routine that accesses the data for you

COBOL85 and FORTRAN routines access heap data by calling C, Pascal, or TAL routines.

In the following example, TAL routine DO\_IT calls C routine GETSPACE. GETSPACE gets a block from the heap, stores a value in the first word of the block, and returns to DO\_IT a pointer that points to the data:

#### C Code

```
#include <stddef.h> nolist
#include <stdlib.h> nolist
int *GETSPACE (int nbytes)
{ int *p;
 p = (int *) malloc (nbytes);
 if (p)
 *p = 100;
 return p;
}
```

#### TAL Code

```
INT(32) PROC tal_malloc = "GETSPACE" (nbytes) LANGUAGE C;
 INT nbytes;
 EXTERNAL;

PROC do_it;
 BEGIN
 INT .EXT t_ptr;
 @t_ptr := tal_malloc (1000);
 IF (@t_ptr = 0d) OR (t_ptr <> 100) THEN
 BEGIN
 !Handle error ...
 END;
 END;
```

**Including Library Files** To run in the CRE, your program needs the TALLIB library file. Your program can also use the CRELIB and CLULIB library files.

TALLIB provides the TAL\_CRE\_INITIALIZER\_ routine. This routine is described in the next subsection.

CRELIB provides routines that enable your routine to perform tasks such as the following. These routines are described in the *CRE Programmer's Guide*:

- Share files
- Manipulate \$RECEIVE
- Terminate the CRE
- Handle exceptions
- Perform standard math functions
- Manipulate strings
- Manage memory blocks

CLULIB (as of the D20 release) contains SMU functions that enable your routine to manipulate saved startup, ASSIGN, and PARAM message values. These routines are described in the *CRE Programmer's Guide*.

The following table lists the files that your program must include depending on which set of routines it uses. The table also indicates the run-time environment in which you can use the library routines.

**Table 17-4. Including Library and External Declaration Files**

| Library File | Including Library Routines         | Routine Prefix | Declaration File to Source In | Run-Time Environment          |
|--------------|------------------------------------|----------------|-------------------------------|-------------------------------|
| TALLIB       | SEARCH directive or Binder command | TAL_           | TALDECS                       | CRE                           |
| CRELIB       | No action *                        | CRE_           | CREDECS                       | CRE                           |
| CRELIB       | No action *                        | RTL_           | RTLDECS                       | CRE                           |
| CLULIB       | SEARCH directive or Binder command | SMU_           | CLUDECS                       | CRE, COBOL85, FORTRAN, or TAL |

\* The CRELIB file is configured into the system library.

CREDECS and RTLDECS contain blocked global data declarations (that is, data declared inside BLOCK declarations). If you include blocked data declarations in a compilation unit, a NAME declaration is required. You must specify global declarations in the following order:

1. NAME declaration
2. Unblocked global data declarations, if any
3. Blocked global data declarations from CREDECS, RTLDECS, and user code, if any
4. Procedure declarations

The following example shows SEARCH directives for TALLIB and CLULIB, followed by SOURCE directives for TALDECS, CREDECS, and CLUDECS:

```
?ENV COMMON
NAME cre_example;
?SEARCH $SYSTEM.SYSTEM.TALLIB
?SEARCH $SYSTEM.SYSTEM.CLULIB
?SOURCE $SYSTEM.SYSTEM.TALDECS (TAL_CRE_INITIALIZER_)
?SOURCE $SYSTEM.SYSTEM.CREDECS (CRE_TERMINATOR_)
?SOURCE $SYSTEM.SYSTEM.CLUDECS (SMU_ASSIGN_DELETE_)
```

You can specify the SEARCH directive in the command to start the compiler, instead of in the source file. Here is an example:

```
TAL /IN mysrc, OUT $s.#mylst, NOWAIT/ myprog;
SEARCH $SYSTEM.SYSTEM.TALLIB
```

If you do not use the SEARCH directive for TALLIB and CLULIB, you can bind the files into the object file by issuing Binder commands. Here is an example:

```
ADD * FROM myprog
SELECT SEARCH $SYSTEM.SYSTEM.TALLIB
SELECT SEARCH $SYSTEM.SYSTEM.CLULIB
BUILD myprog
```

**Initializing the CRE** To initialize the CRE and optionally save system messages, call `TAL_CRE_INITIALIZER_` in the first statement of your MAIN routine. `TAL_CRE_INITIALIZER_` does the following actions:

- It calls the ARMTRAP procedure and establishes a trap handler.
- It optionally saves startup, ASSIGN and PARAM messages.
- It determines the name of your program's standard log.

The TALLIB file contains `TAL_CRE_INITIALIZER_`. The TALDECS file contains the external declaration for `TAL_CRE_INITIALIZER_`, as follows:

```
PROC TAL_CRE_INITIALIZER_ (options) EXTENSIBLE;
 INT options; !Input, optional
EXTERNAL;
```

The OPTION parameter lets you save the startup, ASSIGN, and PARAM messages for manipulation by your routine. OPTION flags you can specify for the actual parameter are provided in the CREDECS file in the INITIALIZATION section, as follows:

```
CRE^Save^all^messages
CRE^Save^startup^message
CRE^Save^assign^message
CRE^Save^param^message
```

The following example initializes the CRE without saving messages:

```
?ENV COMMON
NAME initialize_CRE;
?SEARCH $SYSTEM.SYSTEM.TALLIB
?SOURCE $SYSTEM.SYSTEM.TALDECS (TAL_CRE_INITIALIZER_)

PROC mymain MAIN;
 BEGIN
 CALL TAL_CRE_INITIALIZER_ !Initialize the CRE
 !Lots of code
 END;
```

The following example initializes the CRE and saves ASSIGN and PARAM messages but not the startup message:

```
?ENV COMMON
NAME initialize_CRE;
?SEARCH $SYSTEM.SYSTEM.TALLIB
?SOURCE $SYSTEM.SYSTEM.TALDECS (TAL_CRE_INITIALIZER_)
?SOURCE $SYSTEM.SYSTEM.CREDECS (INITIALIZATION)

PROC mymain MAIN;
 BEGIN
 CALL TAL_CRE_INITIALIZER_
 (CRE^save^assign^message LOR CRE^save^param^message);
 !Lots of code
 END;
```

Your routine can manipulate the saved messages by calling SMU functions, as described in the *CRE Programmer's Manual*.

**Terminating Programs** At the end of execution:

1. A TAL module must call `CRE_FILE_CLOSE_` for each file the TAL module has opened.
2. The TAL module then must call `CRE_TERMINATOR_` to clear the run-time environment (as described in the *CRE Programmer's Guide*).
3. `CRE_TERMINATOR_` calls a run-time library termination routine for each language in your program except TAL.
4. Each termination routine releases any resources and closes files used by routines written in that language as follows:
  - It closes open standard files by calling `CRE_FILE_CLOSE_`.
  - It closes all other files by calling `FILE_CLOSE_`.

Each termination routine then passes control to `CRE_TERMINATOR_`.

5. `CRE_TERMINATOR_` closes any remaining standard files, releases any system resources used by the CRE, and calls `PROCESS_STOP_`.
6. `PROCESS_STOP_` closes any remaining files left open by your program and returns control to the operating system.

**Sharing the Standard Files** To open and access standard input, output, and log files, a TAL routine must call CRE routines directly. The CRE provides the following routines for accessing the standard files:

```
CRE_FILE_CLOSE_
CRE_FILE_CONTROL_
CRE_FILE_INPUT_
CRE_FILE_MESSAGE_
CRE_FILE_OPEN_
CRE_FILE_OUTPUT_
CRE_FILE_RETRYCHECK_
CRE_FILE_SETMODE_
CRE_HOMETERM_OPEN_
CRE_LOG_MESSAGE_
CRE_SPOOL_START_
```

The preceding CRE routines are described in the *CRE Programmer's Guide*. Some examples are given in the following subsections.

### Opening a Standard File

To request an open to a standard file, a TAL routine calls `CRE_FILE_OPEN_`. For example, to request an open to standard log , specify:

```
CALL CRE_FILE_OPEN_ (cre^standard^log);
```

As another example, to request an open to the standard output file, specify:

```
CALL CRE_FILE_OPEN_ (cre^standard^output);
```

`CRE_FILE_OPEN_` does the following tasks:

- If the file is closed, `CRE_FILE_OPEN_` calls `FILE_OPEN_`.
- `FILE_OPEN_` opens the file and returns control to `CRE_FILE_OPEN_`.
- `CRE_FILE_OPEN_` grants the TAL routine a connection to the file open.
- For each subsequent open request, `CRE_FILE_OPEN_` grants a connection to the same file open.

### Writing a Message to Standard Files

To write a record to standard log , a TAL routine calls `CRE_LOG_MESSAGE_`. For example, to write the content of a 15-character array named `MSG` to standard log, you can specify:

```
CALL CRE_LOG_MESSAGE_ (msg:15);
```

To write a record to standard output , a TAL routine calls `CRE_FILE_OUTPUT_`. For example, to write the record "B" to standard output , you can specify:

```
STRING var;
var := "B";
CALL CRE_FILE_OUTPUT_ (cre^standard^output, var:1);
```

### Closing Standard Files

To close standard files, a TAL routine calls `CRE_FILE_CLOSE_`. For example, to close standard log, specify:

```
CALL CRE_FILE_CLOSE_ (cre^standard^log);
```

As another example, to close standard output, specify:

```
CALL CRE_FILE_CLOSE_ (cre^standard^output);
```

### Using Spooling

You can specify a spooler collector as the device for standard output or standard log. For standard output, the CRE uses buffered spooling unless you specify otherwise. For standard log, you cannot use buffered spooling.

To request an open to a spooler collector:

1. The TAL routine calls `CRE_FILE_OPEN_`.
2. If the spooler is closed, `CRE_FILE_OPEN_` sets a flag that `CRE_SPOOL_START_` has not been called.
3. If the flag in step 2 is set, other CRE standard-file routines (such as `CRE_FILE_OUTPUT_`) call `CRE_SPOOL_START_`, which clears the flag and starts the spooler with default settings.

To change the default settings, the TAL routine must call `CRE_SPOOL_START_` directly and specify the new settings, such as the setting for multiple copies.

4. `CRE_FILE_OPEN_` grants the TAL routine a connection to the spooler.
5. For each subsequent request, `CRE_FILE_OPEN_` grants a connection to the same spooler open.

**Accessing \$RECEIVE** \$RECEIVE is a special file through which a routine can receive messages from the operating system, from your backup process, or from another process. Routines written in any language supported by the CRE can read \$RECEIVE. The routines in a program can access \$RECEIVE by calling either CRE routines only or system procedures only.

**CRE Routines**

A program can use CRE routines to access \$RECEIVE if the program accesses \$RECEIVE from:

- TAL routines only
- COBOL routines only
- FORTRAN routines only
- Both COBOL and TAL routines
- Both FORTRAN and TAL routines

CRE Routines for accessing \$RECEIVE include:

| Action                           | CRE Routine                                   |
|----------------------------------|-----------------------------------------------|
| Request an open to \$RECEIVE     | CRE_RECEIVE_OPEN_CLOSE_ with the open variant |
| Read messages from \$RECEIVE     | CRE_RECEIVE_READ_                             |
| Reply to messages from \$RECEIVE | CRE_RECEIVE_WRITE_                            |

If \$RECEIVE is not open when the first request occurs, a CRE routine opens \$RECEIVE for exclusive access. For each subsequent request, the CRE routine grants a connection to the same file open. For more information on the CRE routines, see the *CRE Programmer's Guide*.

**System Procedures**

System procedures for accessing \$RECEIVE include:

| Action                                      | System Procedure           |
|---------------------------------------------|----------------------------|
| Request an open to \$RECEIVE                | FILE_OPEN_                 |
| Read, and reply to, messages from \$RECEIVE | READUPDATE[X] and REPLY[X] |

For more information on the system procedures, see the *Guardian Programmer's Guide* and the *Guardian Procedure Calls Reference Manual*.

**Handling Errors in CRE Math Routines** The CRE provides libraries of math routines, such as sine and cosine routines, that your program can call.

When a CRE math routine receives an invalid parameter or produces an invalid result, an arithmetic fault occurs when control returns to the caller. The caller's run-time library (except TAL) determines the program's behavior.

A TAL caller can determine the effect of the error by setting or resetting the trap-enable bit of the environment register. You should ensure that the trap-enable bit is appropriately set before your program calls a CRE math routine.

If traps are disabled when a CRE math routine detects an error, the system returns control to the caller:

```
REAL r, s;
r := -1.0E0;

CALL disable_overflow_traps; !A user-written routine that
s := RTL_SQRT_REAL32_(r); ! disables overflow traps
IF $OVERFLOW THEN !Control returns here; test
 BEGIN ! error in RTL_SQRT_REAL32_.
 CALL enable_overflow_traps; !Enable overflow traps
 !Lots of code
 END;
CALL enable_overflow_traps; !Enable overflow traps
```

If traps are enabled when a CRE math routine detects an error, the system returns control to the current trap handler:

```
REAL r, s;
r := -1.0E0;

CALL enable_overflow_traps; !A user-written routine that
s := RTL_SQRT_REAL32_(r); ! enables overflow traps
IF $OVERFLOW THEN !If RTL_SQRT_REAL32_ causes
 BEGIN ! overflow, the program does
 !Lots of code! ! not reach this statement
 END; ! because control transfers
 ! to the current trap handler
```

**The Extended Stack** The CRE supports the extended stack, which defines the following data blocks:

- \$EXTENDED#STACK
- EXTENDED#STACK#POINTERS

These data blocks are described in Section 4, "Introducing the Environment."

**CRE Sample Program** Example 17-2 shows the source code of a D-series program written for the CRE. This program contains a TAL routine that calls a C routine, each of which displays a greeting.

---

**Example 17-2. D-Series CRE Sample Program** (Page 1 of 2)

**TAL Code**

```
?SYMBOLS, INSPECT
?ENV COMMON

NAME talsrc;

!Source in CRE run-time library routines:
?PUSHLIST, NOLIST, SOURCE $SYSTEM.SYSTEM CREDECS (FILE)
?POPLIST
?PUSHLIST, NOLIST, SOURCE $SYSTEM.SYSTEM CREDECS (
?
TERMINATION)
?POPLIST
?PUSHLIST, NOLIST, SOURCE $SYSTEM.SYSTEM CREDECS (
?
CRE_FILE_CLOSE_,
?
CRE_FILE_OPEN_,
?
CRE_LOG_MESSAGE_,
?
CRE_TERMINATOR_)
?POPLIST

!Source in TAL run-time library routine:
?PUSHLIST, NOLIST, SOURCE $SYSTEM.SYSTEM TALDECS (
?
TAL_CRE_INITIALIZER_)
?POPLIST

PROC c_routine LANGUAGE C;
EXTERNAL; !Declare EXTERNAL C routine

PROC talmain MAIN;
BEGIN
STRING .msg[0:13] := "Hello from TAL";

CALL TAL_CRE_INITIALIZER_;
CALL CRE_FILE_OPEN_ (cre^standard^log);
CALL CRE_LOG_MESSAGE_ (msg:14);
CALL c_routine;
CALL CRE_FILE_CLOSE_ (cre^standard^log);
CALL CRE_TERMINATOR_ (cre^completion^normal);
END;
```

---

**Example 17-2. D-Series CRE Sample Program (Page 2 of 2)****C Code**

```
#pragma symbols, inspect
#include <stdio.h> nolist

/*file name: csrc */

void C_ROUTINE () /* C routine called by TAL */
{
 short result;
 short stderr = 2;
 short error = 0;

 result = fopen_std_file(stderr, error);
 if ((result == 0) || (result == 1))
 fprintf(stderr, "Hello from C\n");
}
```

---

After you compile the TAL and C source files, you must bind the object files and the TAL and C run-time libraries into a new object file. The resulting object file can then run in the CRE, if the object file meets the requirements of the CRE.

For example, to bind object files named TALOBJ and COBJ and run-time library files named TALLIB and CLARGE into a new object file named EXAMPLE, issue the following Binder commands:

```
CLEAR
ADD * FROM talobj
ADD * FROM cobj
SELECT SEARCH $SYSTEM.SYSTEM.TALLIB
SELECT SEARCH $SYSTEM.SYSTEM.CLARGE
SELECT LIST * OFF
BUILD example!
```

---

# Appendix A Sample Programs

---

This appendix includes the following examples:

- String-display sample program
- String-entry sample program
- Binary-to-ASCII conversion sample program
- Modular programming example

---

## String-Display Programs

Example A-1 shows the source code for a string-display program that displays "Hello, World" on the terminal.

---

### Example A-1. C- or D-Series String-Display Program

```
!Global data declarations:
INT .out_file_name[0:11];

?PUSHLIST, NOLIST, SOURCE $SYSTEM.SYSTEM.EXTDECS (CLOSE,
? INITIALIZER, OPEN, WRITE) !Include system procedures,
 ! but suppress their listings
?POPLIST !Resume listing

PROC startup_proc (rucb, passthru, message, msglen, match)
 VARIABLE; !Declare STARTUP_PROC
 INT .rucb, .passthru, .message, msglen, match;
 BEGIN
 out_file_name := message[21] FOR 12 WORDS;
 !Move statement
 END; !End STARTUP_PROC

PROC myproc MAIN; !Declare MYPROC
 BEGIN
 INT out_file_number;
 STRING .EXT buffer[0:79]; !Array for output message
 INT length; !Length of output message

 CALL INITIALIZER (! rucb !, ! passthru !, startup_proc,
 ! paramsproc !, ! assignproc !, ! flags !);
 !Get OUT file name
 CALL OPEN (out_file_name , out_file_number);
 !Open OUT file; get number
 buffer := "Hello, World"; !Move statement
 length := 12; !Assignment statement
 CALL WRITE (out_file_number, buffer, length);
 !Write message to OUT file
 CALL CLOSE (out_file_number); !Close OUT file
 END; !End MYPROC
```

---

Example A-2 shows a C-series version of the previous string-display program.

---

**Example A-2. C-Series String-Display Program)**

```
?NOLIST, SOURCE $SYSTEM.SYSTEM.EXTDECS (CLOSE, INITIALIZER,
? MYTERM, OPEN, WRITE) !Include system procedures,
 ! but suppress their listings
?LIST !Resume listing

PROC myproc MAIN; !Declare MYPROC procedure
 BEGIN
 !Data declarations
 INT termname[0:11]; !Terminal name
 INT filename; !File number for terminal
 STRING buffer[0:79]; !Array for output message
 INT length; !Variable for message length

 CALL INITIALIZER; !Process startup
 ! initialization
 CALL MYTERM (termname); !Get terminal name
 CALL OPEN (termname, filenum); !Open terminal; get number
 buffer := "Hello, World"; !Move statement
 length := 12; !Assignment statement
 CALL WRITE (filenum, buffer, length);
 !Write message to terminal)
 CALL CLOSE (filenum); !Close terminal
 END; !End MYPROC
```

---

**String-Entry Program** This string-entry program opens the home terminal and then loops forever. In each loop iteration, the program:

1. Displays "ENTER STRING" and accepts a character string of up to 68 characters.
2. Scans the input string for an asterisk. If an asterisk occurs, the program displays a circumflex at the position of the first asterisk.

Example A-3 shows the D-series source code for the string-entry program.

### Example A-3. D-Series String-Entry Program

```

LITERAL maxlength = 68; !Maximum length of BUFFER
INT termnum, !File number of home terminal
 left_side, !BUFFER address of first
 ! character after prompt
 num_xferred, !Number of bytes transferred
 count, !General-purpose variable
 asterisk; !Location of asterisk
STRING .buffer[0:maxlength], !Input-output buffer
 .blanks[0:79] := 80 * [" "]; !Blanks for initialization
?PUSHLIST, NOLIST, SOURCE $SYSTEM.SYSTEM.EXTDECS0 (
? PROCESS_GETINFO_, FILE_OPEN_, WRITEREADX, WRITE(X)
?POPLIST

PROC main_proc MAIN; !Declare MAIN_PROC
BEGIN
CALL PROCESS_GETINFO_ (
 , , , , ,buffer:maxlength+1, num_xferred);
CALL FILE_OPEN_(buffer:num_xferred, termnum);
WHILE 1 DO !Infinite loop
BEGIN
buffer := "ENTER STRING" -> left_side;
CALL WRITEREADX (termnum, buffer, left_side '-' @buffer,
 maxlength, num_xferred);
buffer[num_xferred] := 0; !Delimit the input
SCAN buffer UNTIL "*" -> asterisk;
 !Scan for asterisk
IF NOT $CARRY THEN !Asterisk found
BEGIN
buffer := blanks FOR
 (count := asterisk '-' @buffer +
 (left_side '-' @buffer)) BYTES;
buffer[count] := "^";
CALL WRITE(X) (termnum, buffer, count+1);
END; !End of IF
END; !End of WHILE
END; !End of MAIN_PROC

```

Example A-4 shows the C-series source code for the string-entry program.

---

#### Example A-4. C-Series String-Entry Program

```

LITERAL maxlength = 68; !Maximum length of BUFFER
INT termnum, !File number of home terminal
 left_side, !BUFFER address of first
 ! character after prompt
 num_xferred, !Number of bytes transferred
 count, !General-purpose variable
 asterisk; !Location of asterisk
INT .ibuffer[0:maxlength/2];
STRING .buffer := @ibuffer '<<' 1, !Input-output buffer
 .blanks[0:79] := 80 * [" "]; !Blanks for initialization
?PUSHLIST, NOLIST, SOURCE $SYSTEM.SYSTEM.EXTDECS0 (
? MYTERM, OPEN, WRITEREADX, WRITE(X)
?POPLIST

PROC main_proc MAIN; !Declare MAIN_PROC
BEGIN
CALL MYTERM (ibuffer);
CALL OPEN (ibuffer, termnum);
WHILE 1 DO !Infinite loop
BEGIN
buffer := "ENTER STRING" -> left_side;
CALL WRITEREADX (termnum, buffer, left_side '-' @buffer,
 maxlength, num_xferred);
buffer[num_xferred] := 0; !Delimit the input
SCAN buffer UNTIL "*" -> asterisk;
 !Scan for asterisk
IF NOT $CARRY THEN !Asterisk found
BEGIN
buffer := blanks FOR
 (count := asterisk '-' @buffer +
 (left_side '-' @buffer)) BYTES;
buffer[count] := "^";
CALL WRITE(X) (termnum, buffer, count+1);
END; !End of IF
END; !End of WHILE
END; !End of MAIN_PROC

```

---

---

**Binary-to-ASCII  
Conversion Program**

This binary-to-ASCII conversion program performs a conversion function typical of many algorithms in TAL. The program converts a binary INT value to an ASCII (base 10) value with a maximum length of six characters including the sign, then returns the converted character string and its length to the calling procedure.

Significant items in Example A-5 are keyed to the following discussion:

**Item Discussion**

- !1! Comments preceding the procedure declaration describe the purpose of the procedure. For complex procedures, you can also summarize the input-output characteristics and the main features of the algorithm.
- !2! The formal parameter specifications define the parameters of the procedure. Input parameters V and RJUST are value parameters, and output parameter STG is a reference parameter.
- !3! This local declaration reserves six bytes of memory for the buffer in which the number is converted. The declaration also initializes the first five bytes in the buffer to blanks (using a repetition factor of 5) and sets the last byte to an ASCII 0. Thus, an input of 0 results in an output of five blanks and a 0, rather than six blank characters.
- !4! This IF statement deals with a negative INT binary number. When it encounters a negative number, it sets the negative value flag to 1 and takes the absolute value of the number passed.
- !5! This WHILE loop performs the conversion, character by character, writing each byte to the buffer from right to left.
- !6! This assignment statement converts the binary INT value to an ASCII (base 10) value. It illustrates an arithmetic expression that uses the standard procedure \$UDBL. The statement performs a residue modulo 10 operation, then adds an ASCII 0 to the value of each byte to fit in the numeric range.
- !7! This IF statement uses the assignment form of an arithmetic expression as the condition.
- !8! This IF statement moves the resulting character string from the buffer into the user's target string.
- !9! The RETURN statement returns to the calling procedure the number of characters moved.

**Example A-5. C- or D-Series Binary-to-ASCII Conversion Program**

```

!1! !INT PROC ASCII converts a binary INT value to an ASCII
! (base 10) value of up to six characters (including the
! sign) and returns the ASCII value and its length.

INT PROC ascii (v, rjust, stg);
!2! INT v; !INT value to convert
 INT rjust; !Right justify result flag
 STRING .stg; !Target string
BEGIN
!3! STRING .b[0:5] := [5*[" "], "0"];
 INT n; !String length
 INT sgn := 0; !Nonzero if V is negative
 INT k := 5; !Index for converted digit

!4! IF v < 0 THEN !Value is negative
 BEGIN
 sgn := 1; !Set negative value flag
 v := -v; !Take absolute value
 END;

!5! WHILE v DO !While a value is left . . .
 BEGIN ! (equivalent to V <> 0)
!6! b[k] := $UDBL(v) '\ ' 10 + "0";
 !Convert a character
 v := v / 10; !Compute remainder
 k := k - 1; !Count converted character
 END;

 IF sgn THEN !Number is negative
 BEGIN
 b[k] := "-"; !Insert the sign
 k := k - 1; !Count it as a character
 END;

!7! IF NOT (n:=5-k) THEN !Check for an underflow
 n := 1; !Return 1 character in that case

!8! IF rjust THEN !Move string to target
 stg[n-1] := b[5] FOR n BYTES
 !Reverse move if right justified
 ELSE
 stg := b[6-n] FOR n BYTES;
 !Otherwise forward move

!9! RETURN n; !Return string's length
 END !ascii! ;

```

## Modular Programming Example

This modular programming example illustrates how you can divide the code for a program into separately compilable modules. The program converts records in the input file to a different format and length by reordering fields and adding fields to records. The example includes:

- A brief description of program characteristics
- Partial listings of module code
- Load maps for the program file
- Compilation statistics (compile and bind) for the program file

Selected listings show the handling of data and program structure. The content of global data blocks appear only in the module that declares them. In modules that reference such blocks, NOLIST prevents the listing of the content of the blocks.

Compilation maps and statistics are not shown for each module. Load maps show global-data-block entries that do not exist after compilation, such as LITERALS. The mainline load map does not refer to these blocks.

## Modular Structure

The program consists of five modules, shown in Examples A-6a through A-6e. Each module performs a single operation. The structure of the modules and their procedures allows changes to one operation without the need to recompile the others.

Information is accessible across modules on an as-needed basis. They share named global data blocks and pass information as parameters and local data such as a simple pointer to the locally declared record buffer. The named global data block DEFAULT\_VOL contains shared run-time data. Other named blocks declare structure templates for record definitions and LITERAL declarations, which use no memory.

Procedures within a module share global data in private blocks.

The following table summarizes the blocks used by each module. (The symbol (P) denotes a private block.)

| Module Name           | Blocks Defined                     | Blocks Referenced           |
|-----------------------|------------------------------------|-----------------------------|
| TPR_CONVERT           | RECORD_DEFS                        | MSG_LITERALS                |
| INITIALIZATION_MODULE | DEFAULT_VOL                        |                             |
| MESSAGE_MODULE        | MSG_LITERALS<br>MESSAGE_MODULE (P) | DEFAULT_VOL                 |
| IN_FILE_HANDLER       | IN_DATA<br>IN_FILE_HANDLER (P)     | MSG_LITERALS<br>DEFAULT_VOL |
| OUT_FILE_HANDLER      | OUT_DATA<br>OUT_FILE_HANDLER (P)   | DEFAULT_VOL<br>MSG_LITERALS |

**File-Naming Conventions** This modular program follows these file-naming conventions:

- Source file names end with the character S.
- Object file names correspond to source file names and end with O. For instance, the object file built from the source file INS is named INO.
- Section names ending with D belong to a specific module. For instance, IND is the section that contains LITERAL declarations for INS. Other section names are used to provide a direct means for other source programs to copy individual procedures when they need them. The section names beginning with END\_OF\_ are used solely to mark the ends of actual sections.
- File names ending with P each contains external declarations of the procedures in the module with the corresponding identifier. A module that calls an external procedure includes a SOURCE directive for the P file. For instance, the source for MESSAGE\_MODULE is file MSGS, and source file MSGP declares each external procedure in MSGS. The modules that call MESSAGE\_MODULE specify MSGP in a SOURCE directive.

If any external declarations change, you must recompile both the P file and any module that calls a changed external procedure. The P file enables compile-time consistency checking between procedure declarations and the corresponding external declarations.

A module also uses a P file for its external procedure declarations. Module xxxS uses a SOURCE directive to specify xxxP, which contains external declarations for its procedures. (Otherwise, the consistency check is possible only during a later binding.)

**Compiling and Binding the Modular Program** To compile each source file into an object file, you can issue the following compilation commands. You need do all these steps only once:

```
TAL /IN inits/ inito
TAL /IN ins/ ino
TAL /IN outs/ outo
TAL /IN msgs/ msgo
TAL /IN converts/ converto
```

To bind the modular object files into a single object file, you can issue the following Binder commands. In this example, BINDS is the Binder IN file and the last five lines represent the content of BINDS:

```
BIND / IN binds /

ADD * FROM converto
SELECT SEARCH (ino, outo, msgo, inito)
SELECT SEARCH $SYSTEM.SYSTEM.TALLIB
BUILD convert !
EXIT
```

You can then update the code or private data in a module such as INITS by issuing the following commands:

```
TAL /IN inits/ inito
BIND / IN binds /
```

When you update a shared data block in a module, you must recompile all modules that reference the updated data block. Similarly, when you update the declarations of a shared procedure, you must recompile all modules that reference the updated procedure.

**Source Modules** The following source modules for the modular program illustrate how you can break a program into manageable modules. The source code for these modules are shown in the remainder of this section:

| Example                     | Module Name           | Source File Name |
|-----------------------------|-----------------------|------------------|
| A-6a. Mainline Module       | TPR_CONVERT           | CONVERTS         |
| A-6b. Initialization Module | INITIALIZATION_MODULE | INITS            |
| A-6c. Input File Module     | IN_FILE_HANDLER       | INS              |
| A-6d. Output File Module    | OUT_FILE_HANDLER      | OUTS             |
| A-6e. Message Module        | MESSAGE_MODULE        | MSGS             |

**Mainline Module**

Example A-6a shows the mainline module, which contains the MAIN procedure. The record-definition structures are not listed because they are translations of the Data Definition Language (DDL) source code into TAL.

**Example A-6a. D-Series Mainline Module (Page 1 of 2)**

```

!File name CONVERTS
NAME tpr_convert;

?PUSHLIST, NOLIST, SOURCE recdefs !BLOCK RECORD_DEFS
?POPLIST
?PUSHLIST, NOLIST, SOURCE msgs (msglit)
?POPLIST !BLOCK MSG_LITERALS
 !Following are external procedure declarations:
?PUSHLIST, NOLIST, SOURCE $SYSTEM.SYSTEM.EXTDECS (
?
 PROCESS_STOP_)
?POPLIST
?PUSHLIST, NOLIST, SOURCE inp !IN_FILE_HANDLER
?POPLIST
?PUSHLIST, NOLIST, SOURCE outp !OUT_FILE_HANDLER
?POPLIST
?PUSHLIST, NOLIST, SOURCE msgp !MESSAGE_MODULE
?POPLIST
?PUSHLIST, NOLIST, SOURCE initp !INITIALIZATION_MODULE
?POPLIST

?SECTION out_init
PROC out_init (out_rec:out_rec_len); !Initialize
 STRING .EXT out_rec (out_rec_def); ! output record
 INT out_rec_len;
 BEGIN
 IF out_rec_len '<>' 0 THEN
 out_rec :=' [" "] & out_rec FOR out_rec_len '-' 1 BYTES;
 END;

?SECTION record_convert
PROC record_convert (in_rec, out_rec);
 STRUCT .EXT in_rec (in_rec_def); !Convert between
 STRUCT .EXT out_rec (out_rec_def); ! two records
 BEGIN
 INT i;
 STRING .EXT ch_ptr;
 i := 0;
 @ch_ptr := @in_rec.name; !Copy last name

```

**Example A-6a. D-Series Mainline Module (Page 2 of 2)**

```

WHILE (i < $OCCURS(in_rec.name)) AND (ch_ptr <> ",") DO
 BEGIN
 @ch_ptr := @ch_ptr[1];
 i := i + 1;
 END;
out_rec.name.last_name := in_rec.name FOR i BYTES;

IF ch_ptr = ',' THEN !Copy first name
 BEGIN
 @ch_ptr := @ch_ptr[1]; !Advance past comma
 out_rec.name.first_name := ch_ptr FOR
 $OCCURS(in_rec.name) '-' i '-' 1 BYTES;
 END;
out_rec.address := in_rec.address !Copy address
 FOR $OCCURS(in_rec.address) BYTES;
END;

?SECTION convert
PROC convert;
 BEGIN
 INT record_count := 0;
 STRUCT .in_buffer (in_rec_def);
 STRUCT .out_buffer (out_rec_def);
 WHILE (read_in (in_buffer:$LEN(in_rec_def))) <> 1 !EOF! DO
 BEGIN !Read record,
 ! return EOF
 CALL out_init (out_buffer:$LEN(out_rec_def));
 !Initialize output
 CALL record_convert (in_buffer, out_buffer);
 CALL write_out (out_buffer:$LEN(out_rec_def));
 record_count := record_count + 1;
 END; !Of WHILE loop
 !EOF
 CALL msg (msg_eof, record_count);
 END; !Of CONVERT

?SECTION end_of_code_sections
PROC tprconv MAIN;
 BEGIN
 CALL file_init; !In INITIALIZATION_MODULE
 CALL convert;
 CALL close_all;
 END;
?NOMAP

```

**Initialization Module**

Example A-6b shows the source code for the initialization module. This module defines a primary global data block, DEFAULT\_VOL, which is accessible to all procedures in the modules that declare the block for reference.

**Example A-6b. D-Series Initialization Module (Page 1 of 2)**

```

!File name INITS
NAME initialization_module;

?SECTION default
BLOCK default_vol; !Default volume, subvolume
 LITERAL file_name_max_len = 256;
 INT def_vol_subvol_len;
 STRING .def_vol_subvol[0:file_name_max_len - 1];
 END BLOCK;
?SECTION end_of_data_sections

 !Following are external procedure declarations:
?PUSHLIST, NOLIST, SOURCE $SYSTEM.SYSTEM.EXTDECS (
? OLDFILENAME_TO_FILENAME_)
?POPLIST
?PUSHLIST, NOLIST, SOURCE $SYSTEM.SYSTEM.EXTDECS (
? INITIALIZER)
?POPLIST
?PUSHLIST, NOLIST, SOURCE outp !OUT_FILE_HANDLER
?POPLIST
?PUSHLIST, NOLIST, SOURCE inp !IN_FILE_HANDLER
?POPLIST
?PUSHLIST, NOLIST, SOURCE msgp !MESSAGE_MODULE
?POPLIST
?PUSHLIST, NOLIST, SOURCE initp !INITIALIZATION_MODULE
?POPLIST ! (for consistency checks)

```

---

**Example A-6b. D-Series Initialization Module (Page 2 of 2)**

```
?SECTION startup
PROC startup (rucb, passthru, message, meslen, match)
 VARIABLE;
 INT .rucb, .passthru, .message, meslen, match;
 BEGIN
 INT .def_vol_subvol_internal_fmt[0:11] := [12 * [" "]];
 def_vol_subvol_internal_fmt := message[1] FOR 8 WORDS;
 CALL OLDFILENAME_TO_FILENAME_(def_vol_subvol_internal_fmt,
 def_vol_subvol:file_name_max_len,
 def_vol_subvol:len);

 END;

?SECTION file_init
PROC file_init;
 BEGIN
 CALL INITIALIZER (, , startup);
 CALL msg_init;
 CALL in_file_init;
 CALL out_file_init;
 END;

?SECTION close_all
PROC close_all;
 BEGIN
 CALL in_close;
 CALL out_close;
 CALL msg_close;
 END;

?SECTION end_of_code_sections
?NOMAP
```

---

### Input File Module

Example A-6c shows the source code for the input file module, which contains all procedures that manipulate the input file. If you make I/O changes, only this module needs to be recompiled. The initialization module, for example, calls a procedure in this module. This module declares a private global data block, which is accessible only to the procedures in this module.

---

#### Example A-6c. D-Series Input File Module (Page 1 of 3)

```

!File name INS
NAME in_file_handler;

?PUSHLIST, NOLIST, SOURCE recdefs !BLOCK RECORD_DEFS
?POPLIST
?PUSHLIST, NOLIST, SOURCE inits (default) !BLOCK DEFAULT_VOL
?POPLIST
?PUSHLIST, NOLIST, SOURCE msgslit !BLOCK MSG_LITERALS
?POPLIST

BLOCK PRIVATE;
 INT in_file; !Input file number
END BLOCK;

!Following are external procedure declarations:
?PUSHLIST, NOLIST, SOURCE $SYSTEM.SYSTEM.EXTDECS (
?
 FILE_CLOSE_, FILE_GETINFO_)
?POPLIST
?PUSHLIST, NOLIST, SOURCE $SYSTEM.SYSTEM.EXTDECS (
?
 FILE_OPEN_, FILENAME_RESOLVE_)
?POPLIST
?PUSHLIST, NOLIST, SOURCE $SYSTEM.SYSTEM.EXTDECS (
?
 PROCESS_STOP_, READX)
?POPLIST
?PUSHLIST, NOLIST, SOURCE msgp ! MESSAGE_MODULE
?POPLIST
?PUSHLIST, NOLIST, SOURCE inp ! IN_FILE_HANDLER
?POPLIST
 ! (consistency checks)

```

---

**Example A-6c. D-Series Input File Module (Page 2 of 3)**

```

?SECTION in_file_init
PROC in_file_init;
BEGIN
 INT in_file_name_len := 6;
 STRING .in_file_name[0:file_name_max_len - 1] :=
 ["INFILE"];

 INT status;
 status := FILENAME_RESOLVE_
 (in_file_name:in_file_name_len,
 in_file_name:file_name_max_len,
 in_file_name_len,
 !options!,
 !override_name:override_name_len!,
 !search:search_len!,
 def_vol_subvol:def_vol_subvol_len);
 IF status = 0 !OK! THEN
 BEGIN
 status := FILE_OPEN_(in_file_name:in_file_name_len,
 in_file);
 IF in_file = -1 !unable to open file! THEN
 BEGIN
 CALL msg (msg_in_open, status);
 CALL PROCESS_STOP_ (!phandle!,
 !specifier!,
 !options!,
 3 !Completion code ABEND!,
 !...!);

 END;
 END !STATUS = 0
 ELSE
 BEGIN
 CALL msg (msg_in_name, 0);
 CALL PROCESS_STOP_ (!phandle!,
 !specifier!,
 !options!,
 3 !Completion code ABEND!,
 !...!);

 END;
 END !STATUS <> 0
 END;
 !Of procedure IN_FILE_INIT

```

**Example A-6c. D-Series Input File Module (Page 3 of 3)**

---

```
?SECTION read_in
INT PROC read_in (rec:rec_len);
 STRING .EXT rec;
 INT rec_len;
BEGIN
 INT error;

 CALL READX (in_file, rec, rec_len);
 IF < THEN
 BEGIN
 error := FILE_GETINFO_ (in_file);
 CALL msg (msg_read, error);
 CALL PROCESS_STOP_ (!phandle!,
 !specifier!,
 !options!,
 3 !Completion code ABEND!,
 !...!);
 END
 ELSE
 IF > THEN RETURN 1
 ELSE
 RETURN 0;

?SECTION in_close
PROC in_close;
BEGIN
 CALL FILE_CLOSE_ (in_file);
END;
?SECTION end_of_code_sections
?NOMAP
```

---

**Output File Module**

Example A-6d shows the source code for the output file module. This module declares a private global data block, which is accessible only to procedures in this module. Some code that is parallel to code in the input file handler is not listed.

**Example A-6d. D-Series Output File Module (Page 1 of 3)**

```

!File name OUTS
NAME out_file_handler;

?PUSHLIST, NOLIST, SOURCE recdefs !BLOCK RECORD_DEFS
?POPLIST
?PUSHLIST, NOLIST, SOURCE inits (default) !BLOCK DEFAULT_VOL
?POPLIST
?PUSHLIST, NOLIST, SOURCE msgs (msglit) !BLOCK MSG_LITERALS
?POPLIST

BLOCK PRIVATE;
 INT out_file;
END BLOCK;

!Following are external procedure declarations
?PUSHLIST, NOLIST, SOURCE $SYSTEM.SYSTEM.EXTDECS (
?
 FILE_CLOSE_, FILE_GETINFO_)
?POPLIST
?PUSHLIST, NOLIST, SOURCE $SYSTEM.SYSTEM.EXTDECS (
?
 FILE_OPEN_, FILENAME_RESOLVE_)
?POPLIST
?PUSHLIST, NOLIST, SOURCE $SYSTEM.SYSTEM.EXTDECS (
?
 PROCESS_STOP_, WRITEX)
?POPLIST
?PUSHLIST, NOLIST, SOURCE msgp
?POPLIST

?SECTION out_file_init
PROC out_file_init;
 BEGIN
 INT out_file_name_len := 7;
 STRING .out_file_name[0:file_name_max_len - 1]
 := ["OUTFILE"];

 INT status;
 status := FILENAME_RESOLVE_
 (out_file_name:out_file_name_len,
 out_file_name:file_name_max_len,
 out_file_name_len,
 !options!,
 !override_name:override_name_len!,
 !search:search_len!,
 def_vol_subvol:def_vol_subvol_len);

```

**Example A-6d. D-Series Output File Module (Page 2 of 3)**

---

```
IF status = 0 !FEOK! THEN
 BEGIN
 status := FILE_OPEN_ (out_file_name:out_file_name_len,
 out_file);
 IF out_file = -1 !unable to open file! THEN
 BEGIN
 CALL msg (msg_out_open, status);
 CALL PROCESS_STOP_ (!phandle!,
 !specifier!,
 !options!,
 3 !Completion code ABEND!,
 !...!);
 END;
 END !Of THEN clause
 ELSE
 BEGIN
 CALL msg (msg_out_name, 0);
 CALL PROCESS_STOP_ (!phandle!,
 !specifier!,
 !options!,
 3 !Completion code ABEND!,
 !...!);
 END; !Of ELSE clause
 END;

?SECTION write_out
PROC write_out (rec:rec_len);
 STRING .EXT rec;
 INT rec_len;
 BEGIN
 INT error;
 CALL WRITEX (out_file, rec, rec_len);
 IF < THEN
 BEGIN
 error := FILE_GETINFO_ (out_file);
 CALL msg (msg_write, error);
 CALL PROCESS_STOP_ (!phandle!,
 !specifier!,
 !options!,
 3 !Completion code ABEND!,
 !...!);
 END;
 END;
 END; !Of WRITE_OUT
```

---

**Example A-6d. D-Series Output File Module (Page 3 of 3)**

```

?SECTION out_close
PROC out_close;
 BEGIN
 CALL FILE_CLOSE_ (out_file);
 END;
?SECTION end_of_code_sections
?NOMAP

```

**Message Module**

Example A-6e shows the message module listing. The terminal number in the private global data block is accessible only to procedures in this module.

**Example A-6e. D-Series Message Module (Page 1 of 3)**

```

!File name MSGS
NAME message_module;

?SECTION msglit !Define BLOCK MSG_LITERALS
BLOCK msg_literals;
 LITERAL
 msg_eof = 0,
 msg_in_open = 1,
 msg_in_name = 2,
 msg_read = 3,
 msg_out_open = 4,
 msg_out_name = 5,
 msg_write = 6;
 END BLOCK;
?SECTION end_of_data_sections

BLOCK PRIVATE;
 LITERAL term_name_max_len = 256;
 INT term_file_number;
 LITERAL msg_buf_end = 79;
 END BLOCK;

!Following are external procedure declarations
?PUSHLIST, NOLIST, SOURCE $SYSTEM.SYSTEM.EXTDECS (
?
 FILE_CLOSE_, FILE_OPEN_)
?POPLIST
?PUSHLIST, NOLIST, SOURCE $SYSTEM.SYSTEM.EXTDECS (
?
 NUMOUT, PROCESS_GETINFO_)
?POPLIST
?PUSHLIST, NOLIST, SOURCE $SYSTEM.SYSTEM.EXTDECS (WRITEX)
?POPLIST

```

---

**Example A-6e. D-Series Message Module (Page 2 of 3)**

```

?SECTION msg_init
PROC msg_init;
BEGIN
 INT term_name_len;
 STRING .term_name[0:term_name_max_len - 1];
 CALL PROCESS_GETINFO(!process_handle!,
 !file_name:max_len!,
 !file_name_len!,
 !priority!,
 !moms_phandle!,
 term_name:term_name_max_len,
 term_name_len,
 !...!);
 CALL FILE_OPEN_(term_name:term_name_len, term_file_number);
END;

?SECTION msg
PROC msg (number, altnum);
 INT number, altnum;
 BEGIN
 STRING .buffer[0:msg_buf_end];
 STRING .bufptr := @buffer;
 CASE number OF
 BEGIN
 msg_eof ->
 buffer ':=' " *** End of File " -> @bufptr;
 msg_in_open ->
 buffer ':=' " *** In file open failed " -> @bufptr;
 msg_in_name ->
 buffer ':=' " *** Bad in file name " -> @bufptr;
 msg_read ->
 buffer ':=' " *** Read error " -> @bufptr;
 msg_out_open ->
 buffer ':=' " *** Out file open failed "-> @bufptr;
 msg_out_name ->
 buffer ':=' " *** Bad out file name " -> @bufptr;
 msg_write ->
 buffer ':=' " *** Write error " -> @bufptr;
 OTHERWISE ->
 ;
 END;

 IF altnum <> 0 THEN
 BEGIN
 CALL NUMOUT (bufptr, altnum, 10, 5);
 @bufptr := @bufptr + 5;
 END;

```

---

## Example A-6e. D-Series Message Module (Page 3 of 3)

```

CALL WRITEX (term_file_number, buffer,
 @bufptr '-' @buffer);
END; !Of MSG

?SECTION msg_close
PROC msg_close;
BEGIN
CALL FILE_CLOSE_ (term_file_number);
END;
?SECTION end_of_data_sections
?NOMAP

```

Compilation Maps  
and Statistics

The following compilation maps and statistics are shown for the preceding modular programming example:

- Entry-point load map
- Data-block load map
- Statistics for the mainline compilation

## Entry-Point Load Map

Figure A-1 shows the entry-point load map for the modular programming example:

Figure A-1. Entry-Point Load Map for Modular Program

## ENTRY POINT MAP BY NAME

| SP | PEP | Base   | Limit  | Entry  | Attr | Name           | Date       | Time  | Lang | Source File            |
|----|-----|--------|--------|--------|------|----------------|------------|-------|------|------------------------|
| 00 | 012 | 000737 | 000742 | 000737 |      | CLOSE_ALL      | 1992-04-13 | 13:54 | TAL  | \XX.\$VOL.PRG.INITS    |
| 00 | 004 | 000266 | 000331 | 000266 |      | CONVERT        | 1992-04-13 | 13:54 | TAL  | \XX.\$VOL.PRG.CONVERTS |
| 00 | 011 | 000721 | 000736 | 000721 |      | FILE_INIT      | 1992-04-13 | 13:54 | TAL  | \XX.\$VOL.PRG.INITS    |
| 00 | 017 | 001133 | 001140 | 001133 |      | IN_CLOSE       | 1992-04-13 | 12:59 | TAL  | \XX.\$VOL.PRG.INS      |
| 00 | 015 | 000767 | 001050 | 000773 |      | IN_FILE_INIT   | 1992-04-13 | 12:59 | TAL  | \XX.\$VOL.PRG.INS      |
| 00 | 010 | 000421 | 000720 | 000421 |      | MSG            | 1992-04-13 | 12:59 | TAL  | \XX.\$VOL.PRG.MSGS     |
| 00 | 021 | 001147 | 001154 | 001147 |      | MSG_CLOSE      | 1992-04-13 | 12:59 | TAL  | \XX.\$VOL.PRG.MSGS     |
| 00 | 014 | 000751 | 000766 | 000751 |      | MSG_INIT       | 1992-04-13 | 12:59 | TAL  | \XX.\$VOL.PRG.MSGS     |
| 00 | 020 | 001141 | 001146 | 001141 |      | OUT_CLOSE      | 1992-04-13 | 12:59 | TAL  | \XX.\$VOL.PRG.OUTS     |
| 00 | 016 | 001051 | 001132 | 001055 |      | OUT_FILE_INIT  | 1992-04-13 | 12:59 | TAL  | \XX.\$VOL.PRG.OUTS     |
| 00 | 002 | 000022 | 000111 | 000022 |      | OUT_INIT       | 1992-04-13 | 13:54 | TAL  | \XX.\$VOL.PRG.CONVERTS |
| 00 | 006 | 000340 | 000372 | 000340 |      | READ_IN        | 1992-04-13 | 12:59 | TAL  | \XX.\$VOL.PRG.INS      |
| 00 | 003 | 000122 | 000265 | 000122 |      | RECORD_CONVERT | 1992-04-13 | 13:54 | TAL  | \XX.\$VOL.PRG.CONVERTS |
| 00 | 013 | 000743 | 000750 | 000743 | V    | STARTUP        | 1992-04-13 | 13:52 | TAL  | \XX.PRG.INITS          |
| 00 | 005 | 000332 | 000337 | 000332 | M    | TPRCONV        | 1992-04-13 | 13:54 | TAL  | \XX.\$VOL.PRG.CONVERTS |
| 00 | 007 | 000373 | 000420 | 000373 |      | WRITE_OUT      | 1992-04-13 | 12:59 | TAL  | \XX.\$VOL.PRG.OUTS     |

### Data-Block Load Map

Figure A-2 shows the data-block load map for the modular programming example:

**Figure A-2. Data-Block Load Map for Modular Program**

| DATA BLOCK MAP BY NAME |        |        |      |                  |            |       |      |                     |
|------------------------|--------|--------|------|------------------|------------|-------|------|---------------------|
| Base                   | Limit  | Type   | Mode | Name             | Date       | Time  | Lang | Source File         |
| 000005                 | 000204 | COMMON | WORD | ..DEFAULT_VOL    | 1992-04-13 | 12:59 | TAL  | \XX.\$VOL.PRG.INITS |
| 000003                 | 000012 | COMMON | WORD | DEFAULT_VOL      | 1992-04-13 | 12:59 | TAL  | \XX.\$VOL.PRG.INITS |
| 000000                 | 000000 | COMMON | WORD | IN_FILE_HANDLER  | 1992-04-13 | 12:59 | TAL  | \XX.\$VOL.PRG.INS   |
| 000002                 | 000002 | COMMON | WORD | MESSAGE_MODULE   | 1992-04-13 | 12:58 | TAL  | \XX.\$VOL.PRG.MSGS  |
| 000000                 |        | COMMON | WORD | MSG_LITERALS     | 1992-04-13 | 12:58 | TAL  | \XX.\$VOL.PRG.INP   |
| 000001                 | 000001 | COMMON | WORD | OUT_FILE_HANDLER | 1992-04-13 | 12:59 | TAL  | \XX.\$VOL.PRG.OUTS  |
| 000000                 |        | COMMON | WORD | RECORD_DEFS      | 1992-04-13 | 12:59 | TAL  | \XX.\$VOL.PRG.MSGS  |

### Mainline Compilation Statistics

Figure A-3 shows the mainline compilation statistics for the mainline module of the modular programming example:

**Figure A-3. Compilation Statistics for Mainline Module**

```
BINDER - OBJECT FILE BINDER - T9621D20 - (01JUN93) SYSTEM \XX
Copyright Tandem Computers Incorporated 1982-1993
```

```
Object file \XX.$VOL.PRG.CONVERTO
TIMESTAMP 1993-04-13 17:44:49
```

```
1 Code page
1 Data page
0 Resident code pages
0 Extended data pages

0 Top of stack location in words
1 Code segment

0 Binder Warnings
0 Binder Errors
```

```
TAL - Transaction Application Language - T9250D20 - (01JUN93)
Number of compiler errors = 0
Number of unsuppressed compiler warnings = 0
Number of warnings suppressed by NOWARN = 0
Maximum symbol table space used was = 19944 bytes
Number of source lines = 5646
Compile cpu time - 00:00:01
Total elapsed time - 00:00:19
```

# Appendix B Managing Addressing

This appendix discusses:

- Extended pointer format
- Accessing the upper 32K-word area of the user data segment
- Creating and accessing an extended data segment
- Accessing a code segment

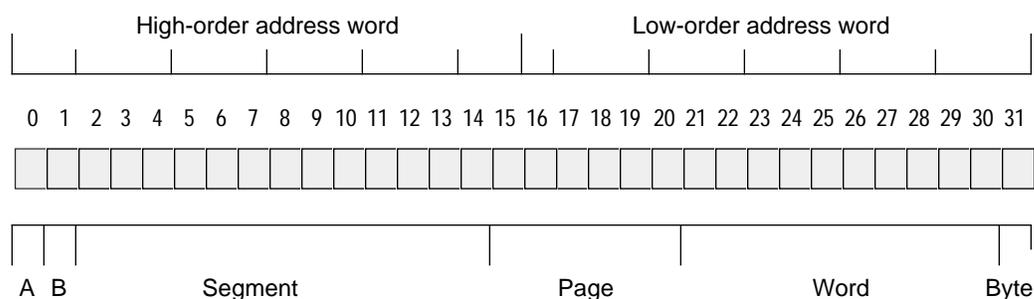
The coding practices covered in this appendix predate automatic extended segment management and are described here in support of existing programs.

## Extended Pointer Format

You can use an extended pointer to access the current user data segment, the system data segment, or the current user code segment. You must use an extended pointer to access an extended data segment.

Figure B-1 shows the format of extended pointers.

Figure B-1. Format of Extended Pointer



335

Table B-1 explains the meanings of the bits in shown in Figure B-1.

Table B-1. Extended Pointer Format

| Bits    | Meaning                     | Values                                                                   |
|---------|-----------------------------|--------------------------------------------------------------------------|
| <0>     | Absolute Mode specifier (A) | 0 for nonprivileged use; 1 for privileged use                            |
| <1>     | Reserved (B)                | 0                                                                        |
| <2:14>  | Segment specifier           | 0:1027 (relative extended address)<br>0:8191 (absolute extended address) |
| <15:20> | Page specifier              | 0:63                                                                     |
| <21:30> | Word specifier              | 0:1023                                                                   |
| <31>    | Byte specifier              | 0 in left byte; 1 in right byte                                          |

The segment specifier in an extended pointer indicates which segment is being accessed by the pointer:

| Segment Specifier | Segment                                                                                 |
|-------------------|-----------------------------------------------------------------------------------------|
| 0                 | Current user data segment                                                               |
| 1                 | System data segment if in privileged mode<br>User data segment if in nonprivileged mode |
| 2                 | Current code segment                                                                    |
| 3                 | Current user code segment (read access only)                                            |
| 4- <i>n</i>       | Base address for the current extended data segment                                      |

An extended pointer, having 32 bits, can access byte addresses anywhere in a segment. (The page, word, and byte fields together require 17 address bits.) All extended addresses are byte addresses.

### Accessing the Upper 32K-Word Area

You can use the upper 32K-word area of the current user data segment if you are not using the CRE.

To use the upper 32K-word area, you must use the DATAPAGES directive. For example, you can specify 33 pages for the user data segment.

```
DATAPAGES 33
```

To access the upper 32K-word area, you must declare a pointer and store an appropriate address in it. To access word addresses (but not byte addresses) in the upper 32K-word area, you can use a standard (16-bit) pointer. To access byte addresses in that area, you must use an extended (32-bit) pointer.

### Storing Addresses in Simple Pointers

To store an address in a simple pointer, you can initialize the pointer when you declare it, or you can assign an address to it in an assignment statement.

#### Initializing Simple Pointers

When you declare a simple pointer, you can initialize it with a standard (16-bit) address or an extended (32-bit) address. For example, you can initialize a simple pointer with the first word address in the upper 32K-word area as follows:

```
INT .std_ptr := %100000; !First 16-bit word address
 ! in upper 32K-word area
```

You can initialize an extended simple pointer with the first extended byte address in the upper 32K-word area as follows:

```
INT .EXT top_ptr := %200000D; !First 32-bit byte address
 ! in upper 32K-word area
```

You can use the `$XADR` standard function to convert the 16-bit address contained in a standard simple pointer to a 32-bit address with which to initialize an extended simple pointer:

```
INT .std_ptr := %100000; !Declare INT standard simple
 ! pointer

INT .EXT ext_ptr := $XADR(std_ptr);
 !Declare extended simple pointer
 ! initialize it with 32-bit
 ! address returned by $XADR for
 ! INT item pointed to by STD_PTR
```

### Assigning Addresses to Simple Pointers

Once you have declared a pointer, you can use assignment statements to assign an address to the pointer.

You can assign to a standard simple pointer the first standard word address in the upper 32K-word area of the current user data segment:

```
INT .std_ptr; !Declare standard simple pointer
@std_ptr := %100000; !Assign first word address
 ! in upper 32K-word area
```

You can assign to an extended simple pointer the first extended byte address in the upper 32K-word area of the current user data segment:

```
INT .EXT top_ptr; !Declare extended simple pointer
@top_ptr := %200000D; !Assign first byte address
 ! in upper 32K-word area
```

You can use the `$XADR` standard function to return the extended address of an INT item to which a standard simple pointer points and then assign the 32-bit address to an extended simple pointer:

```
INT .EXT ext_ptr; !Declare extended simple pointer
INT .std_ptr := %100000; !Declare INT standard simple
 ! pointer
@ext_ptr := $XADR(std_ptr); !Assign 32-bit address of
 ! INT item returned by $XADR
```

**Storing Addresses in Structure Pointers** You can store an address in a structure pointer either by initializing the pointer at declaration or by assigning an address to it after declaration.

#### Initializing Structure Pointers

You can declare an extended structure pointer named `EXT_STRUCT_PTR`, specify a referral named `MY_STRUCT` and then initialize the pointer with the first byte address of the upper 32K-word area of the current user data segment as follows:

```
INT .EXT ext_struct_ptr (my_struct) := %200000D;
 !First byte address in upper
 ! 32K-word area
```

To associate an INT standard structure pointer with a template structure and initialize the structure pointer with the first word address in the upper 32K-word area of the current user data segment, specify:

```
STRUCT names (*); !Declare template structure
BEGIN
 INT array[0:11];
END;

INT .struc_ptr (names) := %100000;
 !Declare structure pointer;
 ! initialize it with first word
 ! address in upper 32K-word area
```

To associate an extended structure pointer with the structure pointer declared in the preceding example and initialize the new structure pointer with the first byte address in the upper 32K-word area of the current user data segment, specify:

```
STRING .EXT ex_strc_ptr (struc_ptr) := %200000D;
 !Declare extended structure
 ! pointer; initialize it with
 ! first byte address in
 ! upper 32K-word area
```

#### Assigning Addresses to Structure Pointers

Once you have declared a structure pointer, you can use an assignment statement to assign an address to the structure pointer.

You can associate an INT structure pointer with a template structure and then assign the first word address in the upper 32K-word area of the current user data segment to the structure pointer:

```
STRUCT names (*); !Declare template structure
BEGIN
 INT array[0:11];
END;

INT .struc_ptr (names); !Declare STRUC_PTR

@struc_ptr := %100000; !Assign first word address in
 ! upper 32K-word area
```

You can associate an extended structure pointer with the structure pointer declared in the preceding example and then assign the first byte address in the upper 32K-word area of the current user data segment to the new structure pointer:

```
STRING .EXT ex_strc_ptr (struc_ptr);
 !Declare EXT_STRC_PTR

@ex_strc_ptr := %200000D; !Assign first byte address in
 ! upper 32K-word area
```

### Managing Data Allocation in Upper 32K-Word Area

When you declare pointers, the compiler allocates storage for the pointers, but you must manage allocation for the data at the address contained in each pointer. You must remember which addresses you have used and the length of the item pointed to by each pointer. When you initialize subsequent pointers, you must allow enough space for the previous items. You can then use an assignment or move statement to copy data to the address contained in the pointer.

For example, you manage standard allocation in the user data segment as follows:

```
PROC std;
BEGIN
 STRING .EXT byte_ptr := %200000D;
 !Initialize extended pointer with
 ! first byte address in upper
 ! 32K-word area for a 46-byte
 ! string constant

 STRING .EXT str_ptr := @byte_ptr + 46D;
 !Initialize extended pointer with
 ! first free byte address in upper
 ! 32K-word area

 byte_ptr ' := '
 "This is a sample string to be scanned for an X";
 !Move statement copies 46-byte
 ! string constant to the byte
 ! address stored in BYTE_PTR

 !Lots of code
END;
```

### Assigning Data to Simple Pointers

You can assign a value to the 32-bit byte address contained in an extended simple pointer:

```
INT .EXT ep := %200000D; !Declare EP and initialize it
 ! with address of first byte
 ! in upper 32K-word area

ep := 5; !Assign 5 to word location at
 ! address %200000D
```

You can use a move statement to copy a constant list to the address contained in a simple pointer. You can then assign one of the values in the constant list to an array element by appending an index to the pointer to access that particular value:

```

INT var[0:4]; !Declare array
INT .ptr := %100000; !Declare simple pointer

var[2] := 5; !Assign 5 to VAR[2]

ptr ':=' [1, 2, 3]; !Copy constant list to location
 ! at address %100000

var[3] := ptr[2]; !Assign 3 to VAR[3]

```

### Copying Data to Structure Pointers

You can use INT structure pointers to copy data to word-addressed structure items in the upper 32K-word area of the current user data segment. To copy data, you use a move statement:

```

?DATAPAGES 64 !Get maximum upper 32K-word area

STRUCT names (*); !Declare template structure
 BEGIN
 INT new_name[0:7];
 END;

INT .name_ptr1(names) := %100000;
 !Point to beginning of upper
 ! 32K-word area

INT .name_ptr2(names) := %100010;
 !Point to next free space in upper
 ! 32K-word area

PROC main_proc MAIN;
 BEGIN
 !Lots of code
 name_ptr1.new_name[0] ':=' "Athersohn, Jutha";
 name_ptr2.new_name[0] ':=' "Zyrphn, Rhod Wen";
 !Move statement copies data
 !Lots of code ! to word-addressed structure
 END; ! items

```

You can use an INT extended structure pointer to copy data to byte-addressed structure items in the upper 32K-word area of the current data user segment:

```
STRUCT name_rec (*); !Declare template structure
 BEGIN
 STRING name[0:25];
 END;

INT .EXT ext_ptr(name_rec) := %200000D;
 !Point to beginning
 ! of upper 32K-word area

ext_ptr.name[0] := "Anasta L. Malatorious";
 !Move statement copies data
 ! to byte-addressed structure
 ! items
```

### Managing Large Blocks of Memory

The DEFINEPOOL, GETPOOL, and PUTPOOL system procedures can help you manage large blocks of memory and build proper addresses:

```
LITERAL head_size = 19D;
INT .EXT poolhead := %200000D; !Pool header
INT .EXT pool := %200000D + head_size; !Points into upper
 ! 32K-word area

INT .EXT block;

status := DEFINEPOOL (poolhead, pool, head_size);
@block := GETPOOL (poolhead, 1024D);
!Lots of processing
CALL PUTPOOL (poolhead, block);
```

The DEFINEPOOL, GETPOOL, and SEGMENT\_USE\_ procedures return both a condition code and a value. If you assign a returned value to a variable, the condition code setting is lost. For more information on system procedures, see the *Guardian Procedure Calls Reference Manual* and the *Guardian Programmer's Guide*.

#### Indexing the Upper 32K-Word Boundary

Although an index for standard indirect structures must fall within the signed INT range, a word offset (from the zeroth structure occurrence) can be in the range -65,535 through 65,535. The following example shows how you can access such offsets:

```

STRUCT x (*);
 BEGIN
 INT i, j;
 END;

PROC m MAIN;
 BEGIN
 INT .y (x) := %70000; !Y[0:18430] spans the
 ! 32K-word boundary

 USE i;
 FOR i := 0 TO 18430 DO
 y[i].i := y[i].j := 0;
 DROP i;
 END;

```

---

**Note** If you use an array of structure occurrences that spans the 32K-word boundary, your program must handle the data stack overflow condition. The compiler issues an error only when the data stack overflows the 32K-word boundary.

---

A byte offset, as opposed to a word offset, can be in the range -65,535 through 65,535 if the structure occurrences are either all in the lower 32K words or all in the upper 32K words of the user data segment. Otherwise, the byte offset is incorrect.

If the structure is in the upper 32K words of the user data segment, you must use \$XADR and extended (32-bit) addressing. Here is an example:

```

STRUCT x (*);
 BEGIN
 STRING s[0:1];
 END;

PROC m MAIN:
 BEGIN
 INT .y (x) := %100000; !Y[0:32767} is in the
 ! upper 32K-word area

 INT .EXT z (x) := $XADR (y);
 INT i;

 FOR i := 0 TO 32767 DO
 z.s[1] := z.s[0] := 0;
 END;

```

**Accessing the User Code Segment**

You can access the user code segment by storing a 32-bit byte address in an extended pointer. To build the address, use:

- The \$DBLL standard function, which returns an INT(32) value from two INT values. The first INT value becomes the upper half of the INT(32) value, and the second value becomes the lower half.
- An unsigned bit-shift operation ('<<' 1), which converts a word address to a byte address.

**Initializing Simple Pointers**

You can initialize an extended simple pointer with a 32-bit byte address for read-only access. The address can point to the seventh word or fourteenth byte of the current user code segment:

```
INT .EXT ext_ptr := ($DBLL (3, 7)) '<<' 1; !Declare and
 ! initialize extended simple
 ! pointer with 32-bit byte
 ! address in user code segment
```

**Assigning Addresses to Simple Pointers**

You can assign an address for read-only access in the current user code segment as follows:

```
INT .EXT ext_ptr; !Declare extended simple
 ! pointer
@ext_ptr := ($DBLL (3, 7)) '<<' 1;
 !Assign current code
 ! segment address
```

#### Using Extended Data Segments

In addition to the user data segment, you can store data in:

- The automatic extended data segment
- One or more explicit extended data segments

You should use only the automatic extended data segment if possible. You should not mix the two approaches. If you must use explicit segments and the automatic segment, however, follow guidelines 4 and 10 in “Using Explicit Extended Segments” that follows.

#### Using the Automatic Extended Segment

When you declare extended indirect arrays or structures, the system automatically creates and manages an extended data segment for you. You have automatic access, however, to only one extended data segment.

To declare extended indirect arrays or structures, you specify the .EXT indirection symbol. To access the array, you use its name in a statement. Declaring and accessing extended indirect arrays and structures is shown throughout this manual, particularly in Sections 7 and 8.

#### Using Explicit Extended Segments

Your program can allocate and deallocate extended data segments explicitly. Since the advent of the automatic extended data segment, however, programs usually need not use explicit extended data segments. The information in this subsection is provided in support of existing programs.

To create and use an explicit extended segment, you call system procedures. You can allocate and manage as many extended segments as you need, but you can use only one extended segment at a time. You can access data in an explicit extended segment only by using extended pointers. The compiler allocates memory space for the extended pointers you declare. You must manage allocation of the data yourself.

Here are guidelines for using explicit extended segments:

1. First declare an extended pointer for the base address of the explicit extended segment.
2. To allocate an explicit extended segment and obtain the segment base address, call `SEGMENT_ALLOCATE_`.
3. To make the explicit extended segment the current segment, call `SEGMENT_USE_`.
4. If the automatic extended segment is already in place, `SEGMENT_USE_` returns the automatic extended segment's number. Save this number so you can later access the automatic segment again.
5. You must keep track of addresses stored in extended pointers. When storing addresses in subsequent pointers, you must allow space for preceding data items.
6. To refer to data in the current segment, call `READX` or `WRITEX`.
7. To move data between extended segments, call `MOVEX`.

8. To manage large blocks of memory, call `DEFINEPOOL`, `GETPOOL`, and `PUTPOOL`.
9. To determine the size of a segment, call `SEGMENT_GETINFO_`.
10. To access data in the automatic extended segment, call `SEGMENT_USE_` and restore the segment number that you saved at step 4.
11. To delete an explicit segment that you no longer need, call `SEGMENT_DEALLOCATE_`.

For information on using these system procedures, see the *Guardian Programmer's Guide* and the *Guardian Procedure Calls Reference Manual*.

If you do not restore the automatic extended segment before you manipulate data in it, any of the following actions can result:

- An assignment statement is beyond the segment's memory limit and causes a trap.
- All assignments within range occur in the hardware base and limit registers of the automatic extended segment. Data in the currently active extended segment is overwritten. The error is undetected until you discover the discrepancy at a later time.
- You get the wrong data from valid addresses in the explicit extended data segment.

#### Storing the Base Address

The base address of the explicit extended segment defines the first storage location that is available for allocating data.

When you call `SEGMENT_ALLOCATE_`, it returns the base address, as shown in the following example.

#### D-Series Extended Segment Allocation Program

Example B-1 shows a D-series version of an extended segment allocation program.

---

**Example B-1. D-Series Extended Segment Allocation**

```

?INSPECT, SYMBOLS
?SOURCE $SYSTEM.SYSTEM.EXTDECS0 (
? SEGMENT_ALLOCATE_, SEGMENT_USE_)

PROC alloc_xsegment MAIN;
 BEGIN
 DEFINE error = ! ...! #; !Error handling routine

 INT .EXT px; !Extended pointer for base
 ! address of extended segment

 INT s;
 !Lots of code

 s := SEGMENT_ALLOCATE_ (0, 4096D, , , , px);
 !Allocate extended segment 0;
 ! assign status value to S;
 ! request 2 pages of extended
 ! memory; obtain base address

 IF s <> 0 THEN error; !Continue if segment 0 is
 !allocated; else return error

 s := SEGMENT_USE_ (0, , px); !Make segment 0 the current
 ! extended segment

 IF s <> 0 THEN error; !Continue if segment 0 is
 !current; else return error

 px := 5; !Assign 5 to first word of
 ! segment 0

 s := SEGMENT_ALLOCATE_ (1, 4096D, , , , px);
 !Allocate extended segment 1;
 ! assign status value to S;
 ! request 2 pages of extended
 ! memory; obtain base address

 IF s <> 0 THEN error; !Continue if segment 1 is
 !allocated; else return error

 s := SEGMENT_USE_ (1, , px); !Make segment 1 the current
 ! extended segment

 IF s <> 0 THEN error; !Continue if segment 1 is
 ! current; else return error

 px := 2; !Assign 2 to first word of
 ! segment 1

 !Lots more code
 END;

```

---

### Managing Data Allocation in Extended Segments

When you declare a pointer, the compiler allocates storage for the pointer itself but does not allocate storage for data at the address that is contained in the pointer. You must manage such allocation yourself. You must remember which addresses you have used and the length of the data item pointed to by each pointer. When you initialize subsequent pointers, you must allow space for the preceding data items.

All data items in an extended data segment are byte addressed. You can manage data allocation in an extended segment as follows:

```
?SOURCE $SYSTEM.SYSTEM.EXTDECS0 (
? SEGMENT_ALLOCATE_, SEGMENT_USE_)

PROC xsegment MAIN;
 BEGIN
 DEFINE error = ! ...! #; !Error handling routine

 !Extended pointer declarations:
 INT .EXT x; !For a 435-word array
 INT .EXT y; !For a 1000-word array
 INT .EXT z; !For a 94-word array

 INT s;
 !Lots of code
 s := SEGMENT_ALLOCATE_ (0, 4096D, , , , x);
 !Allocate extended segment 0;
 ! assign status value to S;
 ! request 2 pages of extended
 ! memory; obtain base address

 IF s <> 0 THEN error; !Continue if segment 0 is
 !allocated; else return error

 s := SEGMENT_USE_ (0, , x); !Make segment 0 the current
 ! extended segment

 IF s <> 0 THEN error; !Continue if segment 0 is
 !current; else return error

 @y := @x + 870D; !Assign pointer Y to
 ! first free space after
 ! area pointed to by X

 @z := @y + 2000D; !Assign pointer Z to the
 ! first free space after
 ! area pointed to by Y

 !Lots of code
 END;
```

### Accessing Data in an Extended Segment

After you declare a pointer and store an address in the pointer, you can use an assignment or move statement to place an item at the location pointed to by the pointer.

For example, you can declare a structure, declare and initialize an INT extended structure pointer, and then access byte-addressed structure items in the current extended data segment. In this case, a move statement copies a character string into an array in the structure:

```
?SOURCE $SYSTEM.SYSTEM.EXTDECS0 (
? SEGMENT_ALLOCATE_, SEGMENT_USE_)

PROC xsegment MAIN;
 BEGIN
 DEFINE error = ! ...! #; !Error handling routine

 STRUCT name_rec (*); !Declare template structure
 BEGIN
 STRING name[0:25];
 END;

 INT .EXT ext_seg(name_rec); !Pointer for base address
 ! of extended segment

 INT s;
 !Lots of code
 s := SEGMENT_ALLOCATE_ (0, 4096D, , , , ext_seg);
 !Allocate extended segment 0;
 ! assign status value to S;
 ! request 2 pages of extended
 ! memory; store base address
 ! in EXT_SEG

 IF s <> 0 THEN error; !Continue if segment 0 is
 !allocated; else return error

 s := SEGMENT_USE_ (0, , ext_seg);
 !Make segment 0 the current
 ! extended segment

 IF s <> 0 THEN error; !Continue if segment 0 is
 !current; else return error

 ext_seg.name[0] := "Octavius Q. Pumpernickle";
 !Copy data into array

 END;
```

### Copying Data Between Segments

You cannot use a move statement to copy data between extended segments. You can use a move statement to copy data from an extended segment to the user data segment and then from there to another extended segment, or you can use the MOVEX system procedure.

### Managing Large Blocks of Memory

To manage large blocks of memory, you can call the following system procedures:

- DEFINEPOOL    Defines the bounds of a memory pool in an extended data segment or in the user data segment.
- GETPOOL       Obtains a block of storage from a memory pool.
- PUTPOOL       Returns a block of storage to a memory pool.

Each of the procedures returns a condition code and a value. If you assign a returned value to a variable, the condition code is lost.

The following example shows how to use memory pools to manage data storage in extended data segments. As shown in the example, you should store -1 or -1D in nil pointers, not 0D because 0D points to the first word in the user data segment.

### D-Series Extended Segment Management Program

Example B-2 shows D-series extended segment management program.

---

#### Example B-2. D-Series Extended Segment Management Program(Page 1 of 3)

```

?INSPECT, SYMBOLS
?NOCODE
?PAGE "dummy page directive"

?PUSHLIST, NOLIST SOURCE $SYSTEM.SYSTEM.EXTDECS0 (
? PROCESS_DEBUG_, DEFINEPOOL, GETPOOL, PUTPOOL,
? SEGMENT_ALLOCATE_, SEGMENT_USE_, SEGMENT_DEALLOCATE_)

?POPLIST

LITERAL dealloc_flags = 1; !For SEGMENT_DEALLOCATE_
 ! later
LITERAL seg_id_zero = 0; !User extended data segment
LITERAL seg_id_two = 2; !IDs need not be contiguous
LITERAL seg_id_zero_len = 2048D;
LITERAL seg_id_two_len = 4096D;
INT .EXT word_ptr := -1D; !Nil pointer
STRING .EXT byte_ptr := -1D; !Nil pointer

INT .EXT pool_head := -1D; !Pointer for 19-word pool
 ! header in extended segment

INT .EXT pool_ptr := -1D; !Pointer for first byte after
 ! pool header

INT .EXT block_ptr1 := -1D; !Pool block general pointer
INT .EXT block_ptr2 := -1D; !Pool block general pointer
STRING .byte_array[-1:100]; !Byte array for local scan
STRING .EXT ba_ptr := -1D; !Extended pointer to byte
 ! array for extended move

STRING .offset_ptr := -1;
INT offset_x := 0;

LITERAL str_len = 47; !Length of string to move
LITERAL array_len = 102; !Length of byte array

INT status := 1000; !Beyond maximum error range
INT old_seg_num := -1; !Not a valid user extended
 ! data segment ID

INT error; !Outcome of system procedure

```

---

**Example B-2. D-Series Extended Addressing Program**(Page 2 of 3)

```
PROC ext_addr_example MAIN;
BEGIN
 status := SEGMENT_ALLOCATE_ (
 seg_id_zero, seg_id_zero_len, , , , byte_ptr);
 IF status <> 0 THEN CALL PROCESS_DEBUG_;

 status := SEGMENT_ALLOCATE_
 (seg_id_two, seg_id_two_len, , , , block_ptr1);
 IF status <> 0 THEN CALL PROCESS_DEBUG_;

 status := SEGMENT_USE_ (seg_id_zero, , byte_ptr);
 IF status <> 0 THEN CALL PROCESS_DEBUG_;

 byte_ptr :='
 "This is a sample string to be scanned for an X.";
 !Put character string into
 ! current extended segment

 byte_array :=' byte_ptr FOR str_len BYTES;
 !Extended move of string
 ! to user stack

 byte_array[-1] := 0; !Delimit the scan area
 byte_array[100] := 0; ! with zeros

 SCAN byte_array[0] UNTIL "X" -> @offset_ptr;
 IF $CARRY THEN CALL PROCESS_DEBUG_;
 !Scan on stack; if scan
 ! stopped by 0, call debugger

 offset_x := @offset_ptr '-' @byte_array[0];
```

---

---

**Example B-2. D-Series Extended Addressing Program** (Page 3 of 3)

```

!USE new extended data segment for more manipulations.

status := SEGMENT_USE_ (seg_id_two, , block_ptr1);
IF status <> 0 THEN CALL PROCESS_DEBUG_;

@pool_ptr := @pool_head + %47D;
 !Store first byte address
 ! after pool header

status := DEFINEPOOL (pool_head, pool_ptr, 4000D);
IF status <> 0 THEN CALL PROCESS_DEBUG_;

@block_ptr1 := GETPOOL (pool_head , 101D);
 !For content of BYTE_ARRAY
IF <> THEN CALL PROCESS_DEBUG_;

block_ptr1 ':= ' byte_array[-1] FOR array_len BYTES;
 !Move BYTE_ARRAY to first
 ! pool in extended segment

@block_ptr2 := GETPOOL (pool_head, 1000D);
 !Get second pool in current
 ! extended segment

IF <> THEN CALL PROCESS_DEBUG_;

block_ptr2 ':= ' [8, 16, 32, 40, 48, 56, 64, 128];
 !Move constant list into
 ! this pool in extended
 ! segment

CALL PUTPOOL (pool_head, block_ptr1);
 !Give first pool back
IF <> THEN CALL PROCESS_DEBUG_;

CALL PUTPOOL (pool_head, block_ptr2);
 !Give second pool back
IF <> THEN CALL PROCESS_DEBUG_;

CALL SEGMENT_DEALLOCATE_ (seg_id_two, dealloc_flags);
CALL SEGMENT_DEALLOCATE_ (seg_id_zero, dealloc_flags);

END;

```

---

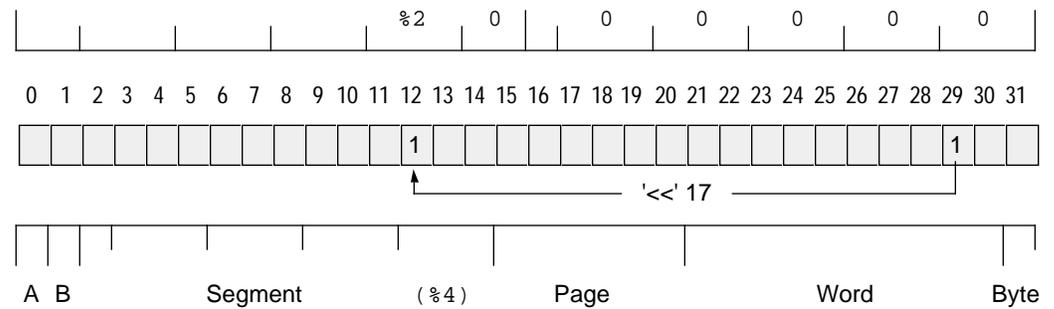
**C-Series Extended Segment Examples**

On a C-series system, you call `ALLOCATESEGMENT` and `USESEGMENT` to create and use an explicit extended data segment. `ALLOCATESEGMENT` does not return the base address of the extended segment. The first segment address you can use in the explicit extended segment is `4D '<<' 17` or `%2000000D`. The following declarations are equivalent:

```
STRING .EXT ptr := 4D '<<' 17;
STRING .EXT ptr := %2000000D;
```

Figure B-2 shows the format of the base address.

**Figure B-2. Format of Extended Segment Base Address**



### C-Series Extended Segment Allocation Program

Example B-3 shows a C-series version of the previous D-series extended segment allocation program. This example is not portable to future software platforms:

---

#### Example B-3. C-Series Extended Segment Allocation Program

```
?INSPECT, SYMBOLS
?SOURCE $SYSTEM.SYSTEM.EXTDECS0 (
? ALLOCATESEGMENT, USESEGMENT)

PROC alloc_xsegment MAIN;
 BEGIN
 DEFINE error = ! ...! #; !Error handling routine
 INT .EXT px := %2000000D; !Initialize extended pointer
 ! to start of extended segment

 INT s;
 !Lots of code
 s := ALLOCATESEGMENT (0, 4096D);
 !Allocate extended segment 0;
 ! assign status value to S;
 ! request 2 pages (4K bytes)
 ! of extended memory

 IF s <> 0 THEN error; !Continue if segment 0 is
 ! allocated else return error

 CALL USESEGMENT (0); !Make segment 0 the current
 ! extended segment

 px := 5; !Assign 5 to first word of
 ! segment 0

 s := ALLOCATESEGMENT (1, 4096D);
 !Allocate extended segment 1;
 ! assign status value to S;
 ! request 2 pages (4K bytes)
 ! of extended memory

 IF s <> 0 THEN error; !Continue if segment 1 is
 ! allocated else return error

 CALL USESEGMENT (1); !Make segment 1 the current
 ! extended segment

 px := 2; !Assign 2 to first word of
 ! extended segment 1

 !Lots more code

 END;
```

---

### Managing Data Allocation in Extended Segments

When you declare a pointer, the compiler allocates storage for the pointer itself but does not allocate storage for data at the address that is contained in the pointer. You must manage such allocation yourself. You must remember which addresses you have used and the length of the data item pointed to by each pointer. When you initialize subsequent pointers, you must allow space for the preceding data items.

An extended data segment begins at the extended byte address %2000000D. All data items in an extended data segment are byte addressed. You can manage data allocation in an extended segment as follows:

```
INT .EXT x := %2000000D; !Initialize extended simple
 ! pointer with first byte
 ! address in extended segment
 ! for 435-word array

INT .EXT y := @x + 870D; !Initialize extended simple
 ! pointer with first free
 ! byte address after array
 ! pointed to by X for
 ! 1000-word array

INT .EXT z := @y + 2000D; !Initialize extended simple
 ! pointer with first free
 ! byte address after array
 ! pointed to by Y for
 ! 94-word array
```

**C-Series Extended Segment Management Program**

Example B-4 shows a C-series version of the previous D-series extended segment management program. This example is not portable to future software platforms.

---

**Example B-4. C-Series Extended Segment Management Program (Page 1 of 3)**

```

?INSPECT, SYMBOLS
?NOCODE
?PAGE "dummy page directive"

LITERAL dealloc_flags = 1; !For DEALLOCATESEGMENT later
LITERAL seg_id_zero = 0; !User extended data segment
LITERAL seg_id_two = 2; !IDs need not be contiguous
LITERAL seg_id_zero_len = 2048D;
LITERAL seg_id_two_len = 4096D;
INT .EXT word_ptr := -1D; !Nil pointer
STRING .EXT byte_ptr := -1D; !Nil pointer
INT .EXT pool_head := %2000000D;
 !Beginning of 19-word pool
 ! header in extended segment
INT .EXT pool_ptr := %2000046D;
 !First byte after pool header
INT .EXT block_ptr1 := -1D; !Pool block general pointer
INT .EXT block_ptr2 := -1D; !Pool block general pointer
STRING .byte_array[-1:100]; !Byte array for local scan
STRING .EXT ba_ptr := -1D; !Extended pointer to byte
 ! array for extended move

STRING .offset_ptr := -1;
INT offset_x := 0;

LITERAL str_len = 47; !Length of string to move
LITERAL array_len = 102; !Length of byte array

INT status := 1000; !Beyond maximum error range
INT old_seg_num := -1; !Not a valid user extended
 ! data segment ID
INT error; !Outcome of system procedure

?PUSHLIST, NOLIST SOURCE $SYSTEM.SYSTEM.EXTDECS0 (
? DEBUG, DEFINEPOOL, GETPOOL, PUTPOOL,
? ALLOCATESEGMENT, USESEGMENT, DEALLOCATESEGMENT)

?POPLIST

```

---

**Example B-4. C-Series Extended Addressing Program** (Page 2 of 3)

```

PROC ext_addr_example MAIN;
BEGIN
 status := ALLOCATESEGMENT (seg_id_zero, seg_id_zero_len);
 IF status <> 0 THEN CALL DEBUG;

 status := ALLOCATESEGMENT (seg_id_two, seg_id_two_len);
 IF status <> 0 THEN CALL DEBUG;

 CALL USESEGMENT (seg_id_zero);
 IF <> THEN CALL DEBUG;

 @byte_ptr := %2000000D; !Set extended pointer to
 ! first byte of current
 ! extended segment

 byte_ptr ':= '
 "This is a sample string to be scanned for an X.";
 !Put character string into
 ! current extended segment

 byte_array ':= ' byte_ptr FOR str_len BYTES;
 !Extended move of string
 ! to user stack

 byte_array[-1] := 0; !Delimit the scan area
 byte_array[100] := 0; ! with zeros

 SCAN byte_array[0] UNTIL "X" -> @offset_ptr;
 IF $CARRY THEN CALL DEBUG;
 !Scan on stack; if scan
 ! stopped by 0, call debugger

 offset_x := @offset_ptr '-' @byte_array[0];

```

---

**Example B-4. C-Series Extended Addressing Program** (Page 3 of 3)

```

!USE new extended data segment for more manipulations.

CALL USESEGMENT (seg_id_two);
IF <> THEN CALL DEBUG;

status := DEFINEPOOL (pool_head, pool_ptr, 4000D);
IF status <> 0 THEN CALL DEBUG;

@block_ptr1 := GETPOOL (pool_head , 101D);
 !For content of BYTE_ARRAY
IF <> THEN CALL DEBUG;

block_ptr1 ':= ' byte_array[-1] FOR array_len BYTES;
 !Move BYTE_ARRAY to first
 ! pool in extended segment

!You cannot use a move statement to copy data directly from
! an extended segment to another extended segment. You can
! use a move statement to copy data from an extended segment
! to the user data segment and then from there to another
! extended segment, or you can use the MOVEX system procedure
! described in the Guardian Procedure Calls Reference Manual.

@block_ptr2 := GETPOOL (pool_head, 1000D);
 !Get second pool in current
 ! extended segment

IF <> THEN CALL DEBUG;

block_ptr2 ':= ' [8, 16, 32, 40, 48, 56, 64, 128];
 !Move constant list into
 ! this pool in extended
 ! segment

CALL PUTPOOL (pool_head, block_ptr1);
 !Give first pool back
IF <> THEN CALL DEBUG;

CALL PUTPOOL (pool_head, block_ptr2);
 !Give second pool back
IF <> THEN CALL DEBUG;

CALL DEALLOCATESEGMENT (seg_id_two, dealloc_flags);
CALL DEALLOCATESEGMENT (seg_id_zero, dealloc_flags);

END;

```

---

---

# Appendix C Improving Performance

---

Although the TAL compiler is a one-pass compiler and is subject to certain limitations inherent in this characteristic, it generates efficient object code for the target computer. If optimum run-time speed is important, however, you can maximize efficiency by following the guidelines given in this appendix.

---

**General Guidelines** The following guidelines describe general practices for achieving efficient code:

- Code programs as cleanly and clearly as possible. Provide structured source code and adequate documentation in the source listing.
- Debug the programs to ensure that they work properly.
- Analyze the programs using performance analysis tools to determine where inefficiencies occur.
- Based on the analysis, change procedures that require modification. Provide comments that describe the changes and why you made them.

The following guidelines apply to addressing, indexing, and arithmetic operations.

---

**Addressing Guidelines** You can use direct and indirect addressing in various ways.

**Direct Addressing** Although direct addressing is limited in the amount of memory it can reference, it is more efficient than indirect addressing. Thus, you should use direct addressing whenever possible.

For example, suppose a procedure expects a reference parameter that is used heavily in calculations within that procedure before it returns a value to the caller. In the procedure, move the value in the indirectly addressed parameter to a local directly addressed storage area and then use that copy in the calculations. At the end of the procedure, store the result in the original parameter, which is returned. Although initially a slight overhead results from copying from parameter to local variable back to parameter, overall execution speed is improved because:

- Indirect addressing is used only twice (once in parameter passing and once in returning the value).
- All other references use direct addressing.

**Indirect Addressing** Indirect arrays, indirect structures, and pointers you declare provide equivalent operation. The advantage of indirect arrays and indirect structures is that the compiler provides a pointer for the array or structure, allocates the data or structure, and initializes the pointer to the beginning of the array or structure. To use pointers you declare, you must initialize the pointer and manage allocation of the data to which the pointer points.

**Extended Addressing** The compiler emits shorter instruction sequences if it can place INT and STRING extended pointers in locations G[0] through G[63] or L[1] through L[63], so you should declare these pointers before you declare other global and local declarations.

**STACK and STORE Statements** STACK and STORE statements do not improve the efficiency of access to data items. These statements are provided primarily for moving operands to and from the register stack when working with the CODE statement.

**Indexing Guidelines** The compiler saves index values in index registers so you can refer to them in later statements. For instance, for the following operation, the compiler saves the value of I in an index register:

```
x[i] := 5;
```

You can then use I in a reference such as Y[I].

Multiple references to the same index value (using the same data type) promote efficiency.

For indexed items in structures, the compiler optimizes references only to adjacent items within the same substructure.

An index on a 16-bit variable is always a signed INT expression. For a STRING variable, an index can access ranges from 32K bytes below to 32K bytes above the zeroth structure occurrence. For any variable except a STRING variable, an index can access ranges from 32K words below to 32K words above the zeroth structure occurrence.

Indexing indirect references is no less efficient than not indexing indirect references, because the hardware requires no extra time to add indexes to address values.

For an INT or STRING extended pointer located below G[63] or L[63] (decimal), a 16-bit index is more efficient than a 32-bit index. A 16-bit index results in a shorter instruction sequence using the LWXX, SWXX, LBXX, and SBXX instructions. (These instructions are described in the *System Description Manual* for your system.)

For all other extended pointers, a 16-bit index is slightly more efficient than a 32-bit index. If, however, the offset of a structure item declared in an extended indirect structure is outside the signed INT range (-32,768 through 32,767), you must use a 32-bit index.

In a program written for a D-series system, you can use the INT32INDEX directive when you index an extended indirect structure item. INT32INDEX suppresses the [NO]INHIBITXX directive and generates a 32-bit index from a 16-bit index. If you use INT32INDEX, you need not calculate the offset of an extended structure item to determine whether to use a 16-bit or a 32-bit index. INT32INDEX always generates correct offsets but is slightly less efficient than using a 32-bit index. For more information, see "Indexing Structures" in Section 8, "Using Structures."

Using a USE register for the 16-bit index of an extended pointer does not provide further efficiency. The compiler must still load the index value from the USE register into register A for use with the LWXX, SWXX, LBXX, and SBXX instructions. For the less efficient extended access, the compiler loads the 16-bit index from the USE register into register A, then converts it to a 32-bit index.

**Arithmetic Guidelines** A single complex arithmetic expression might cause more memory references than several smaller expressions that are equivalent to the single complex expression. The excessive memory references are triggered by register stack overflow, which is especially likely if indexes are involved. Use of an index might cause part of the computation to be pushed on the stack and later popped off. Doubleword or quadrupleword operands fill the register stack quickly.

For quadrupleword operations, do not nest index calculations in larger arithmetic expressions because register stack overflow is likely to result. Use a separate statement for the index calculations, saving the results in a temporary area. The expression can then reference this area.

The IF and CASE forms of arithmetic expressions do not generate efficient machine code, especially when used to test complex conditions. To evaluate a complex condition, include separate IF or CASE statements that perform proper assignments in all possible branches of the condition.

# Appendix D ASCII Character Set

| Char | Left   | Right  | Hex | Dec | Meaning                   |
|------|--------|--------|-----|-----|---------------------------|
| NUL  | 000000 | 000000 | 00  | 0   | Null                      |
| SOH  | 000400 | 000001 | 01  | 1   | Start of heading          |
| STX  | 001000 | 000002 | 02  | 2   | Start of text             |
| ETX  | 001400 | 000003 | 03  | 3   | End of text               |
| EOT  | 002000 | 000004 | 04  | 4   | End of transmission       |
| ENQ  | 002400 | 000005 | 05  | 5   | Enquiry                   |
| ACK  | 003000 | 000006 | 06  | 6   | Acknowledge               |
| BEL  | 003400 | 000007 | 07  | 7   | Bell                      |
| BS   | 004000 | 000010 | 08  | 8   | Backspace                 |
| HT   | 004400 | 000011 | 09  | 9   | Horizontal tabulation     |
| LF   | 005000 | 000012 | A   | 10  | Line feed                 |
| VT   | 005400 | 000013 | B   | 11  | Vertical tabulation       |
| FF   | 006000 | 000014 | C   | 12  | Form feed                 |
| CR   | 006400 | 000015 | D   | 13  | Carriage return           |
| SO   | 007000 | 000016 | E   | 14  | Shift out                 |
| SI   | 007400 | 000017 | F   | 15  | Shift in                  |
| DLE  | 010000 | 000020 | 10  | 16  | Data link escape          |
| DC1  | 010400 | 000021 | 11  | 17  | Device control 1          |
| DC2  | 011000 | 000022 | 12  | 18  | Device control 2          |
| DC3  | 011400 | 000023 | 13  | 19  | Device control 3          |
| DC4  | 012000 | 000024 | 14  | 20  | Device control 4          |
| NAK  | 012400 | 000025 | 15  | 21  | Negative acknowledge      |
| SYN  | 013000 | 000026 | 16  | 22  | Synchronous idle          |
| ETB  | 013400 | 000027 | 17  | 23  | End of transmission block |
| CAN  | 014000 | 000030 | 18  | 24  | Cancel                    |
| EM   | 014400 | 000031 | 19  | 25  | End of medium             |
| SUB  | 015000 | 000032 | 1A  | 26  | Substitute                |
| ESC  | 015400 | 000033 | 1B  | 27  | Escape                    |
| FS   | 016000 | 000034 | 1C  | 28  | File separator            |
| GS   | 016400 | 000035 | 1D  | 29  | Group separator           |
| RS   | 017000 | 000036 | 1E  | 30  | Record separator          |
| US   | 017400 | 000037 | 1F  | 31  | Unit separator            |
| SP   | 020000 | 000040 | 20  | 32  | Space                     |
| !    | 020400 | 000041 | 21  | 33  | Exclamation point         |
| "    | 021000 | 000042 | 22  | 34  | Quotation mark            |
| #    | 021400 | 000043 | 23  | 35  | Number sign               |
| \$   | 022000 | 000044 | 24  | 36  | Dollar sign               |
| %    | 022400 | 000045 | 25  | 37  | Percent sign              |
| &    | 023000 | 000046 | 26  | 38  | Ampersand                 |
| '    | 023400 | 000047 | 27  | 39  | Apostrophe                |
| (    | 024000 | 000050 | 28  | 40  | Opening parenthesis       |
| )    | 024400 | 000051 | 29  | 41  | Closing parenthesis       |
| *    | 025000 | 000052 | 2A  | 42  | Asterisk                  |

| Char | Left   | Right  | Hex | Dec | Meaning                |
|------|--------|--------|-----|-----|------------------------|
| +    | 025400 | 000053 | 2B  | 43  | Plus                   |
| ,    | 026000 | 000054 | 2C  | 44  | Comma                  |
| -    | 026400 | 000055 | 2D  | 45  | Hyphen (minus)         |
| .    | 027000 | 000056 | 2E  | 46  | Period (decimal point) |
| /    | 027400 | 000057 | 2F  | 47  | Right slash            |
| 0    | 030000 | 000060 | 30  | 48  | Zero                   |
| 1    | 030400 | 000061 | 31  | 49  | One                    |
| 2    | 031000 | 000062 | 32  | 50  | Two                    |
| 3    | 031400 | 000063 | 33  | 51  | Three                  |
| 4    | 032000 | 000064 | 34  | 52  | Four                   |
| 5    | 032400 | 000065 | 35  | 53  | Five                   |
| 6    | 033000 | 000066 | 36  | 54  | Six                    |
| 7    | 033400 | 000067 | 37  | 55  | Seven                  |
| 8    | 034000 | 000070 | 38  | 56  | Eight                  |
| 9    | 034400 | 000071 | 39  | 57  | Nine                   |
| :    | 035000 | 000072 | 3A  | 58  | Colon                  |
| ;    | 035400 | 000073 | 3B  | 59  | Semicolon              |
| <    | 036000 | 000074 | 3C  | 60  | Less than              |
| =    | 036400 | 000075 | 3D  | 61  | Equals                 |
| >    | 037000 | 000076 | 3E  | 62  | Greater than           |
| ?    | 037400 | 000077 | 3F  | 63  | Question mark          |
| @    | 040000 | 000100 | 40  | 64  | Commercial at sign     |
| A    | 040400 | 000101 | 41  | 65  | Uppercase A            |
| B    | 041000 | 000102 | 42  | 66  | Uppercase B            |
| C    | 041400 | 000103 | 43  | 67  | Uppercase C            |
| D    | 042000 | 000104 | 44  | 68  | Uppercase D            |
| E    | 042400 | 000105 | 45  | 69  | Uppercase E            |
| F    | 043000 | 000106 | 46  | 70  | Uppercase F            |
| G    | 043400 | 000107 | 47  | 71  | Uppercase G            |
| H    | 044000 | 000110 | 48  | 72  | Uppercase H            |
| I    | 044400 | 000111 | 49  | 73  | Uppercase I            |
| J    | 045000 | 000112 | 4A  | 74  | Uppercase J            |
| K    | 045400 | 000113 | 4B  | 75  | Uppercase K            |
| L    | 046000 | 000114 | 4C  | 76  | Uppercase L            |
| M    | 046400 | 000115 | 4D  | 77  | Uppercase M            |
| N    | 047000 | 000116 | 4E  | 78  | Uppercase N            |
| O    | 047400 | 000117 | 4F  | 79  | Uppercase O            |
| P    | 050000 | 000120 | 50  | 80  | Uppercase P            |
| Q    | 050400 | 000121 | 51  | 81  | Uppercase Q            |

| Char | Left   | Right  | Hex | Dec | Meaning         |
|------|--------|--------|-----|-----|-----------------|
| R    | 051000 | 000122 | 52  | 82  | Uppercase R     |
| S    | 051400 | 000123 | 53  | 83  | Uppercase S     |
| T    | 052000 | 000124 | 54  | 84  | Uppercase T     |
| U    | 052400 | 000125 | 55  | 85  | Uppercase U     |
| V    | 053000 | 000126 | 56  | 86  | Uppercase V     |
| W    | 053400 | 000127 | 57  | 87  | Uppercase W     |
| X    | 054000 | 000130 | 58  | 88  | Uppercase X     |
| Y    | 054400 | 000131 | 59  | 89  | Uppercase Y     |
| Z    | 055000 | 000132 | 5A  | 90  | Uppercase Z     |
| [    | 055400 | 000133 | 5B  | 91  | Opening bracket |
| \    | 056000 | 000134 | 5C  | 92  | Back slash      |
| ]    | 056400 | 000135 | 5D  | 93  | Closing bracket |
| ^    | 057000 | 000136 | 5E  | 94  | Circumflex      |
| _    | 057400 | 000137 | 5F  | 95  | Underscore      |
| `    | 060000 | 000140 | 60  | 96  | Grave accent    |
| a    | 060400 | 000141 | 61  | 97  | Lowercase a     |
| b    | 061000 | 000142 | 62  | 98  | Lowercase b     |
| c    | 061400 | 000143 | 63  | 99  | Lowercase c     |
| d    | 062000 | 000144 | 64  | 100 | Lowercase d     |
| e    | 062400 | 000145 | 65  | 101 | Lowercase e     |
| f    | 063000 | 000146 | 66  | 102 | Lowercase f     |
| g    | 063400 | 000147 | 67  | 103 | Lowercase g     |
| h    | 064000 | 000150 | 68  | 104 | Lowercase h     |
| i    | 064400 | 000151 | 69  | 105 | Lowercase i     |
| j    | 065000 | 000152 | 6A  | 106 | Lowercase j     |
| k    | 065400 | 000153 | 6B  | 107 | Lowercase k     |
| l    | 066000 | 000154 | 6C  | 108 | Lowercase l     |
| m    | 066400 | 000155 | 6D  | 109 | Lowercase m     |
| n    | 067000 | 000156 | 6E  | 110 | Lowercase n     |
| o    | 067400 | 000157 | 6F  | 111 | Lowercase o     |
| p    | 070000 | 000160 | 70  | 112 | Lowercase p     |
| q    | 070400 | 000161 | 71  | 113 | Lowercase q     |
| r    | 071000 | 000162 | 72  | 114 | Lowercase r     |
| s    | 071400 | 000163 | 73  | 115 | Lowercase s     |
| t    | 072000 | 000164 | 74  | 116 | Lowercase t     |
| u    | 072400 | 000165 | 75  | 117 | Lowercase u     |
| v    | 073000 | 000166 | 76  | 118 | Lowercase v     |
| w    | 073400 | 000167 | 77  | 119 | Lowercase w     |
| x    | 074000 | 000170 | 78  | 120 | Lowercase x     |
| y    | 074400 | 000171 | 79  | 121 | Lowercase y     |
| z    | 075000 | 000172 | 7A  | 122 | Lowercase z     |

| Char | Left   | Right  | Hex | Dec | Meaning       |
|------|--------|--------|-----|-----|---------------|
| {    | 075400 | 000173 | 7B  | 123 | Opening brace |
|      | 076000 | 000174 | 7C  | 124 | Vertical line |
| }    | 076400 | 000175 | 7D  | 125 | Closing brace |
| ~    | 077000 | 000176 | 7E  | 126 | Tilde         |
| DEL  | 077400 | 000177 | 7F  | 127 | Delete        |

---

# Appendix E File Names and TACL Commands

---

This appendix describes how you specify disk file names and TACL commands. Topics include:

- Disk file names
- TACL DEFINE commands
- TACL PARAM commands
- TACL ASSIGN commands
- TACL ASSIGN SSV commands

For information on process or device file names, see the *Guardian Programmer's Guide*.

---

## Disk File Names

A disk file name identifies a file that contains data or a program. A disk file name reflects the specified file's location on a Tandem system. The location of a disk file on a Tandem system is analogous to the location of a form in a file cabinet. To find the form, you must know:

- Which file cabinet it is in
- Which drawer it is in
- Which folder it is in
- Which form it is

Analogously, to find a disk file on a Tandem system, you must know:

- Which node (system) it is on
- Which volume it is on
- Which subvolume it is on
- Which disk file it is

In general, disk file names:

- Cannot contain spaces
- Can contain ASCII characters only
- Are not case-sensitive; the following names are equivalent:

```
myfile
MyFile
MYFILE
```

Language functions and system procedures that return file names might return them in uppercase (even if the file name was originally in lowercase). Check the description of the function or procedure that you are using.

**Parts of a Disk File Name** A disk file has a unique file name that consists of four parts, with each part separated by a period:

- A D-series node name or a C-series system name
- A volume name
- A subvolume name
- A file ID

Here is an example of a disk file name:

```
\mynode.$myvol.mysubvol.myfile
```

You can name your own subvolumes and file IDs, but nodes (systems) and volumes are named by the system manager.

All parts of the file name except the file ID are optional except as noted in the following discussion. If you omit any part of the file name, the system uses values as described in “Partial File Names” later in this appendix.

#### **Node or System Name**

The node or system name, such as `\MYNODE`, is the name of the node or system where the file resides. If specified, the node or system name must begin with a backslash (`\`) followed by one to seven alphanumeric characters. The character following the backslash must be an alphabetic character.

#### **Volume Name**

The volume name, such as `$MYVOL`, is the name of the disk volume where the file resides. If specified, the volume name must begin with a dollar sign (`$`), followed by one to six or one to seven alphanumeric characters as follows. The character following the dollar sign must be an alphabetic character.

On a D-series system, the volume name can contain one to seven alphanumeric characters.

On a C-series system, the volume name can contain:

- One to six alphanumeric characters if you include the system name
- One to seven alphanumeric characters if you omit the system name

On a C-series system, if you specify the system name, you must also specify the volume name. If you omit the system name, specifying the volume name is optional.

#### **Subvolume Name**

The subvolume name, such as `MYSUBVOL`, is the name of the set of files, on the disk volume, within which the file resides. The subvolume name can contain from one to eight alphanumeric characters, the first of which must be alphabetic.

On a D-series system, if you specify the volume name, you must also specify the subvolume name. If you omit the volume name, specifying the subvolume name is optional.

**File ID**

The file ID, such as MYFILE, is the identifier of the file in the subvolume. The file ID can contain from one to eight alphanumeric characters, the first of which must be alphabetic.

The file ID is required.

**Partial File Names**

A partial file name contains at least the file ID, but does not contain all the file-name parts. When you specify a partial file name, the operating system or other process fills in the missing file-name parts by using your current default values. Following are the optional file-name parts and their default values:

| File-Name Part | Default                                          |
|----------------|--------------------------------------------------|
| node (system)  | Node (system) on which your program is executing |
| volume         | Current default volume                           |
| subvolume      | Current default subvolume                        |

Following are all the partial file names you can specify for a disk file named \BRANCH.\$DIV.DEPT.EMP:

| Omitted File-Name Parts          | Partial File Name | D-Series System | C-Series System |
|----------------------------------|-------------------|-----------------|-----------------|
| Node (system)                    | \$div.dept.emp    | Yes             | Yes             |
| Node (system), volume            | dept.emp          | Yes             | Yes             |
| Node (system), volume, subvolume | emp               | Yes             | Yes             |
| Volume                           | \branch.dept.emp  | Yes             | No              |
| Volume, subvolume                | \branch.emp       | Yes             | No              |
| Subvolume                        | \branch.\$div.emp | No              | Yes             |
| Node (system), subvolume         | \$div.emp         | No              | Yes             |

You can change your current default values in various ways:

- You can change the volume and subvolume with the VOLUME command of, for example, the Binder, Inspect, and TACL products.
- In some cases, you can specify node (system), volume, and subvolume names by issuing TACL ASSIGN SSV commands, described later in this appendix.

**Logical File Names**

You can use a logical file name in place of the disk file name. A logical file name is an alternate name you specify in a TACL DEFINE or TACL ASSIGN command, described later in this appendix.

**Internal File Names** The C-series operating system uses the internal form of a file name when passing it between your program and the operating system. The D-series operating system uses the internal form only if your program has not been converted to use D-series features.

For information on converting external file names to internal file names in a program, see the *Guardian Programmer's Guide* and the *Guardian Procedure Calls Reference Manual*.

---

**TACL Commands** You can send information to the compiler by using the following TACL commands:

- DEFINE
- PARAM
- ASSIGN

These commands are summarized in the remainder of this appendix. For complete information on these commands, see the following manuals:

- TACL Reference Manual* (syntactic information)
  - TACL Programmer's Guide* (programmatic information)
  - Guardian User's Guide* (interactive information)
  - Guardian Programmer's Guide* (programmatic information)
- 

**TACL DEFINE Commands** By issuing TACL DEFINE commands before starting the compiler, you can:

- Substitute an actual file name for a logical file name used in the source file
- Specify spooler attributes
- Specify file attributes on a labeled tape
- Specify process defaults, such as default volume and subvolume

**Substituting File Names** You can substitute a file name for a logical (or TACL DEFINE) name being passed by a nonprivileged program to a system procedure. To substitute a file name, issue the following TACL commands:

| TACL Command     | Purpose                                                    |
|------------------|------------------------------------------------------------|
| SET DEFMODE ON   | Enable DEFINE processing                                   |
| SET DEFINE CLASS | Set the initial attribute of a DEFINE command to CLASS MAP |
| SET DEFINE       | Set the working attributes                                 |
| ADD DEFINE       | Specify a file name to substitute for a DEFINE name        |

**TACL DEFINE Names** TACL DEFINE names:

- Are not case-sensitive
- Consist of 2 to 24 characters
- Begin with an equals sign (=) followed by an alphabetic character
- Continue with any combination of letters, digits, hyphens (-), underscores (\_), and circumflexes (^)

Some examples of valid DEFINE names are:

```
=A
=The_chosen_file
=Long-but-not-too-long
=The-File-of-The-Week
```

DEFINE names that begin with an equals sign followed by an underscore (=\_) are reserved by Tandem. For example, do not use DEFINE names such as =\_DEFAULT.

**Setting DEFINE  
CLASS Attributes**

To create a DEFINE message or set its attributes, you must set a CLASS attribute for the DEFINE. The CLASS attributes are MAP, TAPE, SORT/SUBSORT, SPOOL, and DEFAULTS. Each attribute has an initial setting based on whether the attribute is required, optional, or default.

#### MAP DEFINE

When you log on, the default CLASS attribute is MAP, which requires a file name. A MAP DEFINE substitutes a file name for a DEFINE name used in the source file. For example, suppose that your current CLASS attribute is MAP and your source file includes the DEFINE name =MULTI in a SOURCE directive:

```
?SOURCE =multi
```

Before running the compiler, you can associate file name \BRIG.\$ULLX.CABLE.PORT with =MULTI:

```
ADD DEFINE =multi, FILE \brig.$ullx.cable.port
```

During compilation, the compiler passes the DEFINE name to a system procedure, which makes the file available to the compiler. If the system procedure cannot make the file available, the open operation fails.

#### TAPE DEFINE (D-Series Systems Only)

The TAPE DEFINE lets you specify attributes for labeled magnetic tapes. For instance, it lets you specify attributes such as block length, recording density, record format and length, number of reels, and labeling.

#### SPOOL DEFINE

The SPOOL DEFINE lets you specify spooler settings or attributes, such as number of copies, form name, location, owner, report name, and priority.

**DEFAULTS DEFINE**

In the DEFAULTS class, a permanently built-in DEFINE named =\_DEFAULTS has the following attributes, which are active regardless of any DEFMODE setting:

| Attribute | Required | Purpose                                                                                                                              |
|-----------|----------|--------------------------------------------------------------------------------------------------------------------------------------|
| VOLUME    | Yes      | Contains the default node, volume, and subvolume names for the current process as set by the TACL VOLUME, SYSTEM, and LOGON commands |
| SWAP      | No       | Contains the node and volume name in which the operating system is to store swap files                                               |
| CATALOG   | No       | Contains a substitute name for a catalog as described in the <i>NonStop SQL Reference Manual</i>                                     |

**TACL PARAM Commands**

Compilers accept TACL PARAM commands that you issue before starting the compilers. PARAM commands are BINSERV, SAMECPU, SWAPVOL, and SYMSERV.

**PARAM BINSERV Command**

The PARAM BINSERV command lets you specify which BINSERV process you want to use. You can specify a file name or a TACL DEFINE name.

For example, you can specify the BINSERV file on a particular node, volume, and subvolume as follows:

```
PARAM BINSERV \mynode.$myvol.mysubvol.BINSERV
```

If the specified node is not the one the TAL compiler runs on, the compiler ignores the command. If you omit the volume and subvolume, the compiler uses the current default volume and subvolume. If you omit the file ID, the compiler uses the file ID BINSERV. If you specify a TACL DEFINE name, it must refer to a disk file of class MAP.

If you use this command, the error file PDERROR must be on the same subvolume as BINSERV. If you omit this command, the compiler uses the BINSERV process on its own volume and subvolume.

**PARAM SAMECPU Command**

The PARAM SAMECPU command causes the compiler, BINSERV, and SYMSERV to all to run in the same processor if you specify any number but 0. For example:

```
PARAM SAMECPU 32767
TAL /CPU 6/
```

Specifying 0 means the compiler, BINSERV, and SYMSERV need not run on the same processor. For example:

```
PARAM SAMECPU 0
```

**PARAM SWAPVOL Command** The PARAM SWAPVOL command lets you specify the volume that the compiler, BINSERV, and SYMSERV use for temporary files. For example:

```
PARAM SWAPVOL $myvol
```

The compiler ignores any node specification and allocates temporary files on its own node. If you omit the volume, the compiler uses the default volume for temporary files; BINSERV and SYMSERV use the volume that is to receive the object file.

Use this command when:

- The volumes normally used for temporary files might not have sufficient space.
- The default volume or the volume to receive the object file is on a different node from the compiler.

**PARAM SYMSERV Command** The PARAM SYMSERV command lets you specify which SYMSERV process you want to use. You can specify a file name or a TACL DEFINE name.

For example, to specify the SYMSERV file on a particular volume and subvolume:

```
PARAM SYMSERV \mynode.$myvol.mysubvol.SYMSERV
```

If the node is not the one the compiler runs on, the compiler ignores the command. If you omit the volume or subvolume, the compiler uses the current default volume or subvolume. If you omit the file name, the compiler uses the name SYMSERV. If you specify a TACL DEFINE name, the name must refer to a disk file of class MAP.

If you omit this command, the compiler uses the volume and subvolume specified in the PARAM BINSERV command. If you omit both PARAM SYMSERV and PARAM BINSERV commands, the compiler uses the SYMSERV process on the compiler's volume and subvolume.

**Using PARAM Commands** You can specify one or more PARAM commands before starting the compiler. For example, you can specify that:

- The compiler use the BINSERV process located on MYSUBVOL

```
PARAM BINSERV mysubvol
```

- The compiler, BINSERV, and SYMSERV all run in the same processor

```
PARAM SAMECPU 1
```

- The compiler, BINSERV, and SYMSERV allocate temporary files on volume SJUNK

```
PARAM SWAPVOL $junk
```

Then you can issue the TAL compilation command:

```
TAL /IN mysource/ myprog
```

**TACL ASSIGN Commands** You can issue the TACL ASSIGN command before starting the compiler to substitute actual file names for logical file names used in the source file. The TACL product stores the file-name mapping until the compiler requests it.

ASSIGN commands fall into two categories:

- Ordinary ASSIGN commands
- ASSIGN SSV commands

**Ordinary ASSIGN Command** The ordinary ASSIGN command equates a file name with a logical file name used in ERRORFILE, SAVEGLOBALS, SEARCH, SOURCE, and USEGLOBALS directives. The compiler accepts only the first 75 ordinary ASSIGN messages.

In each ASSIGN command, specify a logical identifier followed by a comma and the file name or a TACL DEFINE name:

```
ASSIGN dog, \a.$b.c.dog
ASSIGN cat, =mycat
```

If the file name is incomplete, the TACL product completes it from your current default node, volume, and subvolume. For example, if your current defaults are \X.\$Y.Z, the TACL product completes the incomplete file names in ASSIGN commands as follows:

| Incomplete File Names | Complete File Names      |
|-----------------------|--------------------------|
| ASSIGN qq, cat        | ASSIGN qq, \x.\$y.z.cat  |
| ASSIGN ss, b.dog      | ASSIGN ss, \x.\$y.b.dog  |
| ASSIGN tt, \$a.b.rat  | ASSIGN tt, \x.\$a.b.rat. |

If you use a TACL DEFINE name in place of a file name, the TACL product qualifies the file name specified in the ADD DEFINE command when it processes the ASSIGN command. Even if you specify new node, volume, and subvolume defaults between the ADD DEFINE command and the ASSIGN command, the ASSIGN mapping still reflects the ADD DEFINE settings.

#### Processing ASSIGN File Names

If you issue the following commands:

```
ASSIGN aa, $a.b.cat
ASSIGN bb, $a.b.dog
ASSIGN cc, =my_zebra
ADD DEFINE =my_zebra, CLASS MAP, FILE $a.b.zebra

TAL /IN mysource, OUT $s/ obj
```

the compiler equates SOURCE directives in MYSOURCE to files as follows:

```
?SOURCE aa !Equates to ?SOURCE $a.b.cat
?SOURCE cc !Equates to ?SOURCE $a.b.zebra
?SOURCE bb !Equates to ?SOURCE $a.b.dog
```

You can name new source files at each compilation without changing the contents of the source file.

**ASSIGN SSV Command** The ASSIGN SSV (Search SubVolume) command lets you specify which node, volume, and subvolume to take files from. The compiler uses ASSIGN SSV information to resolve incomplete file names in the SEARCH, SOURCE, and USEGLOBALS directives.

For each ASSIGN SSV command, append to the SSV keyword a value in the range 0 through 49. Values in the range 0 through 9 can appear with or without a leading 0.

For example, if you specify:

```
ASSIGN SSV1, oldfiles
```

and the compiler encounters the directive:

```
?SOURCE myutil
```

the compiler looks for OLDFILES.MYUTIL.

If you then specify:

```
ASSIGN SSV1, newfiles
```

and run the compiler again, it looks for NEWFILES.MYUTIL.

If you omit the node or volume, the TACL product uses the current default node or volume. If you omit the subvolume, the compiler ignores the command. TACL DEFINE names are not allowed.

The ASSIGN SSV command also lets you specify the order in which subvolumes are searched. You can specify ASSIGN SSV commands in any order. If the same SSV value appears more than once, the TACL product stores only the last command having that value.

For example, if you issue the following commands, the TACL product stores only two of the messages :

| Assign SSV Command  | Stored |
|---------------------|--------|
| ASSIGN SSV28, \$a.b | Yes    |
| ASSIGN SSV7, \$c.d  | No     |
| ASSIGN SSV7, \$e.f  | No     |
| ASSIGN SSV07, \$g.h | Yes    |

The compiler stores ASSIGN SSV messages in its SSV table in ascending order.

### Processing ASSIGN SSV Commands

For each file name the compiler processes, the compiler scans the SSVs in ascending order from SSV0 until it finds a subvolume that holds the file.

For example, if you issue the following ASSIGN commands before running the compiler:

```
ASSIGN SSV7, $aa.b3
ASSIGN SSV10, $aa.grplib
ASSIGN SSV8, mylib
ASSIGN SSV20, $cc.divlib
ASSIGN trig, $sp.math.xtrig
```

and the compiler encounters the following SOURCE directive:

```
?SOURCE unpack
```

the compiler first looks for an ASSIGN message having the logical name UNPACK. If there is none, the compiler looks for the file in subvolumes in the following order:

```
$aa.b3.unpack (SSV7)
$default-volume.mylib.unpack (SSV8)
$aa.grplib.unpack (SSV10)
$cc.divlib.unpack (SSV20)
$default-volume.default-subvolume.unpack
```

The compiler uses the first file it finds. If it finds none named UNPACK, it issues an error message.

When the compiler encounters the following directive:

```
?SOURCE trig
```

it tries only \$SP.MATH.XTRIG; if it does not find that exact file, it issues an error message.

---

# Appendix F Data Type Correspondence

---

The tables in this appendix provide information on:

- Data type correspondence among Tandem languages
- The return value size generated by each data type

These tables are useful if your programs:

- Use data from files created in another language
- Pass parameters to programs written in callable languages

The return value sizes given in these tables do not correspond to the storage size of SQL data types. For a complete list of SQL data type correspondence, see the appropriate NonStop SQL programmer's guide.

---

**Note** COBOL includes COBOL74, COBOL85, and SCREEN COBOL unless otherwise noted.

---

If you use the Data Definition Language (DDL) utility to describe your files, you might not need these tables. For more information, see the *Data Definition Language (DDL) Reference Manual*.

Table F-1. Integer Types, Part 1

|                                      | 8-Bit Integer                                                                                                            | 16-Bit Integer                                                                                                                                                                                                                                | 32-Bit Integer                                                                                                                                                             |
|--------------------------------------|--------------------------------------------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| BASIC                                | STRING                                                                                                                   | INT<br>INT(16)                                                                                                                                                                                                                                | INT(32)                                                                                                                                                                    |
| C                                    | char [1]<br>unsigned char<br>signed char                                                                                 | int<br>short<br>unsigned                                                                                                                                                                                                                      | long<br>unsigned long                                                                                                                                                      |
| COBOL                                | Alphabetic<br>Numeric DISPLAY<br>Alphanumeric-Edited<br>Alphanumeric<br>Numeric-Edited                                   | PIC S9(n) COMP or PIC 9(n) COMP<br>without P or V, $1 \leq n \leq 4$<br>Index Data Item [2]<br>NATIVE-2 [3]                                                                                                                                   | PIC S9(n) COMP or PIC 9(n) COMP<br>without P or V, $5 \leq n \leq 9$<br>Index Data Item [2]<br>NATIVE-4 [3]                                                                |
| FORTRAN                              | —                                                                                                                        | INTEGER [4]<br>INTEGER*2                                                                                                                                                                                                                      | INTEGER*4                                                                                                                                                                  |
| Pascal                               | BYTE<br>Enumeration, unpacked,<br>$\leq 256$ members<br>Subrange, unpacked,<br>$n \dots m, 0 \leq n$ and<br>$m \leq 255$ | INTEGER<br>INT16<br>CARDINAL [1]<br>BYTE or CHAR value parameter<br>Enumeration, unpacked, $> 256$ members<br>Subrange, unpacked, $n \dots m, -32,768 \leq n$<br>and $m \leq 32,767$ , but at least $n$ or $m$<br>outside $0 \dots 255$ range | LONGINT<br>INT32<br>Subrange, unpacked $n \dots m,$<br>$-2147483648 \leq n$ and<br>$m \leq 2147483647,$<br>but at least $n$ or $m$ outside<br>$-32,768 \dots 32,767$ range |
| SQL                                  | CHAR                                                                                                                     | NUMERIC(1)...NUMERIC(4)<br>PIC 9(1) COMP...PIC 9(4) COMP<br>SMALLINT                                                                                                                                                                          | NUMERIC(5)...NUMERIC(9)<br>PIC 9(1) COMP...PIC 9(9) COMP<br>INTEGER                                                                                                        |
| TAL                                  | STRING<br>UNSIGNED(8)                                                                                                    | INT<br>UNSIGNED(16)                                                                                                                                                                                                                           | INT(32)                                                                                                                                                                    |
| <b>Return Value<br/>Size (Words)</b> | 1                                                                                                                        | 1                                                                                                                                                                                                                                             | 2                                                                                                                                                                          |

[1] Unsigned Integer.

[2] Index Data Item is a 16-bit integer in COBOL 74 and a 32-bit integer in COBOL85.

[3] Tandem COBOL85 only.

[4] INTEGER is normally equivalent to INTEGER\*2. The INTEGER\*4 and INTEGER\*8 compiler directives redefine INTEGER.

Table F-2. Integer Types, Part 2

|                                      | 64-Bit Integer                                                                         | Bit Integer of 1 to 31 Bits                                   | Decimal Integer                             |
|--------------------------------------|----------------------------------------------------------------------------------------|---------------------------------------------------------------|---------------------------------------------|
| BASIC                                | INT(64)<br>FIXED(0)                                                                    | —                                                             | —                                           |
| C                                    | long long                                                                              | —                                                             | —                                           |
| COBOL                                | PIC S9(n) COMP or PIC 9(n) COMP<br>without P or V, $10 \leq n \leq 18$<br>NATIVE-8 [1] | —                                                             | Numeric DISPLAY                             |
| FORTRAN                              | INTEGER*8                                                                              | —                                                             | —                                           |
| Pascal                               | INT64                                                                                  | UNSIGNED(n), $1 \leq n \leq 16$<br>INT(n), $1 \leq n \leq 16$ | DECIMAL                                     |
| SQL                                  | NUMERIC(10)...NUMERIC(18)<br>PIC 9(10) COMP...PIC 9(18) COMP<br>INTEGER                | —                                                             | DECIMAL (n,s)<br>PIC 9(n) DISPLAY           |
| TAL                                  | FIXED(0)                                                                               | UNSIGNED(n), $1 \leq n \leq 31$                               | —                                           |
| <b>Return Value<br/>Size (Words)</b> | 4                                                                                      | 1, 1 or 2 in TAL                                              | 1 or 2, depends on<br>declared pointer size |

[1] Tandem COBOL85 only.

Table F-3. Floating, Fixed, and Complex Types

|                                      | 32-Bit Floating | 64-Bit Floating  | 64-Bit Fixed Point                                                    | 64-Bit Complex |
|--------------------------------------|-----------------|------------------|-----------------------------------------------------------------------|----------------|
| BASIC                                | REAL            | REAL(64)         | FIXED(s), $0 \leq s \leq 18$                                          | —              |
| C                                    | float           | double           | —                                                                     | —              |
| COBOL                                | —               | —                | PIC S9(n-s)v9(s) COMP or<br>PIC 9(n-s)v9(s) COMP, $10 \leq n \leq 18$ | —              |
| FORTRAN                              | REAL            | DOUBLE PRECISION | —                                                                     | COMPLEX        |
| Pascal                               | REAL            | LONGREAL         | —                                                                     | —              |
| SQL                                  | —               | —                | NUMERIC (n,s)<br>PIC 9(n-s)v9(s) COMP                                 | —              |
| TAL                                  | REAL            | REAL(64)         | FIXED(s), $-19 \leq s \leq 19$                                        | —              |
| <b>Return Value<br/>Size (Words)</b> | 2               | 4                | 4                                                                     | 4              |

**Table F-4. Character Types**

|                                  | Character                                                                                                              | Character String                                                                       | Varying Length Character String                              |
|----------------------------------|------------------------------------------------------------------------------------------------------------------------|----------------------------------------------------------------------------------------|--------------------------------------------------------------|
| BASIC                            | STRING                                                                                                                 | STRING                                                                                 | —                                                            |
| C                                | signed char<br>unsigned char                                                                                           | pointer to char                                                                        | struct {<br>int len;<br>char val [n]<br>};                   |
| COBOL                            | Alphabetic<br>Numeric DISPLAY<br>Alphanumeric-Edited<br>Alphanumeric<br>Numeric-Edited                                 | Alphabetic<br>Numeric DISPLAY<br>Alphanumeric-Edited<br>Alphanumeric<br>Numeric-Edited | 01 name.<br>03 len USAGE IS NATIVE-2 [1]<br>03 val PIC X(n). |
| FORTRAN                          | CHARACTER                                                                                                              | CHARACTER array<br>CHARACTER*n                                                         | —                                                            |
| Pascal                           | CHAR or BYTE value parameter<br>Enumeration, unpacked, ≤ 256 members<br>Subrange, unpacked n...m,<br>0 ≤ n and m ≤ 255 | PACKED ARRAY OF CHAR<br>FSTRING(n)                                                     | STRING(n)                                                    |
| SQL                              | PIC X<br>CHAR                                                                                                          | CHAR(n)<br>PIC X(n)                                                                    | VARCHAR(n)                                                   |
| TAL                              | STRING                                                                                                                 | STRING array                                                                           | —                                                            |
| <b>Return Value Size (Words)</b> | 1                                                                                                                      | 1 or 2, depends on declared pointer size                                               | 1 or 2, depends on declared pointer size                     |

[1] Tandem COBOL85 only.

**Table F-5. Structured, Logical, Set, and File Types**

|                                  | Byte-Addressed Structure                    | Word-Addressed Structure                    | Logical (true or false)                  | Boolean | Set | File |
|----------------------------------|---------------------------------------------|---------------------------------------------|------------------------------------------|---------|-----|------|
| BASIC                            | —                                           | MAP buffer                                  | —                                        | —       | —   | —    |
| C                                | —                                           | struct                                      | —                                        | —       | —   | —    |
| COBOL                            | —                                           | 01-level RECORD                             | —                                        | —       | —   | —    |
| FORTRAN                          | RECORD                                      | —                                           | LOGICAL [1]                              | —       | —   | —    |
| Pascal                           | RECORD, byte-aligned                        | RECORD, word-aligned                        | —                                        | BOOLEAN | Set | File |
| SQL                              | —                                           | —                                           | —                                        | —       | —   | —    |
| TAL                              | Byte-addressed standard<br>STRUCT pointer   | Word-addressed standard<br>STRUCT pointer   | —                                        | —       | —   | —    |
| <b>Return Value Size (Words)</b> | 1 or 2, depends on<br>declared pointer size | 1 or 2, depends on<br>declared pointer size | 1 or 2, depends on<br>compiler directive | 1       | 1   | 1    |

[1] LOGICAL is normally defined as 2 bytes. The LOGICAL\*2 and LOGICAL\*4 compiler directives redefine LOGICAL.

Table F-6. Pointer Types

|                                      | Procedure Pointer                           | Byte Pointer                                | Word Pointer                                | Extended Pointer                            |
|--------------------------------------|---------------------------------------------|---------------------------------------------|---------------------------------------------|---------------------------------------------|
| BASIC                                | —                                           | —                                           | —                                           | —                                           |
| C                                    | function pointer                            | byte pointer                                | word pointer                                | extended pointer                            |
| COBOL                                | —                                           | —                                           | —                                           | —                                           |
| FORTRAN                              | —                                           | —                                           | —                                           | —                                           |
| Pascal                               | Procedure pointer                           | Pointer, byte-addressed<br>BYTEADDR         | Pointer, byte-addressed<br>WORDADDR         | Pointer, extended-addressed<br>EXTADDR      |
| SQL                                  | —                                           | —                                           | —                                           | —                                           |
| TAL                                  | —                                           | 16-bit pointer,<br>byte-addressed           | 16-bit pointer,<br>word-addressed           | 32-bit pointer                              |
| <b>Return Value<br/>Size (Words)</b> | 1 or 2, depends on<br>declared pointer size | 1 or 2, depends on<br>declared pointer size | 1 or 2, depends on<br>declared pointer size | 1 or 2, depends on declared<br>pointer size |

---

# Glossary

---

**actual parameter.** An argument that a calling procedure or subprocedure passes to a called procedure or subprocedure.

**addressing mode.** The mode in which a variable is to be accessed—direct addressing, standard indirect addressing, or extended indirect addressing—as specified in the data declaration.

**AND.** A Boolean operator that produces a true state if both adjacent conditions are true.

**arithmetic expression.** An expression that computes a single numeric value.

**array.** A variable that represents a collectively stored set of elements of the same data type.

**ASSERT statement.** A statement that conditionally calls an error-handling procedure.

**assignment expression.** An expression that stores a value in a variable.

**assignment statement.** A statement that stores a value in a variable.

**ASSIGN Command.** A TACL command that lets you associate a logical file name with a Tandem file name. The Tandem file name is a fully qualified file ID. See “file name” and “file ID.”

**ASSIGN SSV Command.** A TACL command that lets you specify the D-series node (or C-series system), volume, and subvolume from which the compiler is to resolve incomplete file names specified in SEARCH, SOURCE, and USEGLOBALS directives.

**automatic extended data segment.** A segment that is automatically allocated by the compiler when you declare extended indirect arrays or extended indirect structures.

**Binder.** A stand-alone binder you can use to bind separately compiled object files (or modules) into a new object file.

**BINSERV.** A binder that is integrated with the TAL compiler.

**bit deposit.** The assignment of a value to a bit field in a previously allocated STRING or INT variable, but not in an UNSIGNED(1–16) variable. A bit-deposit field has the form  $\langle n \rangle$  or  $\langle n:n \rangle$ .

**bit extraction.** The access of a bit field in an INT expression (which can include STRING, INT, or UNSIGNED(1–16) variables). A bit-extraction field has the form  $\langle n \rangle$  or  $\langle n:n \rangle$ .

**bit field.** One of the following units:

- An  $n$ -bit storage unit that is allocated for a variable of the UNSIGNED data type. For an UNSIGNED simple variable, the bit field can be 1 to 31 bits wide. For an UNSIGNED array element, the bit field can be 1, 2, 4, or 8 bits wide.
- A bit field in the form  $\langle n \rangle$  or  $\langle n:n \rangle$ , used in bit-deposit or bit-extraction operations.

**bit shift.** The shifting of bits within an INT or INT(32) expression a specified number of positions to the left or right. An INT expression can consist of STRING, INT, or UNSIGNED(1–16) values. An INT(32) expression can consist of INT(32) or UNSIGNED(17–31) values.

**bit-shift operators.** Unsigned ('<<', '>>') or signed (<<, >>) operators that left shift or right shift a bit field within an INT or INT(32) expression.

**bitwise logical operator.** The LOR, LAND, or XOR operator, which performs a bit-by-bit operation on INT expressions.

**blocked global data.** Data you declare within BLOCK declarations. See “BLOCK declaration.”

**BLOCK declaration.** A means by which you can group global data declarations into a relocatable data block that is either shareable with all compilation units in a program or private to the current compilation unit.

**Boolean operator.** The NOT, OR, or AND operator, which sets the state of a single value or the relationship between two values.

**breakpoint.** A location in a program at which execution is suspended so that you can examine and modify the program state. Breakpoints are set by Inspect or Debug commands.

**built-in function.** See “standard function.”

**byte.** An 8-bit storage unit; the smallest addressable unit of memory.

**CALL statement.** A statement that invokes a procedure or a subprocedure.

**CALLABLE procedure.** A procedure you declare using the CALLABLE keyword; a procedure that can call a PRIV procedure. (A PRIV procedure can execute privileged instructions.)

**CASE expression.** An expression that selects an expression based on a selector value.

**CASE statement.** A statement that selects a statement based on a selector value.

**central processing unit.** See “CPU.”

**character string constant.** A string of one or more ASCII characters that you enclose within quotation mark delimiters. Also referred to as a character string.

**CISC.** Complex instruction set computing. A processor architecture based on a large instruction set, characterized by numerous addressing modes, multicycle machine instructions, and many special-purpose instructions. Contrast with “RISC.”

**CLUDECS.** A file, provided by the CRE, that contains external declarations for CLULIB functions. See also “CLULIB.”

**CLULIB.** A library file, provided by the CRE, that contains Saved Messages Utility (SMU) functions for manipulating saved startup, ASSIGN, and PARAM messages.

**code segment.** A segment that contains program instructions to be executed, plus related information. Applications can read code segments but cannot write to them.

**code space.** A part of virtual memory that is reserved for user code, user library code, system code, and system library code. The current code space of your process consists of an optional library code space and a user code space.

**CODE statement.** A statement that specifies machine codes or constants for inclusion in the object code.

**comment.** A note that you insert into the source code to describe a construct or operation in your source code. The compiler ignores comments during compilation. A comment must either:

- Begin with two hyphens (--) and terminate with the end of the line
- Begin with an exclamation point (!) and terminate with either another exclamation point or the end of the line

**Common Run-Time Environment.** See "CRE."

**compilation unit.** A source file plus source code that is read in from other source files by SOURCE directives, which together compose a single input to the compiler.

**compiler directive.** A compiler option that lets you control compilation, compiler listings, and object code generation. For example, compiler directives let you compile parts of the source file conditionally or suppress parts of a compiler listing.

**compiler listing.** The listing produced by the compiler after successful compilation. A compiler listing can include a header, banner, warning and error messages, source listing, maps, cross-references, and compilation statistics.

**completion code.** A value used to return information about a process to its caller when the process completes execution. For example, the compiler returns to the TACL product a completion code indicating the status of the compilation.

**complex instruction set computing.** See "CISC."

**condition.** An operand that represents a true or false state.

**condition code.** A status returned by expressions and by some file system procedure calls as follows:

| Condition Code | Meaning                     | Expression Status | Procedure Call Status |
|----------------|-----------------------------|-------------------|-----------------------|
| CCG            | Condition-code-greater-than | Positive          | Warning               |
| CCL            | Condition-code-less-than    | 0                 | Error                 |
| CCE            | Condition-code-equal-to     | Negative          | Successful execution  |

**conditional expression.** An expression that establishes the relationship between values and results in a true or false value; an expression that consists of relational or Boolean conditions and conditional operators.

**constant.** A number or a character string.

**constant expression.** An arithmetic expression that contains only constants, LITERALS, and DEFINES as operands.

**CPU.** Central processing unit. Historically, the main data processing unit of a computer. A Tandem system has multiple cooperating processors rather than a single CPU; processors are sometimes called CPUs.

**CRE.** Common Run-Time Environment. Services that facilitate D-series mixed-language programs.

**CREDECS.** A file, provided by the CRE, that contains external declarations for CRELIB functions whose names begin with CRE\_. See also "CRELIB."

**CRELIB.** A library file, provided by the CRE, that contains functions for sharing files, manipulating \$RECEIVE, terminating the CRE, and performing standard math functions and other tasks.

**Crossref.** A stand-alone product that collects cross-reference information for your program.

**CROSSREF.** A compiler directive that collects cross-reference information for your program.

**cross-references.** Source-level cross-reference information produced for your program by the CROSSREF compiler directive or the Crossref stand-alone product.

**C-series system.** A system that is running a C-series release version of the Guardian 90 operating system.

**data declaration.** A means by which to allocate storage for a variable and to associate an identifier with a variable, a DEFINE, or a LITERAL.

**data segment.** A segment that contains information to be processed by the instructions in the related code segment. Applications can read and write to data segments. Data segments contain no executable instructions.

**data space.** The area of virtual memory that is reserved for user data and system data. The current data space of your process consists of a user data segment, an automatic extended data segment if needed, and any user-defined extended data segments.

**data stack.** The local and sublocal storage areas of the user data segment.

**data type.** A part of a variable declaration that determines the kind of values the variable can represent, the operations you can perform on the variable, and the amount of storage to allocate. TAL data types are STRING, INT, INT(32), UNSIGNED, FIXED, REAL, and REAL(64).

**data type alias.** An alternate way to specify INT, REAL, and FIXED(0) data types. The respective aliases are INT(16), REAL(32), and INT(64).

**Debug.** A machine-level interactive debugger.

**DEFINE command.** A TACL command that lets you specify a named set of attributes and values to pass to a process.

**DEFINE.** A TAL declaration that associates an identifier with text such as a sequence of statements.

**definition structure.** A declaration that describes a structure layout and allocates storage for the structure layout. Contrast with “referral structure” and “template structure.”

**dereferencing operator.** A period (.) prefixed to an INT simple variable, which causes the content of the variable to become the standard word address of another data item.

**direct addressing.** Data access that requires only one memory reference and that is relative to the base of the global, local, or sublocal area of the user data segment.

**directive.** See “compiler directive.”

**DO statement.** A statement that executes a posttest loop until a true condition occurs.

**doubleword.** A 32-bit storage unit for the INT(32) or REAL data type.

**DROP statement.** A statement that frees a reserved index register or removes a label from the symbol table.

**D-series system.** A system that is running a D-series release version of the operating system.

**entry point.** An identifier by which a procedure can be invoked. The primary entry point is the procedure identifier specified in the procedure declaration. Secondary entry points are identifiers specified in entry-point declarations.

**entry-point declaration.** A declaration within a procedure that provides a secondary entry point by which that procedure can be invoked. The primary entry point is the procedure identifier specified in the procedure declaration.

**environment register.** A facility that contains information about the current process, such as the current RP value and whether traps are enabled.

**equivalenced variable.** A declaration that associates an alternate identifier and description with a location in a primary storage area.

**expression.** A sequence of operands and operators that, when evaluated, produces a single value.

**EXTDECS.** A file, provided by the operating system, that contains external declarations for system procedures. System procedures, for example, manage files, activate and terminate programs, and monitor the operations of processes.

**extended data segment.** A segment that provides up to 127.5 megabytes of indirect data storage. A process can have more than one extended data segment:

- The compiler allocates an extended data segment when you declare extended indirect arrays or indirect structures.
- Your process can also allocate explicit extended data segments.

**extended indirect addressing.** Data access through an extended (32-bit) pointer.

**extended pointer.** A 32-bit simple pointer or structure pointer. An extended pointer can contain a 32-bit byte address of any location in virtual memory.

**extended stack.** A data block named \$EXTENDED#STACK that is created in the automatic extended data segment by the compiler when you declare extended indirect arrays and structures.

**EXTENSIBLE procedure.** A procedure that you declare using the EXTENSIBLE keyword; a procedure to which you can add formal parameters without recompiling its callers; a procedure for which the compiler considers all parameters to be optional, even if some are required by your program. Contrast with “VARIABLE procedure.”

**external declarations file.** A file that contains declarations for procedures declared in other source files.

**EXTERNAL procedure declaration.** A procedure declaration that includes the EXTERNAL keyword and no procedure body; a declaration that enables you to call a procedure that is declared in another source file.

**file ID.** The last of the four parts of a file name.

**file name.** A fully qualified file ID. A file name contains four parts separated by periods:

- Node name (system name)
- Volume name
- Subvolume name
- File ID

**file system.** A set of operating system procedures and data structures that allows communication between a process and a file, which can be a disk file, a device, or a process.

**filler bit.** A declaration that allocates a bit place holder for data or unused space in a structure.

**filler byte.** A declaration that allocates a byte place holder for data or unused space in a structure.

**FIXED.** A data type that requires a quadrupleword of storage and that can represent a 64-bit fixed-point number.

**FOR statement.** A statement that executes a pretest loop *n* times.

**formal parameter.** A specification, within a procedure or subprocedure, of an argument that is provided by the calling procedure or subprocedure.

**FORWARD procedure declaration.** A procedure declaration that includes the FORWARD keyword but no procedure body; a declaration that allows you to call a procedure before you declare the procedure body.

**fpoint.** An integer in the range -19 through 19 that specifies the implied decimal point position in a FIXED value. A positive fpoint denotes the number of decimal places to the right of the decimal point. A negative fpoint denotes the number of integer places to the left of the decimal point; that is, the number of integer digits to replace with zeros leftward from the decimal point.

**function.** A procedure or subprocedure that returns a value to the calling procedure or subprocedure.

**global data.** Data declarations that appear before the first procedure declaration; identifiers that are accessible to all compilation units in a program, unless the data declarations appear in a BLOCK declaration that includes the PRIVATE keyword.

**GOTO statement.** A statement that unconditionally branches to a label within a procedure or subprocedure.

**group comparison expression.** An expression that compares a variable with another variable or with a constant.

**high PIN.** A process identification number (PIN) that is greater than 255. Contrast with “low PIN.”

**home terminal.** Usually the terminal from which a process was started.

**Identifier.** A name you declare for an object such as a variable, LITERAL, or procedure.

**IF expression.** An expression that selects the THEN expression for a true state or the ELSE expression for a false state.

**IF statement.** A statement that selects the THEN statement for a true state or the ELSE statement for a false state.

**implicit pointer.** A pointer the compiler provides when you declare an indirect array or indirect structure. See also “pointer.”

**index register.** Register R5, R6, or R7 of the register stack.

**index.** An element (byte, word, doubleword, or quadrupleword) offset or an occurrence offset as follows:

- Array index—an element offset from the zeroth element
- Simple pointer index —an element offset from the address stored in the pointer
- Structure or substructure index—an occurrence offset from the zeroth occurrence

**indexing.** Data access through an index appended to a variable name.

**Inspect product.** A source-level and machine-level interactive debugger.

**INITIALIZER.** A system procedure that reads and processes messages during process startup.

**instruction register.** A facility that contains the instruction currently executing the current code segment.

**INT.** A data type that requires a word of storage and that can represent one or two ASCII characters or a 16-bit integer.

**INT(16).** An alias for INT.

**INT(32).** A data type that requires a doubleword of storage and that can represent a 32-bit integer.

**INT(64).** An alias for FIXED(0).

**INTERRUPT attribute.** A procedure attribute (used only for operating system procedures) that causes the compiler to generate an IXIT (interrupt exit) instruction instead of an EXIT instruction at the end of execution.

**keyword.** A term that has a predefined meaning to the compiler.

**label.** An identifier you place before a statement for access by other statements within the encompassing procedure, usually a GOTO statement.

**labeled tape.** A magnetic tape file described by standard ANSI or IBM file labels.

**LAND.** A bitwise logical operator that performs a bitwise logical AND operation.

**LANGUAGE attribute.** A procedure attribute that lets you specify in which language (C, COBOL, FORTRAN, or Pascal) a D-series EXTERNAL procedure is written.

**large-memory-model program.** A C or Pascal program that uses 32-bit addressing and stores data in an extended data segment.

**LITERAL.** A declaration that associates an identifier with a constant.

**local data.** Data that you declare within a procedure; identifiers that are accessible only from within that procedure.

**local register.** A facility that contains the address of the beginning of the local data area for the most recently called procedure.

**logical operator.** See “bitwise logical operator.”

**LOR.** A bitwise logical operator that performs a bitwise logical OR operation.

**low PIN.** A process identification number (PIN) in the range 0 through 254. Contrast with “high PIN.”

**lower 32K-word area.** The lower half of the user data segment. The global, local, and sublocal storage areas.

**MAIN procedure.** A procedure that you declare using the MAIN keyword; the procedure that executes first when you run the program regardless of where the MAIN procedure appears in the source code.

**memory page.** A unit of virtual storage. TAL supports the 1048-byte memory page regardless of the memory-page size supported by the system hardware.

**mixed-language program.** A program that contains source files written in different Tandem programming languages.

**modular program.** A program that is divided into smaller, more manageable compilation units that you can compile separately and then bind together.

**move statement.** A statement that copies a group of elements from one location to another.

**multidimensional array.** A structure that contains nested substructures.

**NAME declaration.** A declaration that associates an identifier with a compilation unit (and with its private global data block if any).

**named data block.** A BLOCK declaration that specifies a data-block identifier. The global data declared within the BLOCK declaration is accessible to all compilation units in the program. Contrast with “private data block.”

**network.** Two or more nodes linked together for intersystem communication.

**node.** A computer system connected to one or more computer systems in a network.

**NonStop SQL.** A relational database management system that provides efficient online access to large distributed databases.

**NOT.** A Boolean operator that tests a condition for the false state and that performs Boolean negation.

**object file.** A file, generated by a compiler or binder, that contains machine instructions and other information needed to construct the executable code spaces and initial data for a process. The file can be a complete program ready for execution, or it can be incomplete and require binding with other object files before execution.

**offset.** Represents, when used in place of an index, the distance in bytes of an item from either the location of a direct variable or the location of the pointer of an indirect variable, not from the location of the data to which the pointer points. Contrast with “index.”

**operand.** A value that appears in an expression. An operand can be a constant, a variable identifier, a LITERAL identifier, or a function invocation.

**operator.** A symbol—such as an arithmetic or conditional operator—that performs a specific operation on operands.

**OR.** A Boolean operator that produces a true state if either adjacent condition is true.

**output listing.** See “compiler listing.”

**page.** See “memory page.”

**PARAM command.** A TACL command that lets you associate an ASCII value with a parameter name.

**parameter.** An argument that can be passed between procedures or subprocedures.

**parameter mask.** A means by which the compiler keeps track of which actual parameters are passed by a procedure to an EXTENSIBLE or VARIABLE procedure.

**parameter pair.** Two parameters connected by a colon that together describe a single data type to some languages.

**PIN.** A process identification number; an unsigned integer that identifies a process in a processor module.

**pointer.** A variable that contains the address of another variable. Pointers include:

- Simple pointers and structure pointers that you declare and manage
- Implicit pointers (pointers the compiler provides and manages when you declare indirect arrays and indirect structures)

See also “extended pointer” and “standard pointer.”

**precedence of operators.** The order in which the compiler evaluates operators in expressions.

**primary storage area.** The area of the user data segment that can store pointers and directly addressed variables. Contrast with “secondary storage area.”

**PRIV procedure.** A procedure you declare using the PRIV keyword; a procedure that can execute privileged instructions. Normally only operating system procedures are PRIV procedures.

**private data area.** The part of the data space that is reserved for the sole use of a procedure or subprocedure while it is executing.

**private data block.** A BLOCK declaration that specifies the PRIVATE keyword. Global data declared within such a BLOCK declaration is accessible only to procedures within the current compilation unit. Contrast with “named data block.”

**procedure.** A program unit that can contain the executable parts of a program and that is callable from anywhere in a program; a named sequence of machine instructions.

**procedure declaration.** Declaration of a program unit that can contain the executable parts of a program and that is callable from anywhere in a program. Consists of a procedure heading and either a procedure body or the keyword FORWARD or EXTERNAL.

**process.** An instance of execution of a program.

**process environment.** The software environment that exists when the processor module is executing instructions that are part of a user process or a system process.

**process identification number.** See “PIN.”

**program.** A set of instructions that a computer is capable of executing.

**program register.** A facility that contains the address of the next instruction to be executed in the current code segment.

**program structure.** The order and level at which major components such as data declarations and statements appear in a source file.

**public name.** A specification within a procedure declaration of a procedure name to use in Binder, not within the compiler. Only a D-series EXTERNAL procedure declaration can include a public name. If you do not specify a public name, the procedure identifier becomes the public name.

**quadrupleword.** A 64-bit storage unit for the REAL(64) or FIXED data type.

**read-only array.** An array that you can read but cannot modify; an array that is located in the user code segment.

**REAL.** A data type that requires a doubleword of storage and that can represent a 32-bit floating-point number.

**REAL(32).** An alias for REAL.

**REAL(64).** A data type that requires a quadrupleword of storage and that can represent a 64-bit floating-point number.

**recursion.** The ability of a procedure or subprocedure to call itself.

**redefinition.** A declaration, within a structure, that associates a new identifier and sometimes a new description with a previously declared item in the same structure.

**reduced instruction set computing.** See “RISC.”

**reference parameter.** An argument for which a calling procedure (or subprocedure) passes an address to a called procedure (or subprocedure). The called procedure or subprocedure can modify the original argument in the caller’s scope. Contrast with “value parameter.”

**referral structure.** A declaration that allocates storage for a structure whose layout is the same as the layout of a specified structure or structure pointer. Contrast with “definition structure” and “template structure.”

**register.** A facility that stores information about a running process. Registers include the program register, the instruction register, the local register, the stack register, the register stack, and the environment register.

**register stack.** A facility that contains the registers R0 through R7 for arithmetic operations, of which R5, R6, and R7 also serve as index registers.

**register pointer (RP).** An indicator that points to the top of the register stack.

**relational operator.** A signed (<, =, >, <=, >= <>) or unsigned (<', '=' ,>', '<=' ,>=' ,<>') operator that performs signed or unsigned comparison, respectively, of two operands and then returns a true or false state.

**relocatable data.** A global data block that Binder can relocate during the binding session.

**RESIDENT procedure.** A procedure you declare using the RESIDENT keyword; a procedure that remains in main memory for the duration of program execution. The operating system does not swap pages of RESIDENT code.

**RETURN statement.** A statement that returns control from a procedure or a subprocedure to the caller. From functions, the RETURN statement can return a value. As of the D20 release, RETURN can also return a condition-code value.

**RISC.** Reduced instruction set computing. 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. Contrast with “CISC.”

**RP.** Register pointer. An indicator that points to the top of the register stack.

**RSCAN statement.** A statement that scans sequential bytes, right to left, for a test character.

**RTLDECS.** A file, provided by the CRE, that contains external declarations for CRELIB functions whose names begin with RTL\_. See also “CRELIB.”

**Saved Messages Utility.** See "SMU functions."

**SCAN statement.** A statement that scans sequential bytes, left to right, for a test character.

**scope.** The set of levels—global, local, or sublocal—at which you can access each identifier.

**secondary storage area.** The part of the user data segment that stores the data of indirect arrays and structures. For standard indirection, the secondary storage area is in the user data segment. For extended indirection, the secondary storage area is in the automatic extended data segment. Contrast with "primary storage area."

**segment ID.** A number that identifies an extended data segment and that specifies the kind of extended data segment to allocate.

**signed arithmetic operators.** The following operators: + (unary plus), - (unary minus), + (binary signed addition), - (binary signed subtraction), \* (binary signed multiplication), and / (binary signed division).

**simple pointer.** A variable that contains the address of a memory location, usually of a simple variable or an array element, that you can access with this simple pointer.

**simple variable.** A variable that contains one item of a specified data type.

**small-memory-model program.** A C or Pascal program that uses 16-bit addressing, contains up to 64K bytes of data, and has a limited number of named static variables.

**SMU functions.** Saved Messages Utility (SMU) functions, provided by the CLULIB library, for manipulating saved startup, ASSIGN, and PARAM messages.

**source file.** A file that contains source text such as data declarations, statements, compiler directives, and comments. The source file, together with any source code read in from other source files by SOURCE directives, compose a compilation unit that you can compile into an object file.

**stack register.** A register that contains the address of the last allocated word in the data stack.

**STACK statement.** A statement that loads a value onto the register stack.

**standard function.** A built-in function that you can use for an operation such as type transfer or address conversion.

**standard indirect addressing.** Data access through a standard (16-bit) pointer.

**standard pointer.** A 16-bit simple pointer or structure pointer. A standard pointer can contain a 16-bit address in the user data segment.

**statement.** An executable sequence of keywords, operators, and values. A statement performs a specific action such as assigning a value to a variable or calling a procedure.

**STORE statement.** A statement that stores a value from a register stack element into a variable.

**STRING.** A data type that requires a byte or word of storage and that can represent an ASCII character or an 8-bit integer.

**structure.** A variable that can contain different kinds of variables of different data types. A definition structure, a template structure, or a referral structure.

**structure data item.** An accessible structure field declared within a structure, including a simple variable, array, substructure, simple pointer, structure pointer, or redefinition. Contrast with “structure item.”

**structure item.** Any structure field, including a structure data item, a bit filler, or a byte filler. Also see “structure data item.”

**structure pointer.** A variable that contains the address of a structure that you can access with this structure pointer.

**sublocal data.** Data that you declare within a subprocedure; identifiers that are accessible only from within that subprocedure.

**subprocedure.** A named sequence of machine instructions that is nested (declared) within a procedure and that is callable only from within that procedure.

**substructure.** A structure that is nested (declared) within a structure or substructure.

**SYMSERV.** A process, integrated with the TAL compiler, that on request provides symbol-table information to the object file for use by the Inspect and Crossref products.

**system.** The processors, memory, controllers, peripheral devices, and related components that are directly connected together by buses and interfaces to form an entity that is operated as one computer.

**system procedure.** A procedure provided by the operating system for your use. System procedures, for example, manage files, activate and terminate programs, and monitor the operations of processes.

**TAL.** Transaction Application Language. A high-level, block-structured language that works efficiently with the system hardware to provide optimal object program performance.

**TALDECS.** A file, provided by the TAL compiler, that contains external declarations for TALLIB functions. See also “TALLIB.”

**TALLIB.** A library file, provided by the TAL compiler, that contains procedures for initializing the CRE and for preparing a program for SQL statements.

**Tandem NonStop Series system.** See “TNS system.”

**Tandem NonStop Series/RISC system.** See “TNS/R system.”

**template structure.** A declaration that describes a structure layout but allocates no storage for the structure. Contrast with “definition structure” and “referral structure.”

**TNS system.** Tandem NonStop Series system. Tandem computers that are based on CISC technology. TNS processors implement the TNS instruction set.

**TNS/R system.** Tandem NonStop Series/RISC system. Tandem computers that are based on RISC technology. TNS/R processors implement the RISC instruction set and are upwardly compatible with the TNS system-level architecture.

**Transaction Application Language.** See “TAL.”

**type transfer.** The conversion of a variable from one data type to another data type.

**unblocked global data.** Global data you declare before any BLOCK declarations. Identifiers of such data are accessible to all compilation units in a program.

**UNSIGNED.** A data type that allocates storage for:

- Simple variable bit fields that are 1 to 31 bits wide
- Array element bit fields that are 1, 2, 4, or 8 bits wide

**unsigned arithmetic operators.** The following operators—'+' (unsigned addition) '-', (unsigned subtraction) '\*' (unsigned multiplication), '/' (unsigned division), and '\%' (unsigned modulo division).

**upper 32K-word area.** The upper half of the user data segment. You can use pointers to allocate this area for your data; however, if you use the CRE, the upper 32K-word area is not available for your data.

**USE statement.** A statement that reserves an index register for your use.

**user data segment.** An automatically allocated segment that provides modifiable, private storage for the variables of your process.

**value parameter.** An argument for which a procedure (or subprocedure) passes a value, rather than the address of the argument, to a called procedure (or subprocedure). The called procedure or subprocedure can modify the passed value but not the original argument in the caller's scope. Contrast with “reference parameter.”

**variable.** A symbolic representation of an item or a group of items or elements. A simple variable, array, structure, simple pointer, structure pointer, or equivalenced variable. A variable can store data that can change during program execution.

**VARIABLE procedure.** A procedure that you declare using the VARIABLE keyword; a procedure to which you can add formal parameters but then you must recompile all its callers; a procedure for which the compiler considers all parameters to be optional, even if some are required by your code. Contrast with “EXTENSIBLE procedure.”

**virtual memory.** A range of addresses that processes use to reference physical memory and disk storage.

**volume.** A disk drive; a pair of disk drives that forms a mirrored disk.

**WHILE statement.** A statement that executes a pretest loop during a true condition.

**word.** A 16-bit storage unit for the INT data type. TAL uses a 16-bit word regardless of the word size used by the system hardware.

**XOR.** A bitwise logical operator that performs a bitwise exclusive OR operation.

---

# Index

---

16-bit (standard) addressing 4-7  
16-bit (standard) pointers 9-1  
32-bit (extended) addressing 4-7  
32-bit (extended) pointers 9-1

---

## A

Accelerated object files xxviii  
Actual parameters  
    *See* Parameters  
Addition operator  
    signed 5-16  
    unsigned 5-18  
Address conversions  
    bit-shift operations 5-30  
    byte-to-word  
        simple pointer initialization 9-4  
        structure pointer initialization 9-14  
    reference parameters 11-35  
    standard-to-extended  
        simple pointer assignment 9-8  
        simple pointer initialization 9-5  
        structure pointer assignment 9-17  
        structure pointer initialization 9-16  
    word-to-byte  
        simple pointer assignment 9-8  
        simple pointer initialization 9-4  
        structure pointer initialization 9-14  
Addressability  
    arrays 7-11  
    definition structures 8-6  
    referral structures 8-9  
Addresses  
    as value parameters 11-26  
    in simple pointers 9-3  
    in simple pointers in structures 8-18  
    in structure pointers 9-13  
    in structure pointers in structures 8-20  
    of arrays, assigning 7-13  
    of procedures (PEP number) 11-52  
    of variables 5-27

- Addressing
  - definition structures 8-3
  - performance guidelines C-1
  - referral structures 8-8
  - template structures 8-7
- Addressing modes
  - byte 4-5
  - direct 4-6
  - extended (32-bit) indirect 4-7
  - indexing 4-7
  - indirect 4-6
  - standard (16-bit) indirect 4-7
  - word 4-5
- Aliases, data types 5-6
- Ampersand (&)
  - concatenated move (copy) operations 7-18
  - prefix, template block name
    - allocation 14-17
    - description 14-16
    - listing 15-14
- AND operator 5-22
- Arithmetic expressions
  - description 5-15
  - in conditional expressions 5-21
  - performance guidelines C-3
- Arithmetic operators
  - signed 5-16
  - unsigned 5-18
- Arrays
  - accessing 7-12
  - addressability 7-11
  - as reference parameters 11-30
  - as structure items 8-9
  - assignments 7-13
  - by data type 7-4
  - concatenating 7-18
  - copying 7-14
  - declaring 7-1
  - indexing 7-12
  - indirection 7-2
  - initializing 7-2

---

Arrays (continued)  
  multidimensional  
    simulated by structures 8-13  
    TAL and C guidelines 17-23  
  of arrays 8-9  
  of structures  
    definition structures 8-3  
    referral structures 8-8  
    TAL and C guidelines 17-23  
  redefinitions 8-21  
  scanning 7-19  
  storage allocation 7-8  
  TAL and C guidelines 17-19  
ASCII character set D-1  
ASSERT statement 12-17  
ASSIGN command, TACL product E-8  
ASSIGN message, saving 17-44  
ASSIGN SSV command, TACL product E-9  
Assignment expression 13-2  
Assignment statement  
  arrays 7-13  
  simple pointers 9-7  
  simple variables 6-4  
  structure items 8-34  
  structure pointers 9-16  
Asterisk  
  *See* \*  
AT keyword, BLOCK declaration 14-15

---

**B**

%B prefix, binary constants 5-7  
Banner, compiler listing 15-2  
BEGIN-END construct  
  compound statements 3-16  
  count in compiler listing 15-4  
  procedures 3-16  
  structures 3-16  
  subprocedures 3-16  
BEGINCOMPILATION directive 14-23  
BELOW keyword, BLOCK declaration 14-15  
Binary number base 5-7

- Binary-to-ASCII conversion program A-5
- Binder 14-5
- Binding object files
  - during compilation 14-6
  - modular sample program A-9
  - overview 14-5
  - run-time 14-6
  - stand-alone Binder 14-6
  - TAL and C for CRE 17-51
- BINSERV
  - action of 14-1
  - and SEARCH directive 14-12
  - binding object files 14-6
  - resolving external references 14-13
  - specifying which one E-6
- Bit extractions 5-28
- Bit fields
  - bit-extraction operations 5-28
  - manipulating in TAL and C 17-28
  - UNSIGNED and C packed 17-29
  - UNSIGNED data type 5-6
- Bit operations
  - extractions 5-28
  - shifts 5-29
  - TAL and C guidelines 17-28
- Bit shifts
  - description 5-29
  - division by powers of 2 5-30
  - multiplication by powers of 2 5-30
  - user code segment access B-9
  - word-byte address conversions 5-30
- Bitwise logical operators 5-20
- BIT\_FILLER declaration 8-16
- BLOCK declarations
  - description 14-14
  - in CREDECS declarations file 17-43
  - in program structure 3-5
  - in RTLDECS declarations file 17-43
  - mixed-language programming 17-1

---

BLOCK declarations (continued)  
  modular sample program  
    named block A-12  
    private block A-19  
  storage allocation 14-19  
Boolean operators 5-22  
Bounds, upper and lower  
  arrays 7-1  
  definition structures 8-3  
  referral structures 8-8  
Branch table form, labeled CASE statement 12-6  
BREAK command, Inspect product 16-5  
BREAK key, stopping programs 16-3  
Breakpoints, setting 16-5  
Built-in functions  
  *See* Standard functions  
BY keyword, FOR statement 12-12  
Byte 5-6  
Byte addressing 4-5  
BYTES keyword  
  group comparison expression 13-6  
  move statement 7-15

---

## C

C language  
  calling TAL 17-13  
  TAL and C guidelines 17-9  
C LANGUAGE attribute, TAL procedures 17-2  
C-series system xxviii  
CALL statement 12-19  
Calling procedures  
  C calling TAL 17-13  
  TAL calling C 17-12  
  TAL calling TAL 12-19  
Calling subprocedures 12-19  
\$CARRY function 5-26  
Carry indicator, testing 5-26  
CASE expressions  
  description 13-3  
  performance guidelines C-3

- CASE statement, labeled
  - branch table form 12-6
  - conditional test form 12-6
  - description 12-5
  - modular sample program A-20
  - with CHECK directive 12-7
  - with OPTIMIZE directive 12-7
- CCE (condition code equal to) 5-25
- CCG (condition code greater than) 5-25
- CCL (condition code less than) 5-25
- char variables, TAL and C mixed programming 17-19
- Character set D-1
- Character strings
  - format 5-10
  - initializing with 6-2
  - interlanguage correspondence F-3
  - maximum length 5-10
- CHECK directive, with labeled CASE statement 12-7
- Circumflex (^) in identifiers 5-2
- CISC systems xxviii
- CLEAR command, Inspect product 16-8
- CLUDECS (CRE external declarations file) 17-43
- CLULIB (Common Language Utility library file of CRE) 17-43
- COBOL environment, ENV directive 17-39
- COBOL LANGUAGE attribute, TAL procedures 17-2
- Code segments
  - procedures in 11-1
  - process environment 4-1
- Code space
  - description 4-1
  - items in arithmetic expressions 5-15
- Code-address field, compiler listing 15-3
- Comments
  - format 3-14
  - omitted parameters 3-14
  - skipping over code 3-15
- COMMON keyword, ENV directive 17-39
- Common Run-Time Environment
  - See CRE
- Comparing arrays 7-23
- Compilation command 14-2

- 
- Compilation statistics, compiler
    - listing 15-16
  - Compilation units
    - description 14-2
    - naming 14-14
    - order of components 3-3
    - scope of identifiers 3-3
    - structuring 3-1
  - Compiler
    - processes integrated with 14-1
    - starting 14-2
  - Compiler directives
    - See Directives
  - Compiler listing
    - banner 15-2
    - compilation statistics 15-16
    - compiler messages 15-2
    - cross-references 15-11
    - directives in 15-2
    - innerlisting 15-7
    - maps
      - file name map 15-10
      - global map 15-9
      - load map 15-13
      - local map 15-6
      - sublocal map 15-6
    - octal code 15-9
    - page header 15-1
    - procedure instruction mnemonics 15-9
      - source code 15-3
      - address field 15-3
      - BEGIN-END counter 15-4
      - edit-file line numbers 15-3
      - lexical-level counter 15-4
    - statement instruction
      - mnemonics 15-7
  - Compiling source files
    - command options 14-2
    - getting started 2-4
    - modular sample program A-9
  - Completion codes from compiler 14-5
  - Complex types, interlanguage correspondence F-3

- Compound statements 3-16
- Concatenated move (copy)
  - operations 7-18
- Condition code indicator, testing 5-25
- Conditional compilation
  - asterisk in listing 15-5
  - directives (DEFINETO, IF, ENDIF, IFNOT) 3-15
- Conditional expressions
  - assigning 5-24
  - description 5-21
- Conditional statements 12-1
- Conditional test form, labeled CASE statement 12-6
- Constant expressions 5-1
- Constant lists
  - copying into arrays 7-14
  - group comparison expressions 13-5
  - initializing arrays 7-3
  - move statement 7-14
- Constants
  - arithmetic expressions 5-15
  - group comparison expressions 13-5
  - kinds of
    - character strings 5-10
    - LITERALS 5-11
    - numbers 5-7
  - move statement 7-15
- Continuation directive lines 14-7
- Copying data
  - arrays 7-14
  - simple pointers 9-9
  - structure pointers 9-19
  - structures 8-39
- CPU, specifying for compiler E-6
- CRE guidelines
  - advantages of using CRE 17-37
  - coding guidelines 17-37
  - data blocks of 17-39
  - ENV directive 17-39
  - errors in CRE math routines 17-49
  - extended stack, support of 17-49
  - HEAP directive 17-41
  - initializing the CRE 17-44

---

CRE guidelines (continued)  
  object files, stopping 17-45  
  run-time environment, specifying 17-39  
  sample program 17-50  
  spooling 17-47  
  standard files, accessing 17-46  
  user heap, accessing 17-41  
  \$RECEIVE, accessing 17-48  
CREDECS (CRE external declarations file)  
  including in program 17-43  
  sample program 17-50  
CRELIB (CRE library file) 17-43  
#CRE\_GLOBALS (CRE control block) 17-39  
#CRE\_HEAP (CRE run-time heap) 17-39  
CRE\_TERMINATOR\_ 17-45  
Cross-references  
  collecting with CROSSREF directive 14-26  
  compiler listing 15-11  
CROSSREF directive  
  and USEGLOBALS directive 14-23  
  description 14-26

---

## D

D suffix, nonhexadecimal INT(32) numbers 5-8  
%D suffix, hexadecimal INT(32) numbers 5-8  
D-series system xxviii  
Data access  
  arrays 7-12  
  operands in expressions 5-27  
  pointers in structures 8-36  
  read-only arrays 7-24  
  simple pointers 9-9  
  structure pointers 9-18  
  structures 8-27  
Data blocks  
  CRE 17-39  
  relocatable 14-14  
Data declarations 3-4  
  *See also* Variables  
  blocked 3-5  
  global 3-5

- Data declarations (continued)
  - local 3-8
  - sublocal 3-12
  - unblocked 3-5
- Data Definition Language (DDL)
  - and byte-aligned structures 17-21
  - and type correspondence F-1
- Data sets
  - arrays 7-1
  - structures 8-1
- Data space 4-2
- Data types
  - compatibility with C data types 17-10
  - description 5-4
  - format 5-4
  - interlanguage correspondence F-2
  - of Boolean operands 5-22
  - of expressions 5-6
  - of logical arithmetic operands 5-20
  - of signed arithmetic operands 5-16
  - of signed relational results 5-23
  - of special expressions 13-1
  - of unsigned arithmetic operands 5-19
  - of unsigned relational results 5-23
  - storage units 5-6
- DATAPAGES directive, upper 32K-word area B-2
- \$DBL
  - structure pointers 9-22
  - structures 8-30
- \$DBLL, accessing user code segment B-9
- DDL
  - See Data Definition Language
- Debug product 16-4
- Debugging programs
  - debuggers 16-4
  - displaying values 16-5
  - sample session 16-6
  - setting breakpoints 16-5
  - stepping through 16-5
- Decimal number base 5-7

---

Declarations  
arrays 7-1  
BLOCKs 14-14  
equivalenced variables 10-1  
functions 11-8  
labels 11-48  
LITERALs 5-11  
NAME 14-14  
procedures 11-2  
simple pointers 9-2  
simple variables 6-1  
structure pointers 9-12  
structures 8-3  
subprocedures 11-15  
DEFAULT DEFINE E-6  
DEFINE command, TACL product E-4  
DEFINEPOOL system procedure  
extended data segment B-16  
upper 32K-word area B-7  
DEFINETOg directive 3-15  
Definition structures  
addressability 8-6  
as reference parameters 11-31  
declaring 8-3  
equivalenced 10-10  
storage allocation 8-4  
Definition substructures  
declaring 8-12  
redefinitions 8-23  
storage allocation 8-14  
Dereferencing operator 5-27  
Direct addressing 4-6  
Directive lines 14-7  
Directive stacks, pushing and popping 14-8  
Directives  
BEGINCOMPILATION 14-23  
compiler listing 15-2  
CROSSREF 14-26  
DATAPAGES B-2  
DEFINETOg 3-15  
ENDIF 3-15  
ENV 17-39

Directives (continued)  
  file names 14-9  
  HEAP 17-41  
  IF 3-15  
  IFNOT 3-15  
  overview of 3-18  
  SAVEGLOBALS 14-23  
  SEARCH 14-12, 14-23  
  SOURCE 14-10  
  specifying in compilation command 14-4  
  specifying in source files 14-7  
  USEGLOBALS 14-23  
Disk file names E-1  
DISPLAY command, Inspect product 16-5  
Displaying program values 16-5  
Division by powers of two 5-30  
Division operator  
  signed 5-16  
  unsigned 5-18  
  unsigned modulo 5-18  
DO keyword  
  DO statement 12-10  
  FOR statement 12-12  
  WHILE statement 12-8  
DO statement 12-10  
Documenting source code  
  *See* Comments  
Doubleword 5-6  
DOWNT0 keyword, FOR statement 12-12  
DROP statement with FOR statement 12-15

---

## E

E register 4-5  
E suffix, REAL numbers 5-9  
Edit-file line numbers, compiler listing 15-3  
ELEMENTS keyword  
  group comparison expression 13-7  
  move statement 7-16  
Ellipsis (...) in labeled CASE statement 12-5

- 
- ELSE keyword
    - IF expression 13-4
    - IF statement 12-2
  - ENDIF directive 3-15
  - Entry points
    - entry-point identifiers
      - procedures 3-7
      - subprocedures 3-11
    - procedure identifiers 3-7
    - recorded in PEP and XEP tables 11-1
    - subprocedure identifiers 3-11
  - Enumeration variables in C 17-27
  - ENV directive 17-39
  - Environment register 4-5
  - Equivalenced variables
    - definition structures 10-10
    - indexing 10-19
    - kinds of 10-1
    - referral structures 10-14
    - simple pointers 10-6
    - simple variables 10-2
    - structure pointers 10-16
  - Error handling
    - ASSERT statement 12-17
    - ASSERTION directive 12-17
    - CRE math routine errors 17-49
    - file-system errors 5-25
    - hardware indicators, testing 5-25
    - sample program
      - input file module A-15
      - output file module A-18
  - Errors, run-time 16-3
  - ESE instruction, EXTENSIBLE procedures 11-47
  - Executing programs and object files
    - See Running object files
  - Exponents
    - REAL numbers 5-9
    - REAL(64) numbers 5-9

## Expressions

- arithmetic expression 5-15
- assignment expression 13-2
- Boolean operations 5-22
- CASE expression 13-3
- conditional expression 5-21
- constant expression 5-1
- data types 5-6
- description 5-1
- fixed-point, scaling of 5-17
- group comparison expression 13-5
- IF expression 13-4
- logical operations 5-20
- precedence of operators 5-13
- relational operations 5-23
- .EXT (extended indirection symbol)
  - arrays 7-1
  - definition structures 8-3
  - referral structures 8-8
  - simple pointers 9-2
  - structure pointers 9-12
- EXTDECS (system external declarations file) 14-11
- Extended data segments 4-2
  - automatic
    - allocation 4-10
    - organization of 4-4
    - TAL and C guidelines 17-33
  - explicit
    - accessing data B-14
    - creating B-10
    - managing B-13, B-21
    - TAL and C guidelines 17-33
  - size of 4-4
- Extended indirection
  - description 4-7
  - TAL and C guidelines 17-18
- Extended pointers
  - accessing data
    - simple pointers 9-10
    - structure pointers 9-22
  - accessing explicit extended data segments B-10, B-14
  - accessing user code segment B-9

---

Extended pointers (continued)  
  declaring  
    simple pointers 9-2  
    structure pointers 9-12  
  description 9-1  
  format B-1  
  performance guidelines C-1  
Extended relocatable data blocks 14-17  
Extended stack  
  organization of 4-4  
  pointers (#SX, #MX) 4-11  
  storage allocation 4-11  
\$EXTENDED#STACK 4-11  
EXTENDED#STACK#POINTERS 4-11  
EXTENSIBLE procedures  
  checking for parameters (\$PARAM) 11-10  
  converting from VARIABLE 11-13  
  declaring 11-10  
  parameter area 11-45  
  parameter masks 11-42  
  passing as parameters 11-25  
  procedure entry sequence 11-47  
External declarations (SOURCE directive) 14-11  
External Entry Point table 11-1  
EXTERNAL procedure declarations 11-9  
External references (SEARCH directive) 14-12

---

## F

F suffix, FIXED numbers 5-9  
%F suffix, FIXED numbers 5-9  
File ID E-2  
File name map, compiler listing 15-10  
File names  
  defaults, specifying E-4  
  directives that accept 14-9  
  disk E-1  
  incomplete, resolving E-9  
  internal E-4

- File names (continued)
  - logical
    - directives that accept 14-9
    - TACL ASSIGN command E-8
    - TACL DEFINE command E-4
- File records
  - See Structures
- File type, interlanguage correspondence F-4
- File-system errors, testing for 5-25
- FILLER declaration 8-16
- FIXED data type
  - arrays 7-6
  - fpoint 6-11
  - numeric constants 5-9
  - range of values allowed 5-4
  - scaling of operands 5-17
  - simple variables 6-11
- FIXED parameter type
  - description 11-20
  - reference parameters 11-32
  - value parameters 11-22
- FIXED(\*) data type
  - arrays 7-6
  - range of values allowed 5-4
  - simple variables 6-11
- FIXED(\*) parameter type 11-22
- Fixed-point
  - arithmetic 5-17
  - implied setting 5-5
  - interlanguage correspondence F-3
  - numbers
    - description 5-9
    - ranges by data type 5-4
  - scaling 5-17
  - setting (fpoint) 6-11
- Floating-point
  - interlanguage correspondence F-3
  - numbers
    - description 5-9
    - ranges by data type 5-4

FOR keyword  
  FOR statement 12-12  
  group comparison expression 13-6  
  move statement 7-15  
FOR loops  
  *See* FOR statement  
FOR statement  
  description 12-12  
  optimized 12-15  
  standard 12-13  
Formal parameters  
  *See* Parameters  
Format of programs 3-13  
FORTRAN environment, ENV directive 17-39  
FORTRAN LANGUAGE attribute, TAL procedures 17-2  
FORWARD procedures 11-8  
fpoint  
  of reference parameters 11-32  
  of value parameters 11-22  
  positive or negative 5-5  
  scaling in expressions 5-17  
  specifying 6-11  
Fractions  
  FIXED numbers 5-9  
  REAL numbers 5-9  
  REAL(64) numbers 5-9  
Functions  
  declaring and calling 11-8  
  in arithmetic expressions 5-15  
  RETURN statement 12-21  
  standard, by categories 5-12  
Future software platforms xxviii

---

## G

GETPOOL system procedure  
  extended data segment B-16  
  upper 32K-word area B-7  
Getting started 2-1

- #GLOBAL
  - allocation 14-17
  - description 14-16
  - listing 15-14
- \$#GLOBAL
  - description 14-17
  - listing 15-14
- .#GLOBAL
  - description 14-17
  - listing 15-14
- Global data
  - declaring 3-5
  - scope of 3-3
  - sharing C data with TAL modules
    - using BLOCK declarations 17-18
    - using pointers 17-16
  - sharing TAL data with C modules
    - using pointers 17-15
- Global data blocks
  - and SECTION directive 14-20
  - and SOURCE directive 14-20
  - declaring 14-14
  - sharing among source files 14-20
  - specifying locations of 14-15
  - storage allocation 14-16
- Global data declarations
  - blocked 14-14
  - saving and retrieving 14-23
  - unblocked 14-16
- Global map, compiler listing 15-9
- Global scope
  - data 3-5
  - data blocks, relocatable 3-5
  - identifiers 3-3
  - procedure entry points 3-7
  - procedures 3-6
- Global storage area
  - extended data segment 4-4
  - user data segment 4-9

---

GOTO statement  
  description 12-24  
  local (with local labels) 11-48  
  sublocal (with local labels) 11-49  
  sublocal (with sublocal labels) 11-50  
Group comparison expressions  
  description 13-5  
  in conditional expressions 5-21

---

## H

%H prefix, hexadecimal constants 5-7  
Hardware indicators, testing 5-25  
HEAP directive 17-41  
#HEAP (CRE user heap) 17-41  
Hexadecimal number base 5-7

---

## I

I register 4-5  
I/O  
  *See* Input/output  
Identifiers  
  rules for specifying 5-2  
  scope of 3-3  
  structure pointers, qualifying 9-18  
  structures, qualifying 8-27  
  TAL and C guidelines 17-9  
IF directive 3-15  
IF expressions  
  description 13-4  
  modular sample program A-16  
  performance guidelines C-3  
IF statement  
  description 12-2  
  sample program  
    binary-to-ASCII conversion A-6  
    input file module A-15  
    mainline module A-10  
    message module A-20  
    string entry A-3  
IFNOT directive 3-15  
Implicit pointers 4-6

- Improving performance
  - addressing C-1
  - arithmetic expressions C-3
  - general guidelines C-1
  - indexing C-2
  - STACK and STORE statements C-2
- IN file
  - compilation command 14-2
  - RUN command 16-1
- Index registers, as value parameters 11-27
- Indexing
  - arrays 7-12
  - description 4-7
  - equivalenced variables 10-19
  - performance guidelines C-2
  - simple pointers 9-10
  - structure pointers 9-20, 9-22
  - structures 8-28, 8-30
  - upper 32K-word area B-8
- Indirection
  - arrays 7-2
  - definition structures 8-3
  - description 4-6
  - TAL and C guidelines 17-18
- INHIBITXX directive
  - and indexing
    - structure pointers 9-23
    - structures 8-30
  - and relocatable data blocks 14-21
  - with USEGLOBALS 14-25
- Initialization module, sample program A-12
- Initializations
  - arrays 7-2
  - simple pointers 9-3
  - simple variables 6-2
  - structure pointers 9-13
- INNERLIST compiler listing 15-7
- Input file module, sample program A-14

---

Input/output sample programs  
  input file module A-14  
  output file module A-17  
  string display A-1  
  string entry A-3

Inspect product  
  and SYMBOLS directive 16-4  
  commands  
    BREAK 16-5  
    CLEAR 16-8  
    DISPLAY 16-5  
    RESUME 16-8  
    STEP 16-5  
    STOP 16-5  
  overview 16-4  
  sample session 16-6  
  specifying 16-4  
  starting 16-4  
  stopping 16-5

Instruction register 4-5

INT attribute, structure pointers 9-13

INT data type  
  arrays 7-5  
  numeric constants 5-8  
  range of values allowed 5-4  
  simple variables 6-6

INT parameter type 11-20

INT(16), alias of INT 5-4, 5-6

INT(32) data type  
  arrays 7-5  
  numeric constants 5-8  
  range of values allowed 5-4  
  simple variables 6-8

INT(32) parameter type 11-20

INT(64), alias of FIXED(0) 5-4, 5-6

INT32INDEX directive  
  extended indexing  
    structure pointers 9-24  
    structures 8-32  
  with USEGLOBALS 14-25

Integers

- by data type 5-8
  - interlanguage correspondence F-2
- Interface declaration in C 17-13
- Interlanguage type correspondence
- character strings F-3
  - complex F-3
  - files F-4
  - fixed F-3
  - floating F-3
  - integers F-2
  - logical F-4
  - pointers F-5
  - sets F-4
  - structures F-4
- Internal file names E-4
- 

K

- Keywords 5-3
- 

L

- L register 4-5
- L suffix, REAL(64) numbers 5-9
- Labeled CASE statement
- See CASE statement, labeled
- Labels
- declaring and using 11-48
  - local (with local GOTOs) 11-48
  - local (with sublocal GOTOs) 11-49
  - local (with sublocal variables) 11-49
  - local scope 3-8
  - sublocal (with sublocal GOTOs) 11-50
  - sublocal scope 3-12
  - undeclared 11-51
- LAND operator 5-20
- LANGUAGE attribute, procedures 17-2
- Layout
- definition structures 8-3
  - structures 8-2
  - substructures 8-12
  - template structures 8-7

---

Left shifts, bit 5-29  
Lexical-level counter, compiler listing 15-4  
LIB option, RUN command 16-1  
Library code space 4-1  
LIBRARY directive 14-6  
Library file, binding 14-6  
Limitations  
    *See* Size  
Line length, maximum 3-13  
Listings  
    *See* Compiler listing  
LITERALS  
    declaring 5-11  
    in arithmetic expressions 5-15  
    modular sample program A-19  
    with C enumeration variables 17-27  
Load map, compiler listing 15-13  
Local data  
    accessing 11-4  
    declaring 3-8  
    scope of 3-3  
Local map, compiler listing 15-6  
Local register 4-5  
Local scope  
    data 3-8  
    identifiers 3-3  
    labels 3-8  
    statements 3-8  
    subprocedure entry points 3-11  
    subprocedures 3-10  
Local storage area 4-9  
Logical file names  
    in directives 14-9  
    TACL ASSIGN command E-8  
    TACL DEFINE command E-4  
Logical operators 5-20  
Logical type, interlanguage correspondence F-4  
LOR operator 5-20  
Lower 32K-word area 4-2

**M**

- MAIN procedures 11-7
- Mainline module, sample program A-10
- Manuals
  - program development xxx
  - programming xxx
  - system xxix
- MAP DEFINE command E-5
- #MCB (CRE master control block) 17-39
- MEM option, RUN command 16-2
- Memory models, TAL and C guidelines 17-11
- Message module, sample program A-19
- Minus operator
  - binary signed 5-16
  - binary unsigned 5-18
  - unary 5-16
- Mixed-language programming
  - BLOCK declarations 17-1
  - CRE guidelines 17-37
  - LANGUAGE attribute, procedures 17-2
  - NAME declarations 17-1
  - parameter pairs 17-6
  - procedure public name 17-3
  - routines as parameters 17-4
  - TAL and C guidelines 17-9
- Modular programming
  - compiling with saved global data 14-23
  - declaring relocatable data blocks 14-14
  - description 3-1
  - sample program A-7
- Modulo division operator 5-18
- Move statement
  - arrays 7-14
  - sample program
    - binary-to-ASCII conversion A-6
    - mainline module A-11
    - string display A-1
    - string entry A-3
  - simple pointers 9-9
  - structure pointers 9-19
  - structures 8-39

---

Multidimensional arrays 8-13  
Multiplication by powers of two 5-30  
Multiplication operator  
    signed 5-16  
    unsigned 5-18  
#MX (extended stack pointer) 4-11

---

## N

NAME declarations  
    description 14-14  
    in program structure 3-5  
    mixed-language programming 17-1  
    sample program  
        CRE 17-50  
        mainline module A-10  
Named data blocks  
    *See* NAME declarations  
NEUTRAL keyword, ENV directive 17-39  
Next address  
    group comparison expression 13-7  
    move statement 7-17  
NOCROSSREF directive 14-26  
Node name in file names E-2  
NOLIST directive and SOURCE 14-10  
NOT operator 5-22  
NOWAIT option, RUN command 16-2  
Null statements 3-18  
Number bases, formats of 5-7  
Numbers, description by data type 5-8

---

## O

Object files  
    binding 14-5  
    description 14-1  
    generating 14-2  
    running  
        getting started 2-5  
        with run-time options 16-1  
    stopping  
        CRE\_TERMINATOR\_ 17-45  
        TACL STOP command 16-3

Object files, accelerated xxviii  
OBJECT, default target file 14-4  
Octal code, compiler listing 15-9  
Octal number base 5-7  
OF keyword  
    CASE expression 13-3  
    CASE statement, labeled 12-5  
Offsets, equivalenced variables 10-19  
OLD keyword, ENV directive 17-39  
Operands  
    accessing 5-27  
    in arithmetic expressions 5-15  
    in expressions 5-2  
Operators  
    arithmetic  
        signed 5-16  
        unsigned 5-18  
    bit-shift 5-29  
    Boolean 5-22  
    in expressions 5-13  
    logical 5-20  
    precedence of 5-13  
    relational 5-23  
OPTIMIZE directive, labeled CASE statement 12-7  
\$OPTIONAL function 11-12  
OR operator 5-22  
OTHERWISE keyword  
    CASE expression 13-3  
    CASE statement, labeled 12-5  
OUT file  
    compilation command 14-2  
    RUN command 16-1  
Output file module, sample program A-17  
Overflow  
    causes of 5-26  
    handling CRE math routine errors 17-49

---

**P**

P register 4-5  
P-relative arrays 7-24  
Page header, compiler listing 15-1

---

---

\$PARAM, checking for parameters  
    EXTENSIBLE procedures 11-10  
    VARIABLE procedures 11-9  
PARAM BINSERV command E-6  
PARAM message, saving 17-44  
PARAM SAMECPU command E-6  
PARAM SWAPVOL command E-7  
PARAM SYMSERV command E-7  
Parameter area  
    procedures 11-36  
        EXTENSIBLE 11-45  
        VARIABLE 11-40  
    subprocedures 11-36  
Parameter list  
    procedures 11-3  
    subprocedures 11-15  
Parameter masks  
    EXTENSIBLE procedures 11-42  
    procedures as parameters 11-25  
    VARIABLE procedures 11-38  
Parameter pairs  
    declaring 17-6  
    passing  
        conditionally 11-12  
        mixed-language guidelines 17-7  
        unconditionally 11-11  
Parameter types 11-20  
Parameters  
    *See also* Reference parameters  
    *See also* Value parameters  
    address conversions by compiler 11-35  
    checking with \$PARAM  
        EXTENSIBLE procedures 11-10  
        VARIABLE procedures 11-9  
    declaring  
        in procedures 11-3  
        in subprocedures 11-15  
    passing 12-19  
        conditionally 11-12  
        unconditionally 11-11  
    run-time 16-3

- Parameters (continued)
  - scope of 11-36
  - specifications 11-20
- PASCAL LANGUAGE attribute, TAL procedures 17-2
- Pascal variant parts, emulating 10-21
- Passing parameters
  - CALL statement 12-19
  - conditionally 11-12
  - unconditionally 11-11
- PEP table 11-1
- Performance guidelines
  - addressing C-1
  - arithmetic expressions C-3
  - general C-1
  - indexing C-2
  - STACK and STORE statements C-2
- Platforms, software xxviii
- Plus operator
  - binary signed 5-16
  - binary unsigned 5-18
  - unary 5-16
- Pointers
  - accessing explicit extended data segments B-10
  - accessing upper 32K-word area B-2
  - as reference parameters 11-33
  - as structure items 8-35
  - equivalencing
    - simple pointers 10-6
    - structure pointers 10-16
  - extended (32-bit) pointers 9-1
  - implicit pointers 4-6
  - interlanguage correspondence F-5
  - simple pointers 9-2
  - standard (16-bit) pointers 9-1
  - structure pointers 9-12
  - TAL and C guidelines 17-25
- POP prefix, directives 14-8
- Precedence of operators 5-13
- Primary relocatable data blocks 14-17
- Primary storage (global, local, sublocal)
  - allocation in 4-8
  - description 4-9

---

PRINTSYM directive (with USEGLOBALS) 14-25  
Private data area, as a TAL feature 1-1  
Private data blocks  
  declaring 14-15  
  in program structure 3-5  
PRIVATE keyword, BLOCK declaration 14-15  
PROC keyword  
  function declaration 11-8  
  procedure declaration 11-2  
PROC parameter type  
  description 11-23  
  mixed-language programming 17-4  
  passing C routines to TAL 17-32  
  passing TAL routines to C 17-30  
PROC(32) parameter type  
  description 11-24  
  mixed-language programming 17-5  
  passing C routines to TAL 17-32  
  passing TAL routines to C 17-30  
Procedure Entry Point table 11-1  
Procedure entry sequence 11-47  
Procedure mnemonics, compiler listing 15-9  
Procedures  
  address of (PEP number) 11-52  
  as parameters 11-22  
  *See* Value parameters  
  calling 11-2  
  compared to subprocedures 11-14  
  declaring 11-2  
  EXTENSIBLE 11-10  
  EXTERNAL 11-9  
  FORWARD 11-8  
  functions 11-8  
  LANGUAGE attribute 17-2  
  MAIN 11-7  
  parameters 11-3  
  public name 17-3  
  RETURN statement 12-21  
  scope of 3-6  
  storage allocation 11-5

Procedures (continued)  
  typed  
    *See* Functions  
  VARIABLE 11-9  
Process environment 4-1  
Processes 4-1  
Processor, specifying for compiler E-6  
Program control  
  ASSERT statement 12-17  
  CALL statement 12-19  
  CASE statement, labeled 12-5  
  conditional expressions 5-21  
  DO statement 12-10  
  FOR statement 12-12  
  GOTO statement 12-24  
  IF statement 12-2  
  labeled CASE statement 12-5  
  relational expressions 5-24  
  RETURN statement 12-21  
  WHILE statement 12-8  
Program flow, controlling 12-1  
Program register 4-5  
Programs  
  *See also* Object files  
  formatting 3-13  
  modular 3-1  
  structuring 3-1  
Public name, procedures 17-3  
PUSH prefix, directives 14-8  
PUTPOOL system procedure  
  extended data segment B-16  
  upper 32K-word area B-7

---

**Q**  
Quadrupleword 5-6  
Question mark (?) (directive line symbol) 14-7  
Quotation mark (") (character string delimiter) 5-10

---

## R

Ranges, data types 5-4

Read-only arrays 7-24

REAL data type

- arrays 7-5

- numeric constants 5-9

- range of values allowed 5-4

- simple variables 6-9

REAL parameter type 11-20

REAL(32), alias of REAL 5-4, 5-6

REAL(64) data type

- arrays 7-6

- numeric constants 5-9

- range of values allowed 5-4

- simple variables 6-10

REAL(64) parameter type 11-20

\$RECEIVE, accessing in the CRE 17-48

Records

- See* Structures

Recursion, as a TAL feature 1-2

Redefinitions

- arrays 8-21

- definition substructures 8-23

- referral substructures 8-25

- simple pointers 8-26

- simple variables 8-21

- structure pointers 8-26

- TAL and C guidelines 17-24

Reference parameters

- address conversions by compiler 11-35

- array size 11-30

- arrays 11-30

- declaring 11-3

- description 11-29

- number of structure occurrences 11-32

- pointers 11-33

- simple variables 11-29

- specifications 11-20

- storage allocation 11-29

- structures 11-31

- Referral structures
  - addressability 8-9
  - addressing 8-8
  - as reference parameters 11-32
  - declaring 8-8
  - equivalenced 10-14
  - storage allocation 8-9
- Referral substructures
  - declaring 8-15
  - redefinitions 8-25
  - storage allocation 8-15
- Register pointer 4-5
- Register stack
  - description 4-5
  - performance guidelines C-2
- Registers in process environment 4-5
- Relational expressions
  - description 5-21
  - program flow, controlling 5-24
- Relational operators
  - conditional expressions 5-21
  - group comparison expression 13-9
  - signed 5-23
  - testing condition code indicator 5-25
  - unsigned 5-23
- Relocatable data blocks
  - declaring 14-14
  - directives for 14-21
- RELOCATE directive 14-21
- Reserved keywords 5-3
- RESUME command, Inspect product 16-8
- RETURN statement
  - description 12-21
  - modular sample program A-16
- Return types
  - functions 11-8
  - interlanguage correspondence F-2
- Returning from procedures 11-7
- Right shifts, bit 5-29
- RISC systems xxviii
- Routines 1-3
- RP 4-5

---

RSCAN statement  
  arrays 7-19  
  example 7-21  
RTLDECS (CRE external declarations file) 17-43  
RUN command, TACL product  
  running object files 16-1  
  running the compiler 14-2  
Run-time environment, specifying with ENV directive 17-39  
Run-time errors 16-3  
Run-time libraries  
  CLULIB 17-43  
  CRELIB 17-43  
  TALLIB 17-43  
Run-time library, TALLIB 17-44  
RUND command, TACL product 16-4  
Running object files  
  getting started 2-5  
  with run-time options 16-1  
Running programs  
  *See* Running object files

---

## S

S register 4-5  
Saved Messages Utility routines 17-43  
SAVEGLOBALS directive 14-23  
Saving system messages 17-44  
\$SCALE, scaling FIXED values 5-17  
Scaling, FIXED values 5-17  
SCAN statement  
  arrays 7-19  
  sample program (string entry) A-3  
  simple pointers 9-9  
  structure pointers 9-18  
Scope  
  global 3-5  
  of identifiers 3-3  
  of parameters 11-36  
SEARCH directive  
  compiling with search lists 14-12  
  retrieving saved global initializations 14-23  
Search lists 14-12

- Search subvolume command E-9
- Secondary relocatable data blocks 14-17
- Secondary storage (global, local)
  - allocation in 4-8
  - description 4-10
- SECTION directive
  - and global data blocks 14-20
  - and SOURCE directive 14-10
- Section names, in SOURCE directive 14-10
- Segment specifier, extended pointer B-2
- SEGMENT\_ALLOCATE\_
  - TAL and C guidelines 17-33
  - TAL guidelines B-10
- SEGMENT\_DEALLOCATE\_
  - TAL and C guidelines 17-33
  - TAL guidelines B-10
- SEGMENT\_USE\_
  - TAL and C guidelines 17-33
  - TAL guidelines B-10
- Selector
  - CASE expression 13-3
  - CASE statement, labeled 12-5
- Semicolon 3-17
- Separate compilation
  - compiling with saved global data 14-23
  - declaring relocatable global data blocks 14-14
  - of source files 3-1
  - sample program A-7
- Set type, interlanguage correspondence F-4
- Sharing data
  - C data with TAL modules
    - using BLOCK declarations 17-18
    - using pointers 17-16
  - TAL and C general guidelines 17-15
  - TAL data with C modules
    - using BLOCK declarations 17-18
    - using pointers 17-15
- Shifts, bit 5-29

- 
- 
- Simple pointers
    - accessing data
      - assignment statement 9-7, 9-9
      - indexing 9-10
      - move and SCAN statements 9-9
      - upper 32K-word area B-2
    - addresses in 9-3
    - as structure items 8-17
    - declaring 9-2
    - equivalenced 10-6
    - redefinitions 8-26
    - storage allocation 9-6
    - @ operator 9-3, 9-7
  - Simple variables
    - accessing 6-4
    - as reference parameters 11-29
    - as structure items 8-9
    - as value parameters 11-21
    - assignments 6-4
    - by data type 6-5
    - declaring 6-1
    - dereferencing 5-27
    - equivalenced 10-2
    - initializing 6-2
    - redefinitions 8-21
    - storage allocation 6-3
  - Size
    - character strings 5-10
    - code segments 4-1
    - combined primary global data blocks 14-17
    - data types 5-4
    - definition structures 8-3
    - extended data segments 4-4
    - extended stack 4-11
    - identifiers 5-2
    - indexes
      - structure pointers 9-21
      - structures 8-30
    - parameter area
      - procedures 11-36
      - subprocedures 11-36
    - primary storage areas (global, local, sublocal) 4-9

- Size (continued)
  - procedures 11-1
  - referral structure occurrences 8-8
  - referral structures 8-8
  - secondary storage areas 4-10
  - storage units 5-6
  - sublocal storage area 11-15
  - user data segment 4-2
- SMU (Saved Messages Utility) routines 17-43
- Software platforms, future xxviii
- Source code, compiler listing 15-3
- SOURCE directive
  - and global data blocks 14-20
  - compiling with 14-10
  - effect on other directives 14-10
- Source files 14-1
  - compiling 2-4, 14-2
  - creating 2-2
  - modular 3-1
  - modular sample program A-9
- Special expressions 13-1
  - assignment expression 13-2
  - CASE expression 13-3
  - data types 13-1
  - group comparison expression 13-5
  - IF expression 13-4
- SPOOL DEFINE command E-5
- Spooler
  - accessing in the CRE 17-47
  - settings, specifying
    - CRE\_SPOOL\_START\_ 17-47
    - TACL SPOOL DEFINE command E-5
- SSV, TACL ASSIGN commands E-9
- Stack register 4-5
- STACK statement, performance guidelines C-2
- Standard files, CRE 17-46
- Standard functions
  - by categories 5-12
  - for arrays 7-23
  - for structures 8-43

---

Standard indirection  
  description 4-7  
  TAL and C guidelines 17-18  
Standard input file, CRE 17-46  
Standard log file, CRE 17-46  
Standard output file, CRE 17-46  
Standard pointers  
  accessing data  
    simple pointers 9-9  
    structure pointers 9-18  
  declaring  
    simple pointers 9-2  
    structure pointers 9-12  
  description 9-1  
Startup message, saving 17-44  
Statement mnemonics, compiler listing 15-7  
Statements  
  ASSERT 12-17  
  assignment  
    arrays 7-13  
    simple pointers 9-7  
    simple variables 6-4  
    structure items 8-34  
    structure pointers 9-16  
  CALL 12-19  
  CASE, labeled 12-5  
  DO 12-10  
  FOR 12-12  
  GOTO 12-24  
  IF 12-2  
  local scope 3-8  
  move  
    arrays 7-14  
    simple pointers 9-9  
    structure pointers 9-19  
    structures 8-39  
  RETURN 12-21  
  RSCAN 7-19  
  SCAN  
    arrays 7-19  
    simple pointers 9-9  
    structure pointers 9-18

- Statements (continued)
  - sublocal scope 3-12
  - WHILE 12-8
- STEP command, Inspect product 16-5
- Stepping through programs 16-5
- STOP command
  - Inspect product 16-5
  - TACL product 16-3
- Stopping Inspect product 16-5
- Stopping object files
  - CRE\_TERMINATOR\_ 17-45
  - TACL STOP command 16-3
- Storage allocation
  - arrays 7-8
  - arrays in structures 8-10
  - definition structures 8-4
  - definition substructures 8-14
  - extended data segment
    - automatic 4-10
    - explicit B-13, B-21
  - global data blocks 14-16
  - object files bound with BLOCKs 14-19
  - parameter areas
    - EXTENSIBLE procedures 11-45
    - procedure 11-36
    - subprocedure 11-36
    - VARIABLE procedures 11-40
  - parameters
    - reference 11-29
    - value 11-21
  - procedure variables 11-5
  - referral structures 8-9
  - referral substructures 8-15
  - simple pointers 9-6
  - simple pointers in structures 8-18
  - simple variables 6-3
  - simple variables in structures 8-10
  - structure pointers 9-16
  - structure pointers in structures 8-20

- 
- Storage allocation (continued)
    - user data segment
      - lower 32K-word area 4-8
      - upper 32K-word area B-5
    - variables 4-8
  - Storage units, by data type 5-6
  - STORE statement, performance guidelines C-2
  - STRING and C char variables 17-19
  - STRING attribute, structure pointers 9-13
  - STRING data type
    - arrays 7-4
    - numeric constants 5-8
    - range of values allowed 5-4
    - simple variables 6-5
  - STRING parameter type 11-20, 11-22
  - String-display sample program A-1
  - String-entry sample program A-3
  - Strings, character 5-10
  - STRUCT keyword, structures 8-3
  - Structure data items
    - arrays 8-9
    - definition substructures 8-12
    - referral substructures 8-15
    - simple pointers 8-17
    - simple variables 8-9
    - structure pointers 8-19
  - Structure items
    - See also* Structure data items
    - filler declarations 8-16
  - Structure pointers
    - accessing data
      - indexing 9-20
      - move statement 9-19
      - scan statements 9-18
    - as structure items 8-19
    - assignments
      - to data 9-18
      - to pointers 9-16
    - declaring 9-12
    - equivalenced 10-16
    - identifiers, qualifying 9-18
    - initialization 9-14

- Structure pointers (continued)
  - redefinitions 8-26
  - storage allocation 9-16
  - upper 32K-word area
    - copying data to B-6
    - storing addresses B-4
  - \$OFFSET function 9-14
- Structures
  - accessing 8-27
  - as parameters 11-31
  - assignments 8-34
  - declaring 8-1
  - definition structure 8-3
  - equivalenced
    - definition structures 10-10
    - referral structures 10-14
  - identifiers, qualifying 8-27
  - indexing 8-28
  - indirection
    - definition structures 8-3
    - referral structures 8-8
  - interlanguage correspondence F-4
  - kinds of 8-1
  - layout 8-2
  - referral structures 8-8
  - storage allocation 8-4
  - TAL and C guidelines 17-20
  - template structures 8-7
- Structuring programs 3-1
- Sublocal data
  - declaring 3-12
  - scope of 3-3
- Sublocal map, compiler listing 15-6
- Sublocal scope
  - data 3-12
  - identifiers 3-3, 11-16
  - labels 3-12
  - statements 3-12

- Sublocal storage
  - description 4-9
  - formal parameters 11-15
  - limitations
    - parameters 11-18
    - variables 11-17
  - parameter area 11-36
- Sublocal variables 11-16
- Subprocedures
  - address of (PEP number) 11-52
  - compared to procedures 11-14
  - declaring 11-15
  - in program structure 3-10
  - parameter area 11-36
  - parameters 11-15
  - RETURN statement 12-21
  - sublocal storage area
    - limitations, parameters 11-18
    - limitations, variables 11-17
  - size 11-15
  - TAL and C guidelines 17-21
- Substructures
  - declaring 8-12
  - definition substructures 8-12
  - referral substructures 8-15
  - storage allocation 8-15
  - TAL and C guidelines 17-21
- Subtraction operator
  - signed 5-16
  - unsigned 5-18
- Subvolume name
  - defaults, specifying E-4
  - in file names E-2
- Suspending execution 16-5
- Swap volume
  - DEFAULT DEFINE E-6
  - PARAM SWAPVOL command E-7
- #SX (extended stack pointer) 4-11
- SYMBOLS directive (with USEGLOBALS) 14-25
- SYMSERV
  - description 14-1
  - specifying which one E-7

SYNTAX directive (with USEGLOBALS) 14-25  
System code space 4-5  
System library space 4-5  
System messages, saving 17-44  
System name, in file names E-2  
System procedures  
    overview of 2-3  
    sample program  
        mainline module A-12  
        string display A-1  
        string entry A-3  
    with SOURCE directive 14-11  
System services 1-3  
Systems supported by TAL xxviii

---

## T

TACL commands  
    ASSIGN E-8  
    ASSIGN SSV E-9  
    DEFINE E-4  
    directives that accept 14-9  
    PARAM E-6  
    RUN (running object files) 16-1  
    RUND (debugging object files) 16-4  
    starting the compiler 14-2  
    STOP 16-3  
TAL and C guidelines  
    arrays 17-19  
    arrays of structures 17-23  
    bit-field manipulation 17-28  
    C calling TAL 17-13  
    C enumeration variables 17-27  
    data sharing 17-15  
    data types 17-10  
    identifiers 17-9  
    indirection 17-18  
    memory usage 17-11  
    multidimensional arrays 17-23  
    passing C routines to TAL 17-32  
    passing TAL routines to C 17-30  
    pointers 17-25

---

TAL and C guidelines (continued)  
  redefinitions and C unions 17-24  
  structures 17-20  
  TAL calling C 17-12  
  UNSIGNED and C bit fields 17-29  
TAL calling C 17-12  
TALLIB (TAL library file) 17-43  
TAL\_CRE\_INITIALIZER\_ 17-44  
Tandem NonStop Series system xxviii  
Tandem NonStop Series/RISC system xxviii  
TAPE DEFINE command E-5  
Target file  
  binding 14-6  
  compilation command option 14-4  
Template blocks  
  allocation 14-17  
  description 14-16  
Template structures 8-7  
Temporary files, specifying volume E-7  
Temporary pointer  
  *See* Dereferencing operator  
Terminating  
  *See* Stopping  
THEN keyword  
  IF expression 13-4  
  IF statement 12-2  
TNS system xxviii  
TNS/R system xxviii  
TO keyword, FOR statement 12-12  
Typed procedures  
  *See* Functions

---

## U

Underscore (\_) in identifiers 5-2  
Unions, C version of redefinitions 17-24  
UNSIGNED data type  
  arrays 7-7  
  range of values allowed 5-4  
  simple variables 6-13  
  TAL and C guidelines 17-29  
UNSIGNED parameter type 11-20

UNSPECIFIED language attribute, procedures 17-2  
UNTIL keyword, DO statement 12-10  
Upper 32K-word area  
    accessing B-2  
    indexing B-8  
    managing storage allocation B-5  
    organization 4-2  
USE statement with FOR statement 12-15  
USEGLOBALS directive  
    and CROSSREF directive 14-26  
    compiling with saved globals 14-23  
User code segment  
    accessing B-9  
    description 4-1  
User code space 4-1  
User data segment  
    description 4-2  
    organization 4-3  
    specifying size with DATAPAGES directive B-2  
    storage allocation 4-8  
User heap, CRE 17-41

---

## V

Value parameters  
    addresses 11-26  
    declaring 11-3  
    description 11-21  
    index registers 11-27  
    procedures as parameters  
        EXTENSIBLE 11-25  
        mixed-language programming 17-4  
        passing C routines to TAL 17-32  
        passing TAL routines to C 17-30  
        PROC 11-23  
        PROC(32) 11-24  
        that have parameters 11-24  
        VARIABLE 11-25  
    simple variables 11-21  
    specifications 11-20  
    storage allocation 11-21

---

VARIABLE procedures  
  checking for parameters (\$PARAM) 11-9  
  converting to EXTENSIBLE 11-13  
  declaring 11-9  
  parameter areas 11-40  
  parameter masks 11-38  
  passing as parameters 11-25  
  sample program  
    initialization module A-13  
    string display A-1

Variables  
  getting the address of 5-27  
  in arithmetic expressions 5-15  
  kinds of 5-7  
    arrays 7-1  
    pointers 9-1  
    simple variables 6-1  
    structures 8-1

Variant parts, emulating 10-21

Volume name  
  defaults  
    DEFAULT DEFINE E-6  
    specifying E-4  
  in file names E-2

---

## W

WHILE keyword, WHILE statement 12-8

WHILE statement  
  description 12-8  
  sample program  
    binary-to-ASCII conversion A-6  
    mainline module A-11  
    string entry A-3

width (INT, REAL, UNSIGNED) 5-5

Word 5-6

Word addressing 4-5

WORDS keyword  
  group comparison expression 13-6  
  move statement 7-16

**X****\$XADR**

- procedures as parameters 11-24
- upper 32K-word area B-8

**XEP table 11-1****XOR operator 5-20**

---

**Z****ZZBInnnn target file 14-4**

---

**Special characters****" (character string delimiter) 5-10****#CRE\_GLOBALS (CRE control block) 17-39****#CRE\_HEAP (CRE run-time heap) 17-39****#GLOBAL**

## allocation 14-17

## description 14-16

## listing 15-14

**#HEAP (CRE user heap) 17-41****#MCB (CRE master control block) 17-39****#MX (extended stack pointer) 4-11****#SX (extended stack pointer) 4-11****\$\$GLOBAL**

## description 14-17

## listing 15-14

**\$CARRY function 5-26****\$DBL**

## structure pointers 9-22

## structures 8-30

**\$DBLL, accessing user code segment B-9****\$EXTENDED#STACK 4-11****\$OPTIONAL function 11-12****\$PARAM, checking for parameters**

## EXTENSIBLE procedures 11-10

## VARIABLE procedures 11-9

**\$RECEIVE, accessing in the CRE 17-48****\$\$SCALE, scaling FIXED values 5-17****\$XADR**

## procedures as parameters 11-24

## upper 32K-word area B-8

**% prefix, octal constants 5-7**

---

---

%B prefix, binary constants 5-7  
%D suffix, hexadecimal INT(32) numbers 5-8  
%F suffix, FIXED numbers 5-9  
%H prefix, hexadecimal constants 5-7  
&  
    concatenated move (copy) operations 7-18  
    prefix, template block name  
        allocation 14-17  
        description 14-16,  
        listing 15-14  
'\*' (unsigned multiplication) 5-18  
'+' (unsigned addition) 5-18  
'-' (unsigned subtraction) 5-18  
'/' (unsigned division) 5-18  
'<' (unsigned less than) 5-23  
'<<' (unsigned left shift) 5-29  
'<=' (unsigned less than or equal to) 5-23  
'<>' (unsigned not equal to) 5-23  
'=' (unsigned equal to) 5-23  
':= ' (left-to-right move operator) 7-14  
'=: ' (right-to-left move operator) 7-14  
'>' (unsigned greater than) 5-23  
'>=' (unsigned greater than or equal to) 5-23  
'>>' (unsigned right shift) 5-29  
'\' (unsigned modulo division) 5-18  
( ) in expressions 5-14  
(\*  
    FIXED(\*) data type 5-5  
    template structures 8-7  
\* (asterisk)  
    in compiler listing 15-5  
    repetition factor, constant list 7-14  
    signed multiplication 5-16  
+  
    binary signed addition 5-16  
    unary plus 5-16  
-  
    binary signed subtraction 5-16  
    unary minus 5-16  
-> (CASE statement, labeled) 12-5

- > (next-address symbol)
  - group comparison expression 13-7
  - move statement 7-17
- . (period)
  - bit extractions 5-28
  - dereferencing operator 5-27
  - in file names E-2
- . (standard indirection symbol)
  - arrays 7-1
  - definition structures 8-3
  - referral structures 8-8
  - simple pointers 9-2
  - structure pointers 9-12
- .. (ellipsis in labeled CASE statement 12-5)
- .#GLOBAL
  - description 14-17
  - listing 15-14
- .EXT (extended indirection symbol)
  - arrays 7-1
  - definition structures 8-3
  - referral structures 8-8
  - simple pointers 9-2
  - structure pointers 9-12
- / (signed division) 5-16
- ;(semicolon) 3-17
- < (signed less than) 5-23
- << (signed left shift) 5-29
- <= (signed less than or equal to) 5-23
- <> (signed not equal to) 5-23
- <n:n> (bit-extraction field) 5-28
- = (signed equal to) 5-23
- > (signed greater than) 5-23
- >= (signed greater than or equal to) 5-23
- >> (signed right shift) 5-29
- ? (directive line symbol) 14-7
- @ operator
  - pointers 9-3
  - procedure/subprocedure addresses 11-52
  - procedures as parameters 11-24
- [constant]
  - group comparison expression 13-5
  - move statement 7-15

[n:n] bounds  
  arrays 7-1  
  structures 8-3, 8-8  
:(colon)  
  bit extractions 5-28  
  entry-point identifiers  
  procedures 3-7  
  subprocedures 3-11  
  label identifiers 11-48  
:= (assignment operator)  
  assignment expression 13-2  
  assignment statement 6-4  
^ (circumflex) in identifiers 5-2  
\_ (underscore) in identifiers 5-2