

# TAL Reference Manual

## **Abstract**

This manual provides syntax descriptions and error messages for TAL (Transaction Application Language) for system and application programmers.

## **Product Version**

TAL D40

## **Supported Release Version Updates (RVUs)**

This publication supports D40.00 and all subsequent D-series RVUs, and G01.00 and all subsequent G-series RVUs until otherwise indicated in a new edition.

<b>Part Number</b>	<b>Published</b>
526371-001	September 2003

**Document History**

<b>Part Number</b>	<b>Product Version</b>	<b>Published</b>
15998	TAL C20	March 1989
065722	TAL D10	January 1993
096255	TAL C30, TAL D10 & TAL D20	September 1993
526371-001	TAL D40	September 2003

# TAL Reference Manual

Glossary

Index

Tables

[What's New in This Manual](#) xxxix

[Manual Information](#) xxxix

[New and Changed Information](#) xxxix

[About This Manual](#) xli

[Audience](#) xli

[How to Use this Manual Set](#) xli

[Manual Organization](#) xlvi

[System Dependencies](#) xlvi

[Compiler Dependencies](#) xlvi

[Additional Information](#) xlvi

[Notation Conventions](#) xlvii

[Railroad Diagrams](#) xlvii

[Branching](#) xlvii

[Spacing](#) xlviii

[Case Conventions](#) xlviii

[Example Diagrams](#) xlviii

[Hypertext Links](#) xlxi

[General Syntax Notation](#) xlxi

[Notation for Messages](#) lii

[Notation for Management Programming Interfaces](#) liii

## 1. Introduction

[Applications and Uses](#) 1-1

[Major Features](#) 1-1

[System Services](#) 1-3

[System Procedures](#) 1-3

[TAL Run-Time Library](#) 1-3

[CRE Services](#) 1-4

## 2. Language Elements

<u>Character Set</u>	2-1
<u>Declarations</u>	2-1
<u>Statements</u>	2-2
<u>Keywords</u>	2-2
<u>Identifiers</u>	2-4
<u>Identifier Classes</u>	2-4
<u>Constants</u>	2-5
<u>Constant Expressions</u>	2-5
<u>Number Bases</u>	2-5
<u>Variables</u>	2-6
<u>Symbols</u>	2-6
<u>Indirection Symbols</u>	2-7
<u>Base Address Symbols</u>	2-7
<u>Delimiters</u>	2-7
<u>Operators</u>	2-9

## 3. Data Representation

<u>Data Types</u>	3-1
<u>Specifying Data Types</u>	3-3
<u>Data Type Aliases</u>	3-4
<u>Storage Units</u>	3-5
<u>Address Modes</u>	3-5
<u>Operations by Data Type</u>	3-5
<u>Functions by Data Type</u>	3-6
<u>Address Types</u>	3-7
<u>Syntax for Constants</u>	3-7
<u>Character String Constants</u>	3-8
<u>Character String Length</u>	3-8
<u>Example of Character String Constant</u>	3-9
<u>String Numeric Constants</u>	3-9
<u>Example of STRING Numeric Constants</u>	3-9
<u>INT Numeric Constants</u>	3-10
<u>Examples of INT Numeric Constants</u>	3-11
<u>Storage Format</u>	3-11
<u>INT (32) Numeric Constants</u>	3-11
<u>Examples of INT (32) Numeric Constants</u>	3-12
<u>FIXED Numeric Constants</u>	3-13
<u>Examples of FIXED Numeric Constants</u>	3-14

### 3. Data Representation (continued)

<u>Storage Format</u>	3-14
<u>REAL and REAL (64) Numeric Constants</u>	3-14
<u>Examples of REAL and REAL (64) Numeric Constants</u>	3-15
<u>Storage Format</u>	3-15
<u>Examples of Storage Formats</u>	3-16
<u>Constant Lists</u>	3-16
<u>Examples of Constant Lists</u>	3-17

### 4. Expressions

<u>About Expressions</u>	4-1
<u>Data Types of Expressions</u>	4-2
<u>Precedence of Operators</u>	4-3
<u>Arithmetic Expressions</u>	4-5
<u>    Examples of Arithmetic Expressions</u>	4-6
<u>    Operands in Arithmetic Expressions</u>	4-6
<u>    Signed Arithmetic Operators</u>	4-6
<u>    Unsigned Arithmetic Operators</u>	4-9
<u>    Bitwise Logical Operators</u>	4-11
<u>Conditional Expressions</u>	4-12
<u>    Examples of Conditional Expressions</u>	4-13
<u>    Conditions</u>	4-13
<u>    Boolean Operators</u>	4-14
<u>    Relational Operators</u>	4-14
<u>Testing Hardware Indicators</u>	4-16
<u>    Condition Code Indicator</u>	4-16
<u>    Carry Indicator</u>	4-17
<u>    Overflow Indicator</u>	4-17
<u>Special Expressions</u>	4-18
<u>    Assignment Expression</u>	4-19
<u>    CASE Expression</u>	4-20
<u>    IF Expression</u>	4-21
<u>    Group Comparison Expression</u>	4-23
<u>Bit Operations</u>	4-27
<u>Bit Extractions</u>	4-28
<u>    Usage Considerations</u>	4-28
<u>    Examples of Bit Extractions</u>	4-28
<u>Bit Shifts</u>	4-29
<u>    Usage Considerations</u>	4-30

## 4. Expressions (continued)

Examples of Bit Shifts 4-30

## 5. LITERALs and DEFINEs

LITERAL Declaration 5-1

Usage Considerations 5-2

Examples of LITERAL Declarations 5-2

DEFINE Declaration 5-3

Usage Considerations 5-4

Examples of DEFINE Declarations 5-5

Invoking DEFINEs 5-6

Compiler Action 5-7

Passing Actual Parameters 5-7

Examples of Passing DEFINE Parameters 5-8

## 6. Simple Variables

Simple Variable Declaration 6-1

Usage Considerations 6-2

Examples of Simple Variable Declarations 6-2

## 7. Arrays

Array Declaration 7-1

Usage Considerations 7-3

Examples of Array Declarations 7-3

Read-Only Array Declaration 7-5

Usage Considerations 7-6

Example of Read-Only Array Declaration 7-6

## 8. Structures

Kinds of Structures 8-1

Structure Layout 8-2

Definition Structure Declaration 8-3

Usage Considerations 8-4

Examples of Definition Structure Declarations 8-4

Template Structure Declaration 8-5

Usage Considerations 8-5

Example of Template Structure Declaration 8-5

Referral Structure Declaration 8-6

Usage Considerations 8-7

Example of Referral Structure Declaration 8-7

## 8. Structures (continued)

<u>Simple Variables Declared in Structures</u>	8-7
<u>Usage Considerations</u>	8-8
<u>Example of Simple Variables in Structures</u>	8-8
<u>Arrays Declared in Structures</u>	8-8
<u>Usage Considerations</u>	8-9
<u>Example of Arrays in Structures</u>	8-9
<u>Substructure Declaration</u>	8-9
<u>Definition Substructure Declaration</u>	8-10
<u>Example of Definition Substructure Declaration</u>	8-11
<u>Referral Substructure Definition</u>	8-11
<u>Example of Referral Substructure Declaration</u>	8-12
<u>Filler Declaration</u>	8-12
<u>Usage Considerations</u>	8-13
<u>Examples of Filler Declarations</u>	8-13
<u>Simple Pointers Declared in Structures</u>	8-13
<u>Usage Considerations</u>	8-14
<u>Example of Simple Pointer Declarations</u>	8-15
<u>Structure Pointers Declared in Structures</u>	8-15
<u>Usage Considerations</u>	8-16
<u>Example of Structure Pointer Declaration</u>	8-17
<u>Redefinition Declaration</u>	8-17
<u>Redefinition Rules</u>	8-17
<u>Redefinitions Outside Structures</u>	8-17
<u>Simple Variable Redefinition</u>	8-18
<u>Usage Considerations</u>	8-18
<u>Example of Simple Variable Redefinition</u>	8-18
<u>Array Redefinition</u>	8-19
<u>Usage Considerations</u>	8-19
<u>Example of Array Redefinition</u>	8-20
<u>Definition Substructure Redefinition</u>	8-20
<u>Usage Considerations</u>	8-21
<u>Examples of Definition Substructure Redefinitions</u>	8-21
<u>Referral Substructure Redefinition</u>	8-22
<u>Usage Considerations</u>	8-22
<u>Example of Referral Substructure Declaration</u>	8-23
<u>Simple Pointer Redefinition</u>	8-23
<u>Example of Simple Pointer Redefinition</u>	8-24
<u>Structure Pointer Redefinition</u>	8-24

## 8. Structures (continued)

- Usage Considerations 8-25
- Example of Structure Pointer Redefinitions 8-25

## 9. Pointers

- Simple Pointer Declaration 9-2
- Usage Considerations 9-2
- Examples of Simple Pointer Declarations 9-4
- Structure Pointer Declaration 9-5
- Usage Considerations 9-6
- Examples of Structure Pointer Declarations 9-7

## 10. Equivalenced Variables

- Equivalenced Variable Declarations 10-1
- Equivalenced Simple Variable 10-2
  - Usage Consideration 10-2
  - Examples of Equivalenced Simple Variable Declarations 10-3
- Equivalenced Simple Pointer 10-3
  - Usage Consideration 10-4
  - Example of Equivalenced Simple Pointer Declaration 10-5
- Equivalenced Definition Structure 10-5
  - Usage Considerations 10-6
  - Example of Equivalenced Definition Structure Declaration 10-7
- Equivalenced Referral Structure 10-7
  - Usage Considerations 10-8
  - Example of Equivalenced Referral Structure Declaration 10-9
- Equivalenced Structure Pointer 10-9
  - Usage Considerations 10-10
  - Example of Equivalenced Structure Pointer Declaration 10-11
- Base-Address Equivalenced Variable Declarations 10-11
- Base-Address Equivalenced Simple Variable 10-12
  - Considerations 10-12
  - Example of Base-Address Equivalenced Simple Variable Declaration 10-13
- Base-Address Equivalenced Simple Pointer 10-13
  - Usage Considerations 10-14
- Base-Address Equivalenced Definition Structure 10-14
  - Usage Considerations 10-15
- Base-Address Equivalenced Referral Structure 10-16
  - Usage Considerations 10-17

## 10. Equivalenced Variables (continued)

Base-Address Equivalenced Structure Pointer 10-17  
Usage Considerations 10-18

## 11. NAMES and BLOCKs

NAME Declaration 11-1  
Usage Considerations 11-2  
Example of NAME Declaration 11-2  
BLOCK Declaration 11-2  
Usage Considerations 11-3  
Examples of BLOCK Declarations 11-4  
Coding Data Blocks 11-5  
Unblocked Declarations 11-5

## 12. Statements

Using Semicolons 12-1  
Compound Statements 12-2  
Usage Considerations 12-2  
Examples of Compound Statements 12-2  
ASSERT Statement 12-3  
Usage Considerations 12-3  
Example of ASSERT Statement 12-4  
Assignment Statement 12-4  
Usage Considerations 12-5  
Examples of Assignment Statements 12-6  
Bit-Drop Assignment Statement 12-7  
Usage Considerations 12-7  
Examples of Bit Drop Assignments 12-8  
CALL Statement 12-9  
Usage Considerations 12-10  
Examples of CALL Statements 12-10  
CASE Statement 12-11  
Labeled CASE Statement 12-11  
Usage Considerations 12-12  
Example of Labeled CASE Statement 12-12  
Unlabeled CASE Statement 12-13  
Usage Considerations 12-13  
Examples of Unlabeled CASE Statements 12-14  
CODE Statement 12-15

## 12. Statements (continued)

<u>Usage Considerations</u>	12-16
<u>Examples of CODE Statements</u>	12-18
<u>DO Statement</u>	12-19
<u>Usage Considerations</u>	12-19
<u>Examples of DO Statements</u>	12-19
<u>DROP Statement</u>	12-20
<u>Usage Considerations</u>	12-20
<u>Examples of DROP Statements</u>	12-21
<u>FOR Statement</u>	12-22
<u>Usage Considerations</u>	12-23
<u>Examples of FOR Statements</u>	12-24
<u>GOTO Statement</u>	12-25
<u>Usage Considerations</u>	12-25
<u>Examples of GOTO Statements</u>	12-25
<u>IF Statement</u>	12-26
<u>Usage Considerations</u>	12-26
<u>Example of IF Statements</u>	12-27
<u>MOVE Statement</u>	12-27
<u>Usage Considerations</u>	12-29
<u>Examples of MOVE Statements</u>	12-30
<u>RETURN Statement</u>	12-31
<u>Usage Considerations</u>	12-32
<u>Examples of RETURN Statements</u>	12-33
<u>SCAN Statement</u>	12-34
<u>Usage Considerations</u>	12-35
<u>Example of SCAN Statements</u>	12-36
<u>STACK Statement</u>	12-36
<u>Usage Considerations</u>	12-36
<u>Examples of STACK Statements</u>	12-37
<u>STORE Statement</u>	12-37
<u>Usage Considerations</u>	12-37
<u>Examples of STORE Statements</u>	12-38
<u>USE Statement</u>	12-38
<u>Usage Considerations</u>	12-39
<u>Examples of USE Statements</u>	12-39
<u>WHILE Statement</u>	12-40
<u>Usage Considerations</u>	12-40
<u>Examples of WHILE Statements</u>	12-41

## 13. Procedures

<u>Procedure Declaration</u>	13-2
<u>Usage Considerations</u>	13-4
<u>Examples of Procedure Declarations</u>	13-4
<u>Procedure Attributes</u>	13-5
<u>MAIN</u>	13-5
<u>INTERRUPT</u>	13-6
<u>RESIDENT</u>	13-6
<u>CALLABLE</u>	13-6
<u>PRIV</u>	13-6
<u>VARIABLE</u>	13-7
<u>EXTENSIBLE</u>	13-7
<u>LANGUAGE</u>	13-8
<u>Formal Parameter Specifications</u>	13-8
<u>Usage Considerations</u>	13-11
<u>Examples of Formal Parameter Specification</u>	13-13
<u>Procedure Body</u>	13-13
<u>Usage Consideration</u>	13-14
<u>Examples of Procedure Declarations</u>	13-14
<u>Subprocedure Declaration</u>	13-15
<u>Subprocedure Body</u>	13-17
<u>Usage Considerations</u>	13-17
<u>Example of Subprocedure Declaration</u>	13-18
<u>Entry-Point Declaration</u>	13-18
<u>Usage Considerations</u>	13-19
<u>Examples of Entry-Point Declarations</u>	13-20
<u>Label Declaration</u>	13-21
<u>Usage Considerations</u>	13-22
<u>Examples of Label Declarations</u>	13-22

## 14. Standard Functions

<u>Summary of Standard Functions</u>	14-1
<u>Type-Transfer Functions</u>	14-4
<u>Functions by Data Type</u>	14-4
<u>Rounding by Standard Functions</u>	14-5
<u>Scope of Standard Functions</u>	14-5
<u>Expression Arguments</u>	14-5
<u>Data Types of Expression Arguments</u>	14-6
<u>Signedness of Expression Arguments</u>	14-6

## 14. Standard Functions (continued)

<u>\$ABS Function</u>	14-6
<u>Usage Considerations</u>	14-7
<u>Example of \$ABS Function</u>	14-7
<u>\$ALPHA Function</u>	14-7
<u>Usage Considerations</u>	14-7
<u>Example of \$ALPHA Function</u>	14-8
<u>\$AXADR Function</u>	14-8
<u>\$BITLENGTH Function</u>	14-8
<u>Usage Considerations</u>	14-8
<u>Example of \$BITLENGTH Function</u>	14-9
<u>\$BITOFFSET Function</u>	14-9
<u>Usage Considerations</u>	14-9
<u>Example of \$BITOFFSET Function</u>	14-10
<u>\$BOUNDS Function</u>	14-10
<u>\$CARRY Function</u>	14-10
<u>Usage Considerations</u>	14-10
<u>Example of \$CARRY Function</u>	14-11
<u>\$COMP Function</u>	14-11
<u>Example of \$COMP Function</u>	14-11
<u>\$DBL Function</u>	14-11
<u>Usage Consideration</u>	14-12
<u>Example of \$DBL Function</u>	14-12
<u>\$DBLL Function</u>	14-12
<u>Usage Consideration</u>	14-12
<u>Examples of \$DBLL Function</u>	14-13
<u>\$DBLR Function</u>	14-13
<u>Usage Consideration</u>	14-13
<u>Examples of \$DBLR Function</u>	14-13
<u>\$DFIX Function</u>	14-14
<u>Usage Consideration</u>	14-14
<u>Example of \$DFIX Function</u>	14-14
<u>\$EFLT Function</u>	14-15
<u>Usage Consideration</u>	14-15
<u>Example of \$EFLT Function</u>	14-15
<u>\$EFLTR Function</u>	14-15
<u>Usage Considerations</u>	14-15
<u>Example of \$EFLTR Function</u>	14-16
<u>\$FIX Function</u>	14-16

## **14. Standard Functions (continued)**

<u>Usage Consideration</u>	14-16
<u>Example of \$FIX Function</u>	14-16
<b>\$FIXD Function</b>	14-16
<u>Usage Consideration</u>	14-17
<u>Example of \$FIXD Function</u>	14-17
<b>\$FIXI Function</b>	14-17
<u>Usage Considerations</u>	14-17
<u>Example of \$FIXI Function</u>	14-17
<b>\$FIXL Function</b>	14-18
<u>Usage Considerations</u>	14-18
<u>Examples of \$FIXL Function</u>	14-18
<b>\$FIXR Function</b>	14-18
<u>Usage Considerations</u>	14-18
<u>Example of \$FIXR Function</u>	14-19
<b>\$FLT Function</b>	14-19
<u>Usage Consideration</u>	14-19
<u>Example of \$FLT Function</u>	14-19
<b>\$FLTR Function</b>	14-20
<u>Usage Consideration</u>	14-20
<u>Example of \$FLTR Function</u>	14-20
<b>\$HIGH Function</b>	14-20
<u>Example of \$HIGH Function</u>	14-20
<b>\$IFIX Function</b>	14-21
<u>Usage Consideration</u>	14-21
<u>Example of \$IFIX Function</u>	14-21
<b>\$INT Function</b>	14-21
<u>Usage Considerations</u>	14-22
<u>Examples of \$INT Function</u>	14-22
<b>\$INTR Function</b>	14-22
<u>Usage Considerations</u>	14-22
<u>Example of \$INTR Function</u>	14-23
<b>\$LADR Function</b>	14-23
<u>Usage Considerations</u>	14-23
<u>Example of \$LADR Function</u>	14-24
<b>\$LEN Function</b>	14-24
<u>Usage Considerations</u>	14-24
<u>Examples of \$LEN Function</u>	14-25
<b>\$LFIX Function</b>	14-25

## **14. Standard Functions (continued)**

<u>Usage Consideration</u>	14-26
<u>Example of \$LFIX Function</u>	14-26
<u>\$LMAX Function</u>	14-26
<u>Example of \$LMAX Function</u>	14-26
<u>\$LMIN Function</u>	14-26
<u>Example of \$LMIN Function</u>	14-27
<u>\$MAX Function</u>	14-27
<u>Example of \$MAX Function</u>	14-27
<u>\$MIN Function</u>	14-27
<u>Example of \$MIN Function</u>	14-28
<u>\$NUMERIC Function</u>	14-28
<u>Usage Considerations</u>	14-28
<u>Example of \$NUMERIC Function</u>	14-28
<u>\$OCCURS Function</u>	14-29
<u>Usage Considerations</u>	14-29
<u>Examples of \$OCCURS Function</u>	14-30
<u>\$OFFSET Function</u>	14-30
<u>Usage Considerations</u>	14-31
<u>Examples of \$OFFSET Function</u>	14-31
<u>\$OPTIONAL Function</u>	14-32
<u>Usage Considerations</u>	14-33
<u>Examples of the \$OPTIONAL Function</u>	14-33
<u>\$OVERFLOW Function</u>	14-35
<u>Usage Considerations</u>	14-35
<u>Example of \$OVERFLOW Function</u>	14-36
<u>\$PARAM Function</u>	14-36
<u>Usage Considerations</u>	14-36
<u>Example of \$PARAM Function</u>	14-37
<u>\$POINT Function</u>	14-37
<u>Usage Considerations</u>	14-37
<u>Example of \$POINT Function</u>	14-37
<u>\$READCLOCK Function</u>	14-38
<u>Usage Considerations</u>	14-38
<u>Example of \$READCLOCK Function</u>	14-38
<u>\$RP Function</u>	14-38
<u>Usage Consideration</u>	14-38
<u>Example of \$RP Function</u>	14-38
<u>\$SCALE Function</u>	14-39

## **14. Standard Functions (continued)**

<u>Usage Considerations</u>	14-39
<u>Example of \$SCALE Function</u>	14-39
<u>\$SPECIAL Function</u>	14-40
<u>Usage Considerations</u>	14-40
<u>Example of \$SPECIAL Function</u>	14-40
<u>\$SWITCHES Function</u>	14-40
<u>\$TYPE Function</u>	14-41
<u>Usage Considerations</u>	14-41
<u>Example of \$TYPE Function</u>	14-41
<u>\$UDBL Function</u>	14-41
<u>Usage Consideration</u>	14-42
<u>Example of \$UDBL Function</u>	14-42
<u>\$USERCODE Function</u>	14-42
<u>Usage Considerations</u>	14-42
<u>Example of \$USERCODE Function</u>	14-43
<u>\$XADR Function</u>	14-43
<u>Usage Considerations</u>	14-43
<u>Examples of \$XADR Function</u>	14-44
<u>Built-in Functions</u>	14-44

## **15. Privileged Procedures**

<u>Privileged Mode</u>	15-1
<u>CALLABLE Procedures</u>	15-1
<u>PRIV Procedures</u>	15-1
<u>Nonprivileged Procedures</u>	15-2
<u>Privileged Operations</u>	15-2
<u>System Global Pointer Declaration</u>	15-3
<u>Usage Consideration</u>	15-3
<u>Example of System Global Pointer Declaration</u>	15-3
<u>'SG'-Equivalenced Variable Declarations</u>	15-4
<u>'SG'-Equivalenced Simple Variable</u>	15-4
<u>Example of 'SG'-Equivalenced Simple Variable</u>	15-5
<u>'SG'-Equivalenced Definition Structure</u>	15-5
<u>Usage Consideration</u>	15-6
<u>Example of 'SG'-Equivalenced Definition Structure</u>	15-6
<u>'SG'-Equivalenced Referral Structure</u>	15-6
<u>Usage Considerations</u>	15-7
<u>Example of 'SG'-Equivalenced Referral Structure</u>	15-7

## **15. Privileged Procedures (continued)**

<a href="#">'SG'-Equivalenced Simple Pointer</a>	15-8
<a href="#">Example of 'SG'-Equivalenced Simple Pointer</a>	15-9
<a href="#">'SG'-Equivalenced Structure Pointer</a>	15-9
<a href="#">Usage Considerations</a>	15-10
<a href="#">Example of 'SG'-Equivalenced Simple Pointer</a>	15-10
<a href="#">Functions for Privileged Operations</a>	15-11
<a href="#">\$AXADR Function</a>	15-11
<a href="#">Usage Considerations</a>	15-11
<a href="#">Example of \$AXADR Function</a>	15-11
<a href="#">\$BOUNDS Function</a>	15-12
<a href="#">Usage Considerations</a>	15-12
<a href="#">Example of \$BOUNDS Function</a>	15-12
<a href="#">\$SWITCHES Function</a>	15-13
<a href="#">Usage Considerations</a>	15-13
<a href="#">Example of \$SWITCHES Function</a>	15-13
<a href="#">TARGET Directive</a>	15-13
<a href="#">Usage Considerations</a>	15-14
<a href="#">Examples of TARGET Directive</a>	15-15

## **16. Compiler Directives**

<a href="#">Specifying Compiler Directives</a>	16-1
<a href="#">Compilation Command</a>	16-1
<a href="#">Directive Line</a>	16-2
<a href="#">Directive Stacks</a>	16-3
<a href="#">Pushing Directive Settings</a>	16-3
<a href="#">Popping Directive Settings</a>	16-3
<a href="#">File Names As Directive Arguments</a>	16-4
<a href="#">Partial File Names</a>	16-4
<a href="#">Logical File Names</a>	16-5
<a href="#">Summary of Compiler Directives</a>	16-5
<a href="#">ABORT Directive</a>	16-12
<a href="#">Usage Considerations</a>	16-12
<a href="#">Example of ABORT Directive</a>	16-13
<a href="#">ABSLIST Directive</a>	16-13
<a href="#">Usage Considerations</a>	16-13
<a href="#">Example of ABSLIST Considerations</a>	16-14
<a href="#">ASSERTION Directive</a>	16-14
<a href="#">Usage Considerations</a>	16-14

## 16. Compiler Directives (continued)

<u>Example of ASSERTION Directive</u>	16-15
<u>BEGINCOMPILATION Directive</u>	16-16
<u>Usage Considerations</u>	16-16
<u>Example of BEGINCOMPILATION Directive</u>	16-16
<u>CHECK Directive</u>	16-17
<u>Usage Considerations</u>	16-17
<u>Example of CHECK Directive</u>	16-18
<u>CODE Directive</u>	16-18
<u>Usage Considerations</u>	16-19
<u>Example of CODE Directive</u>	16-19
<u>COLUMNS Directive</u>	16-19
<u>Usage Considerations</u>	16-20
<u>Examples of COLUMNS Directive</u>	16-21
<u>COMPACT Directive</u>	16-21
<u>Usage Considerations</u>	16-21
<u>Example of COMPACT Directive</u>	16-22
<u>CPU Directive</u>	16-22
<u>Usage Considerations</u>	16-22
<u>CROSSREF Directive</u>	16-22
<u>Usage Considerations</u>	16-23
<u>Example of CROSSREF Directive</u>	16-25
<u>DATAPAGES Directive</u>	16-25
<u>Usage Considerations</u>	16-26
<u>Example of DATAPAGES Directive</u>	16-26
<u>DECS Directive</u>	16-26
<u>Usage Considerations</u>	16-27
<u>Example of DECS Directive</u>	16-27
<u>DEFEXPAND Directive</u>	16-27
<u>Usage Considerations</u>	16-28
<u>Example of DEFEXPAND Directive</u>	16-28
<u>DEFINETOZ Directive</u>	16-29
<u>Usage Considerations</u>	16-29
<u>Examples of DEFINETOZ Directive</u>	16-30
<u>DUMPCONS Directive</u>	16-31
<u>Usage Considerations</u>	16-31
<u>Example of DUMPCONS Directive</u>	16-32
<u>ENDIF Directive</u>	16-32
<u>ENV Directive</u>	16-32

## 16. Compiler Directives (continued)

<u>Usage Considerations</u>	16-33
<u>Examples of ENV Directive</u>	16-34
<u>ERRORFILE Directive</u>	16-34
<u>Usage Considerations</u>	16-35
<u>Example of ERRORFILE Directive</u>	16-36
<u>ERRORS Directive</u>	16-37
<u>Usage Considerations</u>	16-37
<u>Example of ERRORS Directive</u>	16-37
<u>EXTENDSTACK Directive</u>	16-37
<u>Usage Considerations</u>	16-38
<u>Example of EXTENDSTACK Directive</u>	16-38
<u>EXTENDTALHEAP Directive</u>	16-38
<u>Usage Considerations</u>	16-38
<u>Example of EXTENDTALHEAP Directive</u>	16-39
<u>FIXUP Directive</u>	16-39
<u>Usage Considerations</u>	16-39
<u>Example of FIXUP Directive</u>	16-40
<u>FMAP Directive</u>	16-40
<u>Usage Considerations</u>	16-40
<u>Examples of FMAP Directive</u>	16-41
<u>GMAP Directive</u>	16-41
<u>Usage Considerations</u>	16-41
<u>Examples of GMAP Directive</u>	16-41
<u>HEAP Directive</u>	16-42
<u>Usage Considerations</u>	16-42
<u>Example of HEAP Directive</u>	16-43
<u>HIGHPIN Directive</u>	16-43
<u>Usage Considerations</u>	16-43
<u>Examples of Running Object Files at HIGHPIN</u>	16-44
<u>HIGHREQUESTERS Directive</u>	16-45
<u>Usage Considerations</u>	16-45
<u>Examples of HIGHREQUESTERS Directive</u>	16-45
<u>ICODE Directive</u>	16-46
<u>Usage Considerations</u>	16-46
<u>Example of ICODE Directive</u>	16-47
<u>IF and ENDIF Directives</u>	16-47
<u>Usage Considerations</u>	16-48
<u>Examples of IF and ENDIF Directives</u>	16-49

## 16. Compiler Directives (continued)

<u>INHIBITXX Directive</u>	16-50
<u>Usage Considerations</u>	16-50
<u>Example of INHIBITXX Directive</u>	16-51
<u>INNERLIST Directive</u>	16-52
<u>Usage Considerations</u>	16-53
<u>Example of INNERLIST Considerations</u>	16-53
<u>INSPECT Directive</u>	16-54
<u>Usage Considerations</u>	16-54
<u>Example of INSPECT Directive</u>	16-54
<u>INT32INDEX Directive</u>	16-55
<u>Usage Considerations</u>	16-55
<u>Example of INT32INDEX Directive</u>	16-56
<u>LARGESTACK Directive</u>	16-57
<u>Usage Considerations</u>	16-57
<u>Example of LARGESTACK Directives</u>	16-57
<u>LIBRARY Directive</u>	16-58
<u>Usage Considerations</u>	16-58
<u>Example of LIBRARY Directive</u>	16-58
<u>About User Libraries</u>	16-58
<u>LINES Directive</u>	16-59
<u>Usage Considerations</u>	16-59
<u>Examples of LINES Directive</u>	16-59
<u>LIST Directive</u>	16-59
<u>Usage Consideration</u>	16-60
<u>Examples of LIST Directive</u>	16-60
<u>LMAP Directive</u>	16-61
<u>Usage Considerations</u>	16-61
<u>Example of LMAP Directive</u>	16-62
<u>MAP Directive</u>	16-62
<u>Usage Considerations</u>	16-62
<u>Example of MAP Directive</u>	16-63
<u>OLDFLTSTDFUNC Directive</u>	16-63
<u>Usage Considerations</u>	16-63
<u>Example of OLDFLTSTDFUNC Directive</u>	16-63
<u>OPTIMIZE Directive</u>	16-64
<u>Usage Considerations</u>	16-64
<u>Examples of OPTIMIZE Directive</u>	16-64
<u>PAGE Directive</u>	16-65

## 16. Compiler Directives (continued)

<u>Usage Considerations</u>	16-65
<u>Example of PAGE Directive</u>	16-65
<u>PEP Directive</u>	16-66
<u>Usage Considerations</u>	16-66
<u>Example of PEP Directive</u>	16-66
<u>PRINTSYM Directive</u>	16-67
<u>Usage Considerations</u>	16-67
<u>Example of PRINTSYM Directive</u>	16-67
<u>RELOCATE Directive</u>	16-67
<u>Usage Considerations</u>	16-68
<u>Example of RELOCATE Directive</u>	16-68
<u>RESETTOG Directive</u>	16-68
<u>Usage Considerations</u>	16-69
<u>Example of RESETTOG Directive</u>	16-70
<u>ROUND Directive</u>	16-70
<u>Usage Considerations</u>	16-70
<u>Example of ROUND Directive</u>	16-71
<u>RP Directive</u>	16-71
<u>Usage Considerations</u>	16-71
<u>Example of RP Directive</u>	16-72
<u>RUNNAMED Directive</u>	16-73
<u>Usage Considerations</u>	16-73
<u>Examples of RUNNAMED Directive</u>	16-73
<u>SAVEABEND Directive</u>	16-73
<u>Usage Considerations</u>	16-74
<u>Example of SAVEABEND Directive</u>	16-74
<u>SAVEGLOBALS Directive</u>	16-75
<u>Usage Considerations</u>	16-75
<u>Examples of SAVEGLOBALS Directive</u>	16-78
<u>SEARCH Directive</u>	16-79
<u>Usage Considerations</u>	16-80
<u>Examples of SEARCH Directive</u>	16-81
<u>SECTION Directive</u>	16-81
<u>Usage Considerations</u>	16-81
<u>Example of SECTION Directive</u>	16-82
<u>SETTOG Directive</u>	16-82
<u>Usage Considerations</u>	16-83
<u>Examples of SETTOG Directive</u>	16-83

## **16. Compiler Directives (continued)**

<u>SOURCE Directive</u>	16-84
<u>Usage Considerations</u>	16-85
<u>Examples of SOURCE Directive</u>	16-86
<u>SQL Directive</u>	16-86
<u>SQLMEM Directive</u>	16-86
<u>STACK Directive</u>	16-87
<u>Usage Considerations</u>	16-87
<u>Example of STACK Directive</u>	16-87
<u>SUBTYPE Directive</u>	16-87
<u>Usage Considerations</u>	16-88
<u>Example of SUBTYPE Directive</u>	16-88
<u>SUPPRESS Directive</u>	16-88
<u>Usage Considerations</u>	16-89
<u>Example of SUPPRESS Directive</u>	16-89
<u>SYMBOLPAGES Directive</u>	16-89
<u>Usage Considerations</u>	16-90
<u>Example of SYMBOLPAGES Directive</u>	16-90
<u>SYMBOLS Directive</u>	16-90
<u>Usage Considerations</u>	16-90
<u>Examples of SYMBOLS Directive</u>	16-91
<u>SYNTAX Directive</u>	16-92
<u>Usage Considerations</u>	16-92
<u>Examples of SYNTAX Directive</u>	16-92
<u>TARGET Directive</u>	16-93
<u>USEGLOBALS Directive</u>	16-93
<u>Usage Considerations</u>	16-94
<u>Example of USEGLOBALS Directive</u>	16-95
<u>WARN Directive</u>	16-95
<u>Usage Considerations</u>	16-96
<u>Example of WARN Directive</u>	16-96

## **A. Error Messages**

<u>Compiler Initialization Messages</u>	A-1
<u>About Error and Warning Messages</u>	A-1
<u>Error Messages</u>	A-2
<u>Warning Messages</u>	A-40
<u>SYMSERV Messages</u>	A-57
<u>BINSERV Messages</u>	A-57

## A. Error Messages (continued)

[Common Run-Time Environment Messages](#) A-57

## B. TAL Syntax Summary (Railroad Diagrams)

[Constants](#) B-1

[Character String Constants](#) B-1

[STRING Numeric Constants](#) B-2

[INT Numeric Constants](#) B-2

[INT\(32\) Numeric Constants](#) B-2

[FIXED Numeric Constants](#) B-2

[REAL and REAL\(64\) Numeric Constants](#) B-3

[Constant Lists](#) B-3

[Expressions](#) B-4

[Arithmetic Expressions](#) B-4

[Conditional Expressions](#) B-5

[Assignment Expressions](#) B-5

[CASE Expressions](#) B-5

[IF Expressions](#) B-6

[Group Comparison Expressions](#) B-6

[Bit Extractions](#) B-6

[Bit Shifts](#) B-7

[Declarations](#) B-7

[LITERAL and DEFINE Declarations](#) B-7

[LITERALs](#) B-7

[DEFINEs](#) B-8

[Simple Variable Declarations](#) B-8

[Array Declarations](#) B-9

[Structure Declarations](#) B-10

[Definition Structures](#) B-10

[Template Structures](#) B-11

[Referral Structures](#) B-11

[Simple Variables Declared in Structures](#) B-11

[Arrays Declared in Structures](#) B-12

[Definition Substructures](#) B-12

[Referral Substructures](#) B-12

[Fillers in Structures](#) B-13

[Simple Pointers Declared in Structures](#) B-13

[Structure Pointers Declared in Structures](#) B-13

[Simple Variable Redefinitions](#) B-13

## B. TAL Syntax Summary (Railroad Diagrams) (continued)

<u>Array Redefinitions</u>	B-14
<u>Definition Substructure Redefinitions</u>	B-14
<u>Referral Substructure Redefinitions</u>	B-14
<u>Simple Pointer Redefinitions</u>	B-15
<u>Structure Pointer Redefinitions</u>	B-15
<u>Pointer Declarations</u>	B-15
<u>Simple Pointers</u>	B-16
<u>Structure Pointers</u>	B-16
<u>Equivalenced Variable Declarations</u>	B-16
<u>Equivalenced Simple Variables</u>	B-17
<u>Equivalenced Definition Structures</u>	B-17
<u>Equivalenced Referral Structures</u>	B-18
<u>Equivalenced Simple Pointers</u>	B-18
<u>Equivalenced Structure Pointers</u>	B-19
<u>Base-Address Equivalenced Variable Declarations</u>	B-19
<u>Base-Address Equivalenced Simple Variables</u>	B-19
<u>Base-Address Equivalenced Definition Structures</u>	B-20
<u>Base-Address Equivalenced Referral Structures</u>	B-21
<u>Base-Address Equivalenced Simple Pointers</u>	B-21
<u>Base-Address Equivalenced Structure Pointers</u>	B-22
<u>NAME and BLOCK Declarations</u>	B-22
<u>NAMEs</u>	B-22
<u>BLOCKs</u>	B-22
<u>Procedure and Subprocedure Declarations</u>	B-23
<u>Procedures</u>	B-23
<u>Subprocedures</u>	B-28
<u>Entry Points</u>	B-31
<u>Labels</u>	B-31
<u>Statements</u>	B-31
<u>Compound Statements</u>	B-31
<u>ASSERT Statement</u>	B-32
<u>Assignment Statement</u>	B-32
<u>Bit Deposit Assignment Statement</u>	B-32
<u>CALL Statement</u>	B-33
<u>Labeled CASE Statement</u>	B-33
<u>Unlabeled CASE Statement</u>	B-34
<u>CODE Statement</u>	B-34
<u>DO Statement</u>	B-35

## B. TAL Syntax Summary (Railroad Diagrams) (continued)

<u>DROP Statement</u>	B-36
<u>FOR Statement</u>	B-36
<u>GOTO Statement</u>	B-36
<u>IF Statement</u>	B-36
<u>Move Statement</u>	B-37
<u>RETURN Statement</u>	B-38
<u>Scan Statement</u>	B-38
<u>STACK Statement</u>	B-38
<u>STORE Statement</u>	B-39
<u>USE Statement</u>	B-39
<u>WHILE Statement</u>	B-39
<u>Standard Functions</u>	B-39
<u>\$ABS Function</u>	B-39
<u>\$ALPHA Function</u>	B-40
<u>\$AXADR Function</u>	B-40
<u>\$BITLENGTH Function</u>	B-40
<u>\$BITOFFSET Function</u>	B-40
<u>\$BOUNDS Function</u>	B-40
<u>\$CARRY Function</u>	B-40
<u>\$COMP Function</u>	B-41
<u>\$DBL Function</u>	B-41
<u>\$DBLL Function</u>	B-41
<u>\$DBLR Function</u>	B-41
<u>\$DFIX Function</u>	B-41
<u>\$EFLT Function</u>	B-42
<u>\$EFLTR Function</u>	B-42
<u>\$FIX Function</u>	B-42
<u>\$FIXD Function</u>	B-42
<u>\$FIXI Function</u>	B-42
<u>\$FIXL Function</u>	B-43
<u>\$FIXR Function</u>	B-43
<u>\$FLT Function</u>	B-43
<u>\$FLTR Function</u>	B-43
<u>\$HIGH Function</u>	B-43
<u>\$IFIX Function</u>	B-44
<u>\$INT Function</u>	B-44
<u>\$INTR Function</u>	B-44
<u>\$LADR Function</u>	B-44

## B. TAL Syntax Summary (Railroad Diagrams) (continued)

<a href="#">\$LEN Function</a>	B-45
<a href="#">\$LFIX Function</a>	B-45
<a href="#">\$LMAX Function</a>	B-45
<a href="#">\$LMIN Function</a>	B-45
<a href="#">\$MAX Function</a>	B-45
<a href="#">\$MIN Function</a>	B-46
<a href="#">\$NUMERIC Function</a>	B-46
<a href="#">\$OCCURS Function</a>	B-46
<a href="#">\$OFFSET Function</a>	B-46
<a href="#">\$OPTIONAL Function</a>	B-46
<a href="#">\$OVERFLOW Function</a>	B-47
<a href="#">\$PARAM Function</a>	B-47
<a href="#">\$POINT Function</a>	B-47
<a href="#">\$READCLOCK Function</a>	B-47
<a href="#">\$RP Function</a>	B-48
<a href="#">\$SCALE Function</a>	B-48
<a href="#">\$SPECIAL Function</a>	B-48
<a href="#">\$SWITCHES Function</a>	B-48
<a href="#">\$TYPE Function</a>	B-48
<a href="#">\$UDBL Function</a>	B-49
<a href="#">\$USERCODE Function</a>	B-49
<a href="#">\$XADR Function</a>	B-49
<a href="#">Privileged Procedures</a>	B-49
<a href="#">System Global Pointers</a>	B-49
<a href="#">'SG'-Equivalenced Simple Variables</a>	B-50
<a href="#">'SG'-Equivalenced Definition Structures</a>	B-50
<a href="#">'SG'-Equivalenced Referral Structures</a>	B-51
<a href="#">'SG'-Equivalenced Simple Pointers</a>	B-51
<a href="#">'SG'-Equivalenced Structure Pointers</a>	B-52
<a href="#">TARGET Directive</a>	B-54
<a href="#">Compiler Directives</a>	B-54
<a href="#">Directive Lines</a>	B-54
<a href="#">ABORT Directive</a>	B-54
<a href="#">ABSLIST Directive</a>	B-55
<a href="#">ASSERTION Directive</a>	B-55
<a href="#">BEGINCOMPILATION Directive</a>	B-55
<a href="#">CHECK Directive</a>	B-55
<a href="#">CODE Directive</a>	B-56

## B. TAL Syntax Summary (Railroad Diagrams) (continued)

<a href="#">COLUMNS Directive</a>	B-56
<a href="#">COMPACT Directive</a>	B-57
<a href="#">CPU Directive</a>	B-57
<a href="#">CROSSREF Directive</a>	B-57
<a href="#">DATAPAGES Directive</a>	B-57
<a href="#">DECS Directive</a>	B-58
<a href="#">DEFEXPAND Directive</a>	B-58
<a href="#">DEFINETOГ Directive</a>	B-58
<a href="#">DUMPCONS Directive</a>	B-59
<a href="#">ENDIF Directive</a>	B-59
<a href="#">ENV Directive</a>	B-59
<a href="#">ERRORFILE Directive</a>	B-60
<a href="#">ERRORS Directive</a>	B-60
<a href="#">EXTENDSTACK Directive</a>	B-60
<a href="#">EXTENDTALHEAP Directive</a>	B-61
<a href="#">FIXUP Directive</a>	B-61
<a href="#">FMAP Directive</a>	B-61
<a href="#">GMAP Directive</a>	B-61
<a href="#">HEAP Directive</a>	B-62
<a href="#">HIGHPIN Directive</a>	B-62
<a href="#">HIGHREQUESTERS Directive</a>	B-62
<a href="#">ICODE Directive</a>	B-62
<a href="#">IF and ENDIF Directives</a>	B-63
<a href="#">INHIBITXX Directive</a>	B-63
<a href="#">INNERLIST Directive</a>	B-63
<a href="#">INSPECT Directive</a>	B-64
<a href="#">INT32INDEX Directive</a>	B-64
<a href="#">LARGESTACK Directive</a>	B-65
<a href="#">LIBRARY Directive</a>	B-65
<a href="#">LINES Directive</a>	B-65
<a href="#">LIST Directive</a>	B-65
<a href="#">LMAP Directive</a>	B-66
<a href="#">MAP Directive</a>	B-66
<a href="#">OLDFLTSTDFUNC Directive</a>	B-67
<a href="#">OPTIMIZE Directive</a>	B-67
<a href="#">PAGE Directive</a>	B-67
<a href="#">PEP Directive</a>	B-67
<a href="#">PRINTSYM Directive</a>	B-68

## B. TAL Syntax Summary (Railroad Diagrams) (continued)

<u>RELOCATE Directive</u>	B-68
<u>RESETTOG Directive</u>	B-68
<u>ROUND Directive</u>	B-69
<u>RP Directive</u>	B-69
<u>RUNNAMED Directive</u>	B-69
<u>SAVEABEND Directive</u>	B-69
<u>SAVEGLOBALS Directive</u>	B-70
<u>SEARCH Directive</u>	B-70
<u>SECTION Directive</u>	B-70
<u>SETTOG Directive</u>	B-71
<u>SOURCE Directive</u>	B-71
<u>SQL Directive</u>	B-71
<u>SQLMEM Directive</u>	B-71
<u>STACK Directive</u>	B-72
<u>SUBTYPE Directive</u>	B-72
<u>SUPPRESS Directive</u>	B-72
<u>SYMBOLPAGES Directive</u>	B-72
<u>SYMBOLS Directive</u>	B-73
<u>SYNTAX Directive</u>	B-73
<u>TARGET Directive</u>	B-73
<u>USEGLOBALS Directive</u>	B-73
<u>WARN Directive</u>	B-74

## C. TAL Syntax Summary (Bracket-and-Brace Diagrams)

<u>General Syntax Notation</u>	C-1
<u>UPPERCASE LETTERS</u>	C-1
<u>lowercase italic letters</u>	C-1
<u>Brackets []</u>	C-1
<u>Braces {}</u>	C-2
<u>Vertical Line  </u>	C-2
<u>Ellipsis ...</u>	C-2
<u>Punctuation</u>	C-2
<u>Item Spacing</u>	C-2
<u>Line Spacing</u>	C-3
<u>Constants</u>	C-3
<u>Character String Constants</u>	C-3
<u>STRING Numeric Constants</u>	C-3
<u>INT Numeric Constants</u>	C-4

## C. TAL Syntax Summary (Bracket-and-Brace Diagrams) (continued)

<u>INT(32) Numeric Constants</u>	C-4
<u>FIXED Numeric Constants</u>	C-4
<u>REAL and REAL(64) Numeric Constants</u>	C-4
<u>Constant Lists</u>	C-4
<u>Expressions</u>	C-5
<u>Arithmetic Expressions</u>	C-5
<u>Conditional Expressions</u>	C-5
<u>Assignment Expressions</u>	C-5
<u>CASE Expressions</u>	C-6
<u>IF Expressions</u>	C-6
<u>Group Comparison Expressions</u>	C-6
<u>Bit Extractions</u>	C-6
<u>Bit Shifts</u>	C-6
<u>Declarations</u>	C-6
<u>LITERAL and DEFINE Declarations</u>	C-7
<u>LITERALs</u>	C-7
<u>DEFINEs</u>	C-7
<u>Simple Variable Declarations</u>	C-7
<u>Simple Variables</u>	C-7
<u>Array Declarations</u>	C-8
<u>Arrays</u>	C-8
<u>Read-Only Arrays</u>	C-8
<u>Structure Declarations</u>	C-9
<u>Definition Structures</u>	C-9
<u>Template Structures</u>	C-9
<u>Referral Structures</u>	C-9
<u>Simple Variables Declared in Structures</u>	C-9
<u>Arrays Declared in Structures</u>	C-9
<u>Definition Substructures</u>	C-10
<u>Referral Substructures</u>	C-10
<u>Fillers in Structures</u>	C-10
<u>Simple Pointers Declared in Structures</u>	C-10
<u>Structure Pointers Declared in Structures</u>	C-10
<u>Simple Variable Redefinitions</u>	C-11
<u>Array Redefinitions</u>	C-11
<u>Definition Substructure Redefinitions</u>	C-11

## C. TAL Syntax Summary (Bracket-and-Brace Diagrams) (continued)

<u>Referral Substructure Redefinitions</u>	C-11
<u>Simple Pointer Redefinitions</u>	C-11
<u>Structure Pointer Redefinitions</u>	C-11
<u>Pointer Declarations</u>	C-12
<u>Simple Pointers</u>	C-12
<u>Structure Pointers</u>	C-12
<u>Equivalenced Variable Declarations</u>	C-12
<u>Equivalenced Simple Variables</u>	C-12
<u>Equivalenced Definition Structures</u>	C-13
<u>Equivalenced Referral Structures</u>	C-13
<u>Equivalenced Simple Pointers</u>	C-13
<u>Equivalenced Structure Pointers</u>	C-14
<u>Base-Address Equivalenced Variable Declarations</u>	C-14
<u>Base-Address Equivalenced Simple Variables</u>	C-14
<u>Base-Address Equivalenced Definition Structures</u>	C-14
<u>Base-Address Equivalenced Referral Structures</u>	C-15
<u>Base-Address Equivalenced Simple Pointers</u>	C-15
<u>Base-Address Equivalenced Structure Pointers</u>	C-15
<u>NAME and BLOCK Declarations</u>	C-16
<u>NAMEs</u>	C-16
<u>BLOCKs</u>	C-16
<u>Procedure and Subprocedure Declarations</u>	C-16
<u>Procedures</u>	C-16
<u>Subprocedures</u>	C-18
<u>Statements</u>	C-19
<u>Compound Statements</u>	C-19
<u>ASSERT Statement</u>	C-19
<u>Assignment Statement</u>	C-19
<u>Bit-Drop Assignment Statement</u>	C-20
<u>CALL Statement</u>	C-20
<u>Labeled CASE Statement</u>	C-20
<u>Unlabeled CASE Statement</u>	C-20
<u>CODE Statement</u>	C-21
<u>DO Statement</u>	C-21
<u>DROP Statement</u>	C-21
<u>FOR Statement</u>	C-21

## C. TAL Syntax Summary (Bracket-and-Brace Diagrams) (continued)

<a href="#">GOTO Statement</a>	C-21
<a href="#">IF Statement</a>	C-22
<a href="#">Move Statement</a>	C-22
<a href="#">RETURN Statement</a>	C-22
<a href="#">SCAN Statement</a>	C-22
<a href="#">STACK Statement</a>	C-23
<a href="#">STORE Statement</a>	C-23
<a href="#">USE Statement</a>	C-23
<a href="#">WHILE Statement</a>	C-23
<a href="#">Standard Functions</a>	C-23
<a href="#">\$ABS Function</a>	C-23
<a href="#">\$ALPHA Function</a>	C-23
<a href="#">\$AXADR Function</a>	C-24
<a href="#">\$BITLENGTH Function</a>	C-24
<a href="#">\$BITOFFSET Function</a>	C-24
<a href="#">\$BOUNDS Function</a>	C-24
<a href="#">\$CARRY Function</a>	C-24
<a href="#">\$COMP Function</a>	C-24
<a href="#">\$DBL Function</a>	C-24
<a href="#">\$DBLL Function</a>	C-24
<a href="#">\$DBLR Function</a>	C-25
<a href="#">\$DFIX Function</a>	C-25
<a href="#">\$EFLT Function</a>	C-25
<a href="#">\$EFLTR Function</a>	C-25
<a href="#">\$FIX Function</a>	C-25
<a href="#">\$FIXD Function</a>	C-25
<a href="#">\$FIXI Function</a>	C-26
<a href="#">\$FIXL Function</a>	C-26
<a href="#">\$FIXR Function</a>	C-26
<a href="#">\$FLT Function</a>	C-26
<a href="#">\$FLTR Function</a>	C-26
<a href="#">\$HIGH Function</a>	C-26
<a href="#">\$IFIX Function</a>	C-26
<a href="#">\$INT Function</a>	C-27
<a href="#">\$INTR Function</a>	C-27
<a href="#">\$LADR Function</a>	C-27

## C. TAL Syntax Summary (Bracket-and-Brace Diagrams) (continued)

<u>\$LEN Function</u>	C-27
<u>\$LFIX Function</u>	C-27
<u>\$LMAX Function</u>	C-27
<u>\$LMIN Function</u>	C-28
<u>\$MAX Function</u>	C-28
<u>\$MIN Function</u>	C-28
<u>\$NUMERIC Function</u>	C-28
<u>\$OCCURS Function</u>	C-28
<u>\$OFFSET Function</u>	C-28
<u>\$OPTIONAL Function</u>	C-28
<u>\$OVERFLOW Function</u>	C-29
<u>\$PARAM Function</u>	C-29
<u>\$POINT Function</u>	C-29
<u>\$READCLOCK Function</u>	C-29
<u>\$RP Function</u>	C-29
<u>\$SCALE Function</u>	C-29
<u>\$SPECIAL Function</u>	C-30
<u>\$SWITCHES Function</u>	C-30
<u>\$TYPE Function</u>	C-30
<u>\$UDBL Function</u>	C-30
<u>\$USERCODE Function</u>	C-30
<u>\$XADR Function</u>	C-30
<u>Privileged Procedures</u>	C-30
<u>System Global Pointers</u>	C-30
<u>'SG'-Equivalenced Simple Variables</u>	C-31
<u>'SG'-Equivalenced Definition Structures</u>	C-31
<u>'SG'-Equivalenced Referral Structures</u>	C-31
<u>'SG'-Equivalenced Simple Pointers</u>	C-31
<u>'SG'-Equivalenced Structure Pointers</u>	C-32
<u>\$AXADR Function</u>	C-32
<u>\$BOUNDS Function</u>	C-32
<u>\$SWITCHES Function</u>	C-32
<u>TARGET Directive</u>	C-33
<u>Compiler Directives</u>	C-33
<u>Directive Lines</u>	C-33
<u>ABORT Directive</u>	C-33

## C. TAL Syntax Summary (Bracket-and-Brace Diagrams) (continued)

<u>ABSLIST Directive</u>	C-33
<u>ASSERTION Directive</u>	C-33
<u>BEGINCOMPILATION Directive</u>	C-34
<u>CHECK Directive</u>	C-34
<u>CODE Directive</u>	C-34
<u>COLUMNS Directive</u>	C-34
<u>COMPACT Directive</u>	C-34
<u>CPU Directive</u>	C-34
<u>CROSSREF Directive</u>	C-35
<u>DATAPAGES Directive</u>	C-35
<u>DECS Directive</u>	C-35
<u>DEFEXPAND Directive</u>	C-35
<u>DEFINETOG Directive</u>	C-35
<u>DUMPCONS Directive</u>	C-36
<u>ENDIF Directive</u>	C-36
<u>ENV Directive</u>	C-36
<u>ERRORFILE Directive</u>	C-36
<u>ERRORS Directive</u>	C-36
<u>EXTENDSTACK Directive</u>	C-36
<u>EXTENDTALHEAP Directive</u>	C-36
<u>FIXUP Directive</u>	C-37
<u>FMAP Directive</u>	C-37
<u>GMAP Directive</u>	C-37
<u>HEAP Directive</u>	C-37
<u>HIGHPIN Directive</u>	C-37
<u>HIGHREQUESTERS Directive</u>	C-37
<u>ICODE Directive</u>	C-37
<u>IF and ENDIF Directive</u>	C-38
<u>INHIBITXX Directive</u>	C-38
<u>INNERLIST Directive</u>	C-38
<u>INSPECT Directive</u>	C-38
<u>INT32INDEX Directive</u>	C-39
<u>LARGESTACK Directive</u>	C-39
<u>LIBRARY Directive</u>	C-39
<u>LINES Directive</u>	C-39
<u>LIST Directive</u>	C-39

## C. TAL Syntax Summary (Bracket-and-Brace Diagrams) (continued)

<a href="#"><u>LMAP Directive</u></a>	C-39
<a href="#"><u>MAP Directive</u></a>	C-40
<a href="#"><u>OLDFLTSTDFUNC Directive</u></a>	C-40
<a href="#"><u>OPTIMIZE Directive</u></a>	C-40
<a href="#"><u>PAGE Directive</u></a>	C-40
<a href="#"><u>PEP Directive</u></a>	C-40
<a href="#"><u>PRINTSYM Directive</u></a>	C-40
<a href="#"><u>RELOCATE Directive</u></a>	C-41
<a href="#"><u>RESETTOG Directive</u></a>	C-41
<a href="#"><u>RP Directive</u></a>	C-41
<a href="#"><u>RUNNAMED Directive</u></a>	C-41
<a href="#"><u>SAVEABEND Directive</u></a>	C-41
<a href="#"><u>SAVEGLOBALS Directive</u></a>	C-42
<a href="#"><u>SEARCH Directive</u></a>	C-42
<a href="#"><u>SECTION Directive</u></a>	C-42
<a href="#"><u>SETTOG Directive</u></a>	C-42
<a href="#"><u>SOURCE Directive</u></a>	C-42
<a href="#"><u>SQL Directive</u></a>	C-43
<a href="#"><u>SQLMEM Directive</u></a>	C-43
<a href="#"><u>STACK Directive</u></a>	C-43
<a href="#"><u>SUBTYPE Directive</u></a>	C-43
<a href="#"><u>SUPPRESS Directive</u></a>	C-43
<a href="#"><u>SYMBOLPAGES Directive</u></a>	C-43
<a href="#"><u>SYMBOLS Directive</u></a>	C-43
<a href="#"><u>SYNTAX Directive</u></a>	C-43
<a href="#"><u>USEGLOBALS Directive</u></a>	C-44
<a href="#"><u>WARN Directive</u></a>	C-44

## Glossary

## Index

## **Tables**

<a href="#"><u>Table i.</u></a>	<a href="#"><u>TAL Manual Set</u></a>	xlii
<a href="#"><u>Table ii.</u></a>	<a href="#"><u>NonStop Systems</u></a>	xliv
<a href="#"><u>Table iii.</u></a>	<a href="#"><u>System Manuals</u></a>	xlv
<a href="#"><u>Table iv.</u></a>	<a href="#"><u>Programming Manuals</u></a>	xlvi

## Tables (continued)

<a href="#">Table v.</a>	<a href="#">Program Development Manuals</a>	xlvi
<a href="#">Table 1-1.</a>	<a href="#">Uses of TAL</a>	1-1
<a href="#">Table 2-1.</a>	<a href="#">TAL Statements</a>	2-2
<a href="#">Table 2-2.</a>	<a href="#">Keywords</a>	2-3
<a href="#">Table 2-3.</a>	<a href="#">Nonreserved Keywords</a>	2-3
<a href="#">Table 2-4.</a>	<a href="#">Identifier Classes</a>	2-5
<a href="#">Table 2-5.</a>	<a href="#">Variables</a>	2-6
<a href="#">Table 2-6.</a>	<a href="#">Indirection Symbols</a>	2-7
<a href="#">Table 2-7.</a>	<a href="#">Base Address Symbols</a>	2-7
<a href="#">Table 2-8.</a>	<a href="#">Delimiters</a>	2-8
<a href="#">Table 2-9.</a>	<a href="#">Operators</a>	2-9
<a href="#">Table 3-1.</a>	<a href="#">Data Types</a>	3-2
<a href="#">Table 3-2.</a>	<a href="#">Storage Units</a>	3-5
<a href="#">Table 3-3.</a>	<a href="#">Operations by Data Type</a>	3-5
<a href="#">Table 3-4.</a>	<a href="#">Standard Functions by Data Type</a>	3-6
<a href="#">Table 3-5.</a>	<a href="#">Address Types</a>	3-7
<a href="#">Table 4-1.</a>	<a href="#">Precedence of Operators</a>	4-3
<a href="#">Table 4-2.</a>	<a href="#">Operands in Arithmetic Expressions</a>	4-6
<a href="#">Table 4-3.</a>	<a href="#">Signed Arithmetic Operators</a>	4-7
<a href="#">Table 4-4.</a>	<a href="#">Signed Arithmetic Operand and Result Types</a>	4-7
<a href="#">Table 4-5.</a>	<a href="#">Unsigned Arithmetic Operators</a>	4-9
<a href="#">Table 4-6.</a>	<a href="#">Unsigned Arithmetic Operand and Result Types</a>	4-10
<a href="#">Table 4-7.</a>	<a href="#">Logical Operators and Result Yielded</a>	4-11
<a href="#">Table 4-8.</a>	<a href="#">Conditions in Conditional Expressions</a>	4-13
<a href="#">Table 4-9.</a>	<a href="#">Boolean Operators and Result Yielded</a>	4-14
<a href="#">Table 4-10.</a>	<a href="#">Signed Relational Operators and Result Yielded</a>	4-15
<a href="#">Table 4-11.</a>	<a href="#">Unsigned Relational Operators and Result Yielded</a>	4-15
<a href="#">Table 4-12.</a>	<a href="#">Special Expressions</a>	4-19
<a href="#">Table 4-13.</a>	<a href="#">Bit - Operations</a>	4-27
<a href="#">Table 4-14.</a>	<a href="#">Bit-Shift Operators</a>	4-30
<a href="#">Table 8-1.</a>	<a href="#">Kinds of Structures</a>	8-1
<a href="#">Table 8-2.</a>	<a href="#">Structure Items</a>	8-2
<a href="#">Table 8-3.</a>	<a href="#">Data Accessed by Simple Pointers</a>	8-14
<a href="#">Table 8-4.</a>	<a href="#">Addresses in Simple Pointers</a>	8-14
<a href="#">Table 8-5.</a>	<a href="#">Addresses in Structure Pointers</a>	8-16
<a href="#">Table 9-1.</a>	<a href="#">Data Accessed by Simple Pointers</a>	9-3
<a href="#">Table 9-2.</a>	<a href="#">Addresses in Simple Pointers</a>	9-3
<a href="#">Table 9-3.</a>	<a href="#">Addresses in Structure Pointers</a>	9-7

**Tables (continued)**

<a href="#"><u>Table 10-1.</u></a>	<a href="#"><u>Equivalenced Variables</u></a>	10-1
<a href="#"><u>Table 12-1.</u></a>	<a href="#"><u>Summary of Statements</u></a>	12-1
<a href="#"><u>Table 13-1.</u></a>	<a href="#"><u>Formal Parameter Specification</u></a>	13-12
<a href="#"><u>Table 14-1.</u></a>	<a href="#"><u>Summary of Standard Functions</u></a>	14-2
<a href="#"><u>Table 14-2.</u></a>	<a href="#"><u>Type-Transfer Functions by Data Type</u></a>	14-4
<a href="#"><u>Table 14-3.</u></a>	<a href="#"><u>Address type-transfer functions</u></a>	14-45
<a href="#"><u>Table 14-4.</u></a>	<a href="#"><u>pTAL built-ins</u></a>	14-45
<a href="#"><u>Table 16-1.</u></a>	<a href="#"><u>Summary of Compiler Directives</u></a>	16-6

## Contents

# What's New in This Manual

## Manual Information

### Abstract

This manual provides syntax descriptions and error messages for TAL (Transaction Application Language) for system and application programmers.

### Product Version

TAL D40

### Supported Release Version Updates (RVUs)

This publication supports D40.00 and all subsequent D-series RVUs, and G01.00 and all subsequent G-series RVUs until otherwise indicated in a new edition.

Part Number	Published
526371-001	September 2003

### Document History

Part Number	Product Version	Published
15998	TAL C20	March 1989
065722	TAL D10	January 1993
096255	TAL C30, TAL D10 & TAL D20	September 1993
526371-001	TAL D40	September 2003

## New and Changed Information

- Since product names are changing over time, this publication might contain both HP and Compaq product names.
- Product names in graphic representations are consistent with the current product interface.
- There was an error in describing the \$OPTIONAL function. Hence, this release ensures the correct description of the \$OPTIONAL function.
- Adding a description as to how to write user library code files was an issue, which has been solved in this release of the manual. Now, it explains the same by referring it to the *Binder* manual.
- Compiler error 88 was very complex. Hence, this release ensures that the explanation has been simplified for easy understanding.

- Added [Built-in Functions](#) on page 14-44 to document that TAL directly implements pTAL built-ins



# About This Manual

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

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

This manual describes the syntax for using TAL and the TAL compiler. It describes:

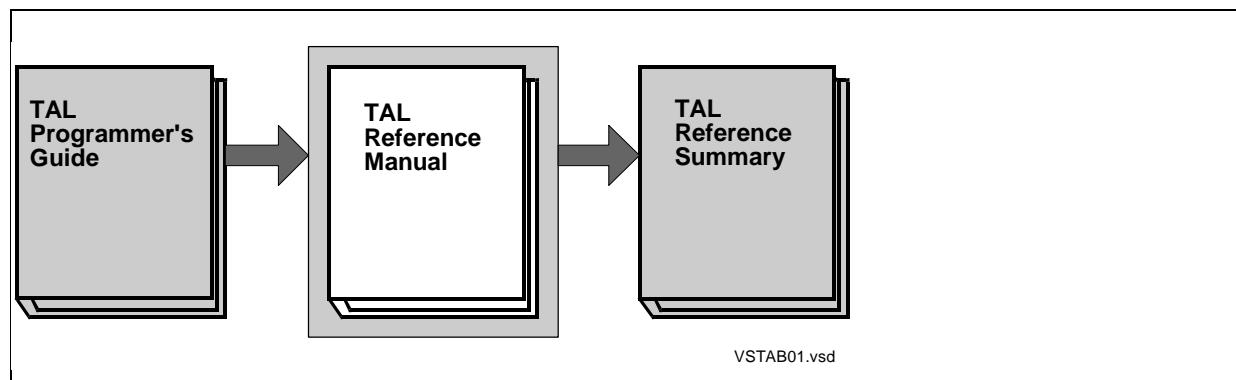
- The syntax for declaring variables and procedures
- The syntax for specifying expressions, statements, standard functions, and compiler directives
- Error and warning messages

## Audience

This manual is intended for system programmers and application programmers who are familiar with NonStop systems and the NonStop Kernel operating system.

## How to Use this Manual Set

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



---

**Table i. TAL Manual Set**

<b>Manual</b>	<b>Description</b>
<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.

For more information about 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.

For more information on 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*.

## Manual Organization

This *TAL Reference Manual* covers these topics:

<a href="#"><u>Section 1, Introduction</u></a>	summarizes the features of TAL
<a href="#"><u>Section 2, Language Elements</u></a>	summarizes language elements such as reserved words, identifiers, constants, number bases, symbols, and operators
<a href="#"><u>Section 3, Data Representation</u></a>	describes data types, storage units, character strings, numeric constants, and constant lists
<a href="#"><u>Section 4, Expressions</u></a>	describes the syntax for specifying arithmetic, conditional, and special expressions
<a href="#"><u>Section 5, LITERALs and DEFINEs</u></a>	describes the syntax for LITERAL and DEFINE declarations
<a href="#"><u>Section 6, Simple Variables</u></a>	describes the syntax for declaring simple variables
<a href="#"><u>Section 7, Arrays</u></a>	describes the syntax for declaring arrays and read-only arrays
<a href="#"><u>Section 8, Structures</u></a>	describes the syntax for declaring structures and structure items
<a href="#"><u>Section 9, Pointers</u></a>	describes the syntax for declaring simple pointers and structure pointers

<a href="#"><u>Section 10, Equivalenced Variables</u></a>	describes the syntax for declaring equivalenced variables
<a href="#"><u>Section 11, NAMEs and BLOCKs</u></a>	describes the syntax for NAME and BLOCK declarations
<a href="#"><u>Section 12, Statements</u></a>	describes the syntax for specifying statements
<a href="#"><u>Section 13, Procedures</u></a>	describes the syntax for declaring procedures, subprocedures, entry points, and labels
<a href="#"><u>Section 14, Standard Functions</u></a>	describes the syntax for using standard functions
<a href="#"><u>Section 15, Privileged Procedures</u></a>	describes the syntax for declaring system global pointers and 'SG'-equivalenced variables and for using privileged standard functions
<a href="#"><u>Section 16, Compiler Directives</u></a>	describes the syntax for specifying compiler directives
<a href="#"><u>Appendix A, Error Messages</u></a>	describes error and warning messages
<a href="#"><u>Appendix B, TAL Syntax Summary (Railroad Diagrams)</u></a>	presents a syntax summary
<a href="#"><u>Appendix C, TAL Syntax Summary (Bracket-and- Brace Diagrams)</u></a>	presents the syntax summary in bracket-and-brace format

## System Dependencies

The features mentioned in this manual are supported on all currently supported systems except where noted. [Table ii](#) on page -xl lists the systems that TAL supports:

**Table ii. NonStop Systems**

System Name	Description	Operating System
HP 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	G-series
HP 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	G-series software on S-series hardware

## 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 S-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 compiler is a disk-resident program on each NonStop 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 iii](#) describes manuals that provide information about NonStop systems.

---

### Table iii. System Manuals

Manual	Description
<i>Introduction to HP 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 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 iv](#) describes manuals about programming in the NonStop environment.

---

#### **Table iv. Programming Manuals**

<b>Manual</b>	<b>Description</b>
<i>Guardian Procedure Calls Reference Manual</i>	Gives the syntax and programming considerations for using system procedures.
<i>Guardian Programmer's Guide</i>	Tells 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>	Tells 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 v](#) describes manuals about program development tools.

---

#### **Table v. Program Development Manuals**

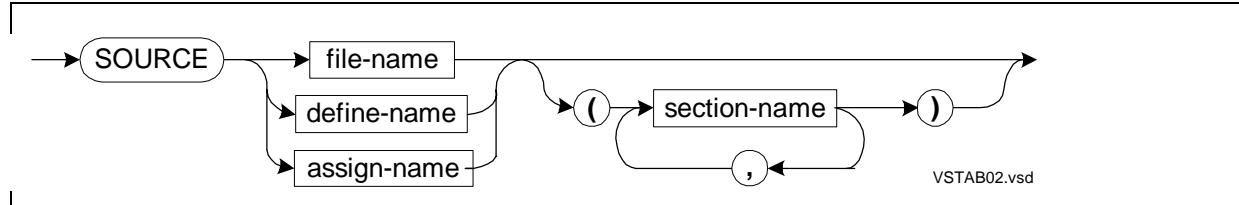
<b>Manual</b>	<b>Description</b>
<i>PS Text Edit Reference Manual</i>	Explains how to create and edit a text file using the PS Text Edit full-screen text editor.
<i>Edit User's Guide and Reference Manual</i>	Explains how to create and edit a text file using the Edit line and virtual-screen text editor.
<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.

---

# Notation Conventions

## Railroad Diagrams

This manual presents syntax in railroad diagrams. To use a railroad diagram, follow the direction of the arrows and specify syntactic items as indicated by the diagram and the term definitions that follow the diagram. Here is an example of a railroad diagram:



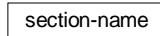
The parts of the diagram have the following meanings:



Specify the keyword as shown, using uppercase or lowercase



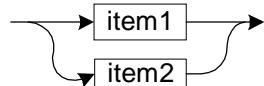
Specify the symbol or punctuation as shown.



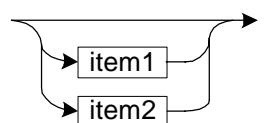
Supply the information indicated, using uppercase or lowercase

## Branching

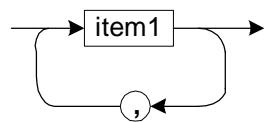
Branching lines indicate a choice, such as:



Required choice. Specify one of the items.



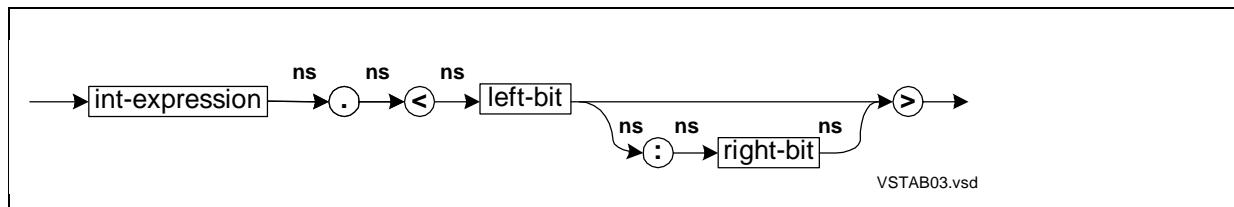
Optional choice. Specify one or none of the items.



Repeatable choice. Specify one or more of the items.

## Spacing

Where no space is allowed, the notation *ns* appears in the railroad diagram. Here is an example of a diagram in which spaces are not allowed:



You can prefix identifiers of standard indirect variables with the standard indirection symbol (.) with no intervening space.

In all other cases, if no separator—such as a comma, semicolon, or parenthesis—is shown, separate syntactic items with at least one space.

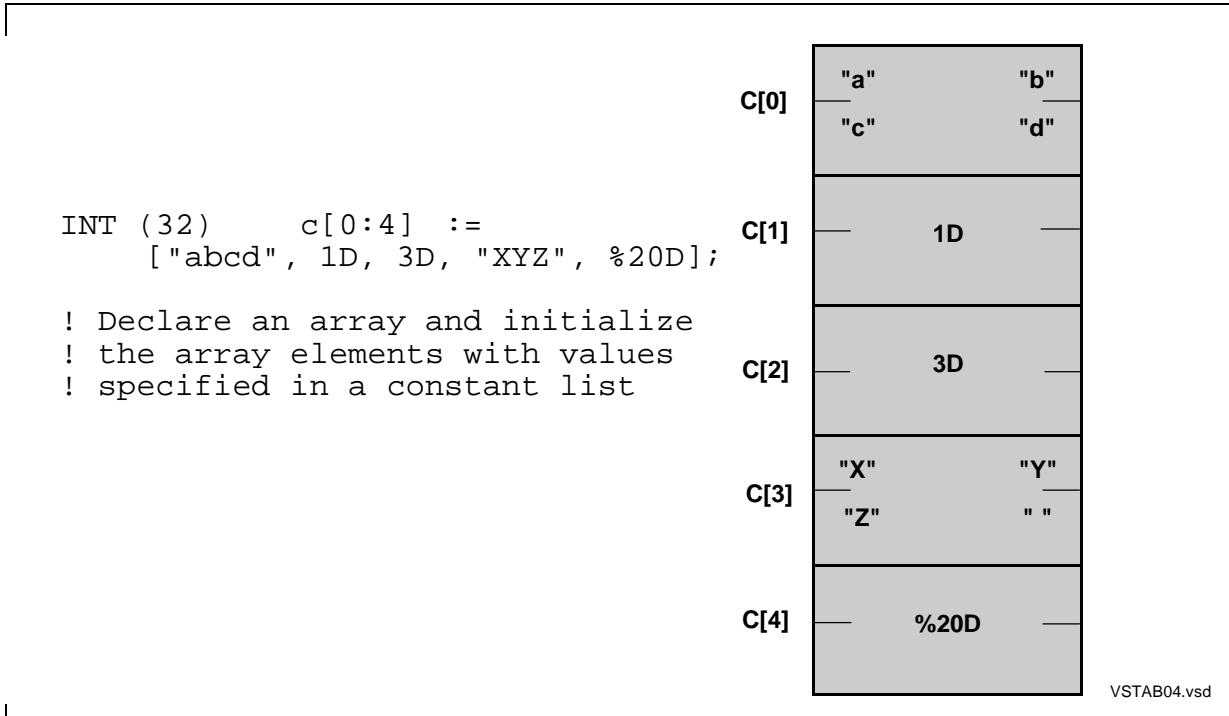
## Case Conventions

Case conventions apply to keywords, variable items (information you supply), and identifiers. This manual presents these terms in uppercase or lowercase as follows:

Term	Context	Case	Example
Keywords	In text and in railroad diagrams	Uppercase	RETURN
Variable Items	In railroad diagrams	Lowercase	file-name
Variable items	In text	Lowercase <i>italics</i>	file-name
Identifiers	In examples	Lowercase	INT total;
Identifiers	In text	Uppercase	TOTAL

## Example Diagrams

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:



## Hypertext Links

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

This requirement is described under [Backup DAM Volumes and Physical Disk Drives](#) on page 3-2.

## General Syntax Notation

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

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

MAXATTACH

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

*file-name*

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

myfile.c

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

*pathname*

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

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

INT[ERRUPTS]

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

FC [ num ]  
[ -num ]  
[ text ]

K [ X | D ] address

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

LISTOPENS PROCESS { \$appl-mgr-name }  
{ \$process-name }  
ALLOWSU { ON | OFF }

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

INSPECT { OFF | ON | SAVEABEND }

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

M address [ , new-value ]...  
[ - ] {0|1|2|3|4|5|6|7|8|9}...

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

"s-char..."

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

error := NEXTFILENAME ( file-name ) ;  
LISTOPENS SU \$process-name.#su-name

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

# Notation for Messages

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

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

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

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

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

```
Backup Up.
```

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

```
p-register  
process-name
```

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

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

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

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

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

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

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

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

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

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

```
%005400
```

```
%B101111
```

```
%H2F
```

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

## Notation for Management Programming Interfaces

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

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

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

**lowercase letters.** Words in lowercase letters are words that are part of the notation, including Data Definition Language (DDL) keywords. For example:

```
token-type
```

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

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

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

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



# 1

# Introduction

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 NonStop Kernel.

## Applications and Uses

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 NonStop data management software Conversion routines that allow data transfer between NonStop software and other applications Procedures that are callable from programs written in other languages Applications that require optimal performance

---

Many NonStop 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. Upon termination, each activation 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 another procedure 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 quadrupletword
REAL, REAL(32)	32-bit floating-point doubleword
REAL(64)	64-bit floating-point quadrupletword
UNSIGNED(n)	n-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 pointers 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.

For more information on System procedures, see 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 in the *TAL Programmer's Guide*)
- 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. For more information on CRE guidelines for TAL programs, see Section 17, “Mixed-Language Programming”. The *CRE Programmer’s Guide* describes the services provided by the CRE, including the math, string, and SMU functions.

# **2** Language Elements

This section lists the elements that make up the TAL language. The elements listed include:

Character Set	<a href="#">2-1</a>
Declarations	<a href="#">2-1</a>
Statements	<a href="#">2-2</a>
Identifiers	<a href="#">2-4</a>
Keywords	<a href="#">2-2</a>
Constants	<a href="#">2-5</a>
Variables	<a href="#">2-6</a>
Indirection symbols	<a href="#">2-7</a>
Address base symbols	<a href="#">2-7</a>
Delimiters	<a href="#">2-8</a>
Operators	<a href="#">2-9</a>

## **Character Set**

TAL supports the complete ASCII character set, which includes:

- Uppercase and lowercase alphabetic characters
- Numeric characters (0 through 9)
- Special characters

For more information on the ASCII character set, see Appendix D of the *TAL Programmer's Guide*.

## **Declarations**

Declarations allocate storage and associate identifiers with variables and other declarable objects in a program.

Variables include data items such as simple variables, arrays, structures, pointers, and equivalenced variables.

Other declarable objects include procedures, LITERALs, DEFINEs, labels, and entry points.

# Statements

Statements specify operations to be performed on declared objects. Statements are summarized in [Table 2-1](#) and described in [Section 12, Statements](#).

**Table 2-1. TAL Statements**

Statement	Operation
ASSERT	Conditionally calls an error-handling procedure
Assignment	Stores a value in a variable
CALL	Calls 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 condition is true
DROP	Frees an index register or removes a label from the symbol table
FOR	Executes a pretest loop <i>n</i> times
GOTO	Unconditionally branches to a label within a procedure or subprocedure
IF	Selects the THEN statement for a true state or the ELSE statement for a false state
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, and can also return a condition code value
RSCAN	Scans data, right to left, for a test character
SCAN	Scans data, 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

# Keywords

Keywords have predefined meanings to the compiler when used as shown in the syntax diagrams in this manual. lists keywords that are reserved by the compiler. Do not use reserved keywords for your identifiers.

**Table 2-2. Keywords**


---

AND	DO	FORWARD	MAIN	RETURN	TO
ASSERT	DOWNTO	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 2-3](#) lists nonreserved keywords, which you can use as identifiers anywhere identifiers are allowed, except as noted in the Restrictions column.

**Table 2-3. 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.
FORTRAN	
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.

---

# Identifiers

Identifiers are names you declare for objects such as variables, LITERALs, DEFINEs, and procedures (including functions). Identifiers must conform to the following rules:

- They can be up to 31 characters long.
- They can begin with an alphabetic character, an underscore (\_), or a circumflex (^).
- They can contain alphabetic characters, numeric characters, underscores, or circumflexes.
- They can contain lowercase and uppercase characters. The compiler treats them all as uppercase.
- They cannot be reserved keywords, which are listed in [on page 2-2](#).
- They can be nonreserved keywords, except as noted in [Table 2-3](#) on page 2-3.

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  
TANDEM  
_23456789012_00  
name_with_exactly_31_characters
```

The following identifiers are incorrect:

2abc	!Begins with number
ab%99	!% symbol not allowed
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_
```

## Identifier Classes

Each identifier is a member of an identifier class such as variable. The compiler determines the identifier class based on how you declare the identifier. The compiler stores the identifier information in the symbol table.

**Table 2-4. Identifier Classes**

<b>Class</b>	<b>Meaning</b>
Block	Global Data Block
Code	Read-only (P-relative) array
Variable	Simple variable, array, simple pointer, structure pointer, structure, or structure data item
DEFINE	Named text
Function	Procedure or subprocedure that returns a value
Label	Statement Label
LITERAL	Named constant
PROC	Procedure or subprocedure that does not return a value
Register	Index register - R5, R6, or R7
Template	Template Structure

## 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 depends on the data type of the variable, as described in [Data Representation](#) on page 3-1. The following examples show constants:

```
654      - !Numeric constant
"abc"     - !Character string constant
```

## Constant Expressions

A constant expression is an arithmetic expression that contains only constants, LITERALS, and DEFINES as operands. You can use a constant expression anywhere a single constant is allowed. The following examples show constant expressions:

```
255
8 * 5 + 45 / 2
```

For more information, see [LITERALS and DEFINES](#) on page 5-1.

## Number Bases

You can specify numeric constants in binary, octal, decimal, or hexadecimal base depending on the data type of the item, as described in [Data Representation](#) on page 3-1. The default number base in TAL is decimal. The following examples show constants in each number base:

Decimal	47
Binary	%B101111
Octal	%57
Hexadecimal	%H2F

## Variables

A variable is a symbolic representation of data. It can be a single-element variable or a multiple-element variable. You use variables to store data that can change during program execution.

The compiler does not automatically initialize variables. Therefore, before you access data stored in a variable, either:

- Initialize the variable with a value when you declare the variable
- Assign a value to the variable after you declare the variable. [Table 2-5](#) summarizes variables.

---

**Table 2-5. 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 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

---

## Symbols

Symbols indicate indirection, address bases, and delimiters.

## Indirection Symbols

Indirection symbols let you use indirect addressing to save space in limited storage areas, as described in the *TAL Programmer's Guide*. Indirect addressing requires two memory references, first to a location that contains an address and then to the data located at the address. [Table 2-6](#) lists indirection symbols.

---

**Table 2-6. Indirection Symbols**

Symbol	Meaning
. (period)	Declares an array or structure as having standard (16-bit) indirect addressing Declares a standard (16-bit) simple pointer or structure pointer
.EXT	Declares an array or structure as having extended (32-bit) indirect addressing Declares an extended (32-bit) simple pointer or structure pointer
.SG	Declares a standard (16-bit) system global pointer

---

## Base Address Symbols

Base address symbols let you associate variables with locations relative to the base address of a storage area, such as the global, local, or sublocal areas of the user data segment. [Table 2-7](#) lists base address symbols.

---

**Table 2-7. Base Address Symbols**

Symbol	Meaning
'P'	P-register addressing (read-only array declaration)
'G'	Base-address equivalencing, global user data area
'L'	Base-address equivalencing, local user data area
'S'	Base-address equivalencing, sublocal user data area
'SG'	Base address equivalencing, system global space (privileged procedures only)

---

The *TAL Programmer's Guide* describes the storage areas of the user data segment.

## Delimiters

Delimiters are symbols that begin, end, or separate fields of information. Delimiters tell the compiler how to handle the fields of information. [Table 2-8](#) on page 2-8 lists delimiters.

**Table 2-8. Delimiters** (page 1 of 2)

<b>Symbol</b>	<b>Character Representation</b>	<b>Uses</b>
!	Exclamation mark	Begins and optionally ends a comment
--	Two hyphens	Begins a comment
,	Comma	Separates fields of information, such as in declarations, statements, directives, and constant lists
;	Semicolon	Terminates data declarations Separates statements Separates declaration options
.	Period	Separates identifier levels in a qualified structure item identifier
< n:n>	Angle brackets	Delimit a bit field in a bit deposit or bit extraction
:	Colon	Denotes a statement label Denotes a procedure entry point Denotes an ASSERT statement assert level Denotes a parameter pair
()	Parentheses	Delimit subexpressions within an expression Delimit instructions in a CODE statement Delimit the parameter list of a DEFINE, procedure, subprocedure, or CALL statement Delimit the referral in a structure pointer declaration Delimit the implied decimal point position in a FIXED variable
[n:n]	Square brackets	Delimit the bounds specification in the declaration of an array, structure, or substructure
->	Hyphen plus right angle bracket	Begins one or more labels in a labeled CASE statement Begins a next-addr clause in a SCAN or RSCAN statement Begins a next-addr clause in a move statement Begins a next-addr clause in a group comparison expression
“string”	Quotation marks	Delimit a character string
““	Contiguous quotation marks	The first quotation mark indicates that the second quotation mark is not a delimiter in a character string

**Table 2-8. Delimiters** (page 2 of 2)

Symbol	Character Representation	Uses
=	Equal sign	Used in LITERAL declarations Used in equivalenced variable declarations Used in redefinition declarations
=body#	Equal sign and hash mark	Delimit the DEFINE body in a DEFINE declaration
'	Single quotation marks	Delimit a comma that is not a delimiter in a DEFINE parameter
\$	Dollar sign	Denotes a standard function, such as \$ABS and \$DBL
?	Question mark	Begins a directive line

## Operators

Operators specify operations—such as arithmetic or assignments—that you want performed on items. [Table 2-9](#) describes operators.

**Table 2-9. Operators** (page 1 of 2)

Context	Operator	Operation
Assignment	:=	Data declaration initialization; assignment statement, FOR statement, and assignment expression
Move statement	:=' '=:' &	Left-to-right move Right-to-left move Concatenated move
Labeled case statement	.. (two periods)	Inclusive range of case labels
Remove indirection	@	Accesses the address contained in a pointer or the address of a non-pointer item
Repetition	* (asterisk)	Repetition factor in a repetition constant list
Template structure	(*)	Template structure declaration
FIXED(*) parameter type	(*)	Value parameter to be treated as FIXED(0)
Dereferencing *	. (period)	Converts value of INT simple variable to standard word address of another data item.
Bit-field access	. (period)	Accesses a bit-deposit or bit-extraction field (< n > or < n:n >)

**Table 2-9. Operators** (page 2 of 2)

Context	Operator	Operation
Bit shift	<<	Signed left shift
	>>	Signed right shift
	'<<'	Unsigned left shift
	'>>'	Unsigned right shift
Arithmetic expression	+	Signed addition
	-	Signed subtraction
	* (asterisk)	Signed multiplication
	/	Signed division
	'+'	Unsigned addition
	'-'	Unsigned subtraction
	'*'	Unsigned multiplication
	'/'	Unsigned division
	'\'	Unsigned modulo division
	LOR	Logical OR bit-wise operation
	LAND	Logical AND bit-wise operation
	XOR	Exclusive OR bit-wise operation
Relational expression	<	Signed less than
	=	Signed equal to
	>	Signed greater than
	<=	Signed less than or equal to
	>=	Signed greater than or equal to
	<>	Singed not equal to
	'<'	Unsigned less than
	'='	Unsigned equal to
	'>'	Unsigned greater than
	'<='	Unsigned less than or equal to
	'>='	Unsigned greater than or equal to
	'<>'	Unsigned not equal to
Boolean expression	AND	Logical conjunction
	OR	Logical disjunction
	NOT	Logical negation

\* Not supported on future software platforms.

# **3 Data Representation**

Data is the information on which a program operates. Your program data includes variables and constants.

Variables hold values that can change during program execution. When you declare a variable, you specify a data type that determines the amount of storage the variable requires, the kind of values it can represent, and other characteristics.

Constants are values that do not change during program execution. The compiler determines the data type of constants from their size and format. You can assign constants to variables. You can declare LITERALS, which associate identifiers with constants.

This section describes:

- Data types of variables and constants
- Storage units in which you can access variables
- Syntax for character string constants, numeric constants, and constant lists

## **Data Types**

When you declare most kinds of variables, you specify a data type. The data type determines:

- The kind of values the variable can represent
- 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 3-1 gives information about each data type.

**Table 3-1. Data Types**

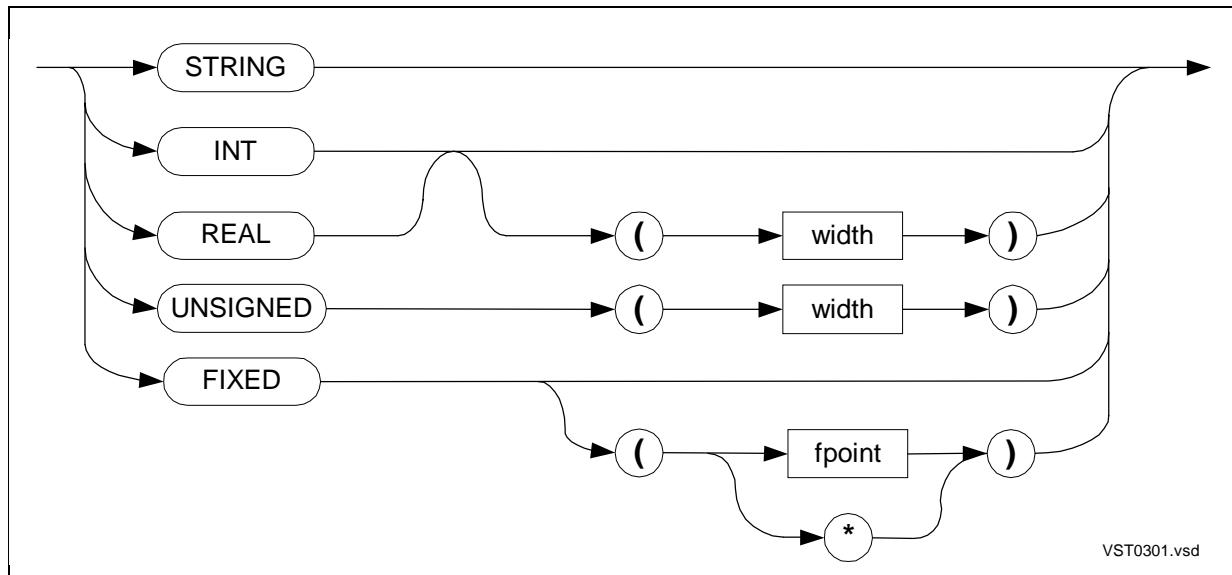
<b>Data Type</b>	<b>Storage Unit</b>	<b>Kind of Values the Data type can Represent</b>
STRING	Byte	An ASCII character An 8-bit integer in the range 0 through 255 unsigned
INT	Word	One or two ASCII characters A 16-bit integer in the range 0 through 65,535 (unsigned) or –32,768 through 32,767 (signed) 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	Double word	A 32-bit floating-point number in the range $\pm 8.6361685550944446 \times 10^{-78}$ through $\pm 1.15792089237316189 \times 10^{77}$ 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.

## Specifying Data Types

The format for specifying data types in declarations is:



### width

is a constant expression that specifies the width, in bits, of the variable. 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, in bit</i>
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 element	1, 2, 4, or 8

\* INT(16), INT(64), and REAL(32) are data type aliases, as described in [Data Type Aliases](#) on page 3-4

### fpoint

is the implied fixed-point setting of a FIXED variable. *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. A negative *fpoint* specifies a number of integer places to the left of the decimal point.

### \* (asterisk)

is a FIXED data type notation. The asterisk prevents scaling of the initialization value.

## Specifying Widths

When you specify the width of the INT, REAL, or UNSIGNED data type, the constant expression can include LITERALS and DEFINEs. Here is an example that includes a LITERAL:

```
LITERAL int_size = (2 * 4) + 8;      !INT_SIZE equals 16
INT(int_size) num;                  !Data type is INT(16)
```

[Section 6, Simple Variables](#) describes [LITERALS and DEFINEs](#) on page 5-1.

## Specifying fpoints

For the FIXED data type, you can specify an *fpoint*, an implied fixed-point setting specified as an integer in the range –19 through 19.

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
```

## Data Type Aliases

The compiler accepts the following aliases for the listed data types:

Data Type	Alias
INT	INT (16)
REAL	REAL (32)
FIXED	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 3-2](#).

**Table 3-2. 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
Quadrupleword	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.

## Address Modes

The data type of a variable determines byte or word addressing and indexing, as discussed elsewhere in this manual and in the *TAL Programmer's Guide*.

## Operations by Data Type

The data type of a variable determines the operations you can perform on the variable. [Table 3-3](#) lists the operations by data type.

**Table 3-3. Operations by Data Type** (page 1 of 2)

Operation	STRING	INT or Unsigned (1-16)	INT (32) or Unsigned (17-31)	FIXED	REAL or REAL (64)
Unsigned arithmetic	Yes	Yes	No	No	No
Signed arithmetic	Yes	Yes	Yes	Yes	Yes
Logical operations	Yes	Yes	No	No	No

**Table 3-3. Operations by Data Type** (page 2 of 2)

<b>Operation</b>	<b>STRING</b>	<b>INT or Unsigned (1-16)</b>	<b>INT (32) or Unsigned (17-31)</b>	<b>FIXED</b>	<b>REAL or REAL (64)</b>
Relational Operations	Yes	Yes	Yes	Yes	Yes
Bit shifts	Yes	Yes	Yes	No	No
Byte scans	Yes	Yes	Yes	Yes	Yes

[Section 4, Expressions](#), explains how the data type of an operand affects its behavior in expressions.

## Addresses as Data

You can store standard (16-bit) addresses in INT variables. Use only unsigned operations for standard addresses.

You can store extended (32-bit) addresses in INT(32) variables.

## Functions by Data Type

The data type of a variable determines the standard function you can use with the variable. [Table 3-4](#) lists the function categories by data type.

**Table 3-4. Standard Functions by Data Type**

<b>Category</b>	<b>STRING</b>	<b>INT or Unsigned (1-16)</b>	<b>INT (32) or Unsigned (17-31)</b>	<b>FIXED</b>	<b>REAL or REAL (64)</b>
Type Transfer	Yes	Yes	Yes	Yes	Yes
Character Test	Yes	No	No	No	No
Minimum or Maximum	Yes	Yes	Yes	Yes	Yes
Scaling	No	No	No	Yes	No
Variable	Yes	Yes	Yes	Yes	Yes
Address Conversion	Yes	Yes	Yes	Yes	Yes

For more information on the descriptions of each standard function, see [Section 14, Standard Functions](#).

## Address Types

TAL supports the following pTAL address types. For further details, see the *pTAL Reference Manual*.

pTAL supports 10 address types that control the addresses you store into pointers. pTAL uses address types to ensure that your program addresses the same relative data locations on a RISC processor as it does on a CISC processor. Address types are like data types except that:

- Address types are used primarily to describe the addresses that you assign to a pointer, not the data your program is processing.
- pTAL implicitly determines the address type of a pointer based on how you declare the pointer. You cannot explicitly declare a pointer's address type.
- Only operations that are meaningful for addresses are valid on address types.
- An address type identifies:
  - The location of the data to which the pointer points.
  - The addressing mode to use when accessing the data.

Address types are summarized in [Table 3-5](#). This table also identifies the target data that applies to each address when you run pTAL on a CISC processor.

---

**Table 3-5. Address Types**

Data Type	Type	Target Data on a CISC Processor
BADDR	Byte	8-bit bytes in the user data segment
WADDR	Word	16-bit words in the user data segment
CBADDR	Byte	8-bit bytes in the user code segment
CWADDR	Word	16-bit words in the user code segment
SGBADDR	Byte	8-bit bytes in system globals
SGWADDR	Word	16-bit words in system globals
SGXBADDR	Byte	8-bit bytes in system globals
SGXWADDR	Word	16-bit words in system globals
EXTADDR	Byte	Data in an extended segment
PROCADDR	N.A.	Index of a procedure in the PEP table

---

## Syntax for Constants

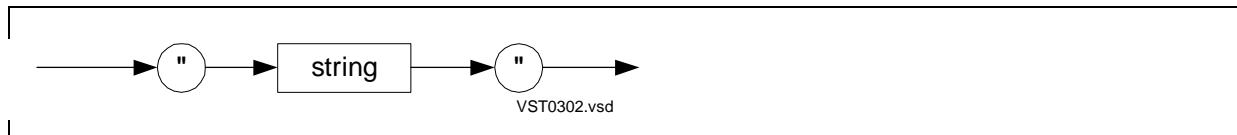
The remaining pages of this section describe the syntax definitions for specifying constants in your program. You can specify the following kinds of constants:

- Character string constants (all data types)
- STRING numeric constants

- INT numeric constants
- INT(32) numeric constants
- FIXED numeric constants
- REAL and REAL(64) numeric constants
- Constant lists

## Character String Constants

A character string constant consists of one or more ASCII characters stored in a contiguous group of bytes.



`string`

is a sequence of one or more ASCII characters enclosed in quotation mark delimiters. If a quotation mark is a character within the sequence of ASCII characters, use two quotation marks (in addition to the quotation mark delimiters). The compiler does not upshift lowercase characters.

## Character String Length

Each character in a character string requires one byte of contiguous storage. The maximum length of a character string you can specify differs for initializations and for assignments.

### Initializations

You can initialize simple variables or arrays of any data type with character strings.

When you initialize a simple variable, the character string can have the same number of bytes as the simple variable or fewer.

When you initialize an array, the character string can have up to 127 characters and must fit on one line. If a character string is too long for one line, use a constant list, described later in this section, and break the character string into smaller character strings.

### Assignments

You can assign character strings to STRING, INT, and INT(32) variables, but not to FIXED, REAL, or REAL(64) variables.

In assignment statements, a character string can contain at most four characters, depending on the data type of the variable.

## Example of Character String Constant

This example declares an INT variable and initializes it with a character string:

```
INT chars := "AB";
```

## String Numeric Constants

Representation: Unsigned 8-bit integer

Range: 0 through 255



**base**

is %, %B, or %H, which indicates a number base:

Octal	%
Binary	%B
Hexadecimal	%H

If you omit the base, the default base is decimal

**integer**

is one or more digits. The digits allowed are:

Decimal	0 through 9
Octal	0 through 7
Binary	0 or 1
Hexadecimal	0 through 9, A through F

## Example of STRING Numeric Constants

Here are examples of STRING numeric constants:

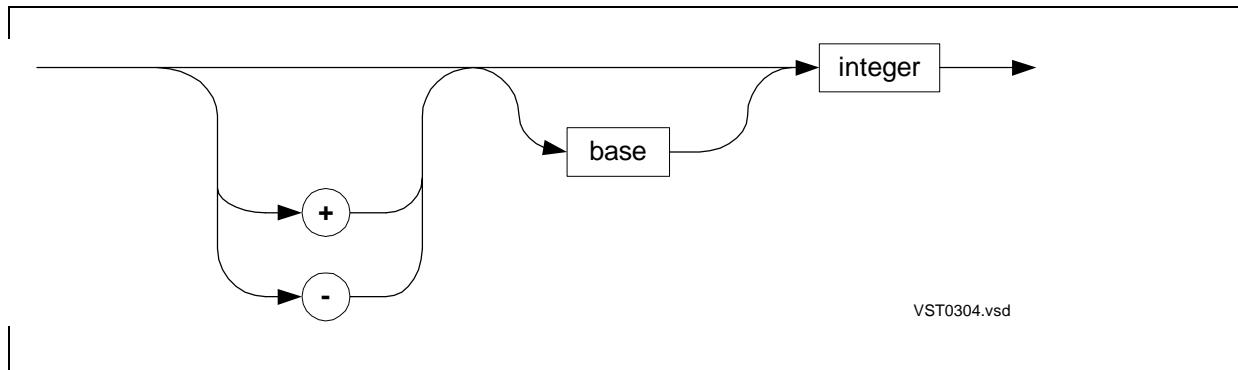
Decimal	255
Octal	%12
Binary	%B101
Hexadecimal	%h2A

# INT Numeric Constants

Representation: Signed or Unsigned 16-bit integer

Range (unsigned): 0 through 65,535

Range (signed): -32,768 through 32,767



base

is %, %B, or %H, which indicates a number base:

Octal	%
Binary	%B
Hexadecimal	%H

The default base is decimal. Unsigned integers greater than 32,767 must be in octal, binary, or hexadecimal base.

integer

is one or more digits. The digits allowed are:

Decimal	0 through 9
Octal	0 through 7
Binary	0 or 1
Hexadecimal	0 through 9, A through F

## Examples of INT Numeric Constants

Here are examples of INT numeric constants:

Decimal	3 -32045
Octal	%177 -%5
Binary	%B01010 %b1001111000010001
Hexadecimal	%H1A %h2f

## Storage Format

The system stores signed integers in two's complement notation. It obtains the negative of a number by inverting each bit position in the number, and then adding 1.

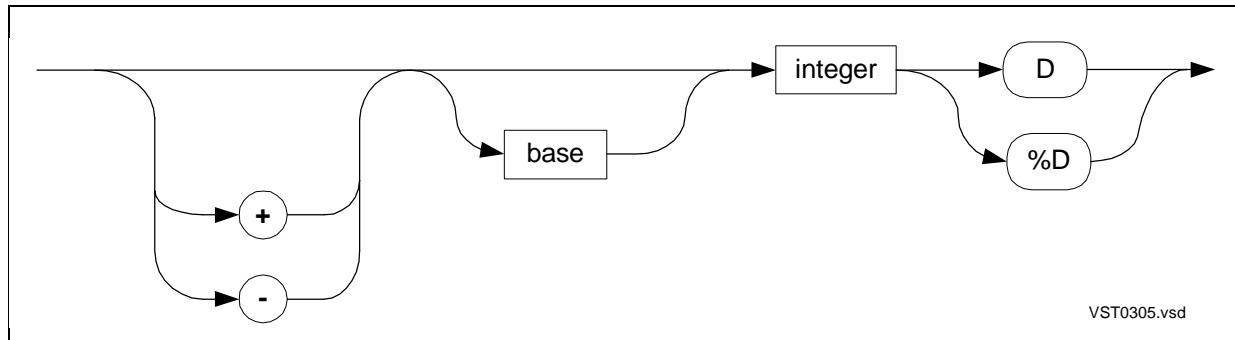
2 is stored as 0000000000000010

-2 is stored as 1111111111111110

## INT (32) Numeric Constants

Representation: Signed or unsigned 32-bit integer

Range: -2,147,483,648 through 2,147,483,647



base

is %, %B, or %H, which indicates a number base:

Octal	%
Binary	%B
Hexadecimal	%H

The default *base* is decimal

integer

is one or more digits. The digits allowed are:

Decimal	0 through 9
Octal	0 through 7
Binary	0 or 1
Hexadecimal	0 through 9, A through F

D and %D

are suffixes that specify INT(32) constants:

Decimal	D
Octal	D
Binary	D
Hexadecimal	%D

## Examples of INT (32) Numeric Constants

Here are examples of INT(32) numeric constants:

Decimal	0D
	+14769D
	-327895066d
Octal	%1707254361d
	-%24700000221D
Binary	%B000100101100010001010001001d
Hexadecimal	%h096228d%D
	-%H99FF29%D

For readability, always specify the % in the %D hexadecimal suffix. The following format, where a space replaces the % in the %D suffix, is allowed but not recommended:

-%H99FF29 D      !Using space instead of %  
                      ! is not recommended

## Storage Format

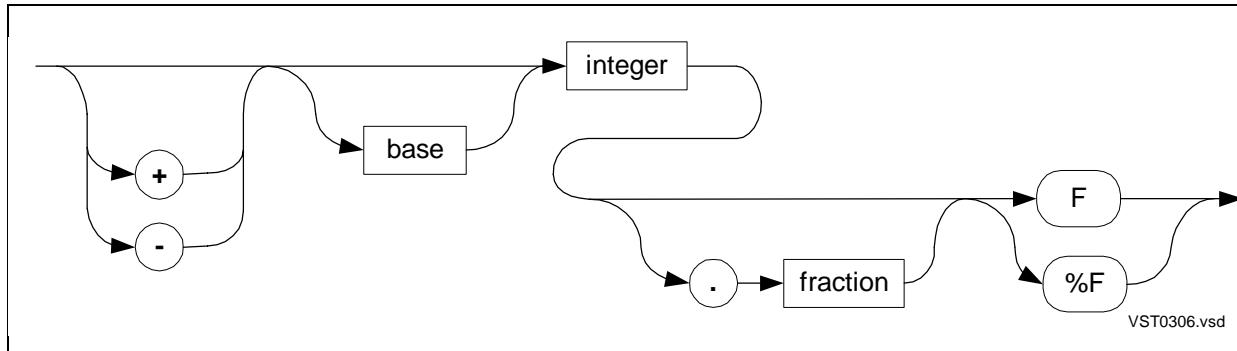
The system stores signed integers in two's complement notation.

# FIXED Numeric Constants

Representation: Signed 64-bit fixed-point number

Range: -9,223,372,036,854,775,808 through

+9,223,372,036,854,775,807



**base**

is %, %B, or %H, which indicates a number base:

Octal           %

Binary          %B

Hexadecimal   %H

The default base is decimal

**integer**

is one or more digits. The digits allowed are:

Decimal       0 through 9

Octal          0 through 7

Binary          0 or 1

Hexadecimal   0 through 9, A through F

**fraction**

is one or more decimal digits. fraction is legal only for decimal base.

**F** and **%F**

are suffixes that specify FIXED constants:

Decimal       F

Octal          F

Binary          F

Hexadecimal   %F

## Examples of FIXED Numeric Constants

Decimal	1200.09F 0.1234567F 239840984939873494F -10.09F
Octal	%765235512F
Binary	%B1010111010101101010110F
Hexadecimal	%H298756%F

## Storage Format

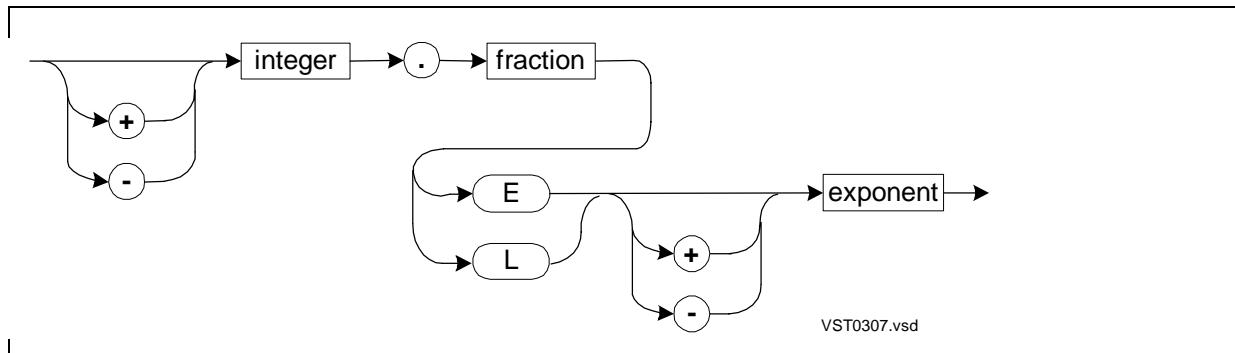
The system stores a FIXED number in binary notation. When the system stores a FIXED number, it scales the constant as dictated by the declaration or expression. Scaling means the system multiplies or divides the constant by powers of 10 to move the decimal.

For more information on scaling of FIXED values in expressions, see [Section 4, Expressions](#).

For more information on scaling of FIXED values in declarations, see [Section 6, Simple Variables](#).

## REAL and REAL (64) Numeric Constants

Representation	Signed 32-bit REAL or 64-bit REAL(64) floating-point number
Range	$\pm 8.6361685550944446 \times 10^{-78}$ through $\pm 1.15792089237316189 \times 10^{+77}$
Precision	REAL—to approximately 7 significant digits REAL(64)—to approximately 17 significant digits



`integer`

is one or more decimal digits that compose the integer part.

`fraction`

is one or more decimal digits that compose the fractional part.

`E` and `L`

are suffixes that specify floating-point constants:

`REAL`            `E`

`REAL(64)`      `L`

`exponent`

is one or two decimal digits that compose the exponential part.

## Examples of REAL and REAL (64) Numeric Constants

Here are the examples of REAL and REAL(64) numeric constants. The examples show the integer part, the fractional part, the E or L suffix, and the exponent part:

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

## Storage Format

The system stores the number in binary scientific notation in the form:

`X * 2 Y`

X is a value of at least 1 but less than 2. Because the integer part of X is always 1, only the fractional part of X is stored.

The exponent can be in the range –256 through 255 (%377). The system adds 256 (%400) to the exponent before storing it as Y. Thus, the value stored as Y is in the range 0 through 511 (%777), and the exponent is Y minus 256.

If the value of the number to be represented is zero, the sign is 0, the fraction is 0, and the exponent is 0.

The system stores the parts of a floating-point constant as follows:

Data Type	Sign Bit	Fraction	Exponent
REAL	<0>	<1:22>	<23:31>
REAL	<0>	<1:54>	<55:63>

## Examples of Storage Formats

- For the following REAL constant, the sign bit is 0, the fraction bits are 0, and the exponent bits contain %400 + 2, or %402:

4 = 1.0 \* 2 2 stored as %000000 %000402

- For the following REAL constant, the sign bit is 1, the fraction bits contain %.2 (decimal .25 is 2/8), and the exponent bits contain %400 + 3, or %403:

-10 = -(1.25 \* 2 3 ) stored as %120000 %000403

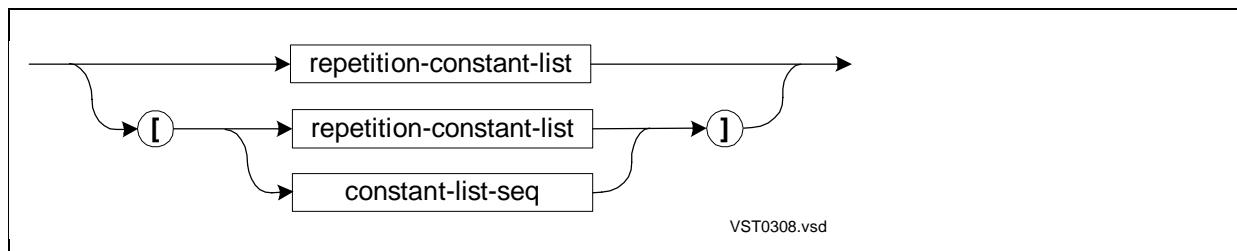
- For the following REAL(64) constant, the sign bit is 0, the fraction bits contain the octal representation of .33333..., and the exponent bits contain %400 – 2, or %376:

1/3 = .33333...\* 2 -2 stored as %025252 %125252 %125252 %125376

## Constant Lists

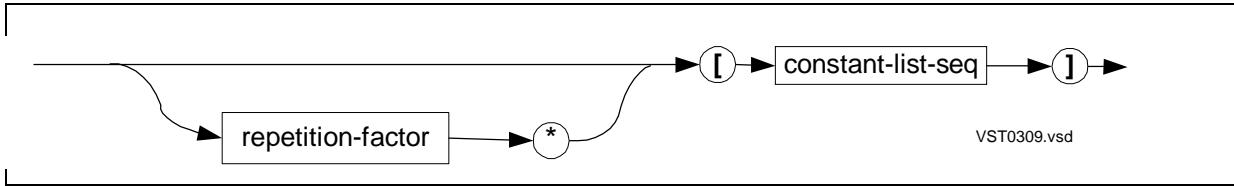
A constant list is a list of one or more constants. You can use constant lists in:

- Initializations of array declarations that are not contained in structures
- Group comparison expressions
- Move statements but not assignment statements



repetition-constant-list

has the form:

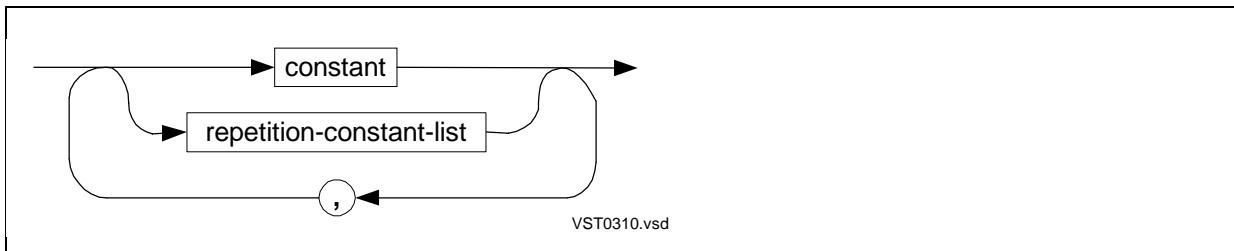


**repetition-factor**

is an INT constant that specifies the number of times constant-list-seq occurs.

**constant-list-seq**

is a list of one or more constants, each stored on an element boundary.



**constant**

is a character string, a number, or a LITERAL specified as a single operand. The range and syntax for specifying constants depends on the data type, as described for each data type on preceding pages.

## Examples of Constant Lists

1. The two examples in each of the following pairs are equivalent:

```
[ "A", "BCD" , "...", "Z" ]
[ "ABCD...Z" ]
10 * [0];
[0,0,0,0,0,0,0,0,0]
[3 * [2 * [1], 2 * [0]]]
[1,1,0,0,1,1,0,0,1,1,0,0]
10 * [ " " ]
[ " " ]
```

2. This example shows how you can break a constant string that is too long to fit on one line into smaller constant strings specified as a constant list. The system stores one character to a byte:

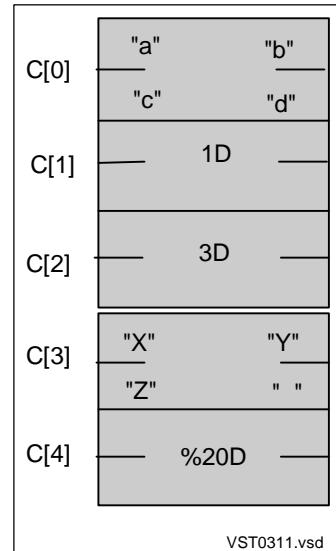
```
STRING a[0:99] := [ "These three constant strings will ",
                     "appear as if they were one constant",
                     "string continued on multiple lines." ];
```

3. This example initializes a STRING array with a repetition constant list:

```
STRING b[0:79] := 80 * [ " " ] ;
```

4. This example initializes an INT(32) array with a mixed constant list containing values of the same data type. The diagram shows how the compiler allocates storage for the variable and the constant list that initializes the variable:

```
INT(32) c[0:4] :=
[ "abcd", 1D, 3D, "XYZ", %20D ];
!Mixed constant list
```



# **4 Expressions**

This section describes the syntax for:

- Arithmetic and conditional expressions
- Special expressions (assignment, CASE, IF, group comparison)
- Bit operations (extraction and shift)

Section 5, “Using Expressions,” in the *TAL Programmer’s Guide* describes:

- Assigning conditional expressions
- Dereferencing simple variables (formerly called temporary pointers)

## **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

The compiler at times requires arithmetic or conditional expressions. Where indicated in this manual, 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

## Data Types of Expressions

The result of an expression can be 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. The system 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 to make them match. (For more information on Type transfer functions, see [Section 14, Standard Functions](#).)

# 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 the operators in the order of precedence. Within each level of precedence, the compiler evaluates the operators from left to right. [Table 4-1](#) shows the level of precedence for each operator, from highest (0) to lowest (9).

---

**Table 4-1. Precedence of Operators** (page 1 of 2)

Operator	Operation	Precedence
[ n ]	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

---

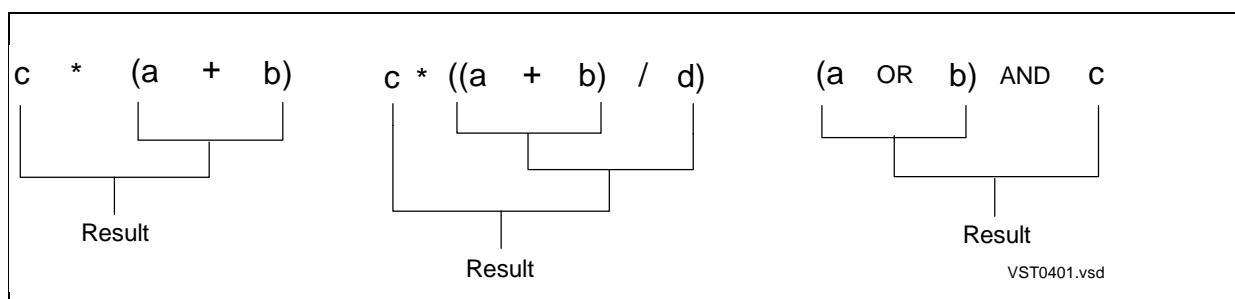
**Table 4-1. Precedence of Operators** (page 2 of 2)

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

\* Described in [Section 12, Statements](#).

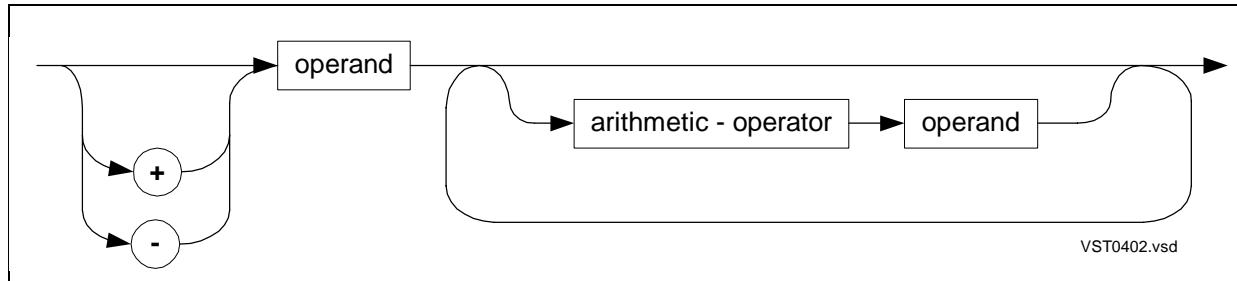
\* Described in the *TAL Programmer's Guide*.

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:



# Arithmetic Expressions

An arithmetic expression is a sequence of operands and arithmetic operators that computes a single numeric value of a specific data type.



+ and -

are unary plus and minus operators applied to the leftmost operand of the expression. If you do not use the unary plus or unary minus operator, the default is unary plus.

operand

is a value in an arithmetic expression. Each operand consists of one or more of the following syntactic elements. Each syntactic element represents a single value:

## Variable

## Constant

LITERAI

## Function invocation

(expression)

## Code space item

### Summary of the following

Is one of the following operators:

Signed at [redacted] on [redacted].

VOLUME 1, NUMBER 1

unsigned arithmetic operator.

Logical operator:

LUR, LAND, LUR

## Examples of Arithmetic Expressions

Following are examples of arithmetic expressions:

var1	! operand
-var1	! - operand
+var1 * 2	! + operand arithmetic-operator operand
var1 / var2	! operand arithmetic-operator operand
var1 * (-var2)	! operand arithmetic-operator operand
2 * 3 + var / 2	
2 * var * 4	

## Operands in Arithmetic Expressions

An operand consists of one or more elements that evaluate to a single value. [Table 4-2](#) describes the operands that can make up an arithmetic expression.

---

**Table 4-2. 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	The invocation of a procedure that returns a value	\$LEN (x)
(expression)	Any expression, enclosed in parentheses	(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 4-3](#) on page 4-7

**Table 4-3. 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](#) on page 4-2.

[Table 4-4](#) 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 4-4. Signed Arithmetic Operand and Result Types**

Operand Type	Operand Type	Result Type	Example
STRING	STRING	INT	byte1+byte2
INT	INT	INT	word1-word 2
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
```

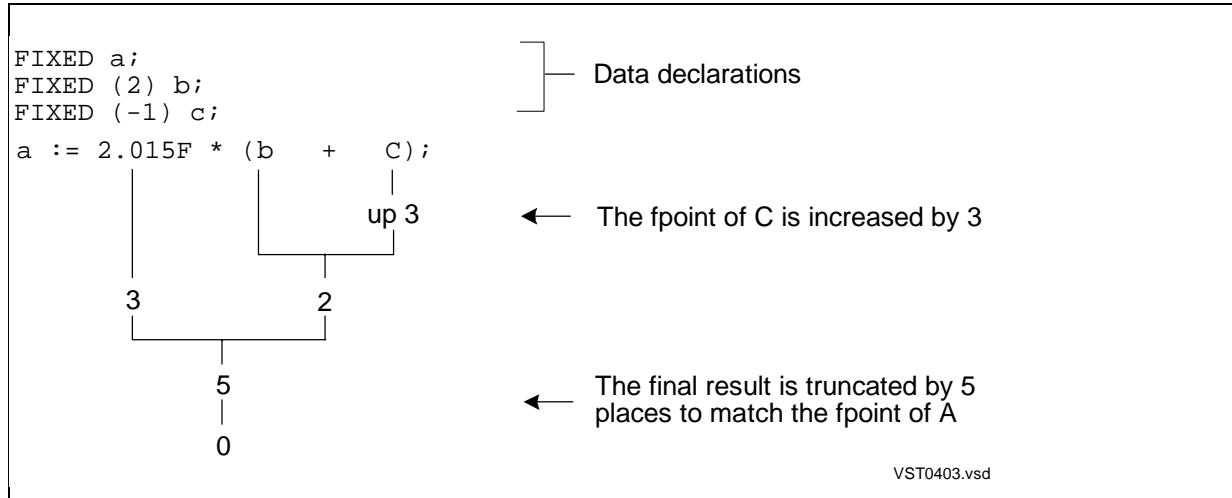
When FIXED operands in an arithmetic expression have different *fpoints*, the system makes adjustments depending on the operator.

- 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*:



## Effect on Hardware Indicators

Signed arithmetic operators affect the hardware indicators as described in [Testing Hardware Indicators](#) on page 4-16.

## Unsigned Arithmetic Operators

Typically, you 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 4-5](#) summarizes unsigned arithmetic operators and the operand types on which they can operate.

**Table 4-5. Unsigned Arithmetic Operators** (page 1 of 2)

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

**Table 4-5. Unsigned Arithmetic Operators** (page 2 of 2)

Operator	Operation	Operand Type	Example
' / '	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 4-6](#) 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 4-6. 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)	wrd1 '*' wrd2
	STRING	INT	INT (32)	byte1 '*' wrd1
	INT	UNSIGNED (1–16)	INT (32)	wrd1 '*' uns9
	STRING	UNSIGNED (1–16)	INT (32)	uns1 '*' uns7
	UNSIGNED (1–16)	UNSIGNED(1–16)	INT (32)	uns1 '*' uns7
' / '	UNSIGNED (17–31)	STRING, INT, or or INT(32) dividend	INT (32)	dbwd '\' word1
' \ '		UNSIGNED(1–16) divisor	INT	

## Effect on Hardware Indicators

Unsigned add and subtract operators affect the carry and condition code indicators as described in [Testing Hardware Indicators](#) on page 4-16.

## 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 4-7](#) gives information about these operators.

**Table 4-7. Logical Operators and Result Yielded**

		Operand		Bit Operations	Example
Operator	Operation	Type			
LOR	Bitwise Logical OR	STRING, INT, or UNSIGNED(1–16)	1 LOR 1 = 1 1 LOR 0 = 1 0 LOR 0 = 0	1 LOR 1 = 1 1 LOR 0 = 1 0 LOR 0 = 0	10 LOR 12 = 14 10 1 0 1 0 <u>12</u> 1 1 0 0 14 1 1 1 0
	Bitwise Logical AND	STRING, INT, or UNSIGNED(1–16)	1 LAND 1 = 1 1 LAND 0 = 0 0 LAND 0 = 0	1 LAND 1 = 1 1 LAND 0 = 0 0 LAND 0 = 0	10 LAND 12 = 8 10 1 0 1 0 <u>12</u> 1 1 0 0 8 1 0 0 0
	Bitwise Exclusive OR	STRING, INT, or UNSIGNED(1–16)	1 XOR 1 = 0 1 XOR 0 = 1 0 XOR 0 = 0	1 XOR 1 = 0 1 XOR 0 = 1 0 XOR 0 = 0	10 XOR 12 = 6 10 1 0 1 0 <u>12</u> 1 1 0 0 6 0 1 1 0

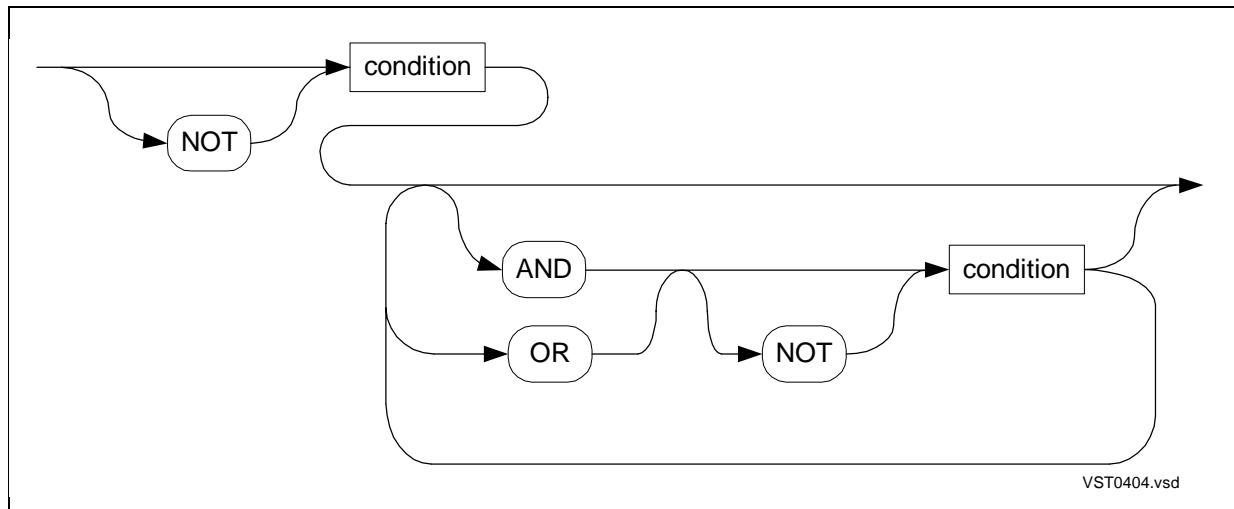
The Bit Operations column in the table shows the bit-by-bit operations that occur on 16-bit values. Each 1-bit operand pair results in a 1-bit result. The bit operands are commutative.

## Effect of Hardware Indicators

Logical operators set the condition code indicator as described in [Testing Hardware Indicators](#) on page 4-16. 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.



## condition

is an operand in a conditional expression. condition can consist of one or more of the following syntactic elements:

Relational expression

Group comparison expression

(conditional expression)

Arithmetic expression

Relational operator

## AND

is a Boolean operator that produces a true state if both adjacent conditions are true.

## OR

is a Boolean operator that produces a true state if either adjacent condition is true.

## NOT

is a Boolean operator that tests the condition for a false state.

## Examples of Conditional Expressions

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 4-8](#).

**Table 4-8. 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 4-9](#) describes the Boolean operators, the operand types you can use with them, and the results that such operators yield.

---

**Table 4-9. 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

---

## Evaluations 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  $\neq$  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](#) on page 4-16.

## Relational Operators

Relational operators are signed or unsigned.

### Signed Relational Operators

Signed relational operators perform signed comparison of two operands and return a true or false state. [Table 4-10](#) describes signed relational operators, operand data types, and the results yielded by such operators.

**Table 4-10. Signed Relational Operators and Result Yielded**

Operator	Meaning	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](#) on page 4-2.

## Unsigned Relational Operators

Unsigned relational operators perform unsigned comparison of two operands and return a true or false state. [Table 4-11](#) describes unsigned relational operators, operand data types, and the results yielded by such operators.

**Table 4-11. Unsigned Relational Operators and Result Yielded**

Operator	Meaning	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

Signed and unsigned operators set the condition code indicator as described in [Testing Hardware Indicators](#).

# 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 indicator. 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;                       ! Test state of carry bit from +
IF $CARRY THEN . . . ;

```

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

<b>Operation</b>	<b>Overflow Indicator</b>
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:

<b>Desired Effect</b>	<b>Method</b>
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 more information on turning off overflow trapping and using \$OVERFLOW, see the description of the \$OVERFLOW function in [Section 14, Standard Functions](#).

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

## Special Expressions

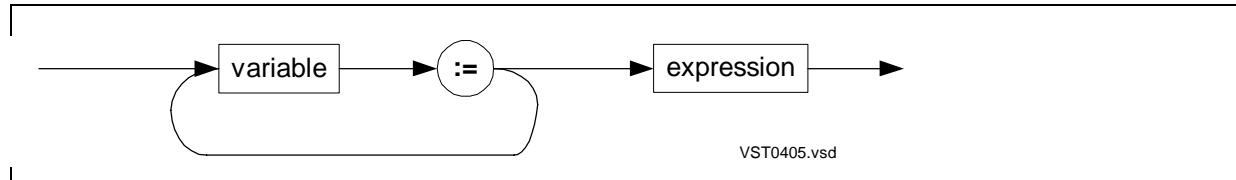
Special expressions allow you to perform specialized arithmetic or conditional operations. [Table 4-12](#) on page 4-19 summarizes these special expressions.

**Table 4-12. Special Expressions**

Expression Form	Kind of Expression	Description	Page
Assignment	Arithmetic	Assigns the value of an expression to a variable	<a href="#">4-19</a>
CASE	Arithmetic	Selects one of several expressions	<a href="#">4-20</a>
IF	Arithmetic	Conditionally selects one of two expressions	<a href="#">4-21</a>
Group comparison	Conditional	Does unsigned comparison of two sets of data	<a href="#">4-23</a>

## Assignment Expression

The assignment expression assigns the value of an expression to a variable.



**variable**

is the identifier of a variable in which to store the result of expression. (variable can have an optional bit-deposit field.)

**expression**

is an expression having the same data type as variable. The result of *expression* becomes the result of the assignment *expression*. Expression is either:

- An arithmetic expression
- A conditional expression (excluding a relational operator with no operands), the result of which has data type INT

## Examples of Assignment Expressions

1. This example decrements A. As long as A– 1 is not 0, the condition is true and the THEN clause is executed:

```
IF (a := a - 1) THEN ... ;
```

2. This example shows the assignment form used as an index. It decrements A and accesses the next array element:

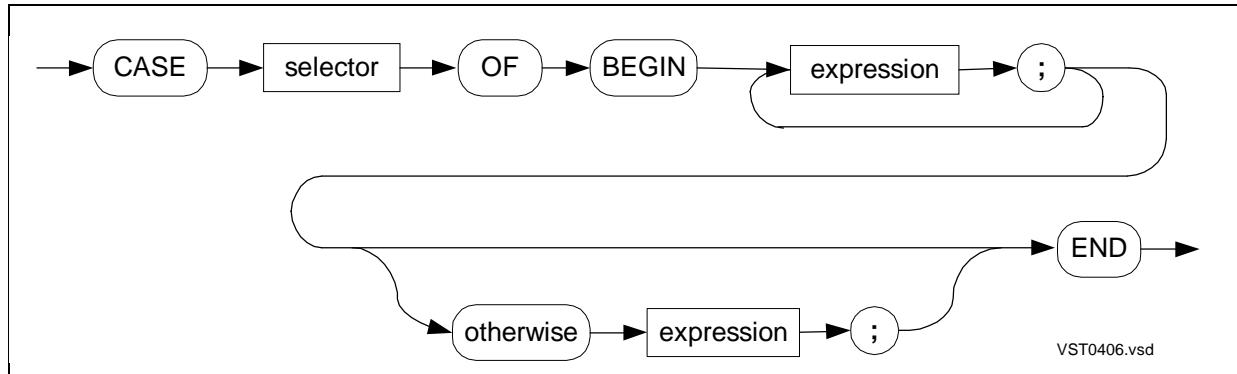
```
IF array[a := a - 1] <> 0 THEN ... ;
```

3. This example mixes the assignment form with a relational form. It assigns the value of B to A, then checks for equality with 0:

```
IF (a := b) = 0 THEN ... ;
```

## CASE Expression

The CASE expression selects one of several expressions.



**selector**

is an INT arithmetic expression that selects the expression to evaluate.

**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

**OTHERWISE expression**

specifies the expression to evaluate if selector does not select an expression in the BEGIN clause. If you omit the OTHERWISE clause and an out-of-range case occurs, results are unpredictable.

## Usage Considerations

All *expressions* in the CASE expression must have the same data type.

The compiler numbers each *expression* in the BEGIN clause consecutively, starting with 0. If the *selector* matches the compiler-assigned number of an *expression*, that *expression* is evaluated and becomes the result of the CASE expression. If the *selector* does not match a compiler-assigned number, the OTHERWISE expression is evaluated.

You can nest CASE expressions. CASE expressions resemble unlabeled CASE statements except that CASE expressions select *expressions* rather than *statements*.

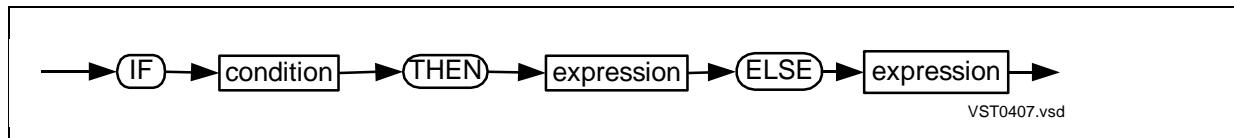
## Example of CASE Expression

This example selects an expression based on the value of A and assigns it to X:

```
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.
```

## IF Expression

The IF expression conditionally selects one of two expressions, usually for assignment to a variable.



condition

is 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

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

## Usage Considerations

If the *condition* is true, the result of the THEN expression becomes 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.

You can nest IF expressions within an IF expression or within other expressions. The IF expression resembles the IF statement except that the IF expression:

- Requires the ELSE clause
- Contains expressions, not statements

## Examples of IF Expressions

1. This example assigns an arithmetic expression to VAR based on the condition LENGTH > 0:

```
var := IF length > 0 THEN 10 ELSE 20;
```

2. This example nests an IF expression (in parentheses) within another expression:

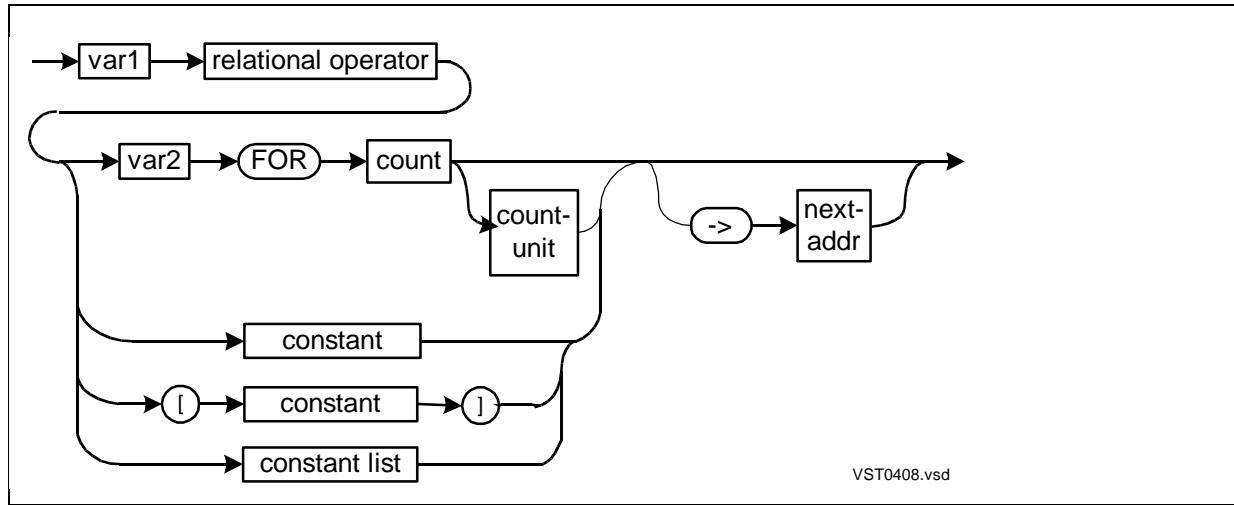
```
var * index +  
(IF index > limit THEN var * 2 ELSE var * 3)
```

3. This example nests an IF expression within another IF expression:

```
var := IF length < 0 THEN -1  
ELSE IF length = 0 THEN 0  
ELSE 1;
```

## Group Comparison Expression

The group comparison expression compares a variable with a variable or constant.



**var1**

is the identifier of a variable, with or without an index, that you want to compare to *var2*, *constant*, or *constant-list*. *var1* can be a simple variable, array, simple pointer, structure, structure data item, or structure pointer, but not a read-only array.

**relational-operator**

is one of the following operators:

Signed relational operator: <, =, >, <=, >=, <>

Unsigned relational operator: '<', '=' , '>', '<=' , '>=' , '<>'

All comparisons are unsigned whether you use a signed or unsigned operator.

**var2**

is the identifier of a variable, with or without an index, to which *var1* is compared. *var2* can be a simple variable, array, read-only array, simple pointer, structure, structure data item, or structure pointer.

count

is a positive INT arithmetic expression that defines the number of units in *var2* to compare. When *count-unit* is not present, the units compared are:

<b>var2</b>	<b>Data Type</b>	<b>Units Compared</b>
Simple variable, array, simple pointer (including those declared in structures)	STRING	Bytes
	INT	Words
	INT(32) or REAL	Doublewords
	FIXED or REAL(64)	Quadruplewords
Structure	Not applicable	Words
Substructure	Not applicable	Bytes
Structure pointer	STRING* INT*	Bytes Words

count-unit

is BYTES, WORDS, or ELEMENTS. count-unit changes the meaning of *count* to the following:

- BYTES      Compares count bytes. If var1 and var2 both have word addresses, BYTES generates a word comparison for  $(\text{count} + 1) / 2$  words
- WORDS      Compares count words
- ELEMENTS    Compares count elements. The elements compared depend on the nature of var2 and its data type as follows:

<b>var2</b>	<b>Data Type</b>	<b>Units Compared</b>
Simple variable, array, simple pointer (including those declared in structures)	STRING	Bytes
	INT	Words
	INT (32) or REAL	Doublewords
	FIXED or REAL (64)	Quadruplewords
Structure	Not applicable	Structure occurrences
Substructure	Not applicable	Substructure occurrences
Structure pointer	STRING* INT*	Structure occurrences Structure occurrences

\* For structure pointers, STRING and INT have meaning only in group comparison expressions and move statements.

If *count-unit* is not BYTES, WORDS, or ELEMENTS, the compiler issues an error. If you specify BYTES, WORDS, or ELEMENTS, the term cannot also appear as a DEFINE or LITERAL identifier in the global declarations or in any procedure or subprocedure in which the group comparison expression appears.

constant

is a number, a character string, or a LITERAL to which *var1* is compared.

If you enclose *constant* in brackets ([ ]) and if the destination has a byte address or is a STRING structure pointer, the system compares a single byte regardless of the size of *constant*. If you do not enclose *constant* in brackets or if the destination has a word address or is an INT structure pointer, the system compares a word, doubleword, or quadrupleword as appropriate for the size of *constant*.

`constant-list`

is a list of one or more constants, which are concatenated and compared to *var1*. Specify *constant-list* in the form shown in [Section 3, Data Representation](#).

`next-addr`

is a variable to contain the address of the first byte or word in *var1* that does not match the corresponding byte or word in *var2*. The compiler returns a 16-bit or 32-bit address as described in [Usage Considerations](#).

## Usage Considerations

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 <i>var1</i> '<' <i>var2</i>
=	CCE if <i>var1</i> = <i>var2</i>
>	CCG if <i>var1</i> '>' <i>var2</i>

The compiler does a standard comparison and returns a 16-bit *next-addr* 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-addr* 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, *next-addr* might point into the middle of an element, rather than at the beginning of the element.

## Examples of Group Comparison Expressions

1. This 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 d_array[0:9];
INT s_array[0:9];
!Code to assign values to arrays
IF d_array = s_array FOR 10 ELEMENTS -> @pointer THEN
    BEGIN
        !They matched
        !Do something
    END
ELSE
    IF < THEN ... ;           !POINTER points to element of
    !Do something else       ! D_ARRAY that is less than the
                            ! corresponding element of
                            ! S_ARRAY

```

2. When you compare array elements (as in the preceding example), the ELEMENTS keyword is optional but provides clearer source code.

3. To 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 here to assign values to structures
IF struct_one = struct_two FOR 10 ELEMENTS THEN ... ;

```

4. This example contrasts a comparison to a bracketed (single-byte) constant with a comparison to an unbracketed (element) constant:

```

STRING var[ 0:1 ];
!Lots of code
IF var = [ 0 ] THEN ... ; !Compare VAR[ 0 ] to one byte
IF var = 0 THEN ... ;     !Compare VAR[ 0:1 ] to two bytes

```

## Bit Operations

You can access individual bits or groups of bits in a STRING or INT variable. [Table 4-13](#) lists bit operations.

---

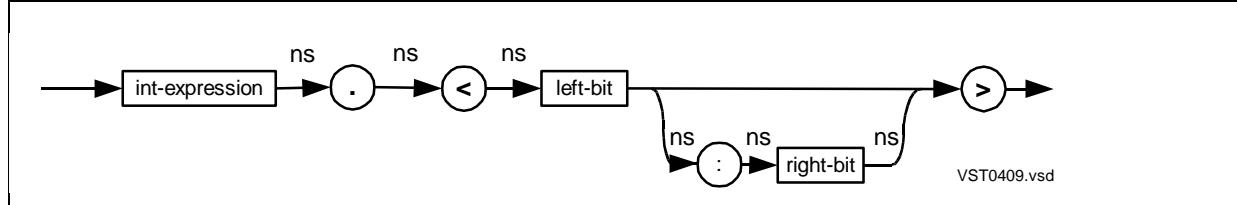
**Table 4-13. Bit - Operations**

Bit Operation	Description
Bit extraction	Access a bit-extraction field in an INT expression without altering the expression
Bit shift	Shift a bit-shift field in an INT or INT(32) expression to the left or to the right by a specified number of bits
Bit deposit	Assign a bit value to a bit-deposit field in a variable (For more information, see <a href="#">Section 12, Statements</a> )

---

# Bit Extractions

A bit extraction lets you access a bit field in an INT expression without altering the expression.



**int-expression**

is an INT expression (which can include STRING, INT, or UNSIGNED(1–16) values).

**left-bit**

is an INT constant in the range 0 through 15 that specifies the bit number of either:

- The leftmost bit of the bit-extraction field
- The only bit (if *right-bit* is the same value as *left-bit* or is omitted)

If *int-expression* is a STRING value, *left-bit* must be in the range 8 through 15. (In a string value, bit <8> is the leftmost bit and bit <15> is the rightmost bit.)

**right-bit**

is an INT constant that specifies the rightmost bit of the bit field. If *int-expression* is a STRING value, *right-bit* must be in the range 8 through 15. *right-bit* must be equal to or greater than *left-bit*. To access a single bit, omit *right-bit* or specify the same value as *left-bit*.

## Usage Considerations

Specify the bit-extraction format with no intervening spaces, as in:

myvar.<0:5>

## Examples of Bit Extractions

1. This assignment accesses bits in an array element:

```

STRING right_byte;
INT array[0:7];
right_byte := array[5].<8:15>;
  
```

2. This example assigns bits <4:7> of the sum of two numbers to RESULT. The parentheses cause the numbers to be added before the bits are extracted:

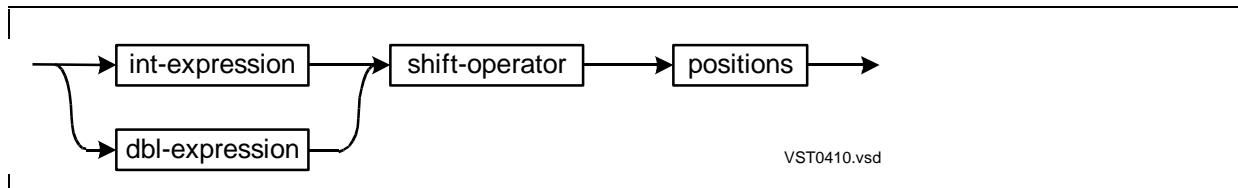
```
INT result;
INT num1 := 51;
INT num2 := 28;
result := (num1 + num2).<4:7>;
```

3. This conditional expression checks bit <15> for a nonzero value:

```
STRING var;
IF var.<15> THEN ... ;
```

## Bit Shifts

A bit shift operation shifts a bit field a specified number of positions to the left or to the right within a variable without altering the variable.



**int-expression**

is an INT arithmetic expression. int-expression can contain STRING, INT, or UNSIGNED(1–16) operands. The bit shift occurs within a word.

**dbl-expression**

is an INT(32) arithmetic expression. dbl-expression can contain INT(32) or UNSIGNED(17–31) operands. The bit shift occurs within a doubleword.

**shift-operator**

is one of the operators ('<<', '>>', <<, >>) described in [Table 4-14](#).

**positions**

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

## Usage Considerations

[Table 4-14](#) lists the bit-shift operators you can specify.

**Table 4-14. 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 ('<<'').

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.

## Examples of Bit Shifts

1. This 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
```

2. This 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
```

3. This 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
```

4. This 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
```

5. These examples show multiplication and division by powers of two:

```
a := b << 1;                      !Multiply by 2
a := b << 2;                      !Multiply by 4
a := b >> 3;                      !Divide by 8
a := b >> 4;                      !Divide by 16
```

6. This unsigned bit shift converts the word address of an INT array to a byte address, which allows byte access to the INT array:

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

7. This example shifts the right-byte value into the left byte of the same word and sets the right byte to a zero:

```
INT b;                            !INT variable
b := b '<<' 8;                  !Shift right-byte value into
                                  ! left byte
```



# **5 LITERALS and DEFINES**

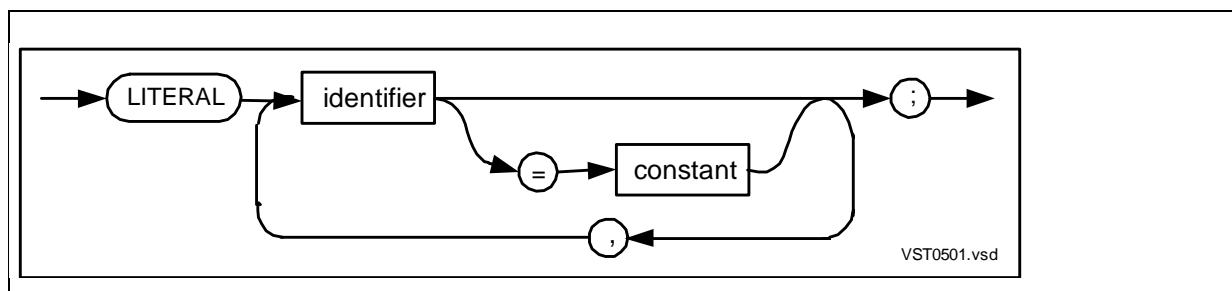
This section describes LITERAL and DEFINE declarations. A LITERAL declaration associates identifiers with constant values.

A DEFINE declaration associates identifiers (and parameters if any) with text. (DEFINE declarations differ from TACL DEFINE commands, which are described in Appendix E, “File Names and TACL Commands,” in the *TAL Programmer’s Guide*.)

You can declare LITERALS and DEFINES once in a program, and then refer to them by identifier many times throughout the program. They allow you to make significant changes in the source code efficiently. You only need to change the declaration, not every reference to it in the program.

## **LITERAL Declaration**

A LITERAL declaration specifies one or more identifiers and associates each with a constant. Each identifier in a LITERAL declaration is known as a LITERAL.



identifier

is the identifier of a LITERAL.

constant

is either:

- An INT, INT(32), FIXED, REAL, or REAL(64) numeric constant expression that evaluates to any value except the address of a global variable. (All global variables are relocatable during binding.)
- A character string of one to four characters.

If you omit any constants, the compiler supplies the omitted numeric constants. The compiler uses unsigned arithmetic to compute the constants it supplies:

- 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.

## Usage Considerations

The compiler allocates no storage for LITERAL constants. It substitutes the constant at each occurrence of the identifier.

You access a LITERAL constant by using its identifier in declarations and statements.

LITERAL identifiers make the source code more readable. For example, identifiers such as BUFFER\_LENGTH and TABLE\_SIZE are more meaningful than their respective constant values of 80 and 128.

## Examples of LITERAL Declarations

1. This example specifies a constant for each identifier:

```
LITERAL true          =      -1,
      false         =      0,
      buffer_length =     80,
      table_size    =     128,
      table_base   = %1000,
      entry_size   =      4,
      timeout      = %100000D,
      CR           = %15,
      LF           = %12;
```

2. This example specifies no numeric constants; the compiler supplies all the constants:

```
LITERAL a,           -- The compiler assigns 0
      b,           -- The compiler assigns 1
      c;           -- The compiler assigns 2
```

3. This example specifies two of eight numeric constants; the compiler supplies the remaining constants:

```
LITERAL d, -- The compiler assigns 0
e, -- The compiler assigns 1
f, -- The compiler assigns 2
g = 0,
h, -- The compiler assigns 1
i = 17,
j, -- The compiler assigns 18
k; -- The compiler assigns 19
```

4. This example uses a LITERAL identifier in an array declaration:

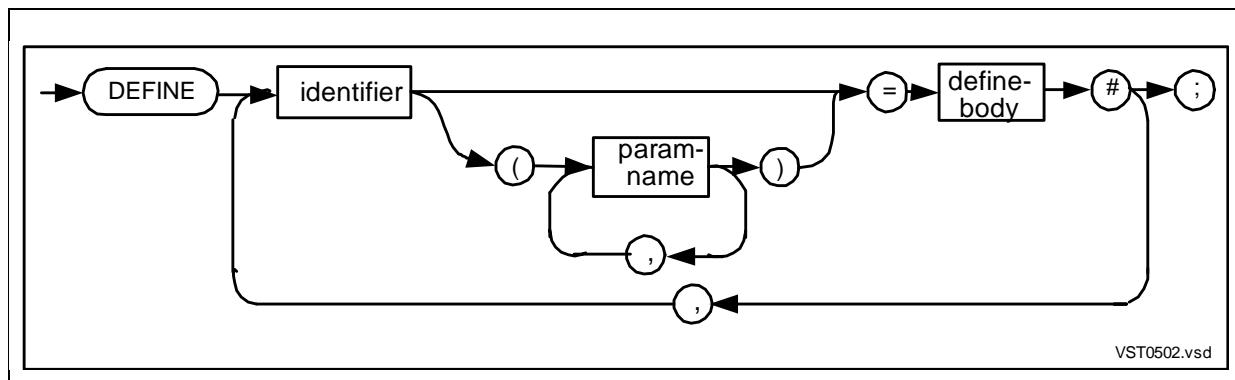
```
LITERAL length = 50; !Length of array
INT buffer[0:length - 1]; !Array declaration
```

5. This example uses LITERAL identifiers in subsequent LITERAL declarations:

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

## DEFINE Declaration

A DEFINE declaration associates an identifier (and parameters if any) with text.



**identifier**

is the identifier of the DEFINE.

**param-name**

is the identifier of a formal parameter. You can specify up to 31 formal parameters. An actual parameter can be up to 500 bytes long.

**define-body**

is all characters between the = and # delimiters. *define-body* can span multiple source lines. Enclose character strings in quotation marks (""). To use # in as part of the *define-body* rather than as a delimiter, enclose the # in quotation marks or embed the # in a character string.

## Usage Considerations

DEFINE declarations have the following characteristics:

- If a DEFINE and a formal parameter have the same identifier, the formal parameter has priority during expansion.
- A DEFINE must not reference itself.
- A DEFINE declaration must not appear within a DEFINE body; that is, do not nest a DEFINE within a DEFINE.
- A DEFINE cannot replace a keyword with another term; for example, a DEFINE cannot replace BEGIN with START.
- To ensure proper grouping and evaluation of expressions in the DEFINE body, use parentheses around each DEFINE parameter used in an expression.
- Within the DEFINE body, place any compound statements within a BEGIN-END construct.
- Directives appearing within a DEFINE body are evaluated immediately; they are not part of the DEFINE itself.
- If the CODE (DECS) and CODE (RP) statements are equivalent to the DECS and RP directives but do not execute until a reference to the DEFINE identifier occurs. Statements are part of the DEFINE itself.

- If expanded DEFINEs must produce correct TAL constructs. To list the expanded DEFINEs in the compiler listing, specify the DEFEXPAND directive before the DEFINE declarations. (For more information, see [Compiler Action](#) on page 5-7.)

**Note.** if you use a DEFINE to name and access a structure item, the DEFINE identifier must be unique among the identifiers of all structure items in the compilation unit. Conversely, if you use the DEFINE identifier to access a structure item and some other structure item has the same identifier as the DEFINE, you access the other structure item and the compiler issues a warning. To access structure bit fields smaller than a byte, use UNSIGNED declarations instead of DEFINEs.

---

## Examples of DEFINE Declarations

1. This example uses parentheses to direct the DEFINE body evaluation:

```
DEFINE value = ( (45 + 22) * 8 / 2 ) #;
```

2. This example provides incrementing and decrementing utilities:

```
DEFINE increment (x) = x := x + 1 #;
DEFINE decrement (y) = y := y - 1 #;
```

3. This example loads numbers into particular bit positions. To ensure proper evaluation, parentheses enclose each parameter used in an expression:

```
DEFINE word_val (a, b) = ((a) '<<' 12) LOR (b) #;
```

4. This example shows a CODE (DECS) statement, which is equivalent to the DECS directive. This example is not portable to future software platforms:

```
DEFINE call_it (x, y) =
BEGIN
  STACK x;
  STACK y;
  CODE (PUSH %711);
  CODE (DPCL);
  CODE (DECS 2);      !Equivalent to DECS directive
END #;

call_it (a, b);      !Expands to: STACK a;
                      ! STACK b;
                      ! CODE (PUSH %711);
                      ! CODE (DPCL);
                      ! ?DECS 2
```

5. In this example, DEFINE MYNAME accesses the structure item named in the DEFINE body. However, the compiler issues a warning because 2 is assigned to MYSTRUCT.YRNAME, not to MYSTRUCT.ITEM2:

```

PROC myproc MAIN;
BEGIN
  DEFINE myname    = item1#,
        yrname     = item2#;
  STRUCT mystruct;
  BEGIN
    INT item1;
    INT item2;
    INT yrname;
  END;                                !Structure item has same
                                         ! identifier as a DEFINE

  mystruct.myname := 1;                  !Okay, 1 is assigned to
                                         ! MYSTRUCT.ITEM1

  mystruct.yrname := 2;                  !Compiler issues warning;
                                         !2 is assigned to
                                         !MYSTRUCT.YRNAME, not to
                                         !MYSTRUCT.ITEM2
  !More code
END;

```

## Invoking DEFINES

You invoke a DEFINE by using its identifier in a statement. The invocation can span multiple lines.

If you invoke a DEFINE within an expression, make sure the expression evaluates as you intend. For instance, if you want the DEFINE body to be evaluated before it becomes part of the expression, enclose the DEFINE body in parentheses.

The following example contrasts expansion of parenthesized and nonparenthesized DEFINE bodies after the identifiers are used in assignment statements:

```

DEFINE expr = (5 + 2) #;
j := expr * 4;                      !Expands to: (5 + 2) * 4;
                                         ! assigns 28 to J

DEFINE expr = 5 + 2 #;
j := expr * 4;                      !Expands to: 5 + 2 * 4;
                                         ! assigns 13 to J

```

DEFINE identifiers are not invoked when specified:

- Within a comment
- Within a character string constant
- On the left side of a declaration

For example, the following declaration can invoke a DEFINE named Y but not a DEFINE named X:

```
INT x := y;
```

## Compiler Action

The compiler allocates no storage for DEFINEs. When the compiler encounters a statement that uses a DEFINE identifier, the compiler expands the DEFINE as follows:

- It replaces the DEFINE identifier with the DEFINE body, replaces formal parameters with actual parameters, compiles the DEFINE, and emits any machine instructions needed.
- It expands quoted character strings intact.
- It expands actual parameters after instantiation. Depending on the order of evaluation, the expansion can change the lexical scope of a DEFINE declaration.

If the DEFEXPAND directive is in effect, the compiler lists each expanded DEFINE in the compiler listing following the DEFINE identifier. The expanded listing includes:

- The DEFINE body, excluding comments
- The lexical level of the DEFINE, starting at 1
- Parameters to the DEFINE

Parameters are listed as  $\$n$  (C-series system) or  $\#n$  (D-series system), where  $n$  is the sequence number of the parameter, starting at 1.

## Passing Actual Parameters

If the DEFINE declaration has formal parameters, you supply the actual parameters when you use the DEFINE identifier in a statement.

The number of actual parameters can be less than the number of formal parameters. If actual parameters are missing, the corresponding formal parameters expand to empty strings. For each missing actual parameter, you can use a placeholder comma. For example:

```
INT PROC d (a, b, c) EXTENSIBLE; EXTERNAL;
DEFINE something (a, b, c) = d (a, b, c) #;
nothing := something ( , , c); !Placeholder commas
```

If a DEFINE has formal parameters and you pass no actual parameters to the DEFINE, you must specify an empty actual parameter list. You can include commas between the list delimiters, but need not. For example:

```
DEFINE something (a, b, c) = anything and everything #;
nothing := something ( ); !Empty parameter list
```

If the number of actual parameters exceeds the number of formal parameters, the compiler issues an error. For example:

```
DEFINE something (a, b, c) = anything and everything #;
nothing := something (a, b, c, d); !Too many parameters
```

You can pass a DEFINE that has no formal parameters as an actual parameter. For example:

```
defmacro (DEFINE x = y + y #); !Invocation
```

If an actual parameter in a DEFINE invocation requires commas, enclose each comma in apostrophes (''). An example is an actual parameter that is a parameter list:

```
DEFINE varproc (proc1, param) = CALL proc1 (param) #;
varproc (myproc, i ', ' j ', ' k); !Expands to:
                                         ! CALL MYPROC (I, J, K);"
```

An actual parameter in a DEFINE invocation can include parentheses. For example:

```
DEFINE varproc (proc1, param) = CALL proc1 (param) #;
varproc (myproc, (i + j) * k); !Expands to:
                                         ! CALL MYPROC ((I+J)*K);
```

## Examples of Passing DEFINE Parameters

Here are more examples of passing actual parameters.

1. This example shows a DEFINE declaration that has one formal parameter and an assignment statement that uses the DEFINE identifier, passing a parameter of 3:

```
DEFINE cube (x) = ( x * x * x ) #;
INT result;

result := cube (3) '>>' 1;
           !Expands to: (3 * 3 * 3) '>>' 1 = 27 '>>' 1 = 13
```

2. This example provides incrementing and decrementing utilities and a statement that uses one of the utilities:

```
DEFINE increment (x) = x := x + 1 #;
DEFINE decrement (y) = y := y - 1 #;
INT index := 0;

increment(index);           !Expands to: INDEX := INDEX + 1;
```

3. This example fills an array with zeros:

```
DEFINE zero_array (array, length) =
BEGIN
array[0] := 0;
array[1] ':=' array FOR length - 1;
END #;

LITERAL len = 50;
INT buffer[0:len - 1];

zero_array (buffer, len);      !Fill buffer with zeros
```

4. This example displays a message, checks the condition code, and assigns an error if one occurs:

```
INT error;
INT file;
INT .buffer[0:50];
INT count_written;
INT i;

DEFINE emit (filenum, text, bytes, count, err) =
BEGIN
CALL WRITE (filenum, text, bytes, count);
IF < THEN
BEGIN
CALL FILEINFO (filenum, err);
!Process errors if any
END;

END #;

!Lots of code.
IF i = 1 THEN
emit (file, buffer, 80, count_written, error);
```



# 6 Simple Variables

A simple variable is a single-element data item of a specified data type. After you declare a simple variable, you can use its identifier in statements to access or change the data contained in the variable. You must declare variables before you use them to access data.

This section defines the syntax for declaring simple variables. The declaration determines:

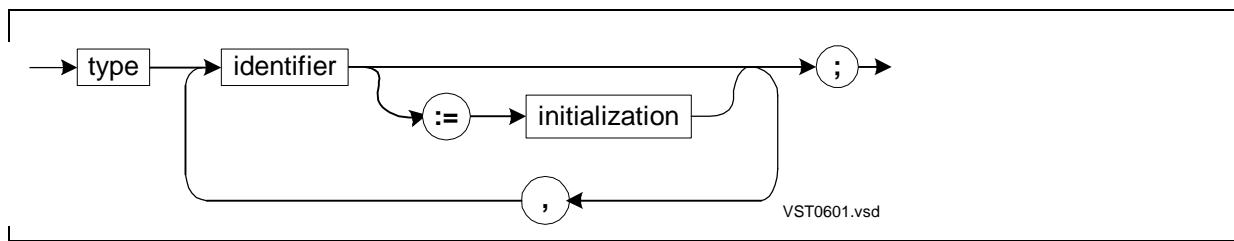
- The kind of values the simple variable can represent
- The amount of storage the compiler allocates for the variable
- The operations you can perform on the variable
- The byte or word addressing mode of the variable
- The direct or indirect addressing mode of the variable

The *TAL Programmer's Guide* describes:

- How the compiler allocates storage for simple variables
- How you access the variables

## Simple Variable Declaration

The simple variable declaration associates an identifier with a single-element data item and optionally initializes it.



**type**

is any data type described in [Section 3, Data Representation](#).

**identifier**

is the identifier of the simple variable, specified in the form described in [Section 2, Language Elements](#).

**initialization**

is an expression that represents the value to store in identifier. The default number base is decimal. The kind of expression you can specify depends on the scope of the simple variable:

- For a global simple variable, use a constant expression.
- For a local or sublocal simple variable, use any arithmetic expression including variables.

You can initialize simple variables of any data type except UNSIGNED.

## Usage Considerations

Simple variables are always directly addressed.

### Initializing With Numbers

When you initialize with a number, it must match the data type specified for the simple variable. The data type determines what kind of values the simple variable can store:

- STRING, INT, and INT(32) simple variables can contain integer constants in binary, decimal, hexadecimal, or octal base.
- REAL and REAL(64) simple variables can contain signed floating-point numbers.
- FIXED simple variables can contain signed 64-bit fixed-point numbers in binary, decimal, hexadecimal, or octal base. For decimal numbers, you can also specify a fractional part, preceded by a decimal point. If a FIXED number has a different decimal setting than the specified fpoint, the system scales the number to match the fpoint. If the number is scaled down, some precision is lost.

For more information on syntax specifying numeric constants in each number base by data type, see [Section 3, Data Representation](#).

### Initializing With Character Strings

When you initialize a simple variable with a character string, the character string can contain the same number of bytes as the simple variable or fewer. Each character in a character string requires one byte of contiguous storage. The values of any uninitialized bytes are undefined.

## Examples of Simple Variable Declarations

1. The following examples declare simple variables of different data types without initializing them:

```
STRING b;  
INT(32) dblwd1;  
REAL(64) long;  
UNSIGNED(5) flavor;
```

2. The following examples declare and initialize simple variables:

```

STRING y := "A";           !Character string
STRING z := 255;          !Byte value
INT a := "AB";            !Character string
INT b := 5 * 2;           !Expression
INT c := %B110;           !Word value
INT(32) dblwd2 := %B1011101D;      !Doubleword value
INT(32) dblwd3 := $DBL(%177775);    !Standard function
REAL flt1 := 365335.6E-3;          !Doubleword value
REAL(64) flt2 := 2718.2818284590452L-3; !Quadrupleword value

```

3. These examples declare FIXED simple variables and show how the fpoint affects storage (and scaling):

```

FIXED(-3) f := 642987F;        !Stored as 642; accessed
                                ! as 642000
FIXED(3) g := 0.642F;          !Stored as 642, accessed
                                ! as 0.642
FIXED(2) h := 1.234F;          !Stored as 123; accessed
                                ! as 1.23

```

4. This example illustrates use of constants (any level) and variables (local or sublocal only) as initialization values:

```

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

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

```



# 7

# Arrays

In TAL, an array is a one-dimensional set of elements of the same data type. Each array is stored as a collective group of elements. Once you declare an array, you can use its identifier to access the array elements individually or as a group.

This section defines the syntax for declaring arrays. The declaration determines:

- The kind of values the array can represent
- The amount of storage the compiler allocates for the array
- The operations you can perform on the array
- The byte or word addressing mode of the array
- The direct or indirect addressing mode of the array

You can declare:

- Arrays—which are stored in the user data segment or in an extended data segment
- Read-only arrays—which are stored in a user code segment

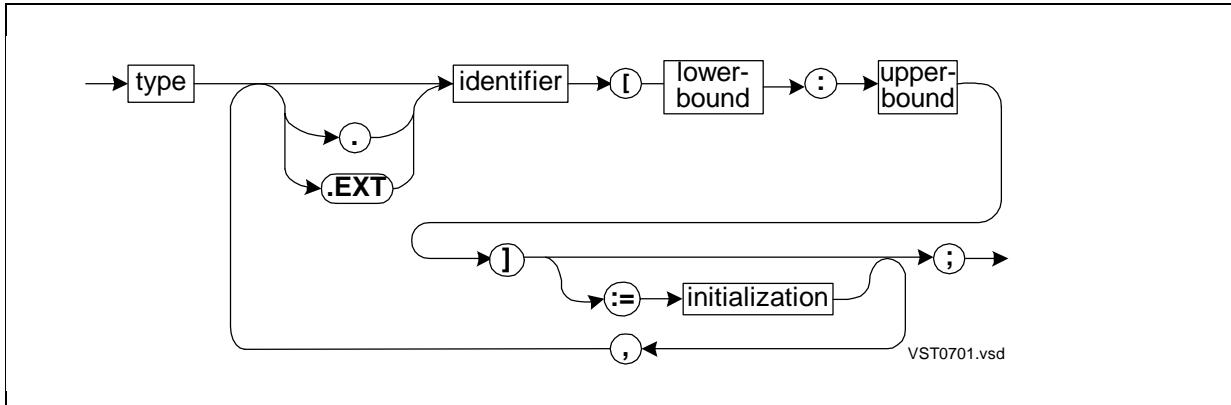
For more information on syntax for declaring arrays within structures, see [Section 8, Structures](#). This section also explains how to declare structures that simulate arrays of arrays or arrays of structures (including multidimensional arrays).

The *TAL Programmer's Guide* describes:

- How to make assignments to arrays
- How to copy, scan, or compare data in arrays
- How the compiler allocates storage for arrays

## Array Declaration

An array declaration associates an identifier with a set of elements of the same data type. The array elements are contiguously stored in the user data segment or in an extended data segment.



**type**

is any data type described in [Section 3, Data Representation](#).

- . (a period)

is the standard indirection symbol and denotes 16-bit indirect addressing. An absence of any indirection symbol (. or .EXT) denotes 16-bit direct addressing. Do not use an indirection symbol with sublocal or UNSIGNED arrays.

.EXT

is the extended indirection symbol and denotes 32-bit indirect addressing. An absence of any indirection symbol (. or .EXT) denotes 16-bit direct addressing. Do not use an indirection symbol with sublocal or UNSIGNED arrays.

**identifier**

is the identifier of the array.

**lower-bound**

is an INT constant expression (in the range –32,768 through 32,767) that specifies the index (relative to the zeroth element) of the first array element you want allocated.

**upper-bound**

is an INT constant expression (in the range –32,768 through 32,767) that specifies the index (relative to the zeroth element) of the last array element you want allocated. For arrays declared outside of structures, *upper-bound* must be equal to or greater than *lower-bound*.

**initialization**

is a constant or a constant list of values to assign to the array elements, beginning with the lower-bound element. (Constant lists are described in [Section 3, Data Representation](#).) If you specify fewer initialization values than the number of

elements, the values of uninitialized elements are undefined. You cannot initialize extended indirect local arrays or UNSIGNED arrays.

Specify initialization values that are appropriate for the data type of the array. For example, if the decimal setting of an initialization value differs from the *fpoint* of a FIXED array, the system scales the initialization value to match the *fpoint*. If the initialization value is scaled down, some precision is lost.

## Usage Considerations

UNSIGNED arrays and sublocal arrays must be directly addressed. For most other arrays, you should use indirection because storage areas for direct global and local data are limited. For very large arrays, use extended indirection. You access indirect data by identifier as you do direct data.

The data type determines:

- The kind of values that are appropriate for the array
- 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

## Examples of Array Declarations

1. These examples declare arrays with various bounds. The arrays are indirectly addressed except the UNSIGNED array, which must be directly addressed:

```

FIXED          .array_a[0:3]; !Four-element array
INT            .array_b[0:49]; !Fifty-element array
UNSIGNED(1)    flags[0:15];   !Array of 16 one-bit elements

```

2. These examples declare arrays and initialize them with constants:

```

INT a_array[0:3]      := -1; !Store -1 in element [0];
                           !values in elements [1:3]
                           !are undefined
INT b_array[0:1]     := "abcd"; !Store one character per byte

```

3. These examples declare and initialize arrays using constant lists:

```

INT .c_array[0:5]      := [1,2,3,4,5,6]; !Constant list
STRING .buffer[0:102]   := [ "A constant list can consist ",
                            "of several character string constants, ",
                            "one to a line, separated by commas." ];
INT(32) .mixed[0:3]    := ["abcd", 1D, %B0101011D, %20D];
                        !Mixed constant list
LITERAL len = 80;          !Length of array
STRING .buffer[0:len - 1] := len * [" "];
                           !Repetition factor
FIXED .f[0:35] := 3*[2*[1F,2F], 4*[3F,4F]];
                     !Repetition factors
LITERAL cr = %15,
       lf = %12;
STRING .err_msg[0:9]     := [cr, lf, "ERROR", cr, lf, 0];
                           !Constant list

```

4. This example initializes all arrays except the local extended indirect array:

```

INT(32) .a[0:1] := [5D, 7D];           !Initialize global array
PROC my_procedure;
BEGIN
  STRING .b[0:1] := ["A", "B"];        !Initialize local standard
                                      ! indirect array
  FIXED .EXT c[0:3];                 !Cannot initialize local
                                      !extended indirect array

SUBPROC my_subproc;
BEGIN
  INT d[0:2] := ["Hello! "];         !Initialize sublocal array
  !Lots of code
END;
END;

```

5. The following examples show how positive and negative fpoints affect storage and access of FIXED values. A positive *fpoint* specifies the number of decimal places to the right of the decimal point for storage and access. The system truncates any value that does not fit:

```

FIXED(2) x[0:1]      := [ 0.64F, 2.348F ];
                         !Stored as 64 and 234; accessed as 0.64 and 2.34

```

6. A negative *fpoint* specifies the number of integer places to the left of the decimal point to truncate when the value is stored. When you access the value, the system replaces the truncated digits with zeros:

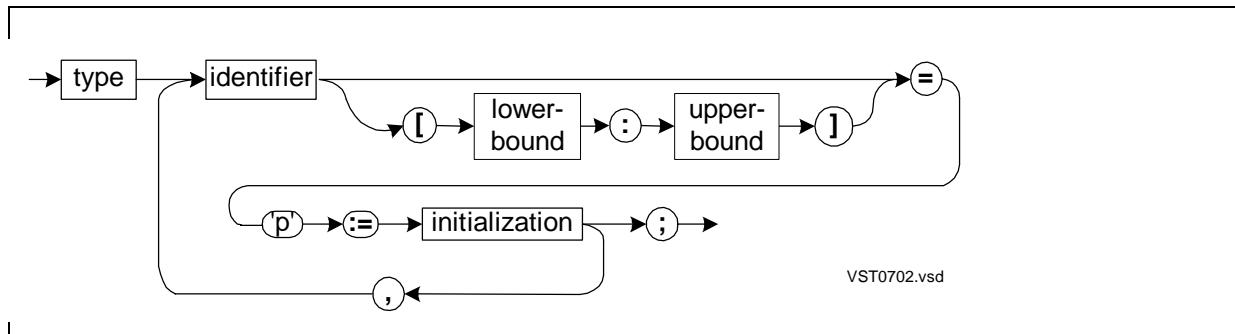
```

FIXED(-3) y[0:1]      := [ 642913F, 1234F ];
                         !Stored as 642 and 1; accessed as 642000 and 1000

```

# Read-Only Array Declaration

A read-only array declaration allocates storage for a nonmodifiable array in a user code segment. Read-only arrays are sometimes referred to as P-relative arrays, because they are addressed using the program counter (the P register).



**type**

is any data type except UNSIGNED.

**identifier**

is the identifier of the read-only array.

**lower-bound**

is an INT constant expression (in the range –32,768 through 32,767) that specifies the index (relative to the zeroth element) of the first array element you want allocated. The default value is 0.

**upper-bound**

is an INT constant expression (in the range –32,768 through 32,767) that specifies the index (relative to the zeroth element) of the last array element you want allocated. The default value is the number of elements initialized minus one.

'P'

specifies a read-only array. Read-only arrays are addressed using the program counter (the P register).

**initialization**

is a constant list to assign to the array elements. You must initialize read-only arrays when you declare them. (Constant lists are described in [Section 3, Data Representation](#).)

Specify initialization values that are appropriate for the data type of the array. For example, if the decimal setting of an initialization value differs from the *fpoint* of a

FIXED array, the system scales the initialization value to match the *fpoint*. If the initialization value is scaled down, some precision is lost.

## Usage Considerations

Because code segments have no primary or secondary areas, read-only arrays must be direct. If you declare an indirect read-only array, the compiler ignores the indirection symbol and issues warning 37 (array access changed from indirect to direct).

You must initialize read-only arrays. UNSIGNED read-only arrays are not allowed, because they cannot be initialized.

If you declare a read-only array in a RESIDENT procedure, the array is also resident in main memory.

Binder binds each global read-only array into any code segment containing a procedure that references the array.

The *TAL Programmer's Guide* gives information on accessing read-only arrays. In summary, you can access read-only arrays 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-addr to read the last character of a string. You can use next-addr to compute the length of the string.

## Example of Read-Only Array Declaration

This example declares read-only arrays using default lower and upper bounds and initializes them with constant lists:

```
STRING prompt = 'P'          := [ "Enter Character: ", 0 ];
INT error = 'P'              := [ "INCORRECT INPUT" ];
```

# **8 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, the unit price, and the quantity on hand.

This section describes the syntax for declaring:

- Definition structures
- Template structures
- Referral structures
- Structure items—simple variables, arrays, substructures, filler bytes, filler bits, simple pointers, structure pointers, and redefinitions

Section 8, “Using Structures,” in the *TAL Programmer’s Guide* describes:

- How the compiler aligns structures and structure items
- How the compiler allocates storage for structures
- How to declare arrays of arrays, arrays of structures, and multidimensional arrays
- How you can access structures and structure items

## **Kinds of Structures**

A structure declaration associates an identifier with any of three kinds of structures, as listed in [Table 8-1](#).

---

**Table 8-1. Kinds of Structures**

<b>Structure</b>	<b>Description</b>
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 (or body) 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 a 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 limit is 64 for direct substructures such as:

```
STRUCT A;
BEGIN
  STRUCT B;
  BEGIN
    ...
  END;
END;
```

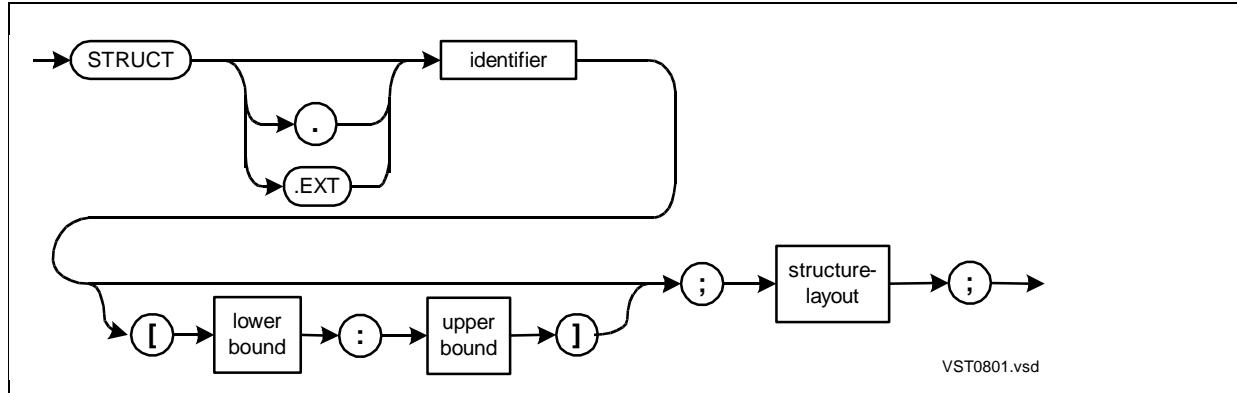
However, for indirect substructures, this limit is smaller depending on the complexity of the structure.

The syntax for declaring each structure item is described after the syntax for declaring structures. 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.

# Definition Structure Declaration

A definition structure declaration describes a structure layout and allocates storage for it.



- . (a period)

is the standard indirection symbol and denotes 16-bit indirect addressing. An absence of any indirection symbol (.) or (.EXT) denotes 16-bit direct addressing. Do not use an indirection symbol with sublocal structures.

- .EXT

is the extended indirection symbol and denotes 32-bit indirect addressing. An absence of any indirection symbol (.) or (.EXT) denotes 16-bit direct addressing. Do not use an indirection symbol with sublocal structures.

**identifier**

is the identifier of the definition structure.

**lower-bound**

is an INT constant expression (in the range –32,768 through 32,767) that specifies the index (relative to the zeroth structure occurrence) of the first structure occurrence you want allocated. Each occurrence is one copy of the structure. The default value is 0.

**upper-bound**

is an INT constant expression (in the range –32,768 through 32,767) that specifies the index (relative to the zeroth structure occurrence) of the last structure occurrence you want allocated. The default value is 0. For a single-occurrence structure, omit both bounds or specify the same value for both bounds.

**structure-layout**

is a BEGIN-END construct that can contain declarations for simple variables, arrays, substructures, filler bits, filler bytes, redefinitions, simple pointers, and structure pointers. The size of one occurrence of the structure is the size of the layout. A single structure occurrence must not exceed 32,767 bytes.

## Usage Considerations

Structures declared in subprocedures must be directly addressed. For most other structures, you should use indirection because storage areas for direct global and local variables are limited. You access indirect structures by identifier as you do direct structures.

For very large structures, you should use the .EXT symbol to declare extended indirect structures. 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 extended data segment yourself, follow the instructions given in the *TAL Programmer's Guide* in Appendix B, "Managing Addressing."

Structures always start on a word boundary.

## Examples of Definition Structure Declarations

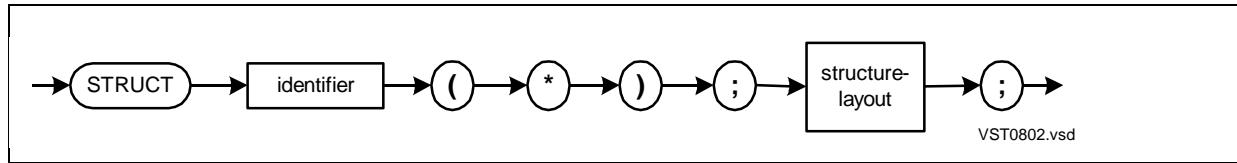
This example declares indirect definition structures:

```
STRUCT .inventory1[0:49];           !Standard indirect structure
BEGIN
  INT item;
  FIXED(2) price;
  INT quantity;
END;

STRUCT .EXT inventory2[0:9999]; !Extended indirect structure
BEGIN
  INT item;
  FIXED(2) price;
  INT quantity;
END;
```

# Template Structure Declaration

A template structure declaration describes a structure layout but allocates no space for it. You use the template layout in subsequent structure, substructure, or structure pointer declarations.



**identifier**

is the identifier of the template structure, with no indirection symbol.

**( \* )**

is the symbol for a template structure.

**structure-layout**

is a BEGIN-END construct that can contain declarations for simple variables, arrays, substructures, filler bits, filler bytes, redefinitions, simple pointers, and structure pointers. The size of one structure occurrence is the size of the layout and must not exceed 32,767 bytes.

## Usage Considerations

A template structure has meaning only when you refer to it in the subsequent declaration of a referral structure, referral substructure, or structure pointer. The subsequent declaration allocates space for a structure whose layout is the same as the template layout.

## Example of Template Structure Declaration

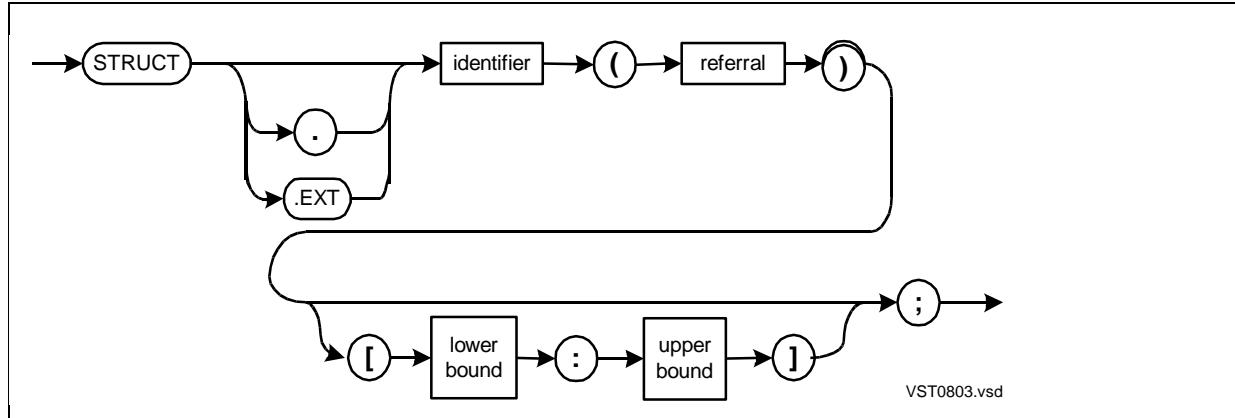
This declaration associates an identifier with a template structure layout but allocates no space for it:

```

STRUCT inventory (*);      !Template structure
BEGIN                      !Structure layout
  INT item;
  FIXED(2) price;
  INT quantity;
END;
  
```

# Referral Structure Declaration

A referral structure declaration allocates storage for a structure whose layout is the same as the layout of a previously declared structure or structure pointer.



- . (a period)

is the standard indirection symbol and denotes 16-bit indirect addressing. An absence of any indirection symbol (.) or (.EXT) denotes 16-bit direct addressing. Do not use an indirection symbol with sublocal structures.

.EXT

is the extended indirection symbol and denotes 32-bit indirect addressing. An absence of any indirection symbol (.) or (.EXT) denotes 16-bit direct addressing. Do not use an indirection symbol with sublocal structures.

identifier

is the identifier of the new referral structure.

referral

is the identifier of a previously declared structure or structure pointer that provides the structure layout for this structure.

lower-bound

is an INT constant expression (in the range –32,768 through 32,767) that specifies the index (relative to the zeroth structure occurrence) of the first structure occurrence you want to allocate. Each occurrence is one copy of the structure. The default value is 0.

upper-bound

is an INT constant expression (in the range –32,768 through 32,767) that specifies the index (relative to the zeroth structure occurrence) of the last structure

occurrence you want to allocate. The default value is 0. For a single-occurrence structure, omit both bounds or specify the same value for both bounds.

## Usage Considerations

The compiler allocates storage for the 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

Structures declared in subprocedures must be directly addressed. For most other structures, you should use indirection because storage areas for direct global and local variables are limited. You access indirect structures by identifier as you do direct structures.

For very large structures, you should use the .EXT symbol to declare extended indirect structures. 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 extended data segment yourself, follow the instructions given in the *TAL Programmer's Guide*, in Appendix B, "Managing Addressing."

Structures always start on a word boundary.

## Example of Referral Structure Declaration

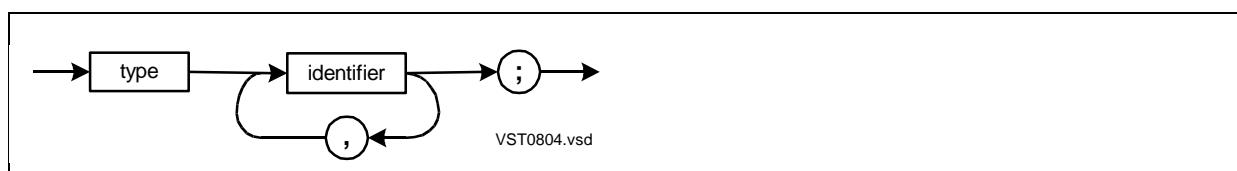
This example declares a template structure and a referral structure that references the template structure. The referral structure imposes its addressing mode and number of occurrences on the layout of the template structure:

```
STRUCT record (*);           !Declare template structure
BEGIN
  STRING name[0:19];
  STRING addr[0:29];
  INT acct;
END;

STRUCT .customer(record)[1:50]; !Declare referral structure
```

## Simple Variables Declared in Structures

The simple variable declaration associates a name with a single-element data item. When you declare a simple variable inside a structure, the form is:



type

is any data type described in [Section 3, Data Representation](#).

identifier

is the identifier of the simple variable.

## Usage Considerations

You cannot initialize a simple variable when you declare it inside a structure. You can subsequently assign a value to the simple variable by using an assignment statement.

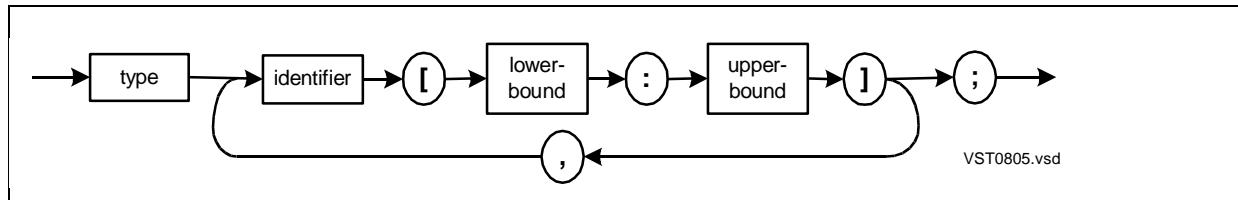
## Example of Simple Variables in Structures

This example declares simple variables in a structure:

```
STRUCT .inventory[0:49];           !Declare definition structure
BEGIN
    INT item;                      !Declare three simple
    FIXED(2) price;                ! variables within the
    INT quantity;                 ! structure layout
END;
```

## Arrays Declared in Structures

An array declaration associates an identifier with a collectively stored set of elements of the same data type. When you declare an array inside a structure, the form is:



type

is any data type described in [Section 3, Data Representation](#).

identifier

is the identifier of the array.

lower-bound

is an INT constant expression (in the range –32,768 through 32,767) that specifies the index (relative to the zeroth array element) of the first array element you want allocated. Both lower and upper bounds are required.

upper-bound

is an INT constant expression (in the range –32,768 through 32,767) that specifies the index (relative to the zeroth array element) of the last array element you want allocated. Both lower and upper bounds are required.

## Usage Considerations

When you declare arrays inside a structure, the following guidelines apply:

- You cannot initialize arrays declared in structures. You can assign values to such arrays only by using assignment statements.
- You cannot declare indirect arrays or read-only arrays in structures.
- You can specify array bounds of [n : n-1] in structures as described in the *TAL Programmer's Guide*. Such an array is addressable but uses no memory.

## Example of Arrays in Structures

This example declares arrays in a structure:

```
STRUCT record;           !Declare definition structure
BEGIN
  STRING name[0:19];    !Declare arrays within the
  STRING addr[0:29];    ! structure layout
  INT acct;
END;
```

## Substructure Declaration

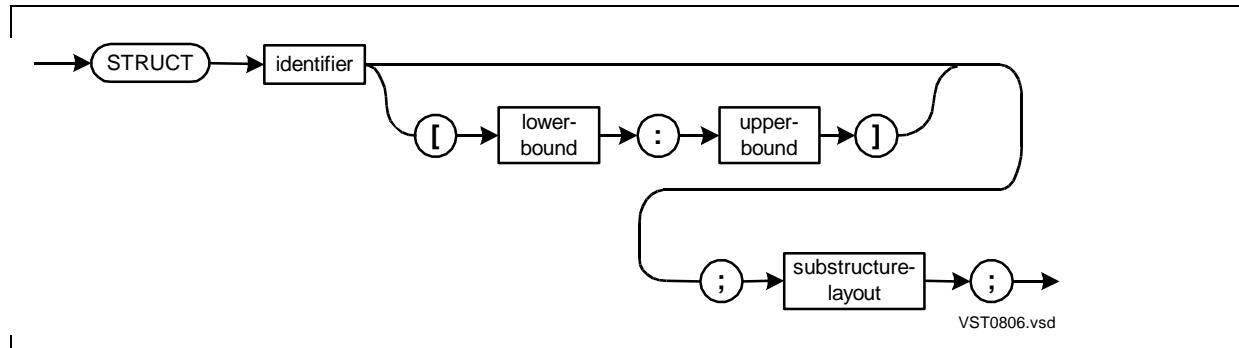
A substructure is a structure embedded within another structure or substructure. You can declare substructures that have the following characteristics:

- Substructures must be directly addressed.
- Substructures have byte addresses, not word addresses.
- Substructures can be nested to a maximum of 64 levels.
- Substructures can have bounds of [n : n-1] as described in the *TAL Programmer's Guide*. Such a substructure is addressable but uses no memory.

You can declare definition substructures or referral substructures, described next.

# Definition Substructure Declaration

A definition substructure describes a layout and allocates storage for it.



**identifier**

is the identifier of the definition substructure.

**lower-bound**

is an INT constant expression (in the range –32,768 through 32,767) that specifies the index (relative to the zeroth substructure occurrence) of the first substructure occurrence you want allocated. Each occurrence is one copy of the substructure. The default value is 0.

**upper-bound**

is an INT constant expression (in the range –32,768 through 32,767) that specifies the index (relative to the zeroth substructure occurrence) of the last substructure occurrence you want allocated. The default value is 0. For a single-occurrence substructure, omit both bounds or specify the same value for both bounds.

**substructure-layout**

is the same BEGIN-END construct as for structures. It 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.

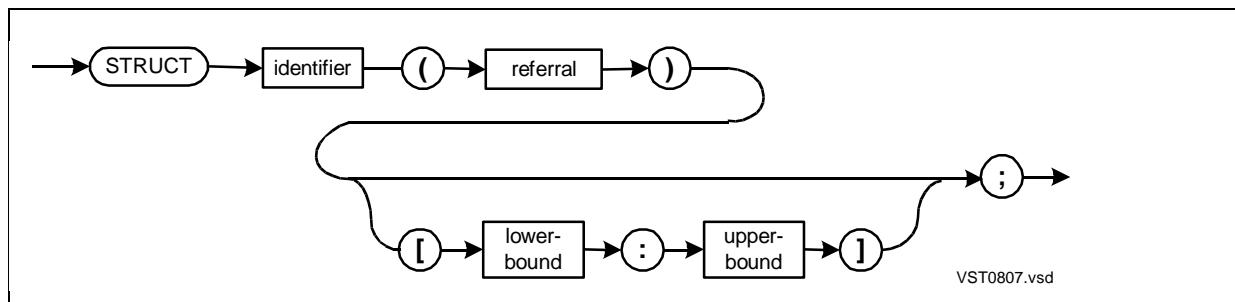
## Example of Definition Substructure Declaration

This example declares two occurrences of a structure, each of which contains 50 occurrences of a definition substructure:

```
STRUCT .warehouse[ 0 :1 ] ;           ! Two warehouses
  BEGIN
    STRUCT inventory [ 0 :49 ] ;       !Definition substructure
      BEGIN
        INT item_number;
        FIXED(2) price;
        INT on_hand;
      END;
  END;
```

## Referral Substructure Definition

A referral substructure allocates storage for a substructure whose layout is the same as the layout of a previously declared structure or structure pointer.



**identifier**

is the identifier of the referral substructure.

**referral**

is the identifier of a structure that provides the structure layout. You can specify any previously declared structure (except the encompassing structure) or structure pointer. If the previous structure has an odd-byte size, the compiler rounds the size of the new substructure up so that it has an even-byte size.

**lower-bound**

is an INT constant expression (in the range –32,768 through 32,767) that specifies the index (relative to the zeroth occurrence) of the first substructure occurrence you want allocated. Each occurrence is one copy of the substructure. The default value is 0.

**upper-bound**

is an INT constant expression (in the range –32,768 through 32,767) that specifies the index (relative to the zeroth occurrence) of the last substructure occurrence you want allocated. The default value is 0. For a single-occurrence substructure, omit both bounds or specify the same value for both bounds.

## Example of Referral Substructure Declaration

This example declares a referral substructure that uses a template structure layout:

```

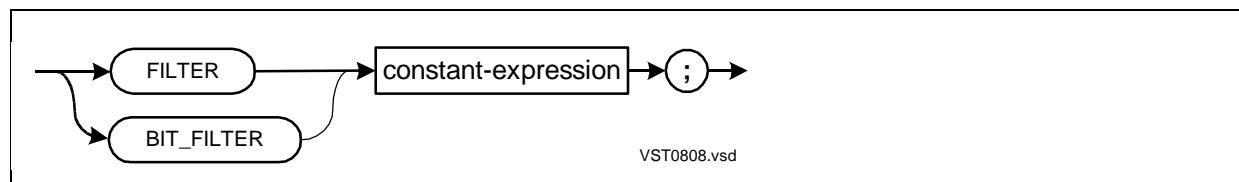
STRUCT temp( * );
    BEGIN
        STRING a[0:2];
        INT b;
        STRING c;
    END;

STRUCT .ind_struct;
    BEGIN
        INT header[0:1];
        STRING abyte;
        STRUCT abc (temp) [0:1]; !Declare referral substructure
    END;                                !The size of IND_STRUCT.ABC[0]
                                         ! is eight bytes
                                         !Space allocated for this layout
                                         !No space allocated for
                                         ! this layout
                                         !Declare template structure
                                         !Declare definition structure
                                         !Space allocated for this layout
                                         !The size of IND_STRUCT.ABC[0]
                                         ! is eight bytes

```

## Filler Declaration

A filler declaration allocates a byte or bit place holder in a structure.



**FILLER**

allocates the specified number of byte place holders.

**BIT\_FILLER**

allocates the specified number of bit place holders.

**constant-expression**

is a positive integer constant value that specifies a number of filler units in one of the following ranges:

<b>FILLER</b> <b>BIT_FILLER</b>	0 through 32,767 bytes 0 through 255 bits
------------------------------------	--

## Usage Considerations

You can declare filler bits and filler bytes, but you cannot access such filler locations.

If the structure layout must match a structure layout defined in another program, your structure declaration need only include data items used by your program and can use filler bits or bytes for the unused space. You can also use filler bytes to document compiler-allocated alignment pad bytes (described in the *TAL Programmer's Guide*).

## Examples of Filler Declarations

1. This example shows filler byte declarations:

```
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;
```

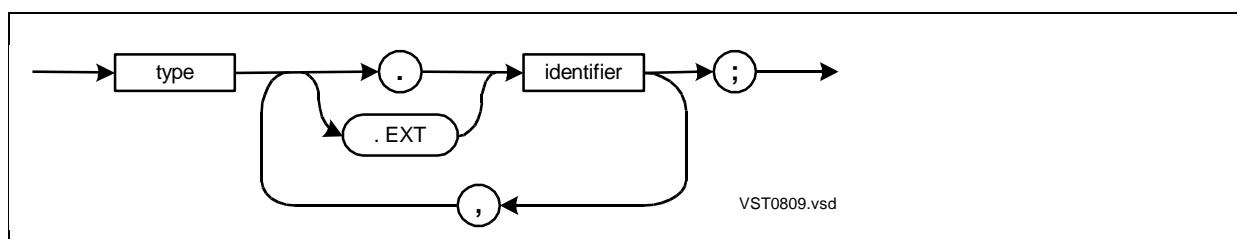
2. This example shows a filler bit declaration:

```
STRUCT .flags;
BEGIN
UNSIGNED(1) flag1;
UNSIGNED(1) flag2;
UNSIGNED(2) state;          !State = 0, 1, 2, or 3
BIT_FILLER 12;              !Place holder for unused space
END;
```

For more information, see the filler byte example in [Definition Substructure Redefinition](#) on page 8-20.

## Simple Pointers Declared in Structures

A simple pointer is a variable that contains the memory address of a simple variable or an array.



**type**

is any data type except UNSIGNED. The data type determines how much data the simple pointer can access at a time—a byte, word, doubleword, or quadrupleword.

- . (period)

is the standard indirection symbol and denotes a standard (16-bit) pointer.

.EXT

is the extended indirection symbol and denotes an extended (32-bit) pointer.

**identifier**

is the identifier of the simple pointer.

## Usage Considerations

The data type determines the size of 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

---

The addressing mode and data type determines the kind of address the simple 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.

---

**Table 8-4. Addresses in Simple Pointers**

Addressing Mode	Data Type	Kind of Addresses
Standard	Any except STRING	16-bit word address 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.)

Before you reference a pointer declared in a structure, be sure to assign an address to it by using an assignment statement, as described in Section 8, “Using Structures,” in the *TAL Programmer’s Guide*.

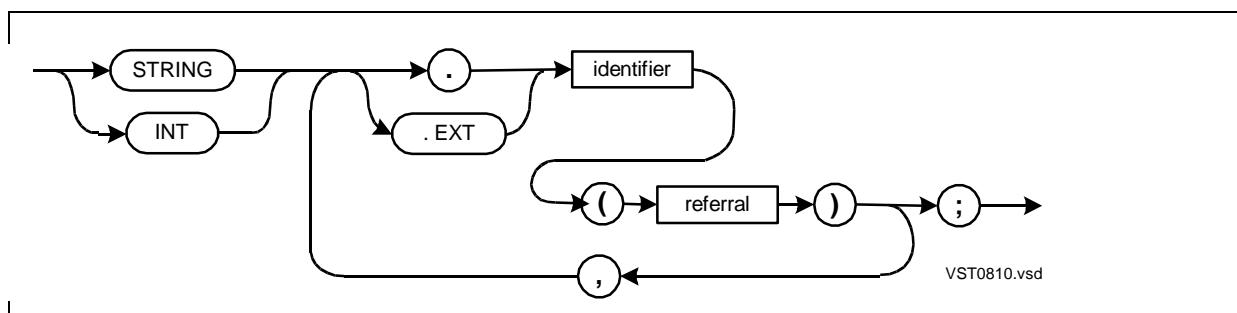
## Example of Simple Pointer Declarations

This example shows simple pointer declarations within a structure:

```
STRUCT my_struct;
BEGIN
  FIXED .std_pointer;           ! Standard simple pointer
  STRING .EXT ext_pointer;     ! Extended simple pointer
END;
```

## Structure Pointers Declared in Structures

A structure simple pointer is a variable that contains the address of a structure. When you declare a structure pointer inside a structure, the form is:



STRING

is the STRING attribute.

INT

is the INT attribute

- . (period)

is the standard indirection symbol and denotes a standard (16-bit) pointer.

- .EXT

is the extended indirection symbol and denotes an extended (32-bit) pointer.

**identifier**

is the identifier of the structure pointer.

**referral**

is the identifier of a structure that provides the structure layout. You can specify any previously declared structure (including the encompassing structure) or structure pointer.

## Usage Considerations

The addressing mode and STRING or INT attribute determine the kind of addresses a structure pointer can contain, as described in [Table 8-5](#).

---

**Table 8-5. Addresses in Structure Pointers**

Addressing Mode	STRING or INT Attribute	Kinds of Address
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 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.

Before you reference a pointer declared in a structure, be sure to assign an address to it by using an assignment statement, as described in Section 8, “Using Structures,” in the *TAL Programmer’s Guide*.

## Example of Structure Pointer Declaration

This example shows a structure pointer declaration within a structure:

```
STRUCT struct_a;                      !Declare STRUCT_A
BEGIN
    INT a;
    INT b;
END;
STRUCT struct_b;                      !Declare STRUCT_B
BEGIN
    INT .EXT struct_pointer (struct_a); !Declare STRUCT_POINTER
    STRING a;
END;
```

## Redefinition Declaration

A redefinition declares a new identifier and sometimes a new description for a previous item in the same structure. You can declare these kinds of redefinitions:

- Simple variable redefinition
- Array redefinition
- Substructure redefinition
- Simple pointer redefinition
- Structure pointer redefinition

## Redefinition Rules

The following rules apply to all redefinitions in structures:

- The new item must be of the same length or shorter than the previous item.
- The new item and the previous item must be at the same BEGIN-END level of a structure.

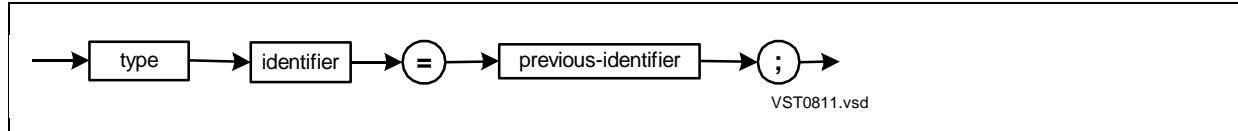
Additional rules are given in subsections that describe each kind of redefinition in the following pages.

## Redefinitions Outside Structures

For information on redefinitions outside structures, see [Section 10, Equivalenced Variables](#).

# Simple Variable Redefinition

A simple variable redefinition associates a new simple variable with a previous item at the same BEGIN-END level of a structure.



**type**

is any data type except UNSIGNED.

**identifier**

is the identifier of the new simple variable.

**previous-identifier**

is the identifier of a simple variable, array, substructure, or pointer previously declared in the same structure. You cannot specify an index with this identifier.

## Usage Considerations

In a redefinition, the new item and the previous (nonpointer) item both must have a byte address or both must have a word address. If the previous item is a pointer, the data it points to must be word addressed or byte addressed to match the new item.

## Example of Simple Variable Redefinition

This declaration redefines the left byte of INT\_VAR as STRING\_VAR:

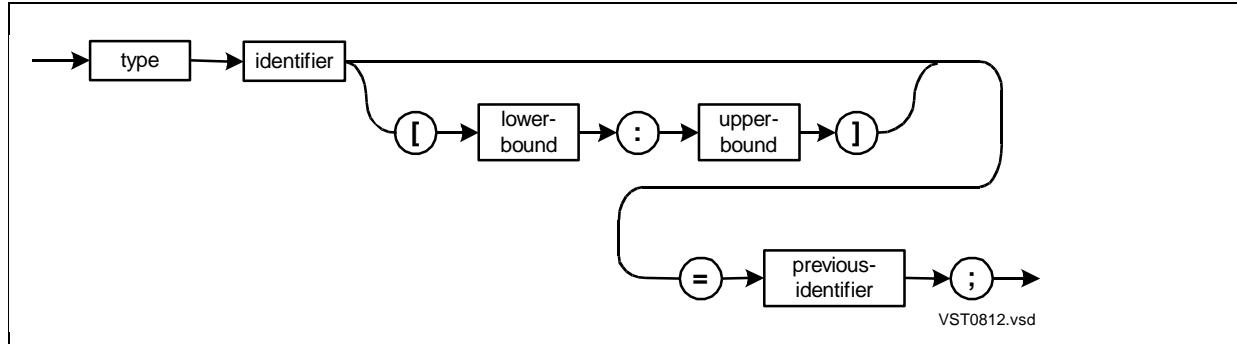
```

STRUCT .mystruct ;
BEGIN
    INT int_var;
    STRING string_var = int_var; !Redefinition
END ;

```

# Array Redefinition

An array redefinition associates a new array with a previous item at the same BEGIN-END level of a structure.



**type**

is any data type except UNSIGNED.

**identifier**

is the identifier of the new array.

**lower-bound**

is an INT constant expression (in the range -32,768 through 32,767) that specifies the index (relative to the zeroth element) of the first array element you want allocated. The default value is 0.

**upper-bound**

is an INT constant expression (in the range -32,768 through 32,767) that specifies the index (relative to the zeroth element) of the last array element you want allocated. The default value is 0.

**previous-identifier**

is the identifier of a simple variable, array, substructure, or pointer previously declared in the same structure. You cannot specify an index with this identifier.

## Usage Considerations

In a redefinition, the new item and the previous (nonpointer) item both must have a byte address or both must have a word address. If the previous item is a pointer, the data it points to must be word addressed or byte addressed to match the new item.

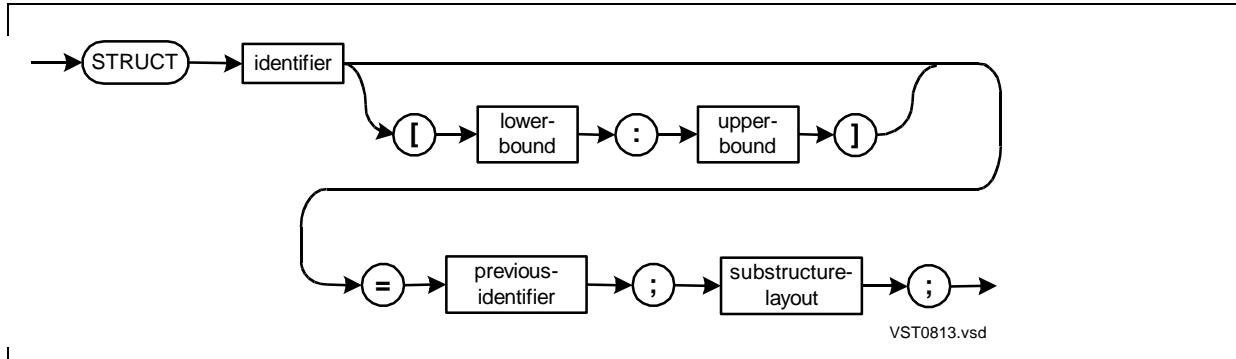
## Example of Array Redefinition

This declaration redefines an INT array as an INT(32) array:

```
STRUCT .s;
BEGIN
  INT a[0:3];
  INT(32) b[0:1] = a;      !Redefinition
END;
```

## Definition Substructure Redefinition

A definition substructure redefinition associates a new definition substructure with a previous item at the same BEGIN-END level of a structure.



**identifier**

is the identifier of the new substructure.

**lower-bound**

is an INT constant expression (in the range -32,768 through 32,767) that specifies the index (relative to the zeroth substructure occurrence) of the first substructure occurrence you want allocated. Each occurrence is one copy of the substructure. The default value is 0.

**upper-bound**

is an INT constant expression (in the range -32,768 through 32,767) that specifies the index (relative to the zeroth substructure occurrence) of the last substructure occurrence you want allocated. The default value is 0. To declare a single-occurrence substructure, omit both bounds or specify the same value for both bounds.

**previous-identifier**

is the identifier of a simple variable, array, substructure, or pointer previously declared in the same structure. No index is allowed with this identifier.

### substructure-layout

is the same BEGIN-END construct as for structures. It 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.

## Usage Considerations

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.

## Examples of Definition Substructure Redefinitions

1. In this example, both substructures (B and C) have odd-byte alignments.

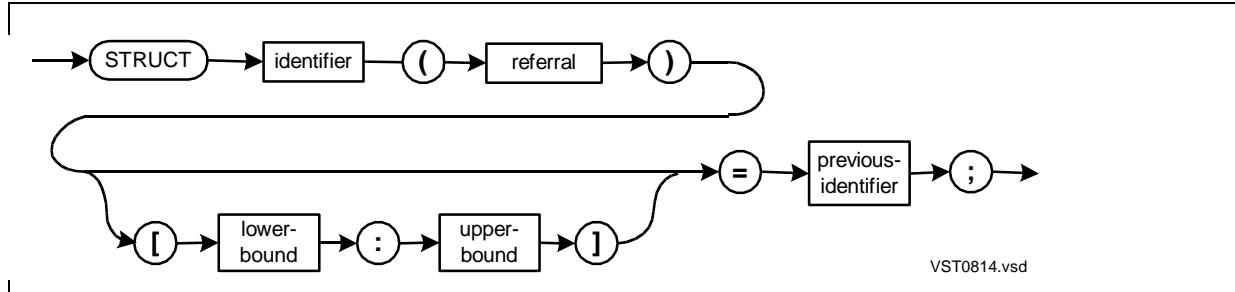
```
STRUCT a;
BEGIN
  STRING x;
  STRUCT b;                      !B starts on odd byte
  BEGIN
    STRING y;
  END;
  STRUCT c = b;                  !Redefine B as C, also on
  BEGIN                           ! odd byte
    STRING z;
  END;
END;
```

2. In this example, MYSUB2 redefines the left byte of the INT variable in MYSUB1 as a STRING variable:

```
STRUCT mystruct;
BEGIN
  STRUCT mysub1;                  !Declare MYSUB1
  BEGIN
    INT int_var;
  END;
  STRUCT mysub2 = mysub1;        !Redefine MYSUB1 as MYSUB2
  BEGIN
    STRING string_var;
  END;
END;
```

# Referral Substructure Redefinition

A referral substructure redefinition associates a new referral substructure with a previous item at the same BEGIN-END level of a structure.



**identifier**

is the identifier of the new substructure.

**referral**

is the identifier of a structure that provides the structure layout. You can specify any previously declared structure (except the encompassing structure) or structure pointer. If the previous structure has an odd-byte size, the compiler rounds the size of the new substructure up so it has an even-byte size.

**lower-bound**

is an INT constant expression (in the range -32,768 through 32,767) that specifies the index (relative to the zeroth substructure occurrence) of the first substructure occurrence you want allocated. Each occurrence is one copy of the substructure. The default value is 0.

**upper-bound**

is an INT constant expression (in the range -32,768 through 32,767) that specifies the index (relative to the zeroth substructure occurrence) of the last substructure occurrence you want allocated. The default value is 0. To declare a single-occurrence substructure, omit both bounds or specify the same value for both bounds.

**previous-identifier**

is the identifier of a simple variable, array, substructure, or pointer previously declared in the same structure. No index is allowed with this identifier.

## Usage Considerations

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.

## Example of Referral Substructure Declaration

This example declares a referral substructure redefinition that uses a template structure layout:

```

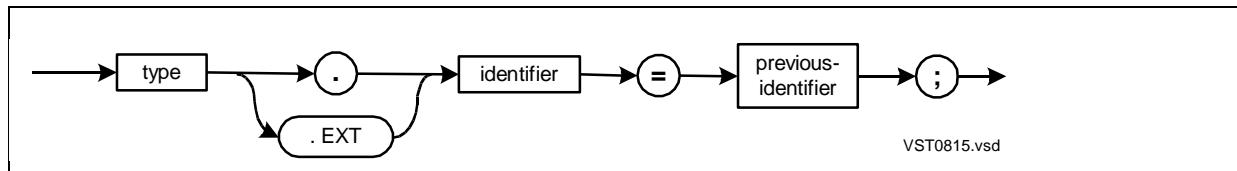
STRUCT temp( * );
    !Declare template structure
    BEGIN
        STRING a[ 0:2 ];
        INT b;
        STRING c;
    END;

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 Pointer Redefinition

A simple pointer redefinition associates a new simple pointer with a previous item at the same BEGIN-END level of a structure.



**type**

is any data type except UNSIGNED. The data type determines how much data the simple pointer can access at a time—a byte, word, doubleword, or quadrupleword.

- . (period)

is the standard indirection symbol and denotes a standard (16-bit) pointer.

.EXT

is the extended indirection symbol and denotes an extended (32-bit) pointer.

**identifier**

is the identifier of the new simple pointer.

**previous-identifier**

is the identifier of a simple variable, array, substructure, or pointer previously declared in the same structure. No index is allowed with this identifier.

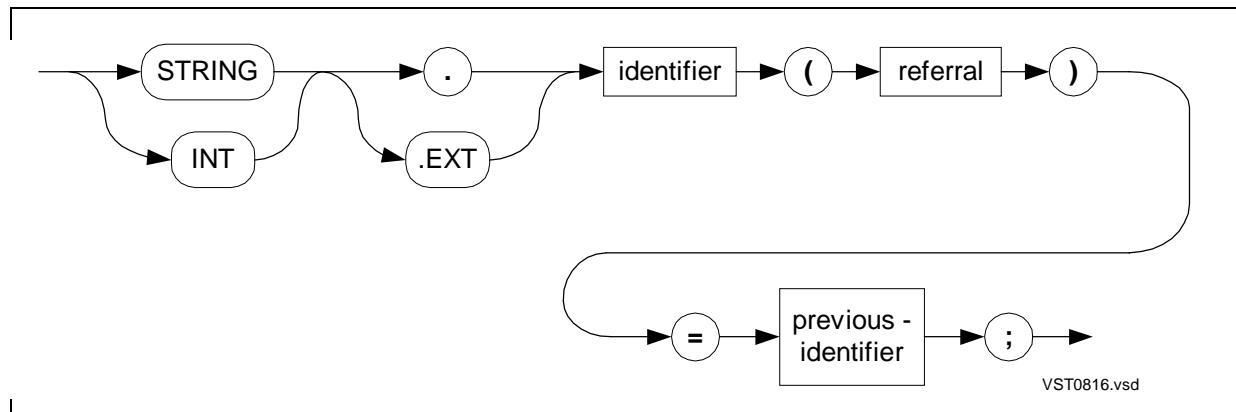
## Example of Simple Pointer Redefinition

This example declares new simple pointer EXT\_POINTER to redefine simple variable VAR:

```
STRUCT my_struct;
BEGIN
  STRING var[0:5];
  STRING .EXT ext_pointer = var;      !Redefinition
END;
```

## Structure Pointer Redefinition

A structure pointer redefinition associates a new structure pointer with a previous item at the same BEGIN-END level of a structure.



STRING

is the STRING attribute.

INT

is the INT attribute.

. (period)

is the standard indirection symbol and denotes a standard (16-bit) pointer.

.EXT

is the extended indirection symbol and denotes an extended (32-bit) pointer.

identifier

is the identifier of the new structure pointer.

referral

is the identifier of a structure that provides the structure layout. You can specify any previously declared structure (including the encompassing structure) or structure pointer.

previous-identifier

is the identifier of a simple variable, array, substructure, or pointer previously declared in the same structure. No index is allowed with this identifier.

## Usage Considerations

The addressing mode and STRING or INT attribute determine the kind of addresses a structure pointer can contain, as described in [Table 8-5](#) on page 8-16.

## Example of Structure Pointer Redefinitions

This example declares 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(32) ext_link_addr;
  INT .EXT ext_link (record) = ext_link_addr; !Redefinition
END;
```



# 9 Pointers

This section describes the syntax for declaring and initializing pointers you manage yourself. You can declare the following kinds of pointers:

- Simple pointer—a variable into which you 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 store the memory address of a structure which you can access with this structure pointer.

Pointers—simple pointers and structure pointers—can be standard or extended:

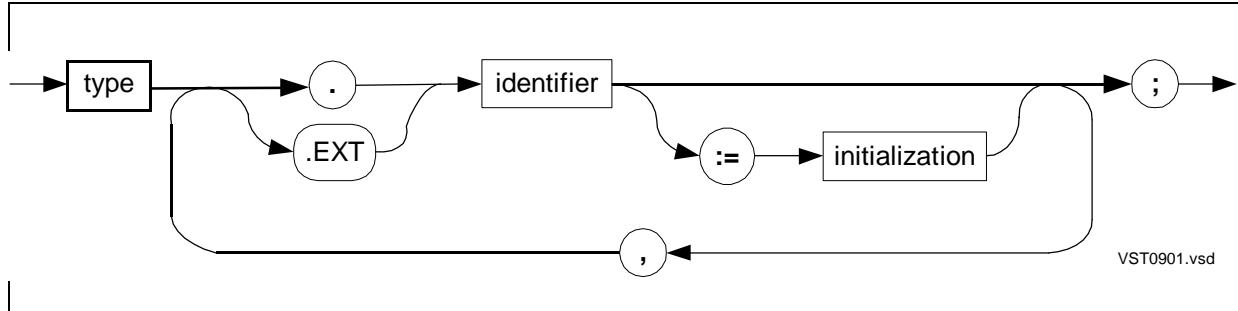
- Standard (16-bit) pointers can access data only in the user data segment.
- Extended (32-bit) pointers can access data in any segment, normally the automatic extended data segment.

Other information on pointers appears in the TAL manual set as follows:

Information	Manual	Section/Appendix
Pointer assignments and access of data to which the pointer ‘points’	<i>TAL Programmer’s Guide</i>	9, “Using Pointers”
Pointers declared inside structures	<i>TAL Programmer’s Guide</i> <i>TAL Reference Manual</i>	8, “Using Structures” <a href="#"><u>Section 8, Structures</u></a>
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”

# Simple Pointer Declaration

A simple pointer declaration associates an identifier with a memory location that contains the user-initialized address of a simple variable or array.



**type**

is any data type except UNSIGNED. The data type determines how much data the simple pointer can access at one time—byte, word, doubleword, or quadrupleword.

- . (period)

is the standard indirection symbol and denotes a standard (16-bit) pointer.

.EXT

is the extended indirection symbol and denotes an extended (32-bit) pointer.

**identifier**

is the identifier of the simple pointer.

**initialization**

is an expression that represents a memory address, as described in [Simple Pointer Initializations](#) on page 9-3.

## Usage Considerations

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].

The data type determines the size of data a simple pointer can access at a time, as listed in [Table 9-1](#) on page 9-3.

---

**Table 9-1. Data Accessed by Simple Pointers**

<b>Data Type</b>	<b>Accessed Data</b>
STRING	Byte
INT	Word
INT (32)	Doubleword
REAL	Doubleword
REAL (64)	Quadrupleword
FIXED	Quadrupleword

---

## Simple Pointer Initializations

The addressing mode and data type of the simple pointer determines the kind of address the pointer can contain, as described in [Table 9-2](#).

---

**Table 9-2. Addresses in Simple Pointers**

<b>Addressing</b>		<b>Kinds of Addresses</b>
<b>Mode</b>	<b>Data Type</b>	
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 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.)

---

Furthermore, the kind of expression you can specify for the address depends on the level at which you declare the pointer:

- At the global level, use a constant expression. See also [Global Standard Pointer Initializations](#).
- At the local or sublocal level, you can use any arithmetic expression.

## Global Standard Pointer Initializations

You can initialize global standard pointers by using constant expressions such as:

<b>Expression</b>	<b>Meaning</b>
<code>@identifier</code>	Accesses address of variable
<code>@identifier '&lt;&lt;'1</code>	Converts word address to byte address

Expression	Meaning
<code>@identifier &gt;&gt;1</code>	Converts byte address to word address
<code>@identifier[index]</code>	Accesses address of variable indicated by index
Standard function	Any that return a constant value, such as \$OFFSET

The following table shows the kinds of global variables to which you can apply the @ operator:

Variable	<code>@identifier?</code>
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

## When Pointers Receive Initial Values

Global simple pointers receive their initial values when you compile the source code. Local or sublocal simple pointers receive their initial values at each activation of the encompassing procedure or subprocedure.

## Examples of Simple Pointer Declarations

1. This example declares but does not initialize a simple pointer:

```
INT(32) .ptr;           !Declare simple pointer
```

2. This example declares a simple pointer and initializes it with the address of the last element in an indirect array:

```
STRING .bytes[0:3];           !Declare indirect array
STRING .s_ptr := @bytes[3];   !Declare simple pointer
                           !initialized with address
                           !of BYTES[3]
```

3. This example declares a STRING simple pointer and initializes it with the converted byte address of an INT array. This allows byte access to the word-addressed array:

```
INT .a[0:39];           !Declare INT array
STRING .ptr := @a[0] ' << ' 1;   !Declare STRING simple
                                !pointer initialized with
                                !byte address of A[0]
```

4. This example declares an array and simple pointers at the local or sublocal level and initializes the pointers with values derived from the array:

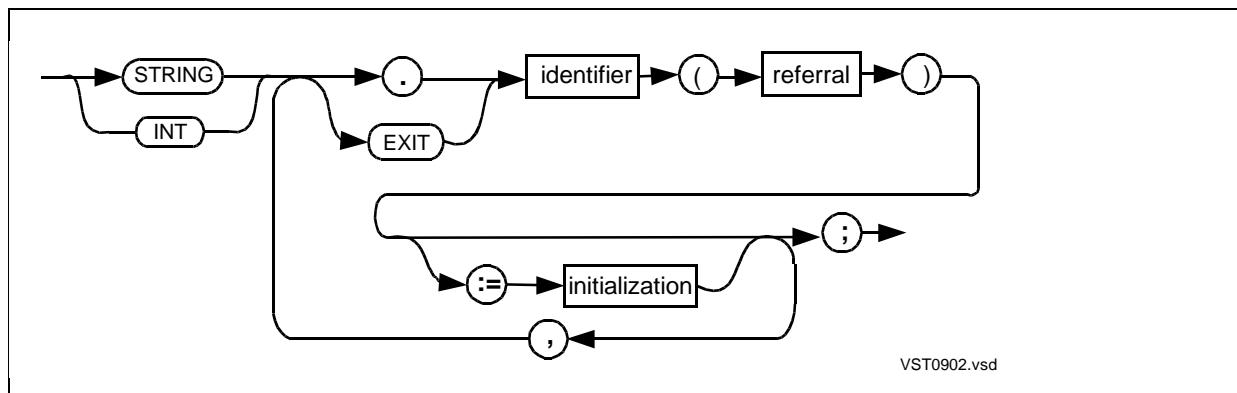
```
INT a[0:1] := [%100000, %110000];  !Declare array
INT .int_ptr1 := a[0];           !Declare simple pointer
                                !initialized with %100000
INT .int_ptr2 := a[1];           !Declare simple pointer
                                !initialized with %110000
```

5. This example declares an array and an extended simple pointer at the local or sublocal level. The pointer is initialized with the byte address of an indexed element, assuming the object being indexed has a 32-bit address:

```
INT .EXT x[-100:100];          !Declare array
INT .EXT x_ptr := @x[-5];       !Declare extended simple
                                !pointer initialized with
                                !32-bit byte address of
                                !X[-5]
```

## Structure Pointer Declaration

The structure pointer declaration associates a previously declared structure with the memory location to which the structure pointer points. You access data in the associated structure by referencing the qualified structure pointer identifier.



STRING

is the STRING attribute.

INT

is the INT attribute.

. (period)

is the standard indirection symbol and denotes a standard (16-bit) pointer.

.EXT

is the extended indirection symbol and denotes an extended (32-bit) pointer.

identifier

is the identifier of the structure pointer.

referral

is the identifier of a previously declared structure or structure pointer that provides the structure layout for this structure pointer.

initialization

is an expression that represents a memory address, as described in [Structure Pointer Initializations](#).

## Usage Considerations

Extended pointer declarations should precede 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].

## Structure Pointer Initializations

The addressing mode and STRING or INT attribute determine the kind of addresses a structure pointer can contain, as described in [Table 9-3](#) on page 9-7.

**Table 9-3. Addresses in Structure Pointers**

<b>Addressing Mode</b>	<b>STRING or INT Attribute</b>	<b>Kind of Addresses</b>
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 the automatic extended data segment
Extended	INT **	32-bit byte address of any structure 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.

Furthermore, the kind of expression you can specify for the address depends on the level at which you declare the pointer:

- At the global level, use a constant expression. See also [Global Standard Pointer Initializations](#) on page 9-3.
- At the local or sublocal level, you can use any arithmetic expression.

If the expression is the address of a structure with an index, the structure pointer points to a particular occurrence of the structure. If the expression is the address of an array, with or without an index, you impose the structure on top of the array.

Global structure pointers receive their initial values when you compile the source code. Local and sublocal structure pointers receive their initial values each time the procedure or subprocedure is activated.

## Examples of Structure Pointer Declarations

1. This example uses the \$OFFSET standard function to include the address of a structure field in the expression of a global initialization:

```
STRUCT t (*);           !Template structure
BEGIN
  INT k;
END;
```

```

STRUCT .st;           !Definition structure
BEGIN
INT j;
STRUCT ss (t);
END;

INT .ip := @st +' $OFFSET (st.j) '>>' 1;
!Simple pointer

INT .stp (t) := @st +' $OFFSET (st.ss) '>>' 1;
!INT structure pointer

STRING .sstp (t) := @st '<<' 1 '+' $OFFSET (st.ss);
!STRING structure pointer

```

2. 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

```

3. This example declares a structure and a structure pointer at the local level. The structure pointer is initialized to point to the second occurrence of the structure:

```

PROC my_proc MAIN;
BEGIN
STRUCT my_struct[0:2];    !Structure
BEGIN
INT array[0:7];
END;

INT .struct_ptr (my_struct) := @my_struct[1];
!Structure pointer contains
! address of MY_STRUCT[1]
END;

```

4. This example initializes 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;
                           !Structure pointer contains
                           !address of substructure
my_name ':= [ "Sue Law" ];

```

5. This example declares an array, a structure, and a structure pointer at the local level. The structure pointer refers to the structure but is initialized to point to the array, thus imposing the structure on the array. You can now refer to the array in two ways:

```

PROC a_proc MAIN;
BEGIN
INT array[0:7];           !Array
STRUCT a_struct ( * );     !Structure
BEGIN
INT var;
INT buffer1[0:3];
STRING buffer2[0:4];
END;

INT .struct_ptr (a_struct) := @array;
END;                      !Structure pointer contains
                           !address of array

```



# **10 Equivalenced Variables**

Equivalencing lets you declare more than one identifier and description for a location in a primary storage area. Equivalenced variables that represent the same location can have different data types and byte-addressing and word-addressing attributes. You can, for example, reference an INT(32) variable as two separate words or four separate bytes.

This section describes the syntax for declaring:

- Equivalenced variables—Variables equivalenced to a previous variable
- Base-address equivalenced variables—Variables equivalenced to a global, local, or top-of-stack base address

Other equivalencing information appears in the TAL manual set as follows:

Information	Manual	Section/ Appendix
Accessing equivalenced variables	<i>TAL Programmer's Guide</i>	10, "Using Equivalenced Variables"
Equivalencing to indexed or offset variables	<i>TAL Programmer's Guide</i>	10, "Using Equivalenced Variables"
Redefinitions (equivalencing within structures)	<i>TAL Programmer's Guide</i> <i>TAL Reference Manual</i>	8, "Using Structures" <a href="#">Section 8, Structures</a> (syntax)
'SG'-equivalencing	<i>TAL Reference Manual</i>	<a href="#">Section 15, Privileged Procedures</a>

## **Equivalenced Variable Declarations**

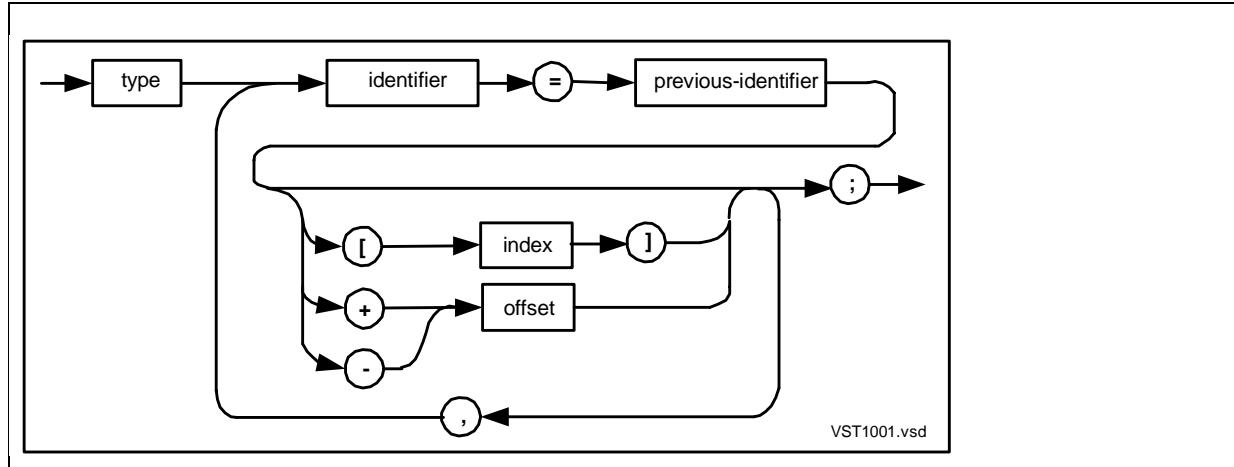
The variables you can equivalence to another variable are listed in [Table 10-1](#). You can equivalence any variable in the first column to any variable in the second column. (You cannot equivalence an array to another variable.)

**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

# Equivalenced Simple Variable

An equivalenced simple variable declaration associates a new simple variable with a previously declared variable.



**type**

is any data type except UNSIGNED.

**identifier**

is the identifier of a simple variable to be made equivalent to *previous-identifier*.

**previous-identifier**

is the identifier of a previously declared simple variable, array, simple pointer, structure, structure pointer, or equivalenced variable.

**index**

is an INT constant that specifies an element offset from *previous-identifier*, which must be a direct variable. The data type of *previous-identifier* dictates the element size. The location represented by *index* must begin on a word boundary.

**offset**

is an INT constant that specifies a word offset from *previous-identifier*, which can be a direct or indirect variable. If *previous-identifier* is indirect, the offset is from the location of the pointer, not from the location of the data pointed to.

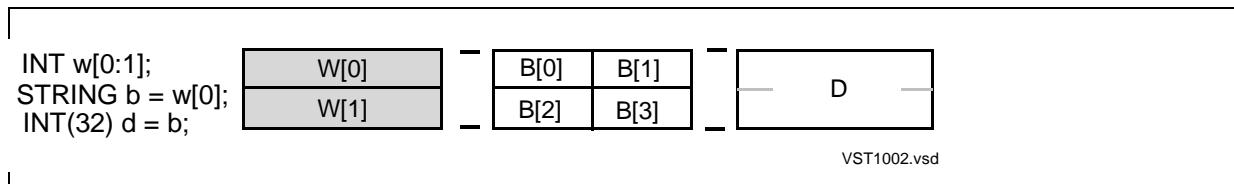
## Usage Consideration

Avoid equivalencing a simple variable to an indirect array or structure. If you do so, the simple variable is made equivalent to the location of the implicit pointer, not the location of the data pointed to.

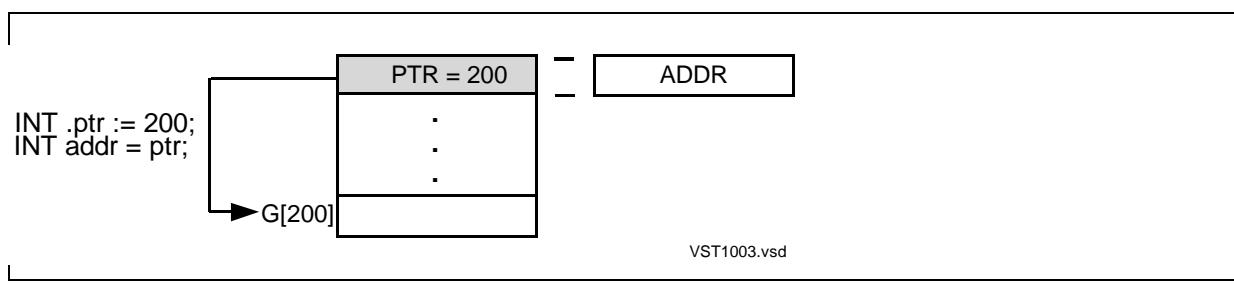
For portability to future software platforms, declare equivalenced variables that fit entirely within the previous variable.

## Examples of Equivalenced Simple Variable Declarations

1. This example equivalences a STRING variable and an INT(32) variable to an INT array:

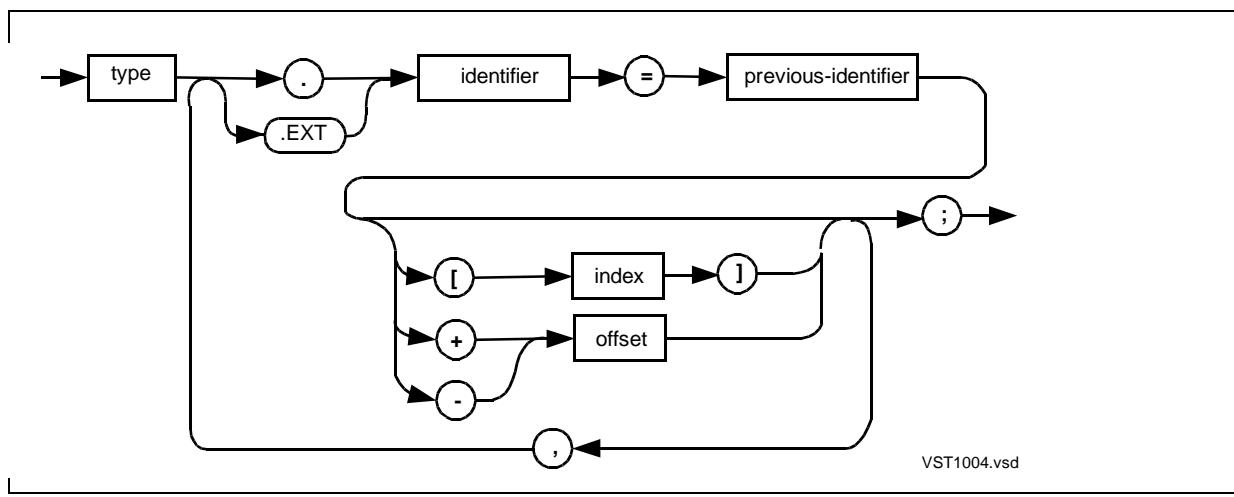


2. This example equivalences a simple variable to a simple pointer. The simple variable is equivalenced to the location occupied by the simple pointer, not to the location whose address is stored in the simple pointer:



## Equivalenced Simple Pointer

An equivalenced simple pointer declaration associates a new simple pointer with a previously declared variable.



**type**

is any data type except UNSIGNED. The data type determines how much data the simple pointer can access at a time (byte, word, doubleword, or quadrupletword).

**. (period)**

is the standard indirection symbol and denotes a standard (16-bit) pointer.

**.EXT**

is the extended indirection symbol and denotes an extended (32-bit) pointer.

**identifier**

is the identifier of a simple pointer to be made equivalent to *previous-identifier*.

**previous-identifier**

is the identifier of a previously declared simple variable, array, simple pointer, structure, structure pointer, or equivalenced variable.

**index**

is an INT constant that specifies an element offset from *previous-identifier*, which must be a direct variable. The data type of *previous-identifier* dictates the element size. The location represented by *index* must begin on a word boundary.

**offset**

is an INT constant that specifies a word offset from *previous-identifier*, which can be a direct or indirect variable. If *previous-identifier* is indirect, the offset is from the location of the pointer, not from the location of the data pointed to.

## Usage Consideration

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:

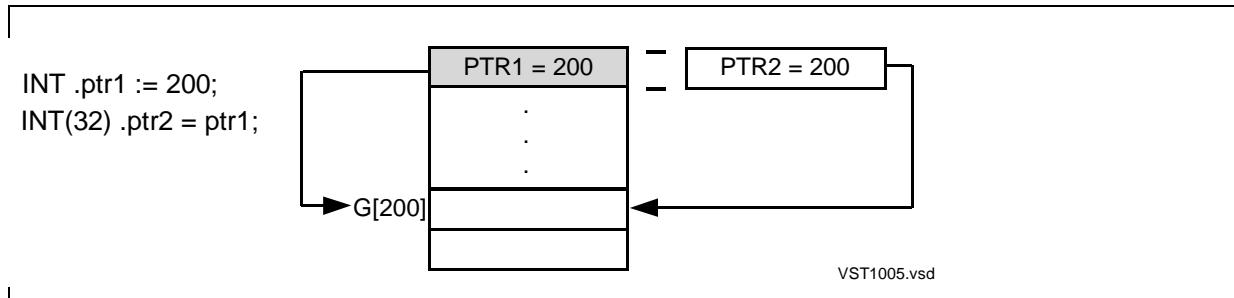
- A standard byte address
- A standard word address
- An extended address

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.

For portability to future software platforms, declare equivalenced variables that fit entirely within the previous variable.

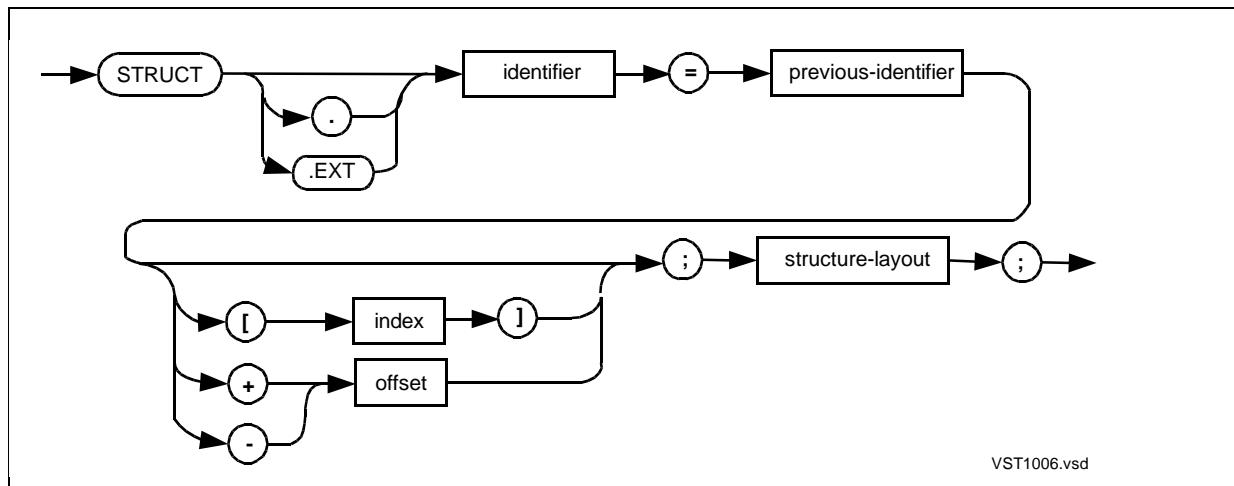
## Example of Equivalenced Simple Pointer Declaration

This example declares an INT(32) simple pointer equivalent to an INT simple pointer. Both contain a word address:



## Equivalenced Definition Structure

An equivalenced definition structure declaration associates a new structure with a previously declared variable.



- . (period)

is the standard indirection symbol and denotes 16-bit indirect addressing. An absence of any indirection symbol (. or .EXT) denotes 16-bit direct addressing.

.EXT

is the extended indirection symbol and denotes 32-bit indirect addressing. An absence of any indirection symbol (. or .EXT) denotes 16-bit direct addressing.

identifier

is the identifier of a definition structure to be made equivalent to *previous-identifier*.

**previous-identifier**

is the identifier of a previously declared simple variable, array, simple pointer, structure, structure pointer, or equivalenced variable.

**index**

is an INT constant that specifies an element offset from *previous-identifier*, which must be a direct variable. The data type of *previous-identifier* dictates the element size. The location represented by *index* must begin on a word boundary.

**offset**

is an INT constant that specifies a word offset from *previous-identifier*, which can be a direct or indirect variable. If *previous-identifier* is indirect, the offset is from the location of the pointer, not from the location of the data pointed to.

**structure-layout**

is a BEGIN-END construct that contains structure item declarations as described in [Section 8, Structures](#).

## Usage Considerations

If the new structure is to occupy the same location as the previous variable, their addressing modes should match. Thus, you can declare a direct or indirect structure equivalent to the following previous variables:

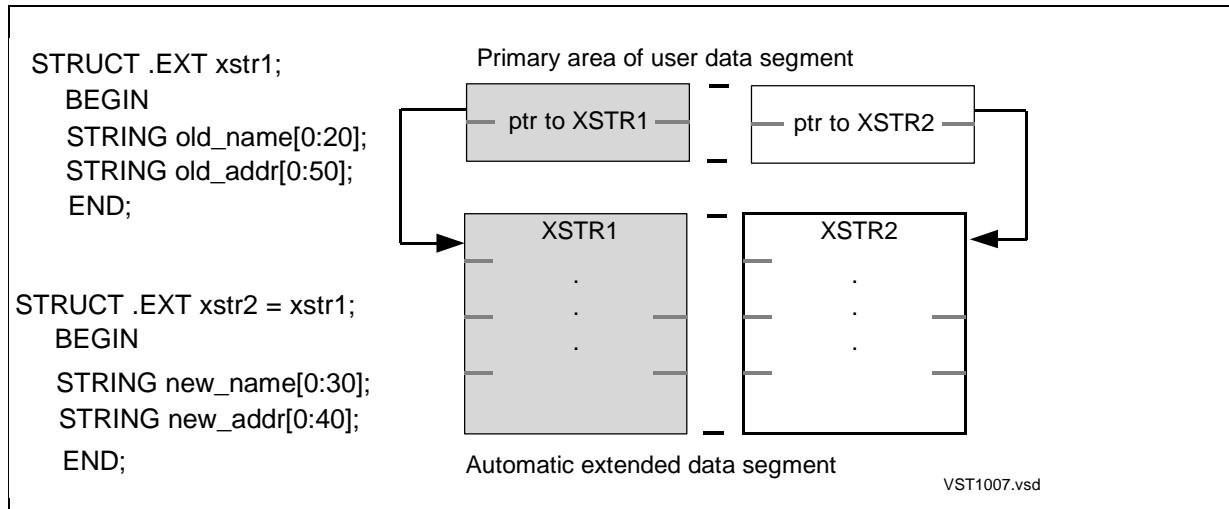
New Structure	Previous Variable
Direct Structure	Simple Variable Direct Structure Direct Array
Standard Indirect Structure	Standard Indirect Structure Standard Indirect Array Standard Structure Pointer
Extended Indirect Structure	Extended Indirect Structure Extended Indirect Array Extended Structure Pointer

If the previous variable is a structure pointer, the new structure is really a pointer, as described in the *TAL Programmer's Guide*.

For portability to future software platforms, declare equivalenced variables that fit entirely within the previous variable.

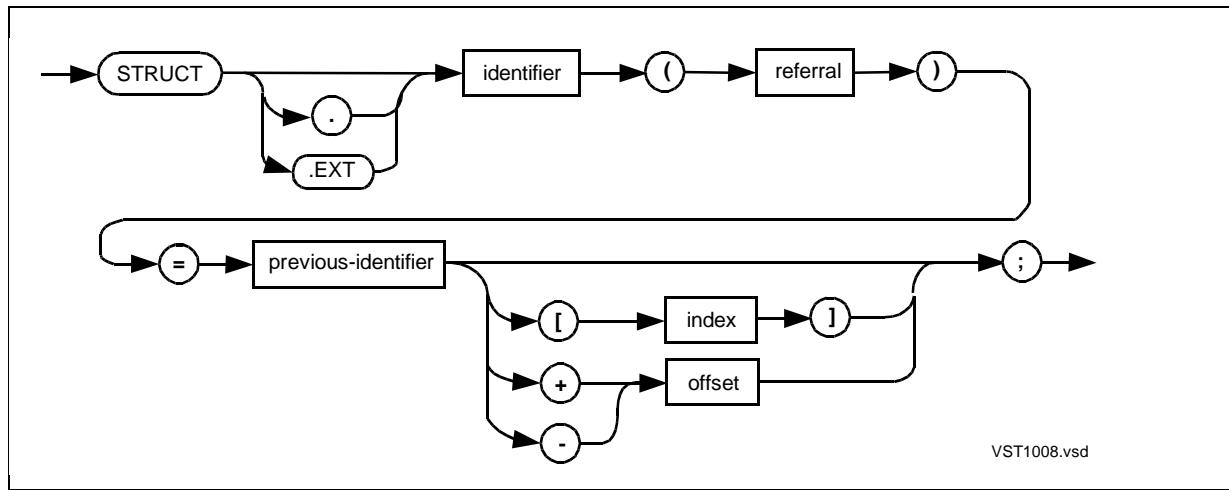
## Example of Equivalenced Definition Structure Declaration

The following example declares an extended indirect definition structure equivalent to a previously declared extended indirect structure:



## Equivalenced Referral Structure

An equivalenced referral structure declaration associates a new structure with a previously declared variable.



- . (period)

is the standard indirection symbol and denotes 16-bit indirect addressing. An absence of any indirection symbol (.) or (.EXT) denotes 16-bit direct addressing.

.EXT

is the extended indirection symbol and denotes 32-bit indirect addressing. An absence of any indirection symbol (.) or (.EXT) denotes 16-bit direct addressing.

`identifier`

is the identifier of a referral structure to be made equivalent to *previous-identifier*.

`referral`

is the identifier of a previously declared structure or structure pointer that provides the structure layout for *identifier*.

`previous-identifier`

is the identifier of a previously declared simple variable, array, simple pointer, structure, structure pointer, or equivalenced variable.

`index`

is an INT constant that specifies an element offset from *previous-identifier*, which must be a direct variable. The data type of *previous-identifier* dictates the element size. The location represented by *index* must begin on a word boundary.

`offset`

is an INT constant that specifies a word offset from *previous-identifier*, which can be a direct or indirect variable. If *previous-identifier* is indirect, the offset is from the location of the pointer, not from the location of the data pointed to.

## Usage Considerations

If the new structure is to occupy the same location as the previous variable, their addressing modes should match. Thus, you can declare a direct or indirect structure equivalent to the following previous variables:

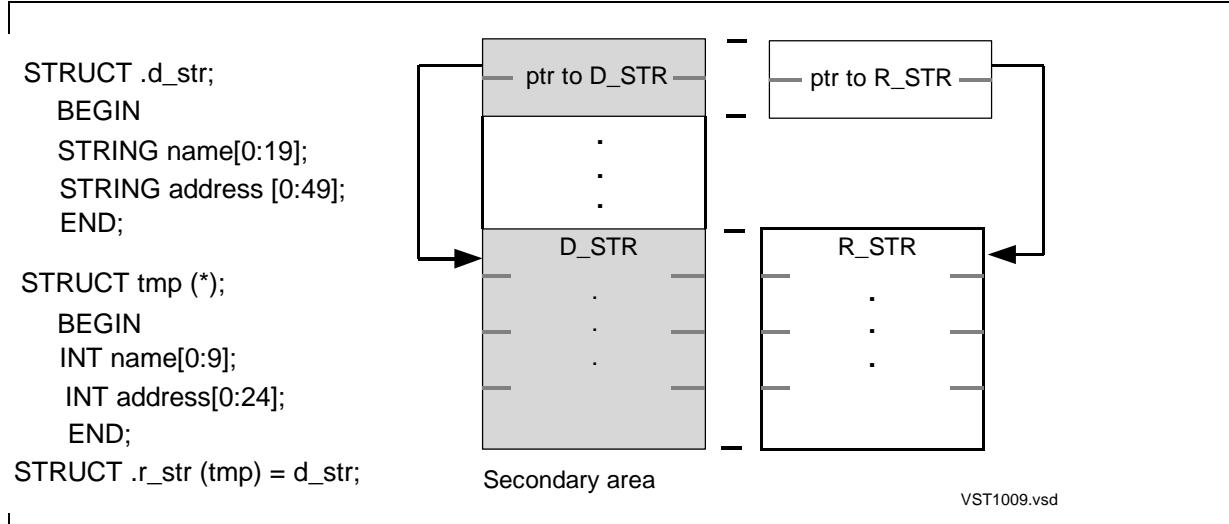
New Structure	Previous Variable
Direct Structure	Simple Variable Direct Structure Direct Array
Standard Indirect Structure	Standard Indirect Structure Standard Indirect Array Standard Structure Pointer
Extended Indirect Structure	Extended Indirect Structure Extended Indirect Array Extended Structure Pointer

If the previous variable is a structure pointer, the new structure is really a pointer, as described in the *TAL Programmer's Guide*.

For portability to future software platforms, declare equivalenced variables that fit entirely within the previous variable.

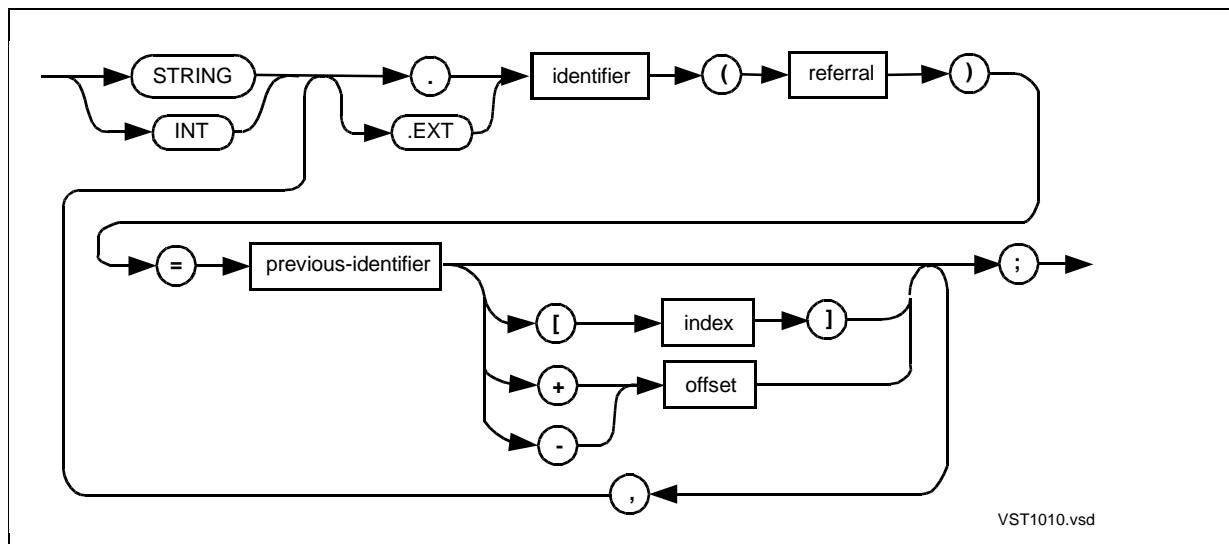
## Example of Equivalenced Referral Structure Declaration

The following example declares a referral structure equivalent to a previously declared definition structure:



## Equivalenced Structure Pointer

An equivalenced structure pointer declaration associates a new structure pointer with a previously declared variable.



STRING

is the STRING attribute.

INT

is the INT attribute.

. (period)

is the standard indirection symbol and denotes a standard (16-bit) pointer.

.EXT

is the extended indirection symbol and denotes an extended (32-bit) pointer.

*identifier*

is the identifier of a structure pointer to be made equivalent to *previous-identifier*.

*referral*

is the identifier of a previously declared structure or structure pointer that provides the structure layout for *identifier*.

*previous-identifier*

is the identifier of a previously declared simple variable, direct array element, simple pointer, structure, structure pointer, or equivalenced variable.

*index*

is an INT constant that specifies an element offset from *previous-identifier*, which must be a direct variable. The data type of *previous-identifier* dictates the element size. The location represented by *index* must begin on a word boundary.

*offset*

is an INT constant that specifies a word offset from *previous-identifier*, which can be a direct or indirect variable. If *previous-identifier* is indirect, the offset is from the location of the pointer, not from the location of the data pointed to.

## Usage Considerations

The STRING or INT attribute and the addressing symbol determine the kind of addresses a structure pointer can contain, as described in [Table 9-3](#) on page 9-7.

You can declare a structure pointer equivalent to the following previous variables.

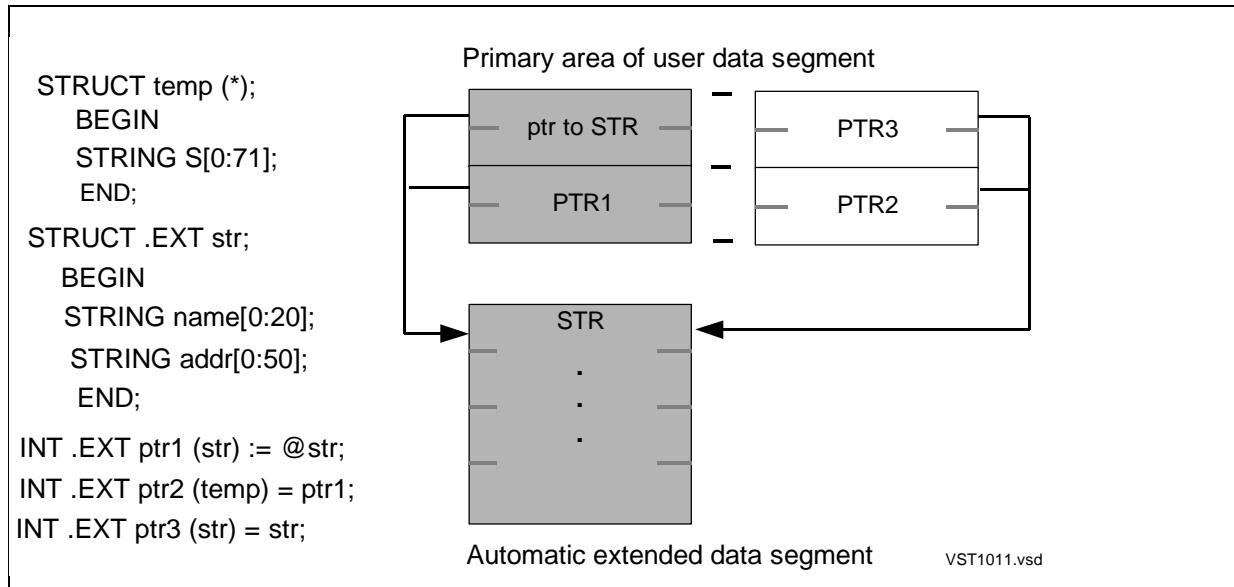
New Structure Pointer	Previous Variable
Standard Structure Pointer	Standard Indirect Structure Standard Indirect Array Standard Structure Pointer
Extended structure pointer	Extended Indirect Structure Extended Indirect Array Extended Structure Pointer

Also, the new structure pointer and the previous pointer must both contain byte addresses or both contain word addresses; otherwise, the pointers point to different locations.

For portability to future software platforms, declare equivalenced variables that fit entirely within the previous variable.

## Example of Equivalenced Structure Pointer Declaration

The following example declares structure pointers equivalent to another structure pointer and to an indirect structure:

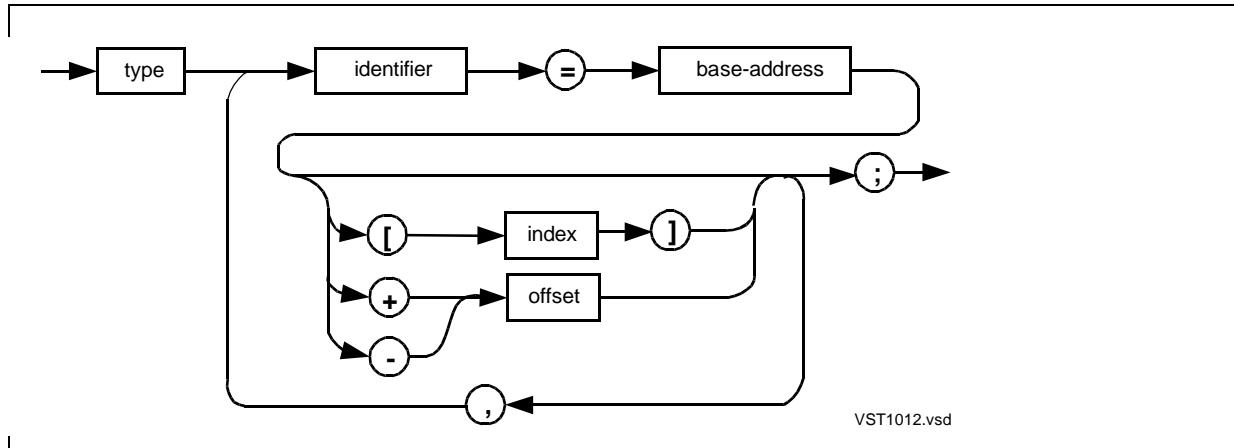


## Base-Address Equivalenced Variable Declarations

Base-address equivalencing lets you declare variables relative to global, local, or top-of-stack base addresses. You can declare base-address equivalenced simple variables, simple pointers, structure pointers, and structures.

# Base-Address Equivalenced Simple Variable

A base-address equivalenced simple variable declaration associates a new simple variable with a global, local, or top-of-stack base address.



**type**

is any data type except UNSIGNED.

**variable**

is the identifier of a simple variable to be made equivalent to *base-address*.

**base-address**

is one of the following base address symbols:

'G' Denotes global addressing relative to G[0]

'L' Denotes local addressing relative to L[0]

'S' Denotes top-of-stack addressing relative to S[0]

**index and offset**

are equivalent INT values giving a location in the following ranges:

0 through 255 For 'G' addressing

-255 through 127 For 'L' addressing

-31 through 0 For 'S' addressing

## Considerations

If you use the Common Run-Time Environment (CRE), locations G[0] and G[1] are not available for your data (as described in the *TAL Programmer's Guide*). References to 'G', 'L', or 'S' are not portable to future software platforms.

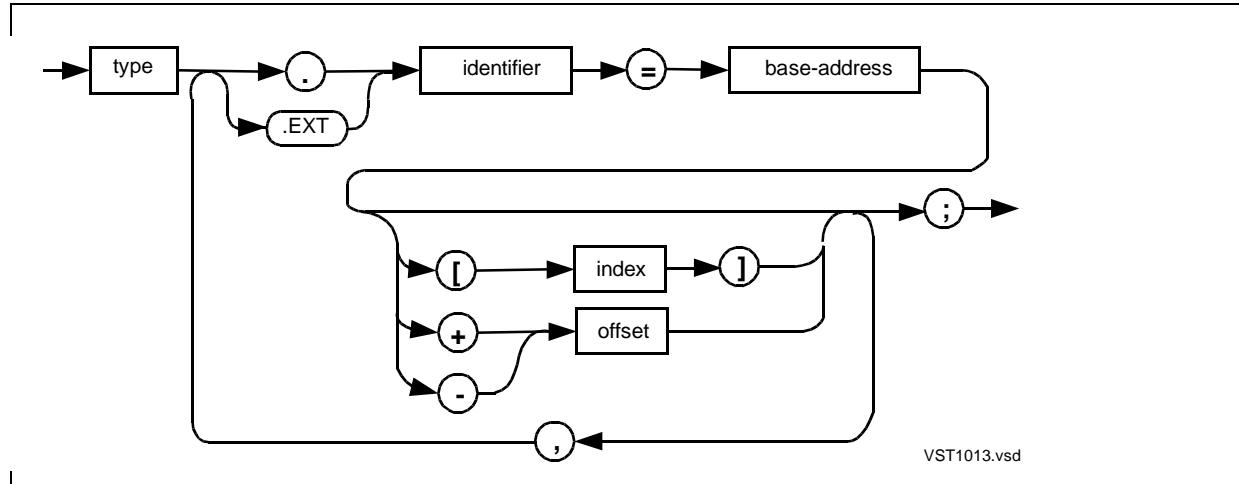
## Example of Base-Address Equivalenced Simple Variable Declaration

The following example declares an INT simple variable equivalent to an 'L'-relative base address:

```
INT var1 = 'L'[5];
INT(32) var2 = 'G'[10];
```

## Base-Address Equivalenced Simple Pointer

A base-address equivalenced simple pointer declaration associates a new simple pointer with a global, local, or top-of-stack base address.



**type**

is any data type except UNSIGNED.

- . (period)

is the standard indirection symbol and denotes a standard (16-bit) pointer.

.EXT

is the extended indirection symbol and denotes an extended (32-bit) pointer.

**identifier**

is the identifier of a simple pointer to be made equivalent to *base-address*.

base-address

is one of the following base address symbols:

- 'G' Denotes global addressing relative to G[0]
- 'L' Denotes local addressing relative to L[0]
- 'S' Denotes top-of-stack addressing relative to S[0]

index and offset

are equivalent INT values giving a location in the following ranges:

- 0 through 255 For 'G' addressing
- 255 through 127 For 'L' addressing
- 31 through 0 For 'S' addressing

## Usage Considerations

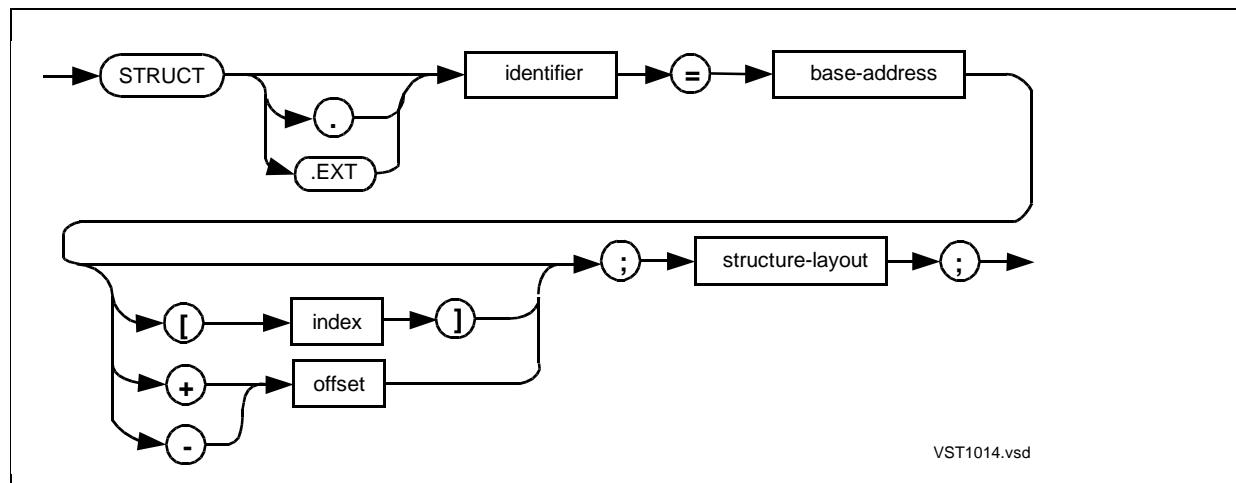
The data type determines how much data the simple pointer can access at a time—byte, word, doubleword, or quadrupleword.

If you use the CRE, locations G[0] and G[1] are not available for your data.

References to 'G', 'L', or 'S' are not portable to future software platforms.

## Base-Address Equivalenced Definition Structure

A base-address equivalenced definition structure declaration associates a new structure with a global, local, or top-of-stack base address.



- . (period)

is the standard indirection symbol and denotes 16-bit indirect addressing. An absence of any indirection symbol (. or .EXT) denotes 16-bit direct addressing.

- .EXT

is the extended indirection symbol and denotes 32-bit indirect addressing. An absence of any indirection symbol (. or .EXT) denotes 16-bit direct addressing.

- identifier

is the identifier of a definition structure to be made equivalent to *base-address*.

- base-address

is one of the following base address symbols:

'G' Denotes global addressing relative to G[0]

'L' Denotes local addressing relative to L[0]

'S' Denotes top-of-stack addressing relative to S[0]

- index and offset

are equivalent INT values giving a location in the following ranges:

0 through 255 For 'G' addressing

-255 through 127 For 'L' addressing

-31 through 0 For 'S' addressing

- structure-layout

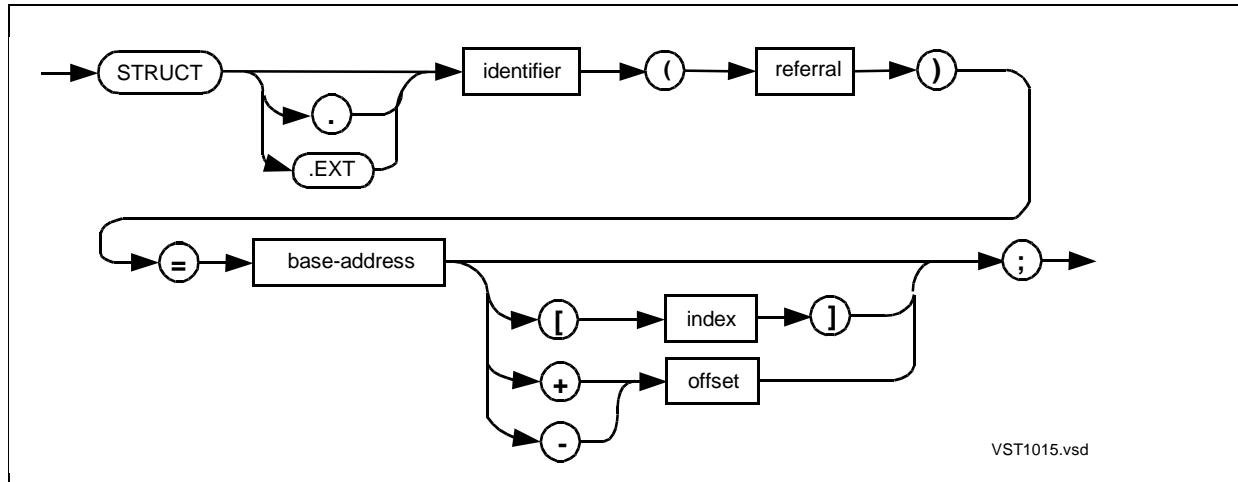
is a BEGIN-END construct that contains declarations as described in [Section 8, Structures](#).

## Usage Considerations

If you use the CRE, locations G[0] and G[1] are not available for your data. References to 'G', 'L', or 'S' are not portable to future software platforms.

# Base-Address Equivalenced Referral Structure

A base-address equivalenced referral structure declaration associates a new structure with a global, local, or top-of-stack base address.



- . (period) is the standard indirection symbol and denotes 16-bit indirect addressing. An absence of any indirection symbol (. or .EXT) denotes 16-bit direct addressing.

• EXT

is the extended indirection symbol and denotes 32-bit indirect addressing. An absence of any indirection symbol (. or .EXT) denotes 16-bit direct addressing.

## identifier

is the identifier of a referral structure to be made equivalent to *base-address*.

## referral

is the identifier of a previously declared structure or structure pointer that provides the structure layout for *identifier*.

## base-address

is one of the following base address symbols:

'G'	Denotes global addressing relative to G[0]
'L'	Denotes local addressing relative to L[0]
'S'	Denotes top-of-stack addressing relative to S[0]

index and offset

are equivalent INT values giving a location in the following ranges:

- |                  |                    |
|------------------|--------------------|
| 0 through 255    | For 'G' addressing |
| -255 through 127 | For 'L' addressing |
| -31 through 0    | For 'S' addressing |

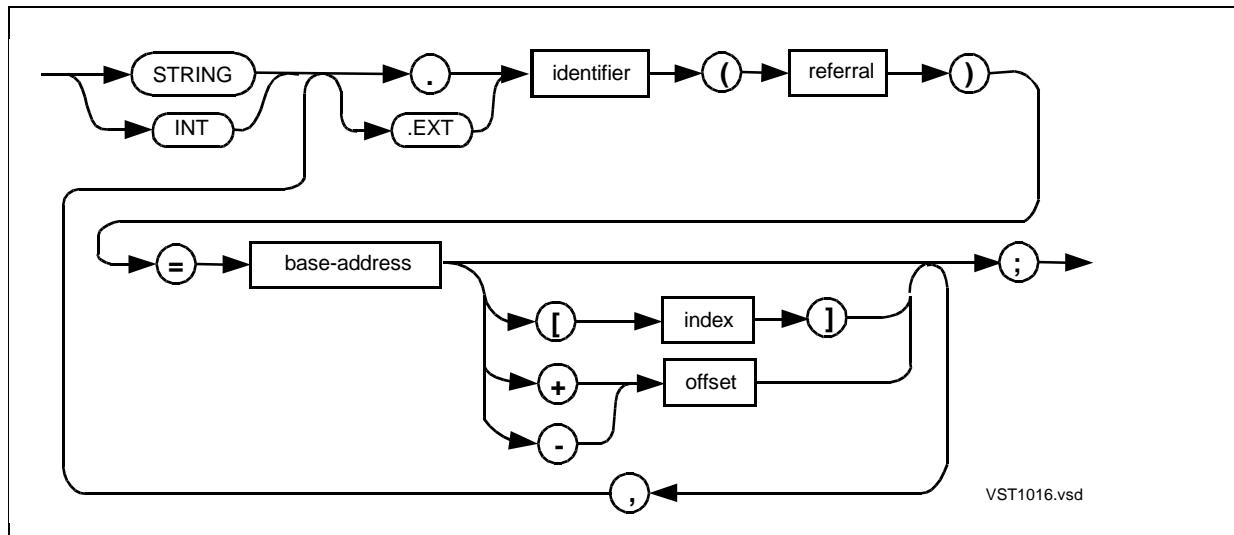
## Usage Considerations

If you use the CRE, locations G[0] and G[1] are not available for your data.

References to 'G', 'L', or 'S' are not portable to future software platforms.

## Base-Address Equivalenced Structure Pointer

A base-address equivalenced structure pointer declaration associates a new structure pointer with a global, local, or top-of-stack base address.



STRING

is the STRING attribute.

INT

is the INT attribute.

. (period)

is the standard indirection symbol and denotes a standard (16-bit) pointer.

.EXT

is the extended indirection symbol and denotes an extended (32-bit) pointer.

**identifier**

is the identifier of a structure pointer to be made equivalent to *base-address*.

**referral**

is the identifier of a previously declared structure or structure pointer that provides the structure layout for *identifier*.

**base-address**

is one of the following base address symbols:

- |     |  |
|-----|--|
| 'G' | Denotes global addressing relative to G[0]       |
| 'L' | Denotes local addressing relative to L[0]        |
| 'S' | Denotes top-of-stack addressing relative to S[0] |

**index and offset**

are equivalent INT values giving a location in the following ranges:

- |                  |                    |
|------------------|--------------------|
| 0 through 255    | For 'G' addressing |
| -255 through 127 | For 'L' addressing |
| -31 through 0    | For 'S' addressing |

## Usage Considerations

The STRING or INT attribute and addressing symbol determine the kind of addresses a structure pointer can contain, as described in [Table 9-3](#) on page 9-7.

If you use the CRE, locations G[0] and G[1] are not available for your data.

References to 'G', 'L', or 'S' are not portable to future software platforms.

# 11 NAMES and BLOCKs

Your input to a compilation session is a single source file. The source file contains declarations, statements, and compiler directives that you can compile into an object file.

The source file and any other source code that is read in by SOURCE directives together compose a **compilation unit**.

The output from a compilation session is an executable or bindable object file that consists of relocatable code and data blocks. You can compile separate object files and then use Binder to bind the object files into a new executable or bindable object file, called the target file.

When you bind object files together, Binder might have to relocate global data. In your compilation unit, you can use the BLOCK declaration to group global data declarations into relocatable data blocks. In the BLOCK declaration, you can specify whether the data block is private to a compilation unit or shareable with other compilation units in a program.

If you use a BLOCK declaration in a compilation unit, you must also use a NAME declaration to give the compilation unit an identifier.

This section describes the syntax for:

- The NAME declaration
- The BLOCK declaration

For more information on the related topics, see Section 14, “Compiling Programs,” in the *TAL Programmer’s Guide*:

- Compiling with relocatable data blocks
- Allocation of global data blocks by the compiler
- Sharing global data blocks

## NAME Declaration

The NAME declaration assigns an identifier to a compilation unit and to its private data block if it has one.



identifier

is the identifier of the compilation unit.

## Usage Considerations

A compilation unit that contains a NAME declaration as its first declaration is called a named compilation unit. If the compilation unit contains no BLOCK declarations, the NAME declaration is optional.

If a compilation unit contains a BLOCK declaration, the NAME declaration must be the first declaration in the compilation unit. NAME is a reserved keyword only within the first declaration; you can use the term NAME elsewhere as an identifier.

The identifier declared in a NAME declaration is accessible as follows:

- If the current compilation unit has a private data block, the NAME identifier has global scope among all compilation units in the target file. No other compilation unit in the target file can use the same identifier at the global level.
- If the current compilation unit has no private data block, the identifier has global scope within the current compilation unit only.

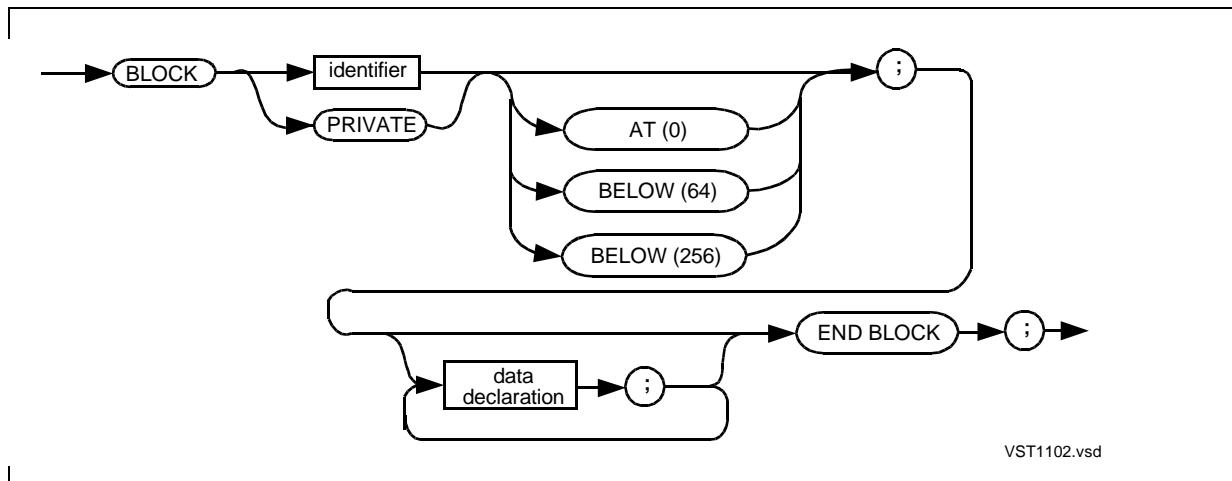
## Example of NAME Declaration

This example assigns the identifier CALC\_MOD to a compilation unit and to its private data block:

```
NAME calc_mod;
```

## BLOCK Declaration

The BLOCK declaration lets you group global data declarations into a named or private relocatable global data block.



**identifier**

is the identifier of the data block. identifier must be unique among all BLOCK and NAME declarations in the target file. A named data block is accessible to other compilation units in the target file.

**PRIVATE**

specifies a private global data block, which is accessible within the current compilation unit only.

**AT ( 0 )**

directs Binder to locate the block at location G[0]. (If you specify this option and run your program in the CRE, conflicts could arise.)

**BELOW ( 64 )**

directs Binder to locate the block below location G[64].

**BELOW ( 256 )**

directs Binder to locate the block below location G[256].

**data-declaration**

is a global declaration of a LITERAL, DEFINE, simple variable, array, simple pointer, structure pointer, structure, or equivalenced variable.

## Usage Considerations

The BLOCK declaration is optional. A compilation unit can contain any number of named (nonprivate) data blocks, but can contain only one private data block.

If you use the BLOCK declaration, the compilation unit must be named and the NAME declaration must be the first declaration in the compilation unit. In a named compilation unit, BLOCK and PRIVATE are reserved words.

The identifier of the BLOCK can be the same as the identifier of a variable declared within the same block. This feature allows a TAL module to share global data with a C module.

The private block, if any, inherits the identifier you specify in the NAME declaration for this compilation unit.

## BLOCK Location Options

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

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.

## Efficient Code

For extended pointers declared within BLOCK declarations that specify AT (0) or BELOW (64), the compiler generates efficient code using the XX instructions (LWXX, SWXX, LBXX, and SBXX). The INHIBITXX directive, which suppresses generation of the XX instructions, has no effect on such BLOCK declarations.

The INT32INDEX directive suppresses the generation of XX instructions regardless of the BLOCK declaration.

For information on the XX instructions, see the *System Description Manual* for your system.

## Examples of BLOCK Declarations

1. This example declares a private global data block, which is accessible only to the current compilation unit. The compiler gives this private block the identifier specified in the NAME declaration for the current compilation unit:

```
BLOCK PRIVATE;
    INT term_num;
    LITERAL msg_buf = 79;
END BLOCK;
```

2. This example declares a named global data block, DEFAULT\_VOL, which is accessible to other compilation units:

```
BLOCK default_vol;
    INT .vol_array [0:7],
    .out_array [0:34];
END BLOCK;
```

3. This example declares a named global data block to be located below G[64]:

```
BLOCK extended_indexed_stuff BELOW (64);
    INT .EXT sym_tab [0:32760],
    .EXT err_tab [0:16380];
END BLOCK;
```

# Coding Data Blocks

Here are guidelines for coding global data blocks:

- Place global declarations, if present, in the following order within a compilation unit:
  1. NAME declaration
  2. Unblocked global data declarations
  3. BLOCK declarations (named or private)
  4. PROC declarations
- Make sure that the target file contains no more than a total of 256 words of primary global blocks.
- Place variable declarations and any declarations that refer to those variables in the same block. For example, place the following declarations in the same block:

```
INT var;           !Variable declaration  
INT .ptr := @var; !Variable reference
```

- Make sure the length of any shared data block matches in all compilation units.

## Unblocked Declarations

Place all unblocked global declarations (those not contained in BLOCK declarations) before the first BLOCK declaration.

The compiler creates an implicit data block named #GLOBAL and places all unblocked declarations (except template structures) in #GLOBAL. A compilation unit can have only one #GLOBAL block.

The compiler creates an implicit data block for each unblocked template structure declaration and gives the block the name of the template structure 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.



# 12 Statements

Statements—also known as executable statements—perform operations in a program. They can modify the program’s data or control the program’s flow. [Table 12-1](#) summarizes statements.

**Table 12-1. Summary of Statements**

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

\* Not portable to future software systems.

## Using Semicolons

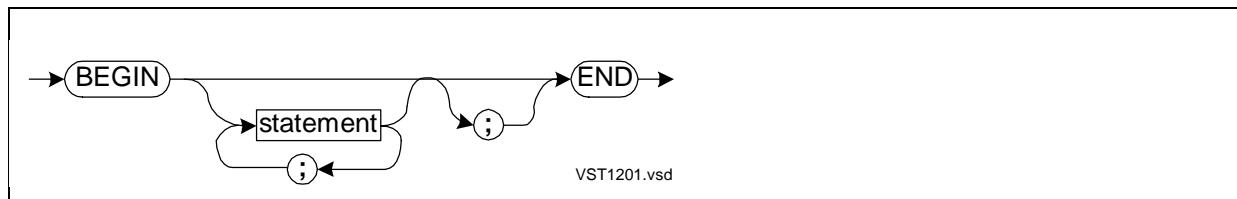
You use semicolons with statements as follows:

- A semicolon is required between successive statements.
- A semicolon is optional before an END keyword that terminates a compound statement.

- A semicolon must not immediately precede an ELSE or UNTIL keyword.
- A semicolon alone in place of a statement creates a null statement. The compiler generates no code for null statements. You can use a null statement wherever you can use a statement except immediately before an ELSE or UNTIL keyword.

## Compound Statements

A compound statement is a BEGIN-END construct that groups statements to form a single logical statement.



BEGIN

indicates the start of the compound statement.

statement

is a statement described in this section.

; (semicolon)

is a statement separator that is required between successive statements. A semicolon before an END that terminates a compound statement is optional and represents a null statement.

END

indicates the end of the compound statement.

## Usage Considerations

You can use compound statements anywhere you can use a single statement. You can nest them to any level in statements such as IF, DO, FOR, WHILE, or CASE.

## Examples of Compound Statements

1. This example shows a null compound statement. One use is in a CASE statement when a case has no action

```

BEGIN
END;

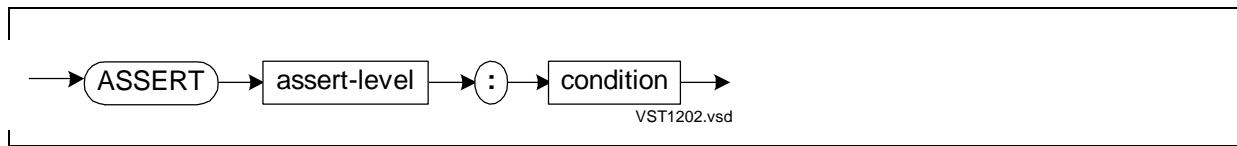
```

2. This example shows a compound statement

```
BEGIN
  a := b + c;
  d := %B101;
  f := d - e;
END;
```

## ASSERT Statement

The ASSERT statement conditionally invokes the procedure specified in an ASSERTION directive.



**assert-level**

is an integer in the range 0 through 32,767. If *assert-level* is equal to or higher than the *assertion-level* specified in the current ASSERTION directive and if *condition* is true, the procedure specified in the ASSERTION directive executes. If the *assert-level* is lower than the *assertion-level*, the procedure is not activated.

**condition**

is an expression that tests a program condition and yields a true or false result.

## Usage Considerations

The ASSERT statement is a debugging or error-handling tool. You use it 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\_ procedure or the C-series Debug procedure:

```
?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 that tests a condition. For example, the standard function \$CARRY returns true if the carry indicator is on and false if it is off:

```
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 the associated condition is true, the compiler invokes 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 an *assertion-level* that is higher than the highest *assert-level* you want to nullify:

```
?ASSERTION 11, PROCESS_DEBUG_
    !Assertion-level nullifies assert-level 10 and below
```

For more information, see [ASSERTION Directive](#) on page 16-14.

## Example of ASSERT Statement

This example invokes PROCESS\_DEBUG\_ whenever 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
```

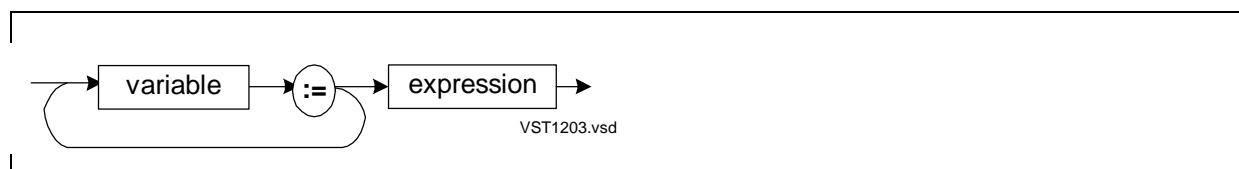
If you change the *assertion-level* in the ASSERTION directive to 15, you nullify the two ASSERT statements that specify *assert-level* 10 and the \$CARRY condition.

If you change the *assertion-level* to 30, you nullify all the ASSERT statements. If ASSERT statements that cover a particular condition all have the same *assert-level*, it is easier to nullify specific levels of ASSERT statements.

## Assignment Statement

The assignment statement assigns a value to a previously declared variable.

A bit-deposit assignment statement is a special form of the assignment statement; its description follows the assignment statement description.



`variable`

is the identifier of a simple variable, array element, simple pointer, structure pointer, or structure data item, with or without a bit deposit field and/or index. To update a pointer's content, prefix the pointer identifier with @.

`expression`

is either

- An arithmetic expression of the same data type as `variable`
- A conditional expression, which always has an INT result

`expression` can be a bit extraction value or an identifier prefixed with @ (the address of a variable). `expression` cannot be a constant list.

## Usage Considerations

In general, the data types of `variable` and `expression` must match. To convert the data type of `expression` to match the data type of `variable`, use a type-transfer function, described in [Section 14, Standard Functions](#).

## Assigning Numbers to FIXED Variables

When you assign a number to a FIXED variable, the system scales the value up or down to match the *fpoint* value. If the system scales the value down, you lose some precision depending on the amount of scaling; for example:

```
FIXED(2) a;  
a := 2.348F;                                !System scales value to 2.34F
```

If the ROUND directive is in effect, the system scales the value as needed, then rounds the value away from zero as follows:

```
(IF value < 0 THEN value - 5 ELSE value + 5) / 10
```

For example, if you assign 2.348F to a FIXED(2) variable, the ROUND directive scales the value by one digit and then rounds it to 2.35F.

## Assigning Character Strings

You can assign a character string to STRING, INT, or INT(32) variables.

If you assign a one-character string such as "A" to an INT simple variable, the system places the value in the right byte of a word and 0 in the left byte. (To store a character in the left byte, assign the character and a space, as in "A ").

If you assign a character string to a FIXED, REAL, or REAL(64) variable, the compiler issues error 32 (type incompatibility).

## Examples of Assignment Statements

- This example shows various assignment statements:

```

INT array[0:2];
INT .ptr;
REAL real_var;
FIXED fixed_var;
array[2] := 255;
@ptr := @array[1];
ptr := array[2];
real_var := 36.6E-3;
fixed_var := $FIX (real_var);

```

!Declare an array  
!Declare a simple pointer  
!Declare a REAL variable  
!Declare a FIXED variable  
!Assign a value to ARRAY[2]  
!Assign address of ARRAY[1]  
!to PTR  
!Assign value of ARRAY[2]  
!to ARRAY[1], to which PTR  
!points  
!Assign a REAL value to a  
!REAL variable  
!Convert a REAL value to  
!FIXED and assign it to a  
!FIXED variable

- Assignment statements can assign character strings but not constant lists, so in this example the three assignment statements together store the same value as the one constant list in the declaration:

```

INT .b[0:2] := [ "ABCDEF" ];
b[0] := "AB";
b[1] := "CD";
b[2] := "EF";

```

!Declare and initialize  
!with constant list  
!Assignment statements  
!cannot use constant lists

- In this example, the first assignment statement (which contains assignment expressions) is equivalent to the three separate assignments that follow it:

```

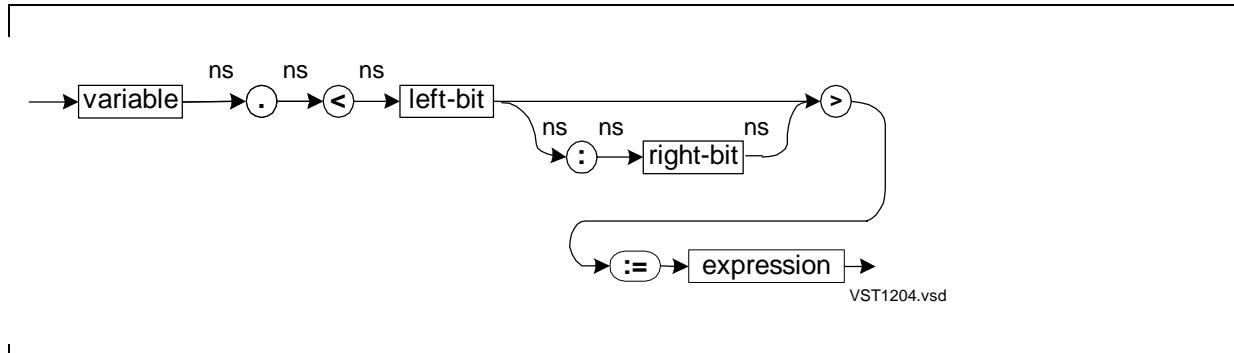
INT int1;
INT int2;
INT int3;
INT var := 16;
int1 := int2 := int3 := var;
int1 := var;
int2 := var;
int3 := var;

```

!Declarations  
!Assignment that contains  
!assignment expressions  
!Separate assignments

# Bit-D eposit Assignment Statement

The bit deposit form of the assignment statement lets you assign a value to an individual bit or to a group of sequential bits.



**variable**

is the identifier of a STRING or INT variable, but not an UNSIGNED(1–16) variable. *variable* can be the identifier of a simple variable, array element, or simple pointer (inside or outside a structure).

**left-bit**

is an INT constant that specifies the leftmost bit of the bit deposit field.

For STRING variables, specify a bit number in the range 8 through 15. Bit 8 is the leftmost bit in a STRING variable; bit 15 is the rightmost bit.

**right-bit**

is an INT constant specifying the rightmost bit of the bit deposit field. *right-bit* must be equal to or greater than *left-bit*.

For STRING variables, specify a bit number in the range 8 through 15. Bit 8 is the leftmost bit in a STRING variable; bit 15 is the rightmost bit.

**expression**

is an INT arithmetic or conditional expression, with or without a bit field specification.

## Usage Considerations

The bit deposit field is on the left side of the assignment operator (:=). The bit deposit assignment changes only the bit deposit field. If the value on the right side has more bits than the bit deposit field, the system ignores the excess high-order bits when making the assignment.

Specify the variable/bit-field construct with no intervening spaces. For example:

```
myvar.<0:5>
```

Do not use bit deposit fields to pack data. Instead, declare an UNSIGNED variable and specify the appropriate number of bits in the bit field.

Bit deposits are not portable to future software platforms. Where possible, isolate bit deposits in a routine that can be modified in the future.

## Examples of Bit Deposit Assignments

1. In this example, the bit deposit assignment sets bits 3 through 7 of the word designated by X:

```
INT x;
x.<3:7> := %B11111;
```

2. This example replaces bits <10> and <11> with zeros:

```
INT old := -1;           !OLD = %b11111111111111
old.<10:11> := 0;       !OLD = %b1111111111001111
```

3. This example sets bit <8>, the leftmost bit of STRNG, to 0:

```
STRING strng := -1;      !STRNG = %b1111111
strng.<8> := 0;          !STRNG = %b0111111
```

4. In this example, the value %577 is too large to fit in bits <7:12> of VAR. The system truncates %577 to %77 before performing the bit deposit:

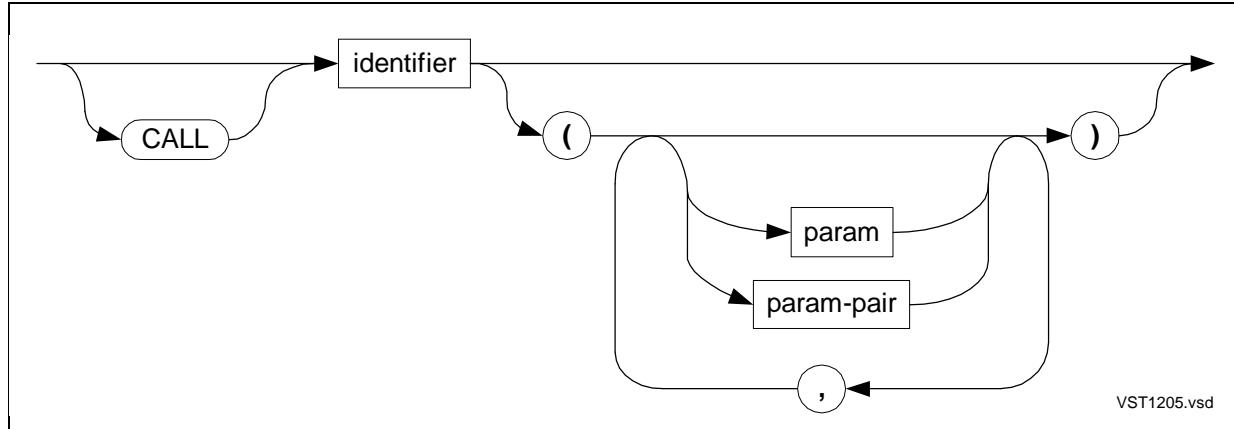
```
INT var := %125252;      !VAR = %b1010101010101010
var.<7:12> := %577;      !%77 = %b000000010111111
                           !VAR = %b1010101111111010
```

5. This example replaces bits <7:8> of NEW with bits <8:9> of OLD:

```
INT new := -1;            !NEW = %b11111111111111
INT old := 0;              !OLD = %b0000000000000000
new.<7:8> := old.<8:9>; !NEW = %b11111100111111
```

# CALL Statement

The CALL statement invokes a procedure, subprocedure, or entry-point identifier, and optionally passes parameters to it.



*identifier*

is the identifier of a previously declared procedure, subprocedure, or entry-point *identifier*.

*param*

is a variable identifier or an expression that defines an actual parameter to pass to a formal parameter declared in *identifier*.

*param-pair*

is an actual parameter pair to pass to a formal parameter pair declared in *identifier*.



*string*

is the identifier of a STRING array or simple pointer declared inside or outside a structure.

*length*

is an INT expression that specifies the length, in bytes, of *string*.

## Usage Considerations

To invoke procedures and subprocedures (but usually not functions), use the CALL statement.

To invoke functions, you usually use their identifiers in expressions. If you invoke a function by using a CALL statement, the caller ignores the returned value of the function.

Actual parameters are value or reference parameters and are optional or required depending on the formal parameter specification in the called procedure or subprocedure declaration (described in [Section 13, Procedures](#)). A value parameter passes the content of a location; a reference parameter passes the address of a location.

The CALL keyword in CALL statements is optional. In a CALL statement to a VARIABLE procedure or subprocedure or to an EXTENSIBLE procedure, you can omit optional parameters in two ways:

- You can omit parameters or parameter pairs unconditionally. Use an empty comma for each omitted parameter or parameter pair up to the last specified parameter or parameter pair. If you omit all parameters, you can specify an empty parameter list (parentheses with no commas) or you can omit the parameter list altogether.
- You can omit parameters or parameter pairs conditionally. Use the \$OPTIONAL standard function as described in [Section 14, Standard Functions](#).

After the called procedure or subprocedure completes execution, control returns to the statement following the CALL statement that invoked the procedure or subprocedure.

For more information on parameters and parameter pairs, see the *TAL Programmer's Guide*.

## Examples of CALL Statements

1. This example invokes procedure ERROR\_HANDLER, which has no formal parameters:

```
CALL error_handler;
```

2. This example omits the CALL keyword from the CALL statement:

```
error_handler;
```

3. This example includes all parameters:

```
CALL compute_tax (item, rate, result);
```

4. This example omits all parameters:

```
CALL extensible_proc ( );
```

5. These examples omit some optional parameters:

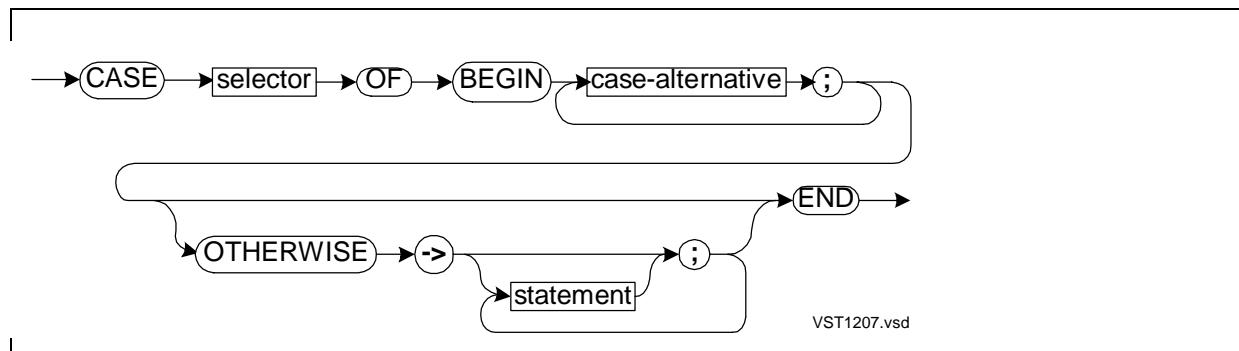
```
CALL variable_proc( , , , , , x);
CALL variable_proc (n, , char, , eof);
CALL variable_proc (n, !name!, char, !type!, !size!, eof);
```

## CASE Statement

The CASE statement executes a choice of statements based on a selector value. Normally, you use labeled CASE statements. Labeled CASE statements are described first, followed by unlabeled CASE statements.

## Labeled CASE Statement

The labeled CASE statement executes a choice of statements when the value of the selector matches a case label associated with those statements.

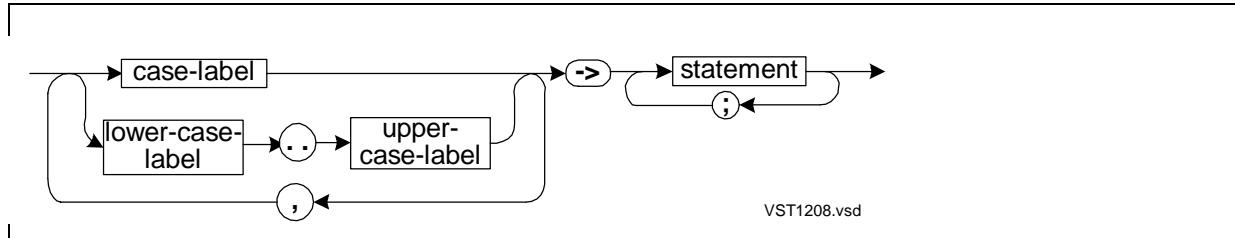


**selector**

is an INT arithmetic expression that uniquely selects the *case-alternative* for the program to execute.

**case-alternative**

associates one or more *case-labels* or one or more ranges of *case-labels* with one or more *statements*. The *statements* of a *case-alternative* are executed if *selector* equals an associated *case-label*.



**case-label**

is a signed INT constant or LITERAL. Each *case-label* must be unique in the CASE statement.

**lower-case-label**

is the smallest value in an inclusive range of signed INT constants or LITERALS.

**upper-case-label**

is the largest value in an inclusive range of signed INT constants or LITERALS.

**statement**

is any statement described in this section.

**OTHERWISE**

specifies an optional sequence of *statements* to execute if *selector* does not select any *case-alternative*. If no OTHERWISE clause is present and *selector* does not match a *case-alternative*, a run-time error occurs. Always include an OTHERWISE clause, even if it contains no *statements*.

## Usage Considerations

The *TAL Programmer's Guide* describes efficiency guidelines and execution of the labeled CASE statement.

## Example of Labeled CASE Statement

This labeled CASE statement has four case alternatives and the OTHERWISE case:

```

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;
    END
END

```

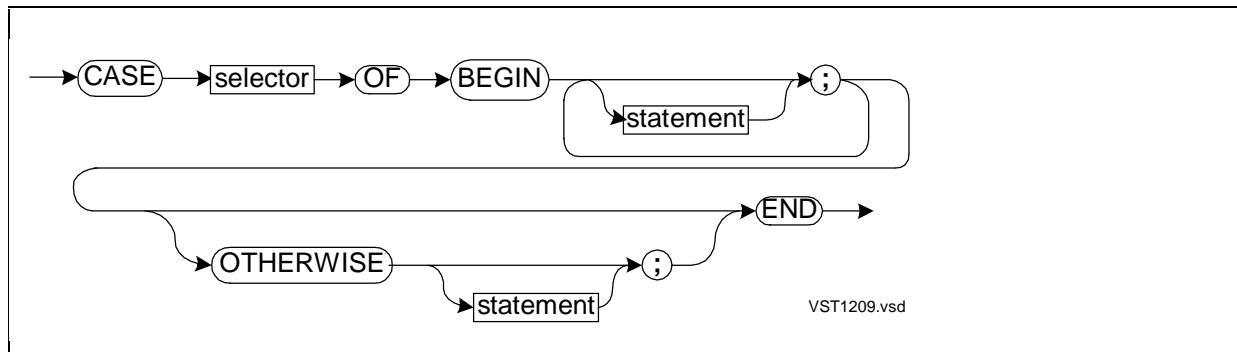
```

213, 818 ->
    location := los_angeles;
808 ->
    location := hawaii;
OTHERWISE ->
    location := elsewhere;
END;                                !End CASE statement
END;                                !End AREA_PROC

```

## Unlabeled CASE Statement

The unlabeled CASE statement executes a choice of statements, based on an inclusive range of implicit selector values, from 0 through *n*, with one statement for each value.



*selector*

is an INT arithmetic expression that selects the statement to execute.

*statement*

is any statement described in this section. Include a *statement* for each value in the implicit *selector* range, from 0 through *n*. If a *selector* has no action, specify a null statement (semicolon with no statement). If you include more than one *statement* for a value, you must use a compound statement.

*OTHERWISE*

indicates the statement to execute for any case outside the range of *selector* values. If the OTHERWISE clause consists of a null statement, control passes to the statement following the unlabeled CASE statement. See [Omitted Otherwise Clause](#).

## Usage Considerations

The compiler numbers each *statement* in the BEGIN clause consecutively, starting with 0. If the *selector* matches the compiler-assigned number of a *statement*, that *statement* is executed. For example, if the *selector* is 0, the first *statement* executes; if the

*selector* is 4, the fifth *statement* executes. Conversely, if the *selector* does not match a compiler-assigned number, the OTHERWISE statement, if any, executes.

For unlabeled CASE statements, the compiler always generates the branch table form (described in “Labeled CASE Statement” in the *TAL Programmer’s Guide*).

## Omitted Otherwise Clause

If you omit the OTHERWISE clause and *selector* is out of range (negative or greater than n), the compiler behaves as follows:

- If the CHECK directive is in effect and your program enables arithmetic traps, a divide-by-zero instruction trap occurs.
- If NOCHECK is in effect or if your program disables arithmetic traps, control passes to the statement following the unlabeled CASE statement and program results are unpredictable.

## Examples of Unlabeled CASE Statements

1. If SELECTOR in the following CASE statement is 0, the first *statement* executes; if SELECTOR is 1, the second *statement* executes. For any other SELECTOR value, the third *statement* executes:

```
INT selector;
INT var0;
INT var1;

CASE selector OF
  BEGIN
    var0 := 0;           !First statement
    var1 := 1;           !Second statement
  OTHERWISE
    CALL error_handler; !Third statement
  END;
```

2. This example selectively moves one of several messages into an array:

```
PROC msg_handler (selector);
  INT selector;
  BEGIN
    LITERAL len = 80;           !Length of array
    STRING .a_array[0:len - 1]; !Destination array
```

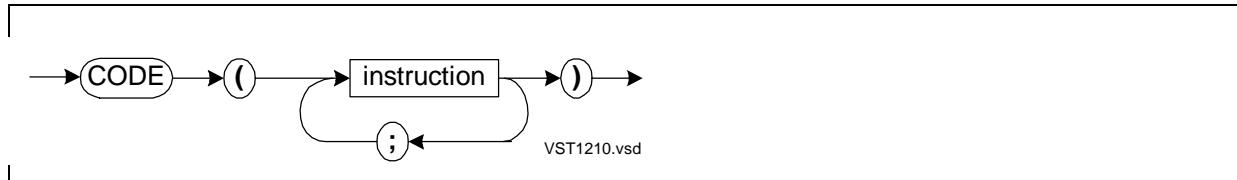
```

CASE selector OF
  BEGIN                                !Move statements
    !0! a_array ':=' "Training Program";
    !1! a_array ':=' "End of Program";
    !2! a_array ':=' "Input Error";
    !3! a_array ':=' "Home Terminal Now Open";
  OTHERWISE
    a_array ':=' "Bad Message Number";
  END;                                     !End of CASE statement
END;                                         !End of procedure

```

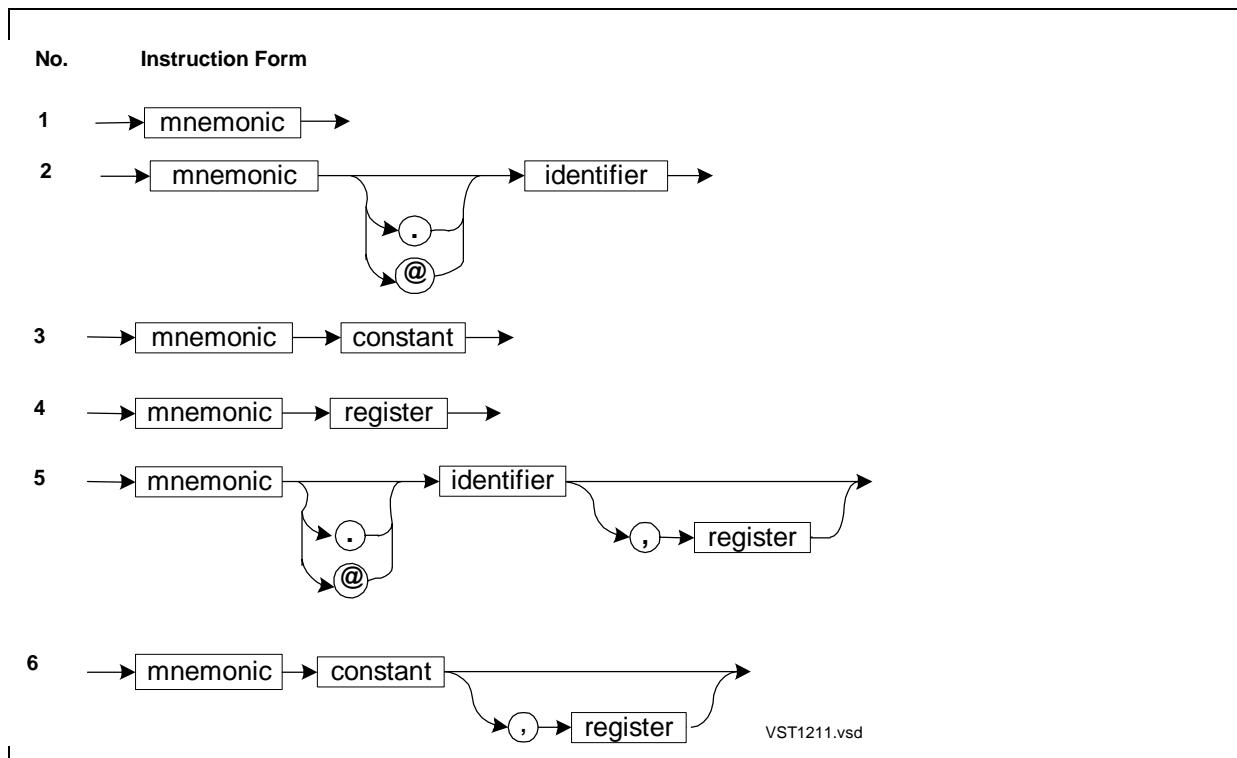
## CODE Statement

The CODE statement enables you to specify machine-level instructions and pseudocodes to compile into the object code.



instruction

is a machine instruction in one of the six forms:



mnemonic

is either an instruction code or a pseudocode.

. (period)

is the dereferencing operator, which converts the value of *identifier* into the standard word address of another data item.

@

removes indirection. If *identifier* is a pointer, @ accesses the address contained in the pointer. Otherwise, @ accesses the address of *identifier*.

identifier

is the identifier of a procedure or label:

- For a PCAL or XCAL instruction, it is a procedure. The procedure identifier must be resolvable by the time the executable object file is created.
- For a branch instruction, it is a label.

An indirect *identifier* specified without @ generates instructions for an indirect reference through *identifier*.

constant

is an INT constant of the same size as the instruction field.

register

is either:

- An INT constant that specifies a register number in the range 0 through 7
- An identifier associated with an index register by a USE statement

If you omit *register*, no indexing occurs.

## Usage Considerations

Because CODE statements are not portable to future software platforms, modularize their use as much as possible to simplify future modification of your program.

## Instruction Codes

You can use instruction codes and pseudocodes as CODE statement mnemonics. The six instruction forms shown in the syntax diagram correlate to instruction codes described in the *System Description Manual* for your system. You must include all required operands for each instruction. The compiler inserts indirect branches around data or instructions emitted in a CODE statement, if needed. Normally, the compiler emits these values after the first unconditional branch instruction occurs.

## Pseudocodes

You can use pseudocodes and instruction codes as CODE statement mnemonics. The form numbers in the following descriptions correlate to form numbers shown in the CODE statement instruction form diagrams:

ACON      A form 3 mnemonic that emits the value specified in *constant*. *constant* is the absolute run-time code address associated with the label in the next instruction location. An absolute code address is relative to the beginning of the code space in which the encompassing procedure resides.

CON      A form 3 mnemonic that emits the value specified in *constant*. *constant* is an offset of a location from the program counter or a character string constant.

The following indirect CODE branch is no longer allowed:

```
CODE (BANZ .test_x);
!Lots of code
test_x: CODE (CON @test_z);
!Error 11 results
```

DECS      A form 3 mnemonic that decrements the compiler's internal S-register counter by the amount specified in *constant*. *constant* is a signed integer to subtract from the compiler's internal S-register counter. This mnemonic emits no code. The CODE (DECS) statement and the DECS directive behave the same except when you include them in a DEFINE declaration. That is, the CODE(DECS) statement is part of the DEFINE, but the DECS directive executes immediately and is not part of the DEFINE.

FULL      A form 1 mnemonic that signals the compiler that the register stack is full and sets the compiler's internal RP counter to 7. This instruction emits no code. RP is the register stack pointer, which points to the top of the register stack.

RP      A form 3 mnemonic that sets the compiler's internal RP counter to the value specified in *constant*. *constant* is a value in the range 0 through 7, where 7 signals the compiler that the register stack is empty. This mnemonic emits no code. The CODE (RP) statement and the RP directive behave the same except when you include them in a DEFINE declaration. That is, the CODE(RP) statement is part of the DEFINE, but the RP directive executes immediately and is not part of the DEFINE.

## DUMPCONS Directive and CODE Statements

When you use CODE statements to create a block of code, the compiler might insert constants or branch labels into the object file in the middle of your block of code. If you need to keep the block of code intact, put a DUMPCONS directive immediately before the CODE statements. The compiler then inserts all pending constants and branch labels into the object code before it compiles the CODE statements.

## FOR Statements and CODE Statements

If you use a CODE statement to make a conditional jump from an optimized FOR loop, the compiler emits warning 78 (TAL cannot set the RP value for the forward branch). To set the RP value for the jump condition, use a CODE (STRP ...) statement or an RP directive. The compiler can generate correct code only for the nonjump condition.

## Examples of CODE Statements

1. This example shows instruction codes that turn off traps:

```
CODE (RDE; ANRI %577; SETE);
```

2. These examples show the six instruction forms:

```
CODE (ZERD; IADD);           !Form 1
CODE (LADR a; STOR .b);     !Form 2
CODE (LDI 21; ADDI -4);      !Form 3
CODE (STAR 7; STRP 2);       !Form 4
CODE (LDX a, 7; LDB stg, x); !Form 5
CODE (LDXI -15 ,5);         !Form 6
```

3. This example scans from a code-relative address to the test character (which is a 0), and then saves the next address:

```
STRING .ptr;
!Some code here
STACK @ptr, 0;
CODE (SBU %040);
STORE @ptr;
```

4. This example decrements the compiler's internal S-register counter by 2:

```
CODE (DECS 2);
```

5. This example decrements the run-time S-register by 2:

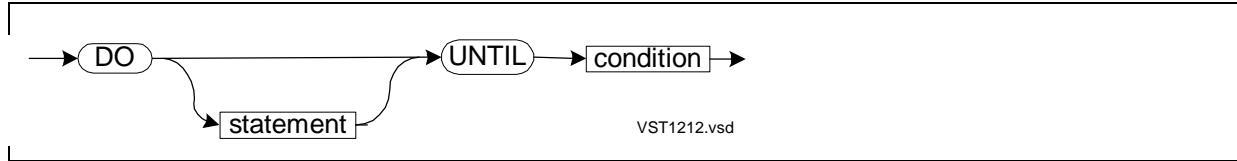
```
CODE (ADDS -2);
```

6. Each of these CON pseudocodes emits data in the next instruction location:

```
CODE (CON %125);          !Emit %125
CODE (CON "the con pseudo operator code"); 
                           !Emit 14 words of constants
CODE (CON @labelid);      !Emit a one-word algebraic signed
                           ! difference between the address
                           !of LABELID and the current content
                           !of the program counter;
                           !that is, of @LABELID - @P.
```

# DO Statement

The DO statement is a posttest loop that repeatedly executes a statement until a specified condition becomes true.



*statement*

is any statement described in this section.

*condition*

is either:

- A conditional expression
- An INT arithmetic expression. If the result of the arithmetic expression is not 0, *condition* is true. If the result is 0, *condition* is false.

If *condition* is false, the DO loop continues to execute. If *condition* is true, the statement following this DO statement executes.

## Usage Considerations

If the *condition* is always false, the loop repeats until a *statement* in the DO loop causes an exit.

A DO statement always executes at least once because the compiler tests *condition* at the end of the loop. Unless you have a special reason to use a DO statement, it is safer to use the WHILE statement.

DO statement execution is shown in the *TAL Programmer's Guide*.

## Examples of DO Statements

1. This DO statement loops through ARRAY\_A, testing each the content of each element until an alphabetic character occurs:

```

index := -1;
DO index := index + 1 UNTIL $ALPHA (array_a[index]);
  
```

2. This DO statement loops through ARRAY\_A, assigning a 0 to each element until all the elements contain a 0:

```

LITERAL limit = 9;
INT index := 0;
STRING .array_a[0:limit];           !Declare array

DO                                !DO statement
BEGIN
  array_a[index] := 0;             !Compound statement to
  index := index + 1;            ! execute in DO loop
END
UNTIL index > limit;             !Condition for ending loop

```

## DROP Statement

The DROP statement disassociates an identifier from either:

- A label
- An index register that you reserved in a previous USE statement



**identifier**

is the identifier of either:

- A label
- An index register that you reserved in a previous USE statement

## Usage Considerations

Following are guidelines for dropping labels and registers.

### Dropping Labels

You can drop a label only if you have declared the label or used it to label a statement. Before you drop a label, be sure there are no further references to the label. If a GOTO statement refers to a dropped label, a run-time error occurs. After you drop a label, you can, however, use the identifier to label a statement preceding the GOTO statement that refers to the label.

## Dropping Registers

You reserve index registers by issuing USE statements. When you no longer need a reserved index register, drop (RVU) it by issuing a DROP statement. After you drop an index register, do not use its identifier without issuing a new USE statement and assigning a value to it.

If you do not drop all reserved index registers, the compiler automatically drops them when the procedure or subprocedure completes execution.

If you reserve an index register for a FOR loop, do not drop the register within the scope of the loop.

## Examples of DROP Statements

1. This example uses and drops a label within a DEFINE declaration:

```
DEFINE loop =
BEGIN
lab:                      !Uses label identifier
    IF a = b
    THEN
        GOTO lab;          !Branches to label
        DROP lab;          !Frees label identifier for reuse
    END #;
```

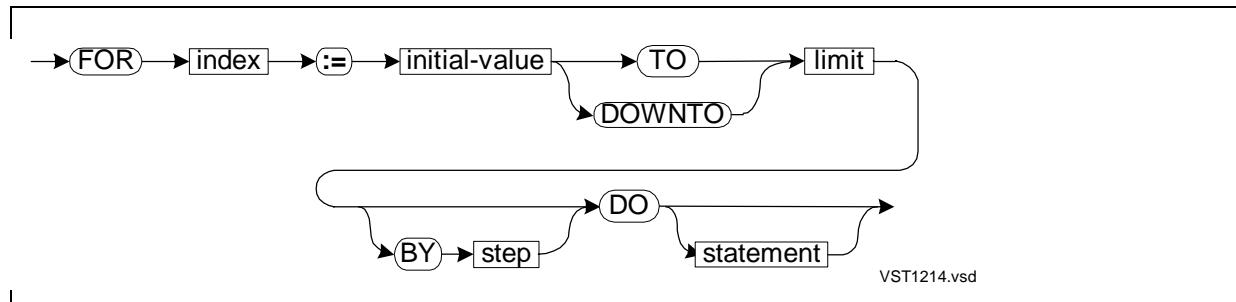
2. This example reserves, uses, and drops an index register:

```
LITERAL len = 100;
INT array[0:len-1];      !Declarations

USE x;                   !Reserves index register named x
FOR x := 0 TO len - 1 DO
    array[x] := 0;        !Uses register X to clear array
    DROP x;              !Drops register X
```

# FOR Statement

The FOR statement is a pretest loop that repeatedly executes a statement while incrementing or decrementing an index automatically. The loop terminates when the index reaches a set value.



**index**

is a value that increments or decrements automatically until it reaches *limit* 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**

is an INT arithmetic expression (such as 0) that initializes *index*.

**TO**

increments *index* each time the loop executes until *index* exceeds *limit*.

**DOWNT0**

decrements *index* each time the loop executes until *index* is less than *limit*.

**limit**

is an INT arithmetic expression that terminates the FOR looping.

**step**

is an INT arithmetic expression by which to increment or decrement *index* each time the loop executes. The default value is 1.

**statement**

is any statement described in this section.

## Usage Considerations

The FOR statement tests *index* at the beginning of each iteration of the loop. If *index* passes *limit* on the first test, the loop never executes.

You can nest FOR loops to any level.

FOR statement execution is shown in the *TAL Programmer's Guide*.

### Standard FOR Loops

For *index*, standard FOR loops specify an INT variable. Standard FOR loops execute as follows:

- When the looping terminates, *index* is greater than *limit* if:
  - The *step* value is 1.
  - You use the TO keyword (not DOWNTO).
  - The *limit* value (not a GOTO statement) terminates the looping.
- *limit* and *step* are recomputed at the start of each iteration of the loop.

### 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 loop.  
(Modifying the register stack, however, is not an operation 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.

Inclusion of procedure calls in the FOR loop slows down the loop because the compiler must emit code to save and restore registers before and after each CALL statement.

The following operations are not portable to future software platforms:

- Using a CODE statement to conditionally jump out of an optimized FOR loop. The compiler generates correct code only for the nonjump condition.
- Jumping into an optimized FOR loop.

## Examples of FOR Statements

1. 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];
```

2. This 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);
```

3. This example compares a standard FOR loop to its optimized equivalent. Both FOR loops clear the array by assigning a 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
```

# GOTO Statement

The GOTO statement unconditionally transfers program control to a statement that is preceded by a label.



**label-name**

is the identifier of a label that is associated with a statement. It cannot be an entry-point identifier.

## Usage Considerations

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.

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.

## Examples of GOTO Statements

1. In this 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;
```

2. In this 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
```

```

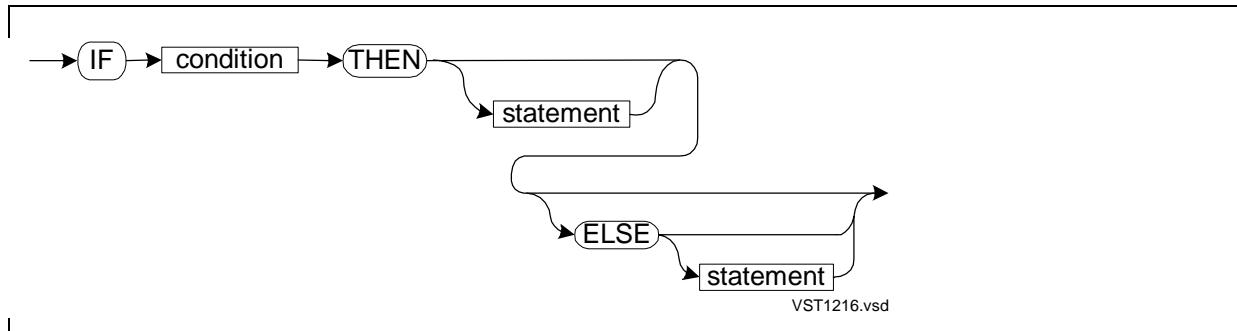
GOTO a;           !Sublocal branch to local label
END;

a :
  i := 0;
  !More code
END;

```

## IF Statement

The IF statement conditionally selects one of two statements.



*condition*

is either:

- A conditional expression
- An INT arithmetic expression. If the result of the arithmetic expression is not 0, *condition* is true. If the result is 0, *condition* is false.

*THEN statement*

specifies the statement to execute if *condition* is true. *statement* can be any statement described in this section. If you omit *statement*, no action occurs for the THEN clause.

*ELSE statement*

specifies the statement to execute if *condition* is false. *statement* can be any statement described in this section.

## Usage Considerations

If the *condition* is true, the THEN *statement* executes. If the *condition* is false, the ELSE *statement* executes. If no ELSE clause is present, the statement following the IF statement executes.

You can nest IF statements to any level.

For more information on the IF statement execution and IF-ELSE pairing, see Section 12, “Controlling Program Flow,” in the *TAL Programmer’s Guide*.

## Example of IF Statements

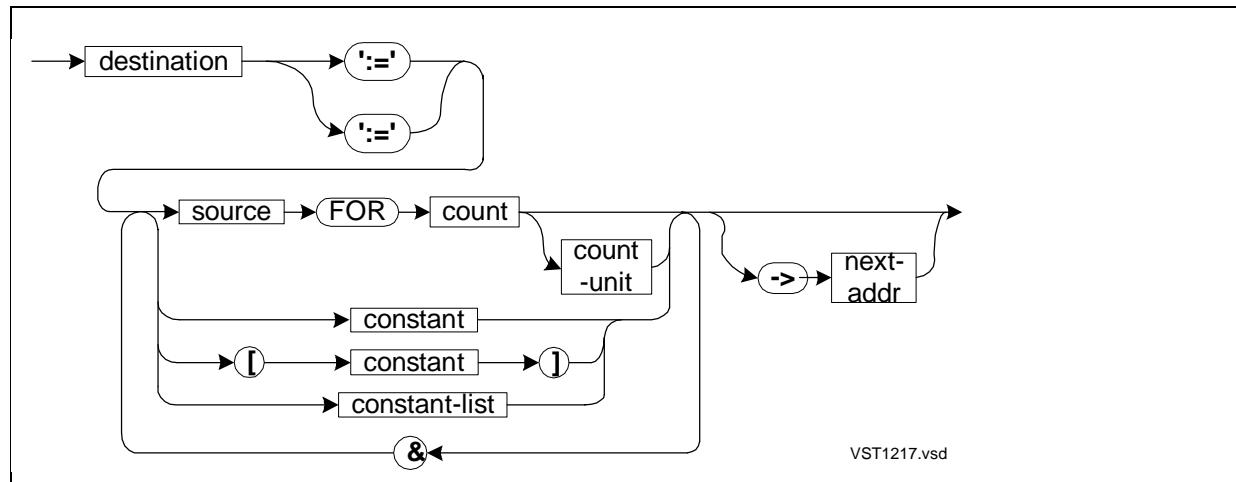
This example compares two arrays:

```
INT .new_array[0:9];
INT .old_array[0:9];
INT item_ok;

IF new_array = old_array FOR 10 WORDS THEN
    item_ok := 1
ELSE
    item_ok := 0;
```

## MOVE Statement

The move statement copies contiguous bytes, words, or elements to a new location.



**destination**

is the identifier, with or without an index, of the variable to which the copy operation begins. It can be a simple variable, array, simple pointer, structure, structure data item, or structure pointer, but not a read-only array.

' := '

specifies a left-to-right sequential move. It starts copying data from the leftmost item in *source*.

' = : '

specifies a right-to-left sequential move. It starts copying data from the rightmost item in *source*.

**source**

is the identifier, with or without an index, of the variable from which the copy operation begins. It can be a simple variable, array, read-only array, simple pointer, structure, structure data item, or structure pointer.

**count**

is an unsigned INT arithmetic expression that defines the number of units in *source* to copy. If you omit *count-unit*, the units copied (depending on the nature of the *source* variable) are:

<b>Source Variable</b>	<b>Data Type</b>	<b>Units Copied</b>
Simple variable, array, simple pointer (including structure item)	STRING INT INT (32) or REAL FIXED or REAL (64)	Bytes Words Doublewords Quadruplewords
Structure	Not applicable	Words
Substructure	Not applicable	Bytes
Structure Pointer	STRING* INT*	Bytes Words

**count-unit**

is the value BYTES, WORDS, or ELEMENTS. *count-unit* changes the meaning of *count* from that described above to the following:

BYTES	Copies <i>count</i> bytes. If both <i>source</i> and <i>destination</i> have word addresses, BYTES generates a word move for $(count + 1) / 2$ words.
WORDS	Copies <i>count</i> words
ELEMENTS	Copies <i>count</i> elements as follows (depending on the nature of the <i>source</i> variable):

<b>Source Variable</b>	<b>Data Type</b>	<b>Units Copied</b>
Simple variable, array, simple pointer (including structure item)	STRING INT INT (32) or REAL FIXED or REAL (64)	Bytes Words Doublewords Quadruplewords
Structure	Not applicable	Structure occurrences
Substructure	Not applicable	Substructure occurrences
Structure Pointer	STRING * INT *	Structure occurrences Structure occurrences

\* For structure pointers, STRING and INT have meaning only in group comparison expressions and move statements.

If *count-unit* is not BYTES, WORDS, or ELEMENTS, the compiler issues an error. If you specify BYTES, WORDS, or ELEMENTS for *count-unit*, that term cannot also appear as a DEFINE or LITERAL identifier in the global declarations or in any procedure or subprocedure in which the move statement appears.

*constant*

is a numeric constant, a character string constant, or a LITERAL to copy.

If you enclose *constant* in brackets ([ ]) and if *destination* has a byte address or is a STRING structure pointer, the system copies *constant* as a single byte regardless of the size of *constant*. If you do not enclose *constant* in brackets or if *destination* has a word address or is an INT structure pointer, the system copies a word, doubleword, or quadrupleword as appropriate for the size of *constant*.

*constant-list*

is a list of constants to copy. Specify *constant-list* in the form shown in [Section 3, Data Representation](#).

*next-addr*

is a variable to contain the location in *destination* that follows the last item copied. The compiler returns a 16-bit or 32-bit address as described in [Usage Considerations](#).

&

is the concatenation operator. It lets you move more than one source or *constant-list*, each separated by the concatenation operator.

## Usage Considerations

The compiler does a standard or extended move (which is slightly less efficient) depending on the addressing modes of the data involved in the move operation.

The compiler does a standard move and returns a 16-bit *next-addr* if:

- Both *source* and *destination* have standard byte addresses
- Both *source* and *destination* have standard word addresses

The compiler does an extended move and returns a 32-bit *next-addr* if:

- Either *source* or *destination* has a standard byte address and the other has a standard word address
- Either *source* or *destination* 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

If the move contains more than one *source* and one of the sources causes an extended move sequence, the compiler emits an extended move sequence for all the sources. The compiler generates the extended address of each variable, generates a byte count corresponding to *count*, and emits a MVBX instruction for the move statement.

After an element move, *next-addr* might point into the middle of an element, rather than at the beginning of the element. If *destination* is word addressed and *source* is byte addressed and you copy an odd number of bytes, *next-addr* will not point to an element boundary.

## Examples of MOVE Statements

1. This example copies spaces into the first five elements of an array, and then uses *next-addr* as *destination* to copy dashes into the next five elements:

```
LITERAL len = 10;           !Length of array
LITERAL num = 5;            !Number of elements
STRING .array[0:len - 1];   !Destination array
STRING .next_addr;          !Next address simple pointer

array[0] ':= num * [" "] -> @next_addr;
                           !Do first copy and capture next-addr
next_addr ':= num * ["-"];
                           !Use next-addr as start of second copy
```

2. This example contrasts copying a bracketed constant versus copying an unbracketed constant. A bracketed constant copies a single byte regardless of the size of the constant. An unbracketed constant copies words, doublewords, or quadruplewords depending on the size of the constant:

```
STRING x[0:8];             !Declare STRING array X
x[0] ':= [0];               !Copy one byte
x[0] ':= 0;                 !Copy two bytes
```

3. This example copies three occurrences of one structure to another:

```
LITERAL copies = 3;           !Number of occurrences
STRUCT .s[0:copies - 1];      !Source structure
BEGIN
  INT a, b, c;
END;

STRUCT .d (s) [0:copies - 1]; !Destination structure

PROC p;
BEGIN
  d ':=' s FOR copies ELEMENTS; !Word move of three
END;                           ! structure occurrences
```

4. This example copies three occurrences of a substructure:

```
LITERAL copies = 3;           !Number of occurrences
STRUCT .s;
BEGIN
  STRUCT s_sub[0:copies - 1]; !Source substructure
  BEGIN
    INT a, b;
  END;
END;

STRUCT .d (s);               !Destination substructure
                             ! is within structure D

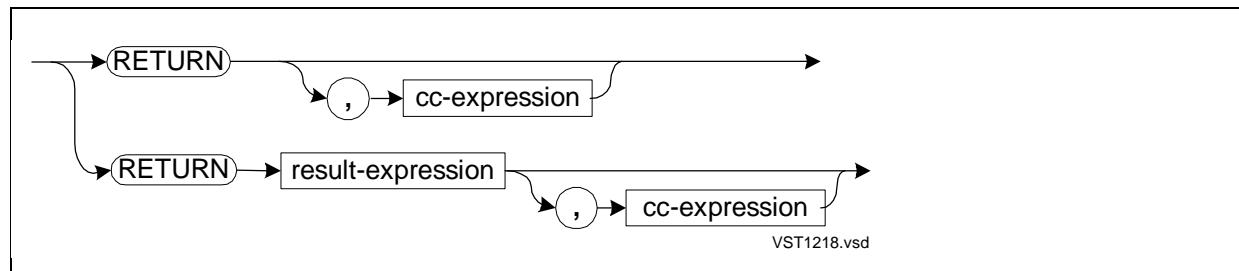
PROC p;
BEGIN
  d.s_sub ':=' s.s_sub FOR copies ELEMENTS;
END;                           !Byte move of three
                               ! substructure occurrences
```

For more examples, see sections 7 and 8 in the *TAL Programmer's Guide*.

## RETURN Statement

The RETURN statement returns control to the caller. If the called procedure or subprocedure is a function, RETURN must return a result expression.

The RETURN statement can also return a program-specified condition code value.



**cc-expression**

is an INT expression whose numeric value specifies the condition code value to return to the caller. If *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 (>)

**result-expression**

is an arithmetic or conditional expression that a function must return to the caller. *result-expression* must be of the same return type as the data type specified in the function header. The data type of a conditional expression is always INT. Specify *result-expression* only when returning from a function.

## Usage Considerations

In general, a procedure or subprocedure returns control to the caller when:

- A RETURN statement is executed.
- The called procedure or subprocedure reaches the end of its code.

## Returning From Functions

A function is a typed procedure or 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.

If *result-expression* is any type except FIXED or REAL(64), a function can return both *result-expression* and *cc-expression*.

A function can also return the condition code value by specifying a value in a CODE or STACK statement; however, CODE and STACK statements are not portable to future software platforms. Furthermore, if the compiler's RP counter setting is not correct, the compiler emits a warning.

## Returning From Nonfunction Procedures and Subprocedures

In procedures and subprocedures that are not functions, a RETURN statement is optional. A nonfunction procedure or subprocedure that returns a condition code value, however, must return to the caller by executing a RETURN statement that includes *cc-expression*.

In a procedure designated MAIN, a RETURN statement stops execution of the procedure and passes control to the operating system.

## Examples of RETURN Statements

1. This function contains two RETURN statements nested in an IF statement:

```
INT PROC other (nuff, more);      !Declare function with
                                  ! return type INT

    INT nuff;
    INT more;
BEGIN
    IF nuff < more THEN          !IF statement
        RETURN nuff * more      !Return a value
    ELSE
        RETURN 0;                !Return a different value
END;
```

2. This 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;
```

3. This function returns a value 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
                                ! condition code
END;
```

4. This 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
                                ! is >; otherwise, it is =
END;
```

5. 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;

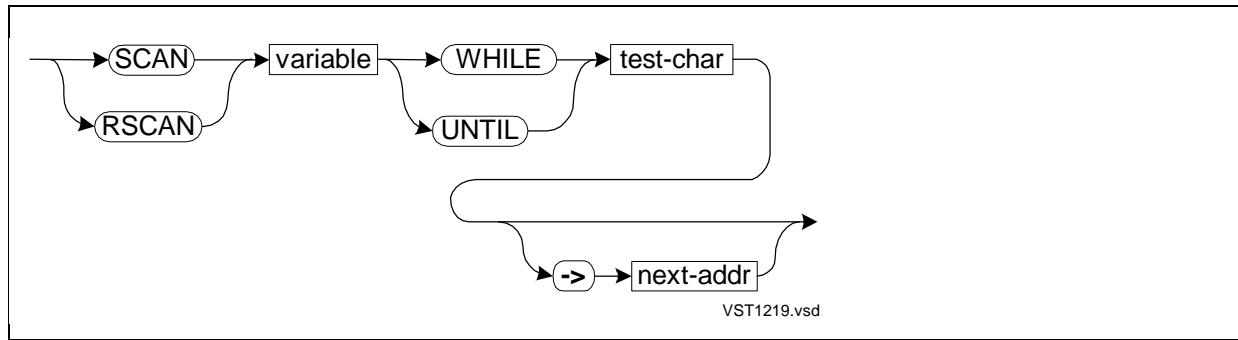
INT PROC p2 (i);
    INT i;
BEGIN
INT j := i + 1;
RETURN i, j;
END;

CALL p1 (i);
IF < THEN ... ;           !Test the condition code
CALL p2 (i);
IF < THEN ... ;           !Test the condition code

```

## SCAN Statement

The SCAN or RSCAN statement searches a scan area for a test character from left to right or from right to left, respectively.



**SCAN**

indicates a left-to-right search.

**RSCAN**

indicates a right-to-left search.

**variable**

is the identifier, with or without an index, of a variable at which to start the scan.  
The following restrictions apply:

- The variable can be a simple variable, array, read-only array, simple pointer, structure pointer, structure, or structure data item.
- The variable can be of any data type but UNSIGNED.
- The variable must be located either:
  - In the lower 32K-word area of the user data segment.
  - In the same 32K-word area of the current code segment as the procedure that accesses the variable.
- The variable cannot have extended indirection.

**WHILE**

specifies that the scan continues until a character other than *test-char* occurs or until a 0 occurs. A scan stopped by a character other than *test-char* resets the hardware carry bit. A scan stopped by a 0 sets the hardware carry bit.

**UNTIL**

specifies that the scan continues either until *test-char* occurs or until a 0 occurs. A scan stopped by *test-char* resets the hardware carry bit. A scan stopped by a 0 sets the hardware carry bit.

**test-char**

is an INT arithmetic expression that evaluates to a maximum of eight significant bits (one byte). A larger value might cause execution errors.

**next-addr**

is a 16-bit variable to contain the 16-bit byte address of the character that stopped the scan, regardless of the data type of *identifier*.

## Usage Considerations

You should delimit the scan area with zeros. Otherwise, a scan operation might continue to the 32K-word boundary if either:

- 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

To delimit the scan area, you can specify zeros as follows:

```
INT .buffer[-1:10] := [0, " John James Jones ",0];
```

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-addr*'-'*byte address of identifier*) by the number of bytes per element, using unsigned arithmetic.

## Example of SCAN Statements

The following example converts the word address of an INT array to a byte address. The assignment statement stores the resulting byte address in 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
```

For more examples, see Section 7, Using Arrays, in the *TAL Programmer's Guide*.

## STACK Statement

The STACK statement loads values onto the register stack.



expression

is a value to load onto the register stack. If you list multiple values, the compiler loads them starting with the leftmost value.

## Usage Considerations

You can use the register stack for temporary storage and for optimizing critical code.

The compiler loads values on the register stack starting at the current setting of RP + 1 and increments RP by the number of words needed by each value. For example, an INT(32) value needs two words; a FIXED value needs four words.

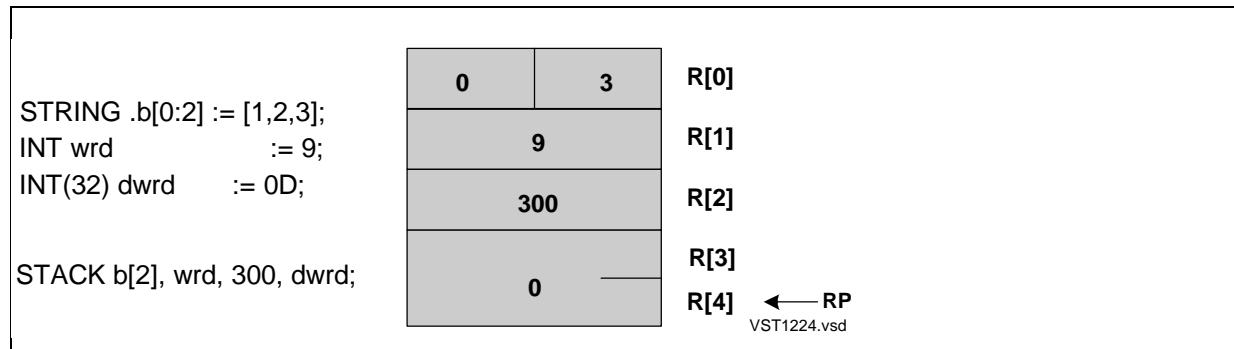
The number of registers needed by a value depends on its data type. If enough registers are not free, the compiler transfers the content of registers R[0] through RP to the data stack and then loads STACK values starting at RP[0]. The compiler keeps

track of the size and data type of stacked values. It loads byte values in bits <8:15> and a 0 in bits <0:7>.

Modularize any use of STACK statements as much as possible; they are not portable to future software platforms.

## Examples of STACK Statements

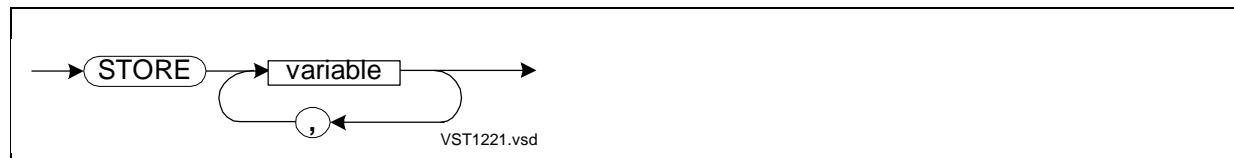
This example loads values of various data types onto the register stack:



For more information on examples of STACK Statements, see the [STORE Statement](#).

## STORE Statement

The STORE statement removes values from the register stack and stores them into variables.



**variable**

is the identifier of a variable—a simple variable, array element, simple pointer, or structure data item—with or without a bit deposit field and/or index. To update the content of a simple pointer, prefix the pointer identifier with @. If you list multiple identifiers, storage begins with the leftmost identifier.

## Usage Considerations

The data type of each variable specified dictates the number of registers to unload, starting at the current setting of the RP. If the RP setting is too small to satisfy the variable data type, the compiler removes the required number of items from the data stack, places them on the register stack, and stores them in the variable.

Modularize any use of STORE statements as much as possible; they are not portable to future software platforms.

## Examples of STORE Statements

1. This example stores the content of the register stack starting at the current setting of the RP into variables of various data types:

```
LITERAL len = 100;
STRING      .byte[0:len - 1];
INT         word;
INT(32)    twowords;

STORE twowords, word, byte[3];
```

2. This example loads the values from two variables onto the register stack, and then stores them back into the same variables:

```
STACK x, y;
!Some code here
STORE y, x;
```

3. This example shows two versions of a swap operation:

```
INT temp;
INT x;
INT y;

temp := x;
x := y;           !Version 1 needs five memory references
y := temp;        ! if you use OPTIMIZE 2

STACK x,y;        !Version 2 needs four memory references,
STORE x,y;        !uses the register stack, and is faster
                  ! on TNS systems
```

## USE Statement

The USE statement optimizes repetitive references to localized expressions or iteration variables. It associates an identifier with an index register and reserves the register for your use.



**identifier**

is an identifier to associate with an index register.

## Usage Considerations

The compiler associates each identifier specified in a USE statement with an index register starting with R[7] down to R[5]. Thus, you can use at most three reserved registers at a time.

Statements that appear between a USE statement and a corresponding DROP statement or the end of the procedure or subprocedure can access the reserved index register. If a global or local item with the same identifier as a USE identifier already exists, the compiler issues an error message.

You can, for example, use a reserved index register to optimize a FOR statement as described in the description of the FOR statement. Before referring to the identifier of a reserved index register, be sure to assign a value to it.

You can also assign a value to a reserved index register and then pass the index register content as an implicit parameter to a procedure or subprocedure. This practice, however, is not portable to future software platforms. For more information, see the *TAL Programmer's Guide*.

If evaluation of an expression overwrites the value in a reserved register, the compiler issues a diagnostic message. For example, multiplication of two FIXED values will overwrite the reserved register.

To determine whether using the index register is beneficial, you can use the INNERLIST directive and then review the code generated by statements in the range of a USE statement.

If the compiler needs an index register and none is available, the compiler emits a diagnostic message.

When you are finished using a reserved index register, release (or drop) it by issuing a DROP statement, as described in [DROP Statement](#) on page 12-20.

## Examples of USE Statements

1. This example reserves two index registers. The compiler associates each identifier with an index register, starting with R[7]:

```
USE a_index;                      !Reserve R[7]
USE b_index;                      !Reserve R[6]

a_index := 0;
b_index := -1;
```

2. This example contrasts a standard FOR loop with an optimized FOR loop (if no procedure or function calls occur within the loop):

```

LITERAL len = 100;
INT .array [0:len - 1];
INT i;

FOR i := 0 TO len - 1 DO
    array[i] := array[i] + 5;           ! Standard FOR loop

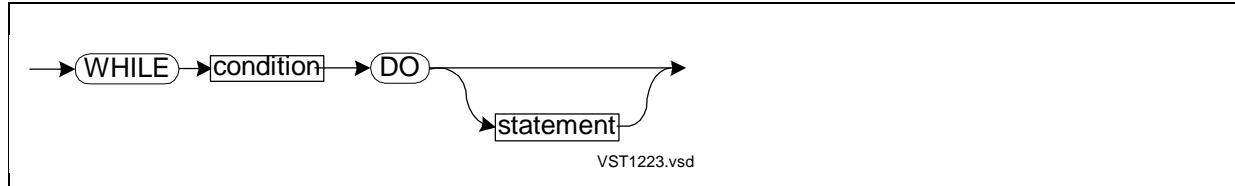
USE x;
FOR x := 0 to len - 1 DO
    array[x] := array[x] + 5;          ! Optimized FOR loop uses
                                         ! an index register

DROP x;                                ! Release the register

```

## WHILE Statement

The WHILE statement is a pretest loop that repeatedly executes a statement while a specified condition is true.



*condition*

is either:

- A conditional expression
- An INT arithmetic expression. If the result of the arithmetic expression is not 0, *condition* is true. If the result is 0, *condition* is false.

*statement*

is any TAL statement.

## Usage Considerations

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.

WHILE statement execution is shown in the *TAL Programmer's Guide*.

## Examples of WHILE Statements

1. 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
```

2. 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 . . . ;
```



# **13 Procedures**

Procedures are program units that contain the executable portions of a TAL program and that are callable from anywhere in the program. Procedures allow you to segment a program into discrete parts that each perform a particular task such as I/O or error handling.

An executable program contains at least one procedure. One procedure in the program has the attribute MAIN, which identifies it as the first procedure to execute when you run the program.

A procedure can contain subprocedures, which are callable from various points within the same procedure.

A function is a procedure or subprocedure that returns a value. A function is also known as a typed procedure or typed subprocedure.

This section describes the syntax for:

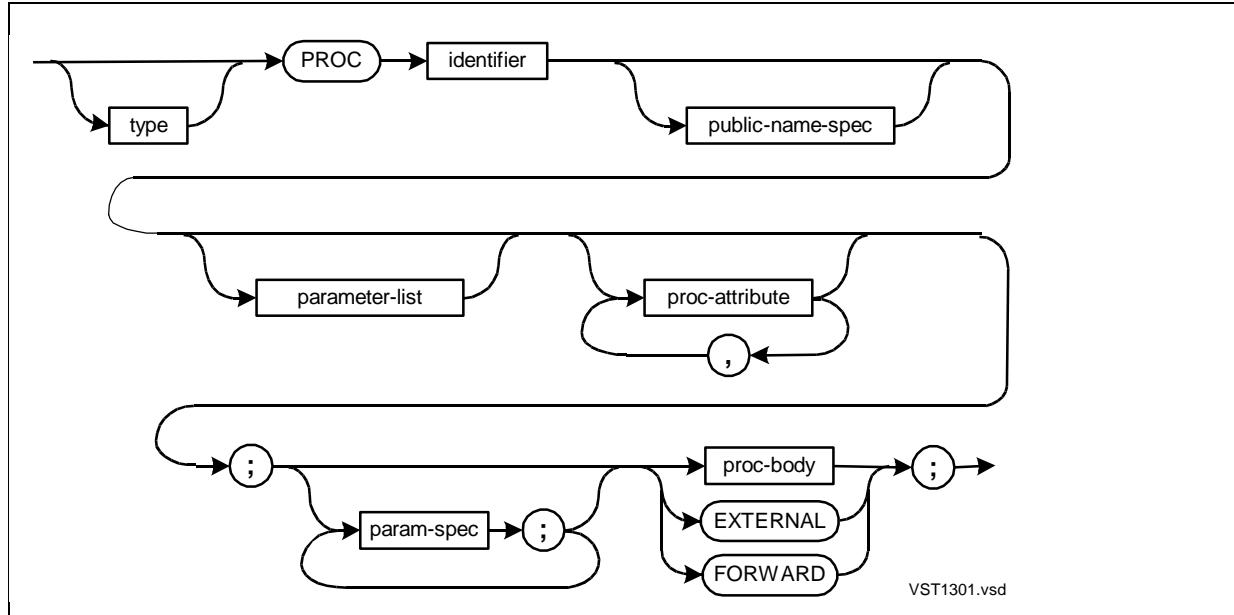
- Procedure declarations
- Subprocedure declarations
- Entry-point declarations
- Label declarations

Section 11, “Using Procedures,” in the *TAL Programmer’s Guide* describes:

- How the compiler allocates storage for procedures and subprocedures
- How you call procedures and subprocedures
- How you pass parameters
- What parameter masks look like

# Procedure Declaration

A procedure is a program unit that is callable from anywhere in the program. You declare a procedure as follows:



`type`

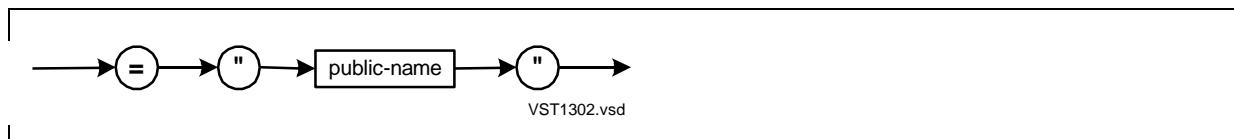
specifies that the procedure is a function that returns a result and indicates the data type of the returned result. `type` can be any data type described in [Section 3, Data Representation](#).

`identifier`

is the procedure identifier to use in the compilation unit.

`public-name-spec`

has the form:

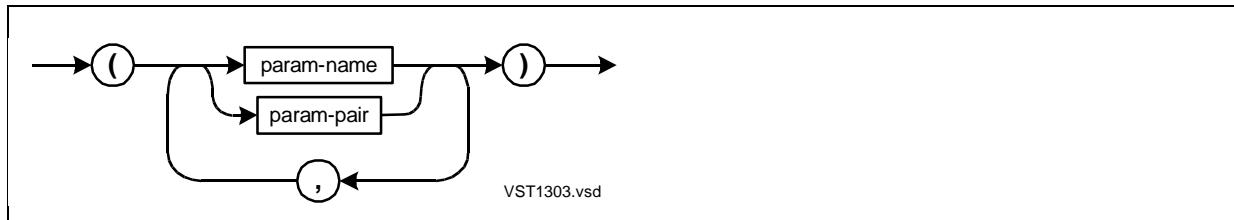


`public-name`

is the procedure name to use in Binder, not in the compilation unit. Use this option only in a D-series `EXTERNAL` procedure declaration. If you omit this option, `identifier` is the default `public-name`. `public-name` must conform to the identifier rules of the language in which the external procedure is written. For all languages except C, the compiler upshifts `public-name` automatically.

parameter-list

has the form:



param-name

is the identifier of a formal parameter. A procedure can have up to 32 formal parameters, with no limit on the number of words of parameters.

param-pair

is a pair of formal parameter identifiers that comprises of a language-independent string descriptor in the form:



string

is the identifier of a standard or extended STRING simple pointer. The actual parameter is the identifier of a STRING array or simple pointer declared inside or outside a structure.

length

is the identifier of a directly addressed INT simple variable. The actual parameter is an INT expression that specifies the length of *string*, in bytes.

proc-attribute

is a procedure attribute, as described in [Procedure Attributes](#) on page 13-5. The TAL compiler ignores extra commas between attributes and before and after the list of attributes.

param-spec

specifies the parameter type of a formal parameter and whether it is a value or reference parameter, as described in [Formal Parameter Specifications](#) on page 13-8.

**proc-body**

is a BEGIN-END construct that contains local declarations and statements, as described in [Procedure Body](#) on page 13-13.

**FORWARD**

specifies that the procedure body is declared later in the source file.

**EXTERNAL**

specifies that the procedure body is declared in another compilation unit.

## Usage Considerations

The maximum size of a single procedure is 32K words minus either the Procedure Entry Point (PEP) table in the lower 32K-word area of the user code segment or the External Entry Point (XEP) table in the upper 32K-word area. The PEP and XEP tables are system tables in which the operating system records entry points of procedures. The PEP table contains the entry points for procedures located in the current code segment. The XEP table contains the entry points of procedures located in other code segments.

For more information on public names and parameter pairs, see Section 17, “Mixed-Language Programming,” in the *TAL Programmer’s Guide*.

## Examples of Procedure Declarations

1. This example declares a function that has two formal parameters:

```
INT PROC mult (var1, var2);
    INT var1, var2;
BEGIN
    RETURN var1 * var2;
END;
```

2. This example shows a FORWARD declaration:

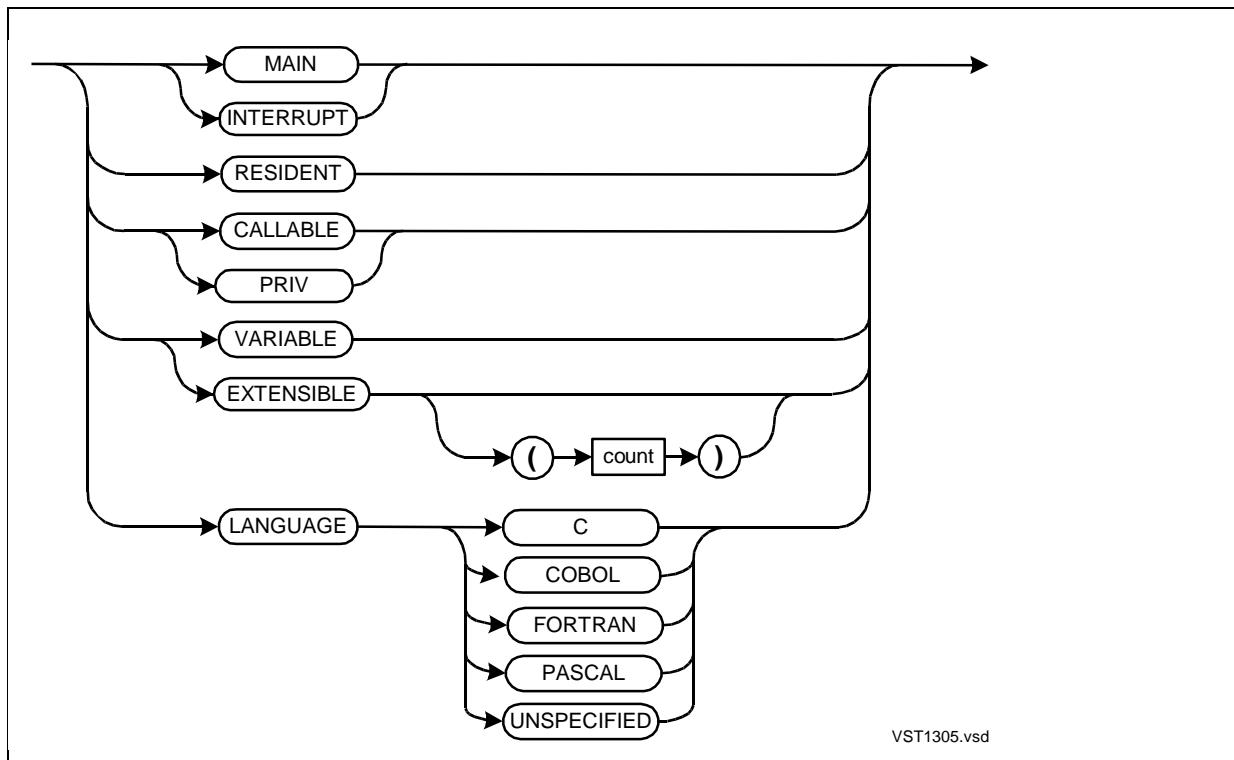
```
PROC to_come (param1);
    INT param1;
FORWARD;
```

3. This example shows EXTERNAL declarations:

```
PROC proc_a; EXTERNAL;
PROC proc_b; EXTERNAL;
```

# Procedure Attributes

Procedures can have the following attributes:



## MAIN

The MAIN attribute causes the procedure to execute first when you run the program. When the MAIN procedure completes execution, it passes control to the PROCESS\_STOP\_ system procedure, rather than executing an EXIT instruction.

If more than one procedure in a compilation has the MAIN attribute, the compiler emits a warning and uses the first MAIN procedure it sees as the main procedure. For example, in the following source code, procedures X and Y have the MAIN attribute, but in the object file only X has the MAIN attribute:

```

PROC main_proc1 MAIN;           !This procedure is MAIN in
BEGIN                         ! the object file
    CALL this_proc;
    CALL that_proc;
END;

PROC main_proc2 MAIN;           !This MAIN procedure is not
BEGIN                         ! MAIN in the object file
    CALL some_proc;
END;

```

## INTERRUPT

The INTERRUPT attribute causes the compiler to generate an IXIT (interrupt exit) instruction instead of an EXIT instruction at the end of execution. Only operating system interrupt handlers use the INTERRUPT attribute. An example is:

```
PROC int_handler INTERRUPT;
BEGIN
    !Do some work
END;
```

## RESIDENT

The RESIDENT attribute causes procedure code to remain in main memory for the duration of program execution. The operating system does not swap pages of this code. Binder allocates storage for resident procedures as the first procedures in the code space. An example is:

```
PROC res_proc RESIDENT;
BEGIN
    !Do some work
END;
```

## CALLABLE

The CALLABLE attribute authorizes a procedure to call a PRIV procedure (described next). Nonprivileged procedures can call CALLABLE procedures, which can call PRIV procedures. Thus, nonprivileged procedures can only access PRIV procedures indirectly by first calling CALLABLE procedures. Normally, only operating system procedures have the CALLABLE attribute. In the following example, a CALLABLE procedure calls the PRIV procedure declared next:

```
PROC callable_proc CALLABLE;
BEGIN
    CALL priv_proc;
END;
```

## PRIV

The PRIV attribute means the procedure can execute privileged instructions. Only PRIV or CALLABLE procedures can call a PRIV procedure. Normally, only operating system procedures have the PRIV attribute. PRIV protects the operating system from unauthorized (nonprivileged) calls to its internal procedures. For more information on privileged mode, see [Section 15, Privileged Procedures](#). The following PRIV procedure is called by the preceding CALLABLE procedure:

```
PROC priv_proc PRIV;
BEGIN
    !Privileged instructions
END;
```

## VARIABLE

The VARIABLE attribute means the compiler treats all parameters of the procedure (or subprocedure) as if they are optional, even if some are required by your code. If you add parameters to the VARIABLE procedure (or subprocedure) declaration, all procedures that call it must be recompiled. The following example declares a VARIABLE procedure:

```
PROC v (a, b) VARIABLE;
  INT a, b;
  BEGIN
    !Lots of code
  END;
```

When you call a VARIABLE procedure (or subprocedure), the compiler allocates space in the parameter area for all the parameters. The compiler also generates a parameter mask, which indicates which parameters are actually passed. The *TAL Programmer's Guide* describes parameter allocation, parameter masks, and testing for the presence of actual parameters.

## EXTENSIBLE

The EXTENSIBLE attribute lets you add new parameters to the procedure declaration without recompiling its callers. The compiler treats all parameters of the procedure as if they are optional, even if some are required by your code. The following example declares an EXTENSIBLE procedure:

```
PROC x (a, b) EXTENSIBLE;
  INT a, b;
  BEGIN
    !Do some work
  END;
```

When you call an EXTENSIBLE procedure, the compiler allocates space in the parameter area for all the parameters. The compiler also generates a parameter mask, which indicates which parameters are actually passed. The *TAL Programmer's Guide* describes parameter allocation, parameter masks, and testing for presence of actual parameters.

### Count Option

You use the *count* option following the EXTENSIBLE attribute only when you convert a VARIABLE procedure to an EXTENSIBLE procedure. The *count* value is the number of formal parameters in the VARIABLE procedure that you are converting to EXTENSIBLE. For the *count* value, specify an INT value in the range 1 through 15. For more information on VARIABLE-to-EXTENSIBLE conversions, see Section 11, Using Procedures, in the *TAL Programmer's Guide*.

## LANGUAGE

In a D-series EXTERNAL procedure declaration, you can use the LANGUAGE attribute to specify that the external routine is a C, COBOL, FORTRAN, or Pascal routine. If you do not know if the external routine is a C, COBOL, FORTRAN, or Pascal routine, you can use the LANGUAGE UNSPECIFIED option. The following example shows the LANGUAGE COBOL option and a public name "A-PROC" (in COBOL identifier format):

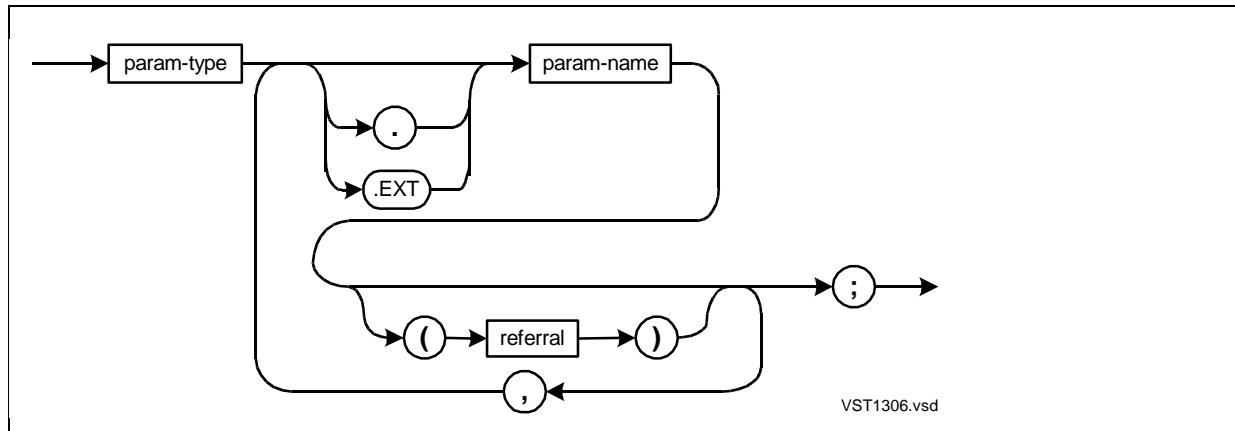
```
PROC a_proc = "a-proc" (a, b, c)      !EXTERNAL declaration
    LANGUAGE COBOL;                      ! for a COBOL procedure
    STRING .a, .b, .c;
    EXTERNAL;
```

Here are rules for using the LANGUAGE attribute:

- Use the LANGUAGE attribute only in a D-series EXTERNAL declaration.
- Specify no more than one LANGUAGE attribute in a declaration.
- For external TAL routines, omit the LANGUAGE attribute.

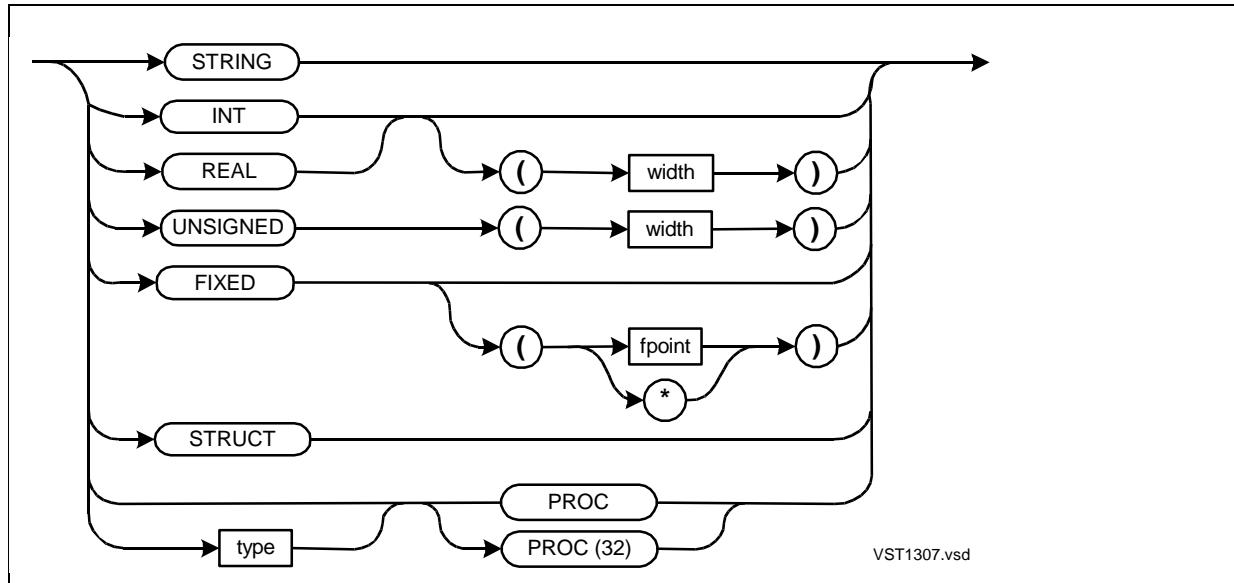
## Formal Parameter Specifications

A formal parameter specification defines the parameter type of a formal parameter and whether the parameter is a value or a reference parameter.



**param-type**

is the parameter type of the formal parameter. [Table 13-1](#) on page 13-12 lists the parameter types you can specify depending on the kind of actual parameter expected for this formal parameter. *param-type* can be one of:

**width**

is a constant expression that specifies the number of bits in the variable. The constant expression can include previously declared LITERALS and DEFINES. The result of the constant expression must be one of the following values:

<b>Data Type</b>	<b>width</b>
INT	16, 32, Or 64
REAL	32 or 64
UNSIGNED	A value in the range 1 through 31

UNSIGNED parameters must be passed by value; you cannot use an indirection symbol (.) or (.EXT) with UNSIGNED parameters.

**fpoint**

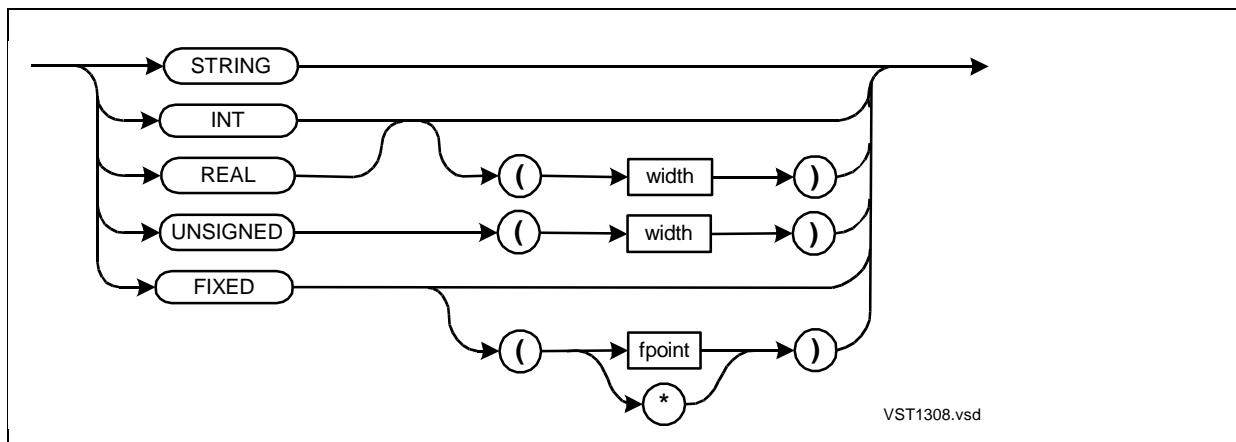
is an integer in the range –19 through 19 that specifies the implied decimal point position. The default is 0 (no decimal places). A positive *fpoint* specifies the number of decimal places to the right of the decimal point. A negative *fpoint* specifies a number of integer places to the left of the decimal point.

**\* (asterisk)**

prevents scaling of the *fpoint* of a FIXED actual parameter to match the *fpoint* in the parameter specification. Such scaling might cause loss of precision. The called procedure treats the actual parameter as having an *fpoint* of 0.

type

specifies that the parameter is a function procedure, the return value of which is one of the following data types:



## STRUCT

means the parameter is one of:

- A standard indirect or extended indirect definition structure (not supported in future software platforms)
  - A standard indirect or extended indirect referral structure

PROC

means the parameter is a 16-bit address that refers to one of:

- A C small-memory-model routine
  - A FORTRAN routine compiled with the NOEXTENDED directive
  - A TAL procedure or subprocedure

PROC(32)

means the parameter is a 32-bit address that refers to one of:

- A C large-memory-model routine
  - A FORTRAN routine compiled with the EXTENDED directive
  - A Pascal routine

Specify PROC(32) only in a D-series compilation unit.

- . (period)

is the standard indirection symbol. Its presence denotes a reference parameter that has a 16-bit address. Specify reference parameters for actual parameters that will be:

- Arrays
- Structures
- Simple variables (when you want to update the original value in the caller's scope)

An absence of any indirection symbol (. or .EXT) denotes 16-bit direct addressing.

.EXT

is the extended indirection symbol. Its presence denotes a reference parameter that has a 32-bit address. Specify reference parameters for actual parameters that are arrays or structures or for actual parameters that you want to update. An absence of any indirection symbol (. or .EXT) denotes 16-bit direct addressing.

param-name

is the identifier of a formal parameter. The identifier has local scope if declared in a procedure body or sublocal scope if declared in a subprocedure body.

referral

is the identifier of a previously declared structure or structure pointer. [Table 13-1](#) on page 13-12 lists the kind of actual parameter for which the formal parameter requires a *referral*.

## Usage Considerations

[Table 13-1](#) on page 13-12 lists the characteristics that you can declare in a formal parameter specification depending on the kind of actual parameter the procedure or subprocedure expects.

**Table 13-1. Formal Parameter Specification**

Formal Parameter Characteristics				
Expected Actual Parameter	Declare Formal Parameter As:	Parameter Type	Indirection Symbol	Referral
Simple variable	A value or reference parameter	STRING * INT INT (32) REAL REAL (64) FIXED (n) FIXED (*)	Value, no; reference, yes	No
Simple Variable	A value parameter	UNSIGNED	No	No
Array or simple pointer	A reference parameter	STRING INT INT (32) REAL REAL (64) FIXED (n)	Yes	No
Definition structure, referral structure, or structure pointer	A reference parameter	INT or STRING	Yes	Yes
Definition structure, * referral structure, or structure pointer	A reference parameter	STRUCT	Yes	Yes
Constant expression ** (including @identifier)	A value parameter	STRING INT INT (32) UNSIGNED REAL REAL (64) FIXED (n)	No	No
Procedure	A value parameter	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.

For more information on declaring, passing, and allocating parameters, see Section 11, Using Procedures, in the *TAL Programmer's Guide*.

For information on using the PROC(32) parameter type, see Section 17, Mixed-Language Programs, in the *TAL Programmer's Guide*.

## Examples of Formal Parameter Specification

1. This example shows a function that has two formal parameters, of which one is a value parameter and the other is a reference parameter. The compiler treats VAR1 as if it were a simple variable and treats VAR 2 as if it were a simple pointer:

```
INT PROC mult (var1, var2);
    INT var1,           !Declare value parameter
        .var2;          !Declare reference parameter
    BEGIN
        var2 := var2 + var1;      !Manipulate parameters
    END;
```

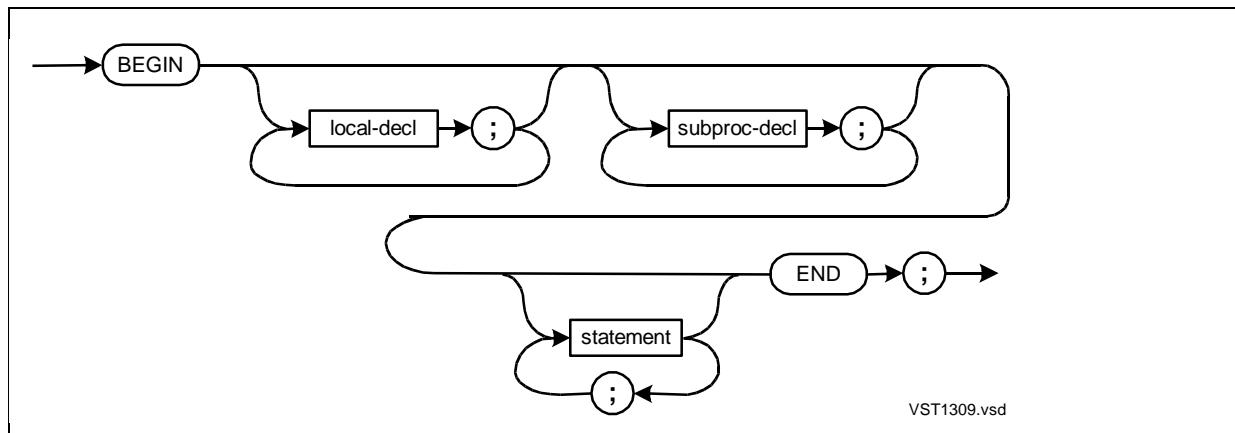
2. This example shows a procedure that declares a reference structure as a formal reference parameter:

```
STRUCT template (*);           !Template structure
BEGIN
    INT a;
    INT b;
END;

PROC p (ref_struct);
    STRUCT ref_struct (template);
    BEGIN
        !Lots of code
    END;
```

## Procedure Body

A procedure body can contain local declarations, subprocedure declarations, and statements.



local-decl

is a declaration for one of:

- Simple variable
- Array (direct, indirect, or read-only)
- Structure (direct or indirect)
- Simple pointer
- Structure pointer
- Equivalenced variable
- LITERAL
- DEFINE
- Label
- Entry point
- FORWARD subprocedure

subproc-decl

is a subprocedure declaration, as described in [Subprocedure Declaration](#) on page 13-15.

statement

is any statement described in [Section 12, Statements](#).

## Usage Consideration

Section 11, “Using Procedures,” in the *TAL Programmer’s Guide* describes:

- How the compiler allocates storage for procedures and their parameters
- How you call procedures

## Examples of Procedure Declarations

1. This example shows two procedures, the second of which calls the first:

```

INT c;                                ! Global declaration

PROC first;
  BEGIN                                     ! Procedure body
    INT a,                                ! Local declarations
    b;
    !Lots of code
  END;

PROC second;
  BEGIN                                     ! Procedure body
    !Lots of code
  CALL first;                               ! Call first procedure
    !More code
  END;

```

2. This example shows a FORWARD declaration for PROCB, a procedure that calls PROCB before its body is declared, and a declaration for the body of PROCB:

```

INT g2;

PROC procb (param1);           ! FORWARD declaration
    INT param1;
    FORWARD;

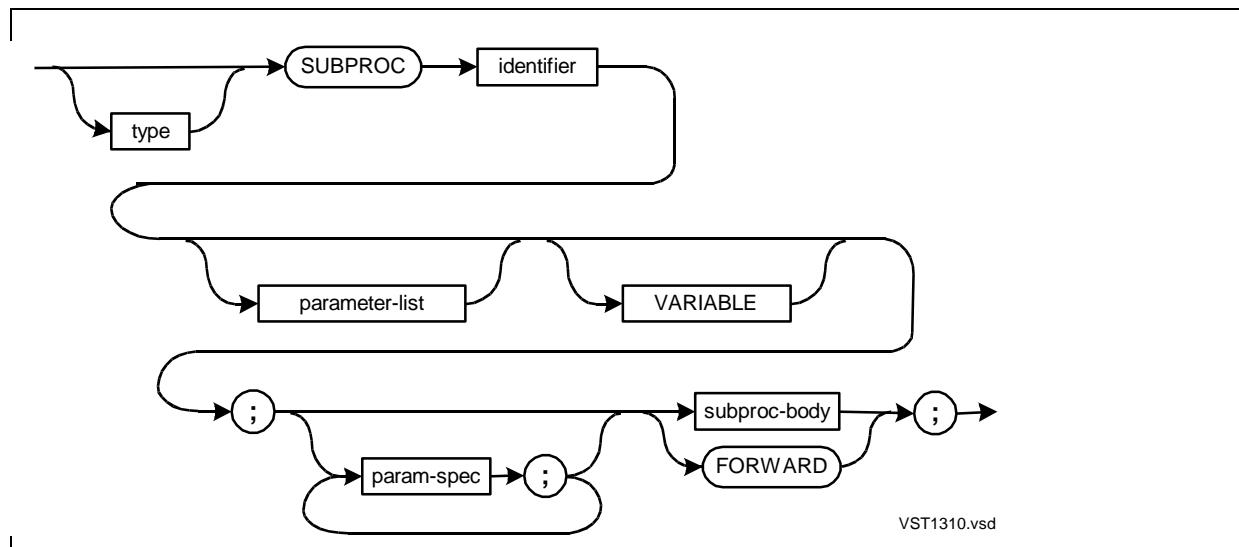
PROC proca;
BEGIN
    INT i1 := 2;
    CALL procb (i1);           ! Call PROCB
END;

PROC procb (param1);           ! Body for PROCB
    INT param1;
BEGIN
    g2 := g2 + param1;
END;

```

## Subprocedure Declaration

You can declare subprocedures within procedures, but not within subprocedures.



**type**

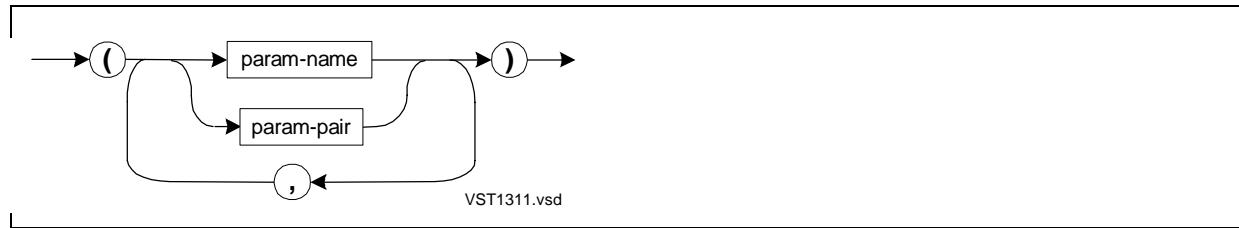
specifies that the subprocedure is a function that returns a result and indicates the data type of the returned result. **type** can be any data type described in [Section 3, Data Representation](#).

**identifier**

is the identifier of the subprocedure.

parameter-list

has the form:

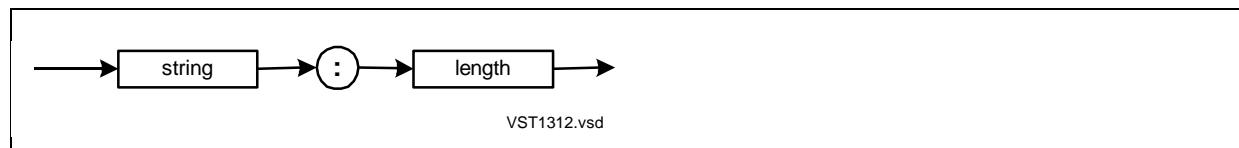


param-name

is the identifier of a formal parameter. The number of formal parameters a subprocedure can have is limited by space available in the parameter area of the subprocedure.

param-pair

is a pair of formal parameter identifiers that comprises of a language-independent string descriptor in the form:



string

is the identifier of a standard or extended STRING simple pointer. The actual parameter is the identifier of a STRING array or simple pointer declare inside or outside a structure.

length

is the identifier of a directly addressed INT simple variable. The actual parameter is an expression that specifies the length of *string*, in bytes.

VARIABLE

specifies that the compiler treats all parameters as optional, even if some are required by your code. The compiler ignores extra commas before or after the VARIABLE keyword.

param-spec

specifies the parameter type of a formal parameter and whether it is a value or reference parameter, as described in [Formal Parameter Specifications](#) on page 13-8.

subproc-body

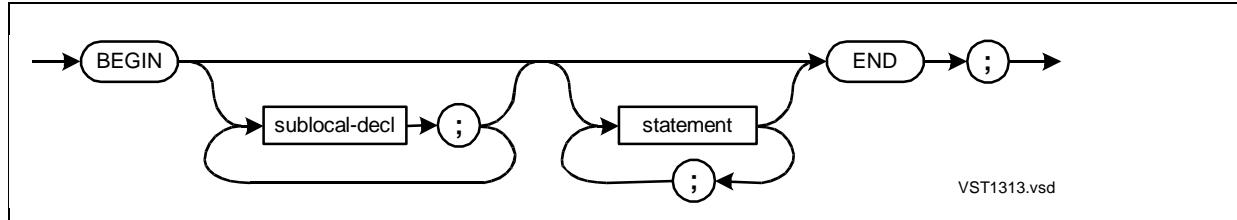
is a BEGIN-END construct that contains sublocal declarations and statements, as described in [Subprocedure Body](#) on page 13-17.

FORWARD

means the subprocedure body is declared later in this procedure.

## Subprocedure Body

A subprocedure body can contain sublocal declarations and statements.



sublocal-decl

is a declaration for one of:

- Simple variable
- Array (direct or read-only)
- Structure (direct only)
- Simple pointer
- Structure pointer
- Equivalenced variable
- LITERAL
- DEFINE
- Label
- Entry point

statement

is any statement described in [Section 12, Statements](#).

## Usage Considerations

Section 11, “Using Procedures,” in the *TAL Programmer’s Guide* describes:

- How the compiler allocates storage for subprocedures and their parameters
- How you call subprocedures

## Sublocal Variables

In subprocedures, declare pointers and directly addressed variables only. Here are examples:

Sublocal Variable	Example
Simple variable (which are always direct)	INT var;
Direct array	INT array[0:5];
Read-only array	INT ro_array = 'P' := [0,1,2,3,4,5];
Direct structure	STRUCT struct_a; BEGIN INT a, b, c; END;
Simple pointer	INT .simple_ptr;
Structure pointer	STRING .struct_ptr (struct_a);

If you use an indirection symbol (. or .EXT) with sublocal arrays and structures, the compiler allocates them as direct arrays and structures and issues a warning.

Because each subprocedure has only a 32-word area for variables, declare large arrays or structures at the global or local level and make them indirect.

## Example of Subprocedure Declaration

This example declares a function subprocedure:

```

PROC myproc;
BEGIN
  INT result;

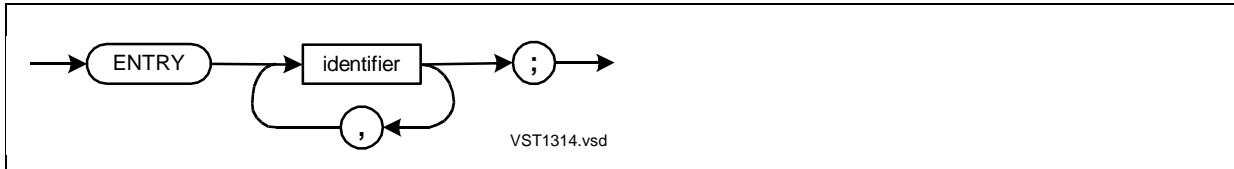
  INT SUBPROC mult (var1, var2); !Declare function
    INT var1, var2;           ! subprocedure
    BEGIN
      RETURN var1 * var2;
    END;                      ! End subprocedure

  result := mult(2, 3);
END;

```

## Entry-Point Declaration

The entry-point declaration associates an identifier with a secondary location in a procedure or subprocedure where execution can start.



identifier

is an entry-point identifier to be placed in the procedure or subprocedure body. It is an alternate or secondary point in the procedure or subprocedure at which to start executing.

## Usage Considerations

An entry point of a procedure or subprocedure is a location at which execution can start. The primary entry point is the procedure or subprocedure identifier. Secondary entry points are entry-point identifiers you declare and place within the procedure or subprocedure. Following are procedure entry-point guidelines, followed by subprocedure entry-point guidelines.

### Procedure Entry-Point Identifiers

Here are guidelines for using procedure entry-point identifiers:

- Declare all entry-point identifiers for a procedure within the procedure.
- Place each entry-point identifier and a colon (:) at a point in the procedure at which execution is to start.
- You can invoke a procedure entry-point identifier from anywhere in the program.
- (For functions, use the entry-point identifier in an expression; for other procedures, use a CALL statement.)
- Pass actual parameters as if you were calling the procedure identifier.
- You cannot use a GOTO statement to branch to a procedure entry-point identifier.
- To obtain the address of a procedure entry-point identifier, preface the identifier with @.
- You can specify FORWARD or EXTERNAL procedure entry-point declarations, which look like FORWARD or EXTERNAL procedure declarations, as shown in Example 2.

## Subprocedure Entry-Point Identifiers

Here are guidelines for using subprocedure entry-point identifiers:

- Declare all entry-point identifiers for a subprocedure within the subprocedure.
- Place each entry-point identifier and a colon (:) at a point in the subprocedure at which execution is to start.
- You invoke a subprocedure entry-point identifier from anywhere in the encompassing procedure, including from within the same subprocedure. (For functions, use the entry-point identifier in an expression; for other subprocedures, use a CALL statement.)
- Pass actual parameters as if you were calling the subprocedure identifier.
- You cannot use a GOTO statement to branch to a subprocedure entry-point identifier.
- To obtain the code address of a subprocedure entry-point identifier, preface the identifier with @.
- You can specify FORWARD subprocedure entry-point declarations, which look like FORWARD subprocedure declarations.

## Examples of Entry-Point Declarations

1. This example illustrates use of procedure entry-point identifiers:

```

INT to_this := 314;                      !Declare global data

PROC add_3 (g2);
    INT .g2;
BEGIN
    ENTRY add_2;                      !Declare entry-point
    ENTRY add_1;                      ! identifiers
    INT m2 := 1;
    g2 := g2 + m2;
add_2:                                     !Location of entry-point
    g2 := g2 + m2;                    ! identifier ADD_2
add_1:                                     !Location of entry-point
    g2 := g2 + m2;                    ! identifier ADD_1
END;

PROC mymain MAIN;                         !Main procedure
BEGIN
    CALL add_1 (to_this);            !Call entry point ADD_1
END;

```

2. This example shows FORWARD declarations for entry points:

```

INT to_this := 314;

PROC add_1 (g2);           ! FORWARD entry-point
    INT .g2;               ! identifier declaration
    FORWARD;

PROC add_2 (g2);           ! FORWARD entry-point
    INT .g2;               ! identifier declaration
    FORWARD;

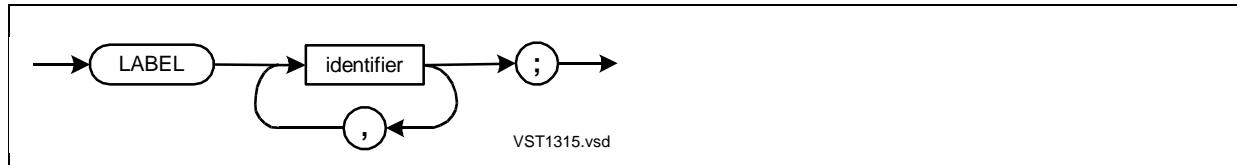
PROC add_3 (g2);           ! FORWARD procedure
    INT .g2;               ! declaration
    FORWARD;

PROC mymain MAIN;
    BEGIN
        CALL add_1 (to_this); ! Main procedure declaration
    END;

    PROC add_3 (g2);        ! Call entry-point identifier
        INT .g2;
        BEGIN
            ENTRY add_2;     ! Body for FORWARD procedure
            ENTRY add_1;     ! Declare entry-point
            INT m2 := 1;     ! identifiers
            g2 := g2 + m2;
            add_2:             ! Location of entry-point
            g2 := g2 + m2;   ! identifier ADD_2
            add_1:             ! Location of entry-point
            g2 := g2 + m2;   ! identifier ADD_1
        END;
    
```

## Label Declaration

The LABEL declaration reserves an identifier for later use as a label within the encompassing procedure or subprocedure.



**identifier**

is the identifier of the label. It cannot be a global declaration.

## Usage Considerations

Labels are the only declarable objects that you need not declare before using them. For best program management, however, declare all labels. For example, declaring a label helps ensure that you access the label, not a variable that has the same identifier.

### Local Labels

You can use a local label as follows:

1. Declare the label inside a procedure at the local level.
2. Place the label identifier and a colon (:) preceding a local statement in the same procedure.
3. Branch to the label by using local or sublocal GOTO statements in the same procedure.

Future software platforms require that you declare local labels that are referenced by sublocal GOTO statements.

### Sublocal Labels

You can use a sublocal label as follows:

1. Declare the label inside a subprocedure.
2. Place the label identifier and a colon (:) preceding a sublocal statement in the same subprocedure.
3. Branch to the label by using sublocal GOTO statements in the same subprocedure.

## Examples of Label Declarations

1. In this example, a local GOTO statement branches to a local label named ADDR:

```
PROC p;
BEGIN
  LABEL addr;                      !Declare label
  INT result;
  INT op1 := 5;
  INT op2 := 28;
addr:
  result := op1 + op2;             !Labeled statement
  op1 := op2 * 299;
  IF result < 100 THEN
    GOTO addr;                    !Branch to label
  !Some code
END;
```

2. In this example, sublocal GOTO statements branch to local labels A and B. The branch to declared label A is portable to future software platforms; the branch to undeclared label B is not portable:

```
PROC p;
BEGIN
LABEL a;           !Declare label A
INT i;

SUBPROC s;
BEGIN
!Lots of code
GOTO a;           !Branch is portable; label A is declared
GOTO b;           !Branch is not portable; label B is not
                  ! declared
!Lots of code
a :
i := 0;
!More code
b :
i := 1;
!Still more code
END;
```



# **14 Standard Functions**

TAL provides a variety of standard functions that perform frequently used operations. This section describes the syntax for calling each standard function.

For information on privileged standard functions—\$AXADR, \$BOUNDS, and \$SWITCHES—see [Section 15, Privileged Procedures](#).

## **Summary of Standard Functions**

Standard functions—also known as built-in functions—perform a variety of operations. The following table describes the operations by category:

<b>Category</b>	<b>Operation</b>
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 fixed-point setting, 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

Identifiers of standard functions begin with a dollar sign (\$).

[Table 14-1](#) on page 14-2 lists the standard functions by category and summarizes what they do. [Table 14-2](#) on page 14-4 cross-references the type-transfer functions by data type.

In both tables, INT and INT(32) expressions can also include operands of the following data types:

- INT expressions can include operands of data types STRING, INT, and UNSIGNED(1–16).
- INT(32) expressions can include operands of data types INT(32) and UNSIGNED(17–31).

For more information, see [Expression Arguments](#) on page 14-5.

**Table 14-1. Summary of Standard Functions** (page 1 of 3)

<b>Functional Group</b>	<b>Function Name</b>	<b>Description</b>
Type transfer	\$DBL	Converts an INT, FIXED(0), REAL, or REAL(64) expression to an INT(32) expression
	\$DBLL	Converts two INT expressions to an INT(32) expression
	\$DBLR	Converts an INT, FIXED(0), REAL, or REAL(64) expression to a rounded INT(32) expression
	\$DFIX	Converts an INT(32) expression to a FIXED( <i>fpoint</i> ) expression
	\$EFLT	Converts an INT, INT(32), FIXED( <i>fpoint</i> ), or REAL expression to a REAL(64) expression
	\$EFLTR	Converts an INT, INT(32), FIXED( <i>fpoint</i> ), or REAL expression to a rounded REAL(64) expression
	\$FIX	Converts an INT, INT(32), REAL, or REAL(64) expression to a FIXED(0) expression
	\$FIXD	Converts a FIXED(0) expression to an INT(32) expression
	\$FIXI	Converts a FIXED(0) expression to a signed INT expression
	\$FIXL	Converts a FIXED(0) expression to an unsigned INT expression
	\$FIXR	Converts an INT, INT(32), REAL, or REAL(64) expression to a rounded FIXED(0) expression
	\$FLT	Converts an INT, INT(32), FIXED( <i>fpoint</i> ), or REAL(64) expression to a REAL expression
	\$FLTR	Converts an INT, INT(32), FIXED( <i>fpoint</i> ), or REAL(64) expression to a rounded REAL expression
	\$HIGH	Converts the high-order 16 bits of an INT(32) expression to an INT expression
	\$IFIX	Converts a signed INT expression to a FIXED( <i>fpoint</i> ) expression
\$INT	\$INT	Converts the low-order 16 bits of an INT(32) or FIXED(0) expression to an INT expression; fully converts a REAL or REAL(64) expression to an INT expression
	\$INTR	Converts the low-order 16 bits of an INT(32) or FIXED(0) expression to an INT expression; fully converts a REAL or REAL(64) expression to a rounded INT expression

**Table 14-1. Summary of Standard Functions** (page 2 of 3)

<b>Functional Group</b>	<b>Function Name</b>	<b>Description</b>
Address conversion	\$LFIX	Converts an unsigned INT expression to a FIXED( <i>fpoint</i> ) expression
	\$UDBL	Converts an unsigned INT expression to an INT(32) expression
Character test	\$LADR	Converts an extended address to a standard address
	\$XADR	Converts a standard address to an extended address
Minimum-maximum	\$ALPHA	Tests an expression for an alphabetic character
	\$NUMERIC	Tests an expression for a numeric character
	\$SPECIAL	Tests an expression for a special character
Carry and overflow	\$LMAX	Returns the maximum of two unsigned INT expressions
	\$LMIN	Returns the minimum of two unsigned INT expressions
	\$MAX	Returns the maximum of two signed INT, INT(32), FIXED( <i>fpoint</i> ), REAL, or REAL(64) expressions
	\$MIN	Returns the minimum of two signed INT, INT(32), FIXED( <i>fpoint</i> ), REAL, or REAL(64) expressions
FIXED expression	\$CARRY	Tests the state of the carry indicator of the environment register
	\$OVERFLOW	Tests the state of the overflow indicator of the environment register
Variable	\$POINT	Returns an INT value that is the fixed-point setting of an expression
	\$SCALE	Moves the position of the implied decimal point in a stored FIXED( <i>fpoint</i> ) value
Variable	\$BITLENGTH	Returns an INT value that is the length, in bits, of a variable
	\$BITOFFSET	Returns an INT value that is the 0-relative bit offset of a structure item from the address of the zeroth structure occurrence
	\$LEN	Returns an INT value that is the unit length, in bytes, of a variable
	\$OCCURS	Returns an INT value that is the number of occurrences of a variable
	\$OFFSET	Returns an INT value that is the offset, in bytes, of a structure item from the address of the zeroth structure occurrence
	\$TYPE	Returns an INT value that indicates the data type of a variable

**Table 14-1. Summary of Standard Functions** (page 3 of 3)

<b>Functional Group</b>	<b>Function Name</b>	<b>Description</b>
Miscellaneous	\$ABS	Returns the absolute value of an expression
	\$COMP	Returns the one's complement of an INT expression
	\$OPTIONAL	Controls passing of a parameter to VARIABLE or EXTENSIBLE procedure in a D20 or later object file
	\$PARAM	Tests for the receipt of an actual parameter
	\$READCLOCK	Returns a FIXED value from an RCLK instruction
	\$RP	Returns an INT value that is the current setting of the compiler's internal register pointer
	\$USERCODE	Returns the content of a word in the current user code space

## Type-Transfer Functions

The type-transfer functions convert a variable of one data type into a variable of another data type.

Functions that convert an expression from a smaller data type to a larger data type perform a sign extension of the expression to the high bits. For example, \$DFIX returns a FIXED(0) expression from an INT(32) expression.

### Functions by Data Type

[Table 14-2](#) cross-references type-transfer functions by data type.

**Table 14-2. Type-Transfer Functions by Data Type** (page 1 of 2)

<b>Result</b>					
<b>Expression</b>	<b>INT</b>	<b>INT (32)</b>	<b>FIXED</b>	<b>REAL</b>	<b>REAL (64)</b>
INT	-	\$DBL	\$FIX	\$FLT	\$EFLT
		\$DBLR	\$FIXR	\$FLTR	\$EFLTR
		\$UDBL	\$IFIX		
			\$LFIX		
INT (32)	\$INT	-	\$DFIX	\$FLT	\$EFLT
	\$INTR		\$FIX	\$FLTR	\$EFLTR
	\$HIGH		\$FIXR		

**Table 14-2. Type-Transfer Functions by Data Type** (page 2 of 2)

Expression	Result				
	INT	INT (32)	FIXED	REAL	REAL (64)
FIXED	\$FIXI	\$DBL	\$FIX	\$FLT	\$EFLT
	\$FIXL	\$DBLR	\$FIXR	\$FLTR	\$EFLTR
	\$INT	\$FIXD			
	\$INTR				
REAL	\$INT	\$DBL	\$FIX	-	\$EFLT
	\$INTR	\$DBLR	\$FIXR		\$EFLTR
REAL (64)	\$INT	\$DBL	\$FIX	\$FLT	-
	\$INTR	\$DBLR	\$FIXR	\$FLTR	

## Rounding by Standard Functions

Type-transfer functions that have names ending in R, such as \$DBLR, round the result. All other type-transfer functions truncate the result.

The functions round values as follows:

(IF value < 0 THEN value - 5 ELSE value + 5) / 10

That is, if the result is negative, 5 is subtracted; if positive, 5 is added. Then, an integer division by 10 truncates the result. Therefore, if the absolute value of the least significant digit of the result after initial truncation is 5 or more, a one is added to the absolute value of the final least significant digit.

Rounding has no effect on INT, INT(32), or FIXED(0) expressions.

## Scope of Standard Functions

You can use the following standard functions at the global level because they return constant values:

- \$BITLENGTH
- \$BITOFFSET
- \$LEN
- \$OCCURS
- \$OFFSET
- \$TYPE

You can use all other standard functions only at the local or sublocal levels.

## Expression Arguments

Many standard functions accept expressions as arguments. The following guidelines regarding the data type and signedness of expressions apply to all standard functions that accept expressions as arguments.

## Data Types of Expression Arguments

Expressions can be any data type except STRING and UNSIGNED. INT and INT(32) expressions, however, can include such operands as follows. In any other expressions, all operands must be of the same data type.

### INT Expressions

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:

- Places a STRING operand in the right byte of a word and sets the left byte to 0.
- Places an UNSIGNED(1–16) operand in the right bits of a word and sets the unused left bits to 0.

### INT(32) Expressions

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. Before evaluating the expression, the system places an UNSIGNED(17–31) operand in the right bits of a doubleword and sets the unused left bits to 0.

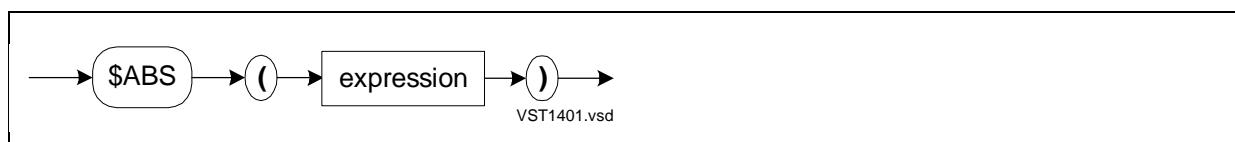
## Signedness of Expression Arguments

The standard function, not the expression or its data type, dictates the signedness or unsignedness of its argument.

For instance, standard functions that expect signed arguments treat unsigned expressions as if they were signed. Conversely, standard functions that expect unsigned arguments treat signed expressions as if they were unsigned.

## \$ABS Function

The \$ABS function returns the absolute value of an expression. The returned value has the same data type as the expression.



expression

is any expression.

## Usage Considerations

If the absolute value of a negative INT, INT(32), or FIXED expression cannot be represented in two's complement form, \$ABS sets the overflow indicator. For example, if X has the INT value -32,768, \$ABS(X) causes an arithmetic overflow.

### Example of \$ABS Function

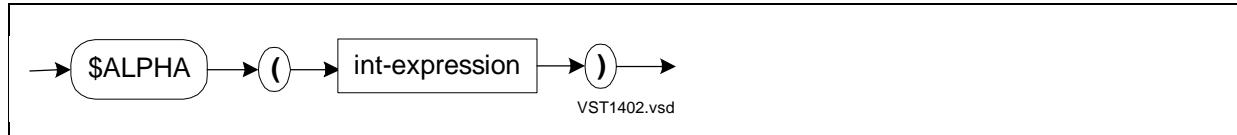
In this example, \$ABS returns an absolute value from an INT expression:

```
INT int_val := -5;           !Declare variables
INT abs_val;

abs_val := $ABS(int_val);   !Return 5, the absolute value
                           ! of -5
```

## \$ALPHA Function

The \$ALPHA function tests the right byte of an INT value for the presence of an alphabetic character.



int-expression  
is an INT expression.

### Usage Considerations

\$ALPHA inspects bits <8:15> of *int-expression* and ignores bits <0:7>. It tests for an alphabetic character according to the following criteria:

*int-expression* >= "A" AND *int-expression* <= "Z" OR

*int-expression* >= "a" AND *int-expression* <= "z"

If an alphabetic character occurs, \$ALPHA sets the condition code indicator to CCE (condition code equal to). If you plan to check the condition code, do so before an arithmetic operation or assignment occurs.

If the character passes the test, \$ALPHA returns a -1 (true); otherwise, it returns a 0 (false).

*int-expression* can include STRING and UNSIGNED(1–16) operands, as described in [Expression Arguments](#) on page 14-5.

## Example of \$ALPHA Function

In this example, \$ALPHA tests for an alphabetic character in a STRING argument:

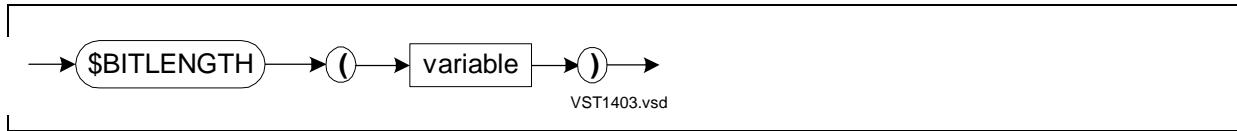
```
STRING some_char;           !Declare variable
IF $ALPHA (some_char) THEN ... ;   !Test for alphabetic
                                  ! character
```

## \$AXADR Function

The \$AXADR function is described in [Section 15, Privileged Procedures](#).

## \$BITLENGTH Function

The \$BITLENGTH function returns the length, in bits, of a variable.



**variable**

is the identifier of a simple variable, array element, pointer, structure, or structure data item.

## Usage Considerations

\$BITLENGTH returns the length, in bits, of a single occurrence of a simple variable, array element, structure, structure data item, or item to which a pointer points.

The length of a structure or substructure occurrence is the sum of the lengths of all items contained in the structure or substructure. Complete the structure before you use \$BITLENGTH to obtain the length of any of the items in the structure.

To compute the total number of bits in an entire array or substructure, multiply the value returned by \$BITLENGTH by the value returned by \$OCCURS. To compute the total number of bits in a structure, first round up the value returned by \$BITLENGTH to the word boundary and then multiply the rounded value by the value returned by \$OCCURS.

You can use \$BITLENGTH in LITERAL expressions and global initializations, because it always returns a constant value.

## Example of \$BITLENGTH Function

In this example, \$BITLENGTH returns the length, in bits, of one occurrence of structure S:

```

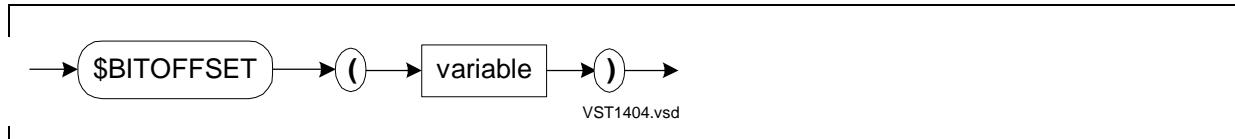
INT s_len;                      !Declare variable
STRUCT .s[0:3];                 !Declare four occurrences of a
BEGIN                           ! structure
UNSIGNED(1) flags[0:15];
UNSIGNED(2) status;
BIT_FILLER 14;
END;

s_len := $BITLENGTH (s);        !Return 32, the number of bits
                                ! in one structure occurrence

```

## \$BITOFFSET Function

The \$BITOFFSET function returns the number of bits from the address of the zeroth structure occurrence to a structure data item.



*variable*

is the fully qualified identifier of a structure data item.

## Usage Considerations

The zeroth structure occurrence has an offset of 0. For items other than substructure, simple variable, array, or pointer declared within a structure, \$BITOFFSET returns a 0.

When you qualify the identifier of *variable*, you can use constant indexes but not variable indexes; for example:

```
$BITOFFSET (struct1.subst[1].item) !1 is a constant index
```

To find the offset of an item in a structure, complete the structure before you use \$BITOFFSET.

You can use \$BITOFFSET in LITERAL expressions and global initializations, because it always returns a constant value.

## Example of \$BITOFFSET Function

In this example, \$BITOFFSET returns the offset (in bits) of the third occurrence of substructure AB:

```

STRUCT a;                                !Declare structure
BEGIN
    INT array[0:40];
    STRUCT ab[0:9];                      !Declare substructure AB
    BEGIN                                   ! with ten occurrences
        UNSIGNED(1) flag;
        UNSIGNED(15) offset;
    END;
END;

INT c;
c := $BITOFFSET (a.ab[2]);                !Return offset of third
                                         ! occurrence of AB

```

## \$BOUNDS Function

The \$BOUNDS function is described in [Section 15, Privileged Procedures](#).

## \$CARRY Function

The \$CARRY function checks the state of the carry bit in the environment register and indicates whether a carry out of the high-order bit position occurred.



## Usage Considerations

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 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 following the operation that affects the carry indicator. If the carry indicator is on, \$CARRY is -1 (true). If the carry indicator is off, \$CARRY is 0 (false).

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

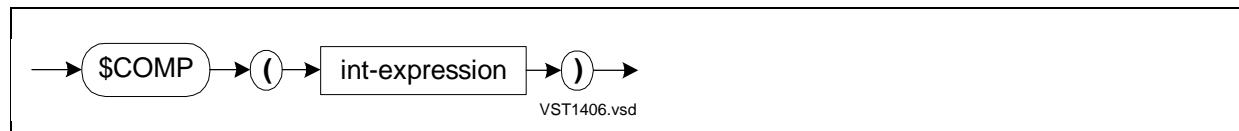
## Example of \$CARRY Function

In this example, \$CARRY tests the state of the carry bit after addition:

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

## \$COMP Function

The \$COMP function obtains the one's complement of an INT expression.



int-expression  
is an INT expression.

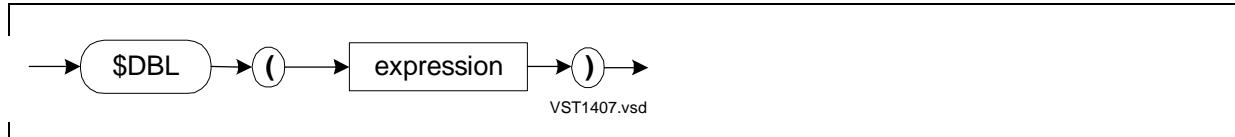
## Example of \$COMP Function

In this example, \$COMP returns a value equal to the one's complement of 10:

```
INT some_int;                      !Declare variable
some_int := $COMP (10);           !Return -11
```

## \$DBL Function

The \$DBL function returns an INT(32) value from an INT, FIXED(0), REAL, or REAL(64) expression.



*expression*

is an INT, FIXED(0), REAL, or REAL(64) expression.

## Usage Consideration

If *expression* is too large in magnitude to be represented by a 32-bit two's complement integer, \$DBL sets the overflow indicator.

## Example of \$DBL Function

In this example, \$DBL returns an INT(32) value from an INT expression:

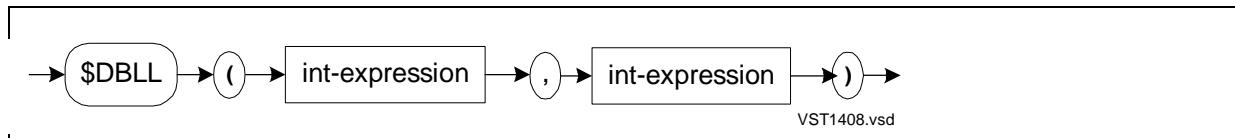
```

INT i2 := %177775;           !Declare variables
INT(32) b32;
b32 := $DBL (i2);           !Return -3D

```

## \$DBLL Function

The \$DBLL function returns an INT(32) value from two INT values.



*int-expression*

is an INT expression.

## Usage Consideration

To form the INT(32) value, \$DBLL places the first *int-expression* in the high-order 16 bits and the second *int-expression* in the low-order 16 bits.

## Examples of \$DBLL Function

1. In this example, \$DBLL returns an INT(32) value formed from two INT variables, FIRST\_INT and SECOND\_INT:

```
INT first_int, second_int;      !Declare variables
INT(32) some_double;

some_double := $DBLL (first_int, second_int);
                !Return INT(32) value
```

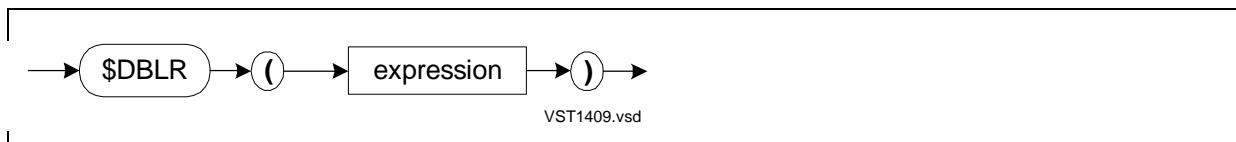
2. In this example, \$DBLL returns an extended (32-bit) address from two INT constants that represent a standard (16-bit) address in the current user code segment:

```
INT .EXT p;                      !Declare 32-bit simple
                                    ! pointer

@p := ($DBLL (2, 7)) '<<' 1;    !Return 32-bit address in
                                    ! user code segment
```

## \$DBLR Function

The \$DBLR function returns an INT(32) value from an INT, FIXED(0), REAL, or REAL(64) expression and applies rounding to the result.



*expression*

is an INT, FIXED(0), REAL, or REAL(64) expression.

## Usage Consideration

If *expression* is too large in magnitude to be represented by a 32-bit two's complement integer, \$DBLR sets the overflow indicator.

## Examples of \$DBLR Function

1. In this example, \$DBLR returns a rounded INT(32) value from a REAL expression:

```
REAL r2 := 1.5e0;          !Declare variables
INT(32) b32;

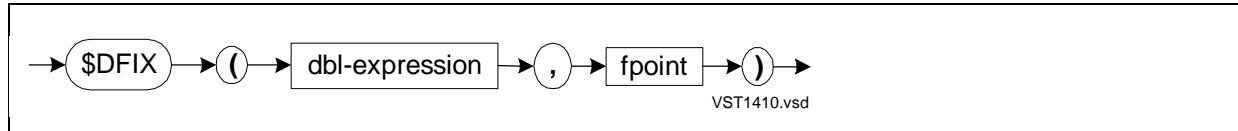
b32 := $DBLR (r2);        !Return 2D
```

2. Here is another example:

```
REAL realnum := 123.456E0;           !Declare variables
INT(32) dblnum;
dblnum := $DBLR (realnum);          !Return 123D
```

## \$DFIX Function

The \$DFIX function returns a FIXED(*fpoint*) expression from an INT(32) expression.



*dbl-expression*

is an INT(32) arithmetic expression.

*fpoint*

is a value in the range –19 through +19 that specifies the position of the implied decimal point in the result. A positive *fpoint* specifies the number of decimal places to the right of the decimal. A negative *fpoint* specifies the number of integer places to the left of the decimal point.

## Usage Consideration

\$DFIX converts an INT(32) expression to a FIXED(*fpoint*) expression by performing the equivalent of a signed right shift of 32 positions from the left 32 bits into the right 32 bits of a quadrupleword unit.

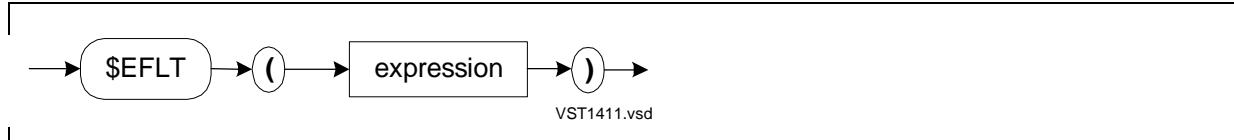
## Example of \$DFIX Function

In this example, \$DFIX returns a FIXED(2) value from an INT(32) expression and an *fpoint* of 2:

```
FIXED(2) fixnum;                      !Declare variables
INT(32) dblnum := -125D;
fixnum := $DFIX (dblnum, 2);           !Return -1.25
```

# \$EFLT Function

The \$EFLT function returns a REAL(64) value from an INT, INT(32), FIXED(*fpoint*), or REAL expression.



expression

is an INT, INT(32), FIXED(*fpoint*), or REAL expression.

## Usage Consideration

If a FIXED expression has a nonzero *fpoint*, the compiler multiplies or divides the result by the appropriate power of ten.

## Example of \$EFLT Function

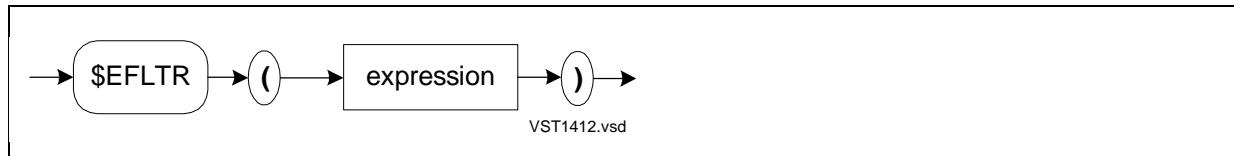
In this example, \$EFLT returns a REAL(64) value from a FIXED(3) expression:

```

REAL(64) dblnum;           !Declare variables
FIXED(3) fixnum := 12345.678F;
dblnum := $EFLT (fixnum);   !Return 12345678L-3
  
```

# \$EFLTR Function

The \$EFLTR function returns a REAL(64) value from an INT, INT(32), FIXED(*fpoint*), or REAL expression and applies rounding to the result.



expression

is an INT, INT(32), FIXED(*fpoint*), or REAL expression.

## Usage Considerations

If a FIXED expression has a nonzero *fpoint*, the compiler multiplies or divides the result by the appropriate power of ten.

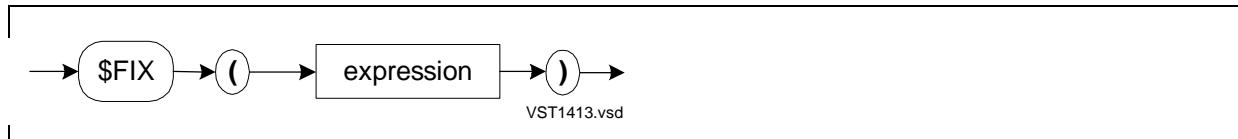
## Example of \$EFLTR Function

In this example, \$EFLTR returns a rounded REAL(64) value from a FIXED(3) expression:

```
REAL(64) rndnum;           !Declare variables
FIXED(3) fixnum := 12345.678F;
rndnum := $EFLTR (fixnum); !Return rounded REAL(64)
                           ! value
```

## \$FIX Function

The \$FIX function returns a FIXED(0) value from an INT, INT(32), REAL, or REAL(64) expression.



*expression*

is an INT, INT(32), FIXED, REAL, or REAL(64) expression.

## Usage Consideration

If *expression* is too large in magnitude to be represented by a 64-bit two's complement integer, \$FIX sets the overflow indicator.

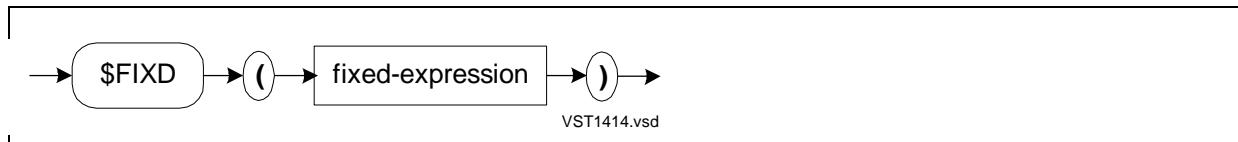
## Example of \$FIX Function

In this example, \$FIX returns a FIXED(0) value from an INT expression:

```
FIXED fixnum;           !Declare variables
INT intnum := 5;
fixnum := $FIX (intnum); !Return 5F
```

## \$FIXD Function

The \$FIXD function returns an INT(32) value from a FIXED(0) expression.



fixed-expression

is a FIXED expression, which \$FIXD treats as a FIXED(0) expression, ignoring any implied decimal point.

## Usage Consideration

If the result cannot be represented in a signed doubleword, \$FIXD sets the overflow indicator.

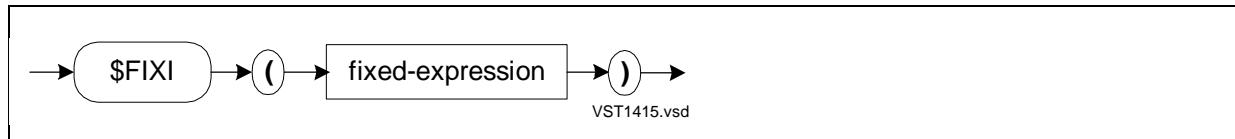
## Example of \$FIXD Function

In this example, \$FIXD returns an INT(32) value from a FIXED(0) expression:

```
INT( 32 ) dblnum;           ! Declare variables
FIXED fixnum := 1234F;
dblnum := $FIXD (fixnum);    ! Return 1234D
```

## \$FIXI Function

The \$FIXI function returns the signed INT equivalent of a FIXED(0) expression.



fixed-expression

is a FIXED expression, which \$FIXI treats as a FIXED(0) expression, ignoring any implied decimal point.

## Usage Considerations

If the result cannot be represented in a signed 16-bit integer, \$FIXI sets the overflow indicator.

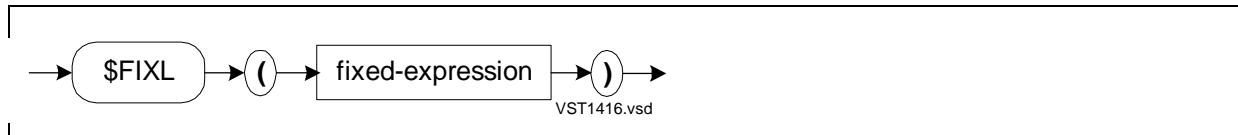
## Example of \$FIXI Function

In this example, \$FIXI returns a signed INT equivalent value from a FIXED(0) expression:

```
INT intnum;           ! Declare variables
FIXED fixnum := %177777F;
intnum := $FIXI (fixnum);    ! Return -1
```

# \$FIXL Function

The \$FIXL function returns the unsigned INT equivalent of a FIXED(0) expression.



**fixed-expression**

is a FIXED expression, which \$FIXL treats as a FIXED(0) expression, ignoring any implied decimal point.

## Usage Considerations

If the result cannot be represented in an unsigned 16-bit integer, \$FIXL sets the overflow indicator.

## Examples of \$FIXL Function

In this example, \$FIXL returns an unsigned INT equivalent value from a FIXED(0) expression:

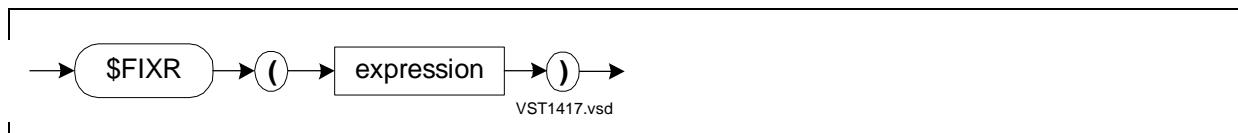
```

INT intnum;                                !Declare variables
FIXED fixnum := 32767F;
intnum := $FIXL (fixnum);                  !Return 32,767

```

# \$FIXR Function

The \$FIXR function returns a FIXED(0) value from an INT, INT(32), FIXED, REAL, or REAL(64) expression and applies rounding to the result.



**expression**

is an INT, INT(32), FIXED, REAL, or REAL(64) expression.

## Usage Considerations

If *expression* is too large in magnitude to be represented by a 64-bit two's complement integer, \$FIXR sets the overflow indicator.

## Example of \$FIXR Function

1. In this example, \$FIXR returns a rounded FIXED(0) value from a REAL(64) expression:

```
FIXED rfixnum;                      !Declare variables
REAL(64) bigrealnum := -1.5L0;
rfixnum := $FIXR (bigrealnum);       !Return -1F
```

2. In this example, \$FIXR returns a rounded FIXED(0) value from a REAL expression:

```
FIXED rndfnum;                      !Declare variables
REAL realnum := 123.456E0;
rndfnum := $FIXR (realnum);          !Return 123F
```

## \$FLT Function

The \$FLT function returns a REAL value from an INT, INT(32), FIXED(*fpoint*), or REAL(64) expression.



expression

is an INT, INT(32), FIXED(*fpoint*), or REAL(64) expression.

## Usage Consideration

If a FIXED expression has a nonzero *fpoint*, the compiler multiplies or divides the result by the appropriate power of ten.

## Example of \$FLT Function

In this example, \$FLT returns a REAL value from an INT(32) expression:

```
REAL realnum;                      !Declare variables
INT(32) dblnum := 147D;
realnum := $FLT (dblnum);          !Return 147E0
```

# \$FLTR Function

The \$FLTR function returns a REAL value from an INT, INT(32), FIXED(*fpoint*), or REAL(64) expression and applies rounding to the result.



**expression**

is an INT, INT(32), FIXED(*fpoint*), REAL, or REAL(64) expression.

## Usage Consideration

If a FIXED expression has a nonzero *fpoint*, the compiler multiplies or divides the result by the appropriate power of ten.

## Example of \$FLTR Function

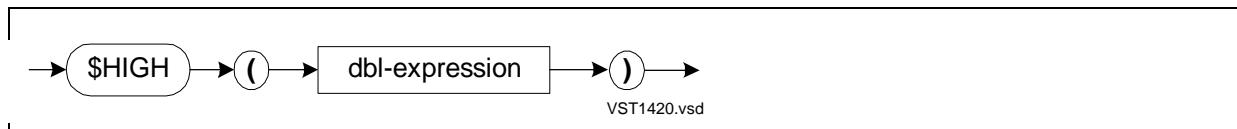
In this example, \$FLTR returns a rounded REAL value from an INT(32) expression:

```

REAL rrlnum;                                !Declare variables
INT(32) dblnum := 147D;
rrlnum := $FLTR (dblnum);                  !Return rounded REAL value
  
```

# \$HIGH Function

The \$HIGH function returns an INT value that is the high-order 16 bits of an INT(32) expression.



**dbl-expression**

is an INT(32) expression.

## Example of \$HIGH Function

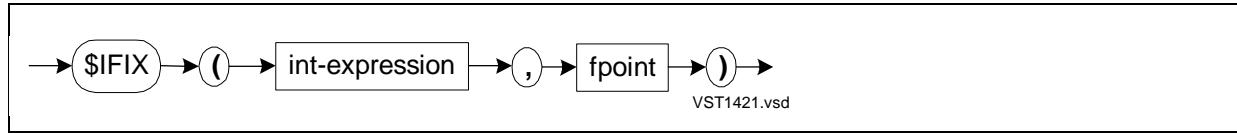
In this example, \$HIGH returns the high-order word of an INT(32) expression:

```

INT intnum;                                !Declare variables
INT(32) dblnum := 65538D;
intnum := $HIGH (dblnum);                  !Return 1
  
```

# \$IFIX Function

The \$IFIX function returns a FIXED( *fpoint*) value from a signed INT expression.



*int-expression*

is a signed INT expression.

*fpoint*

is a value in the range –19 through +19 that specifies the position of the implied decimal point in the result. A positive *fpoint* specifies the number of decimal places to the right of the decimal. A negative *fpoint* specifies the number of integer places to the left of the decimal point.

## Usage Consideration

When \$IFIX converts the signed INT expression to a FIXED value, it performs the equivalent of a signed right shift of 48 positions in a quadrupleword unit.

## Example of \$IFIX Function

In this example, \$IFIX returns a FIXED(2) value from a signed INT expression and an *fpoint* of 2:

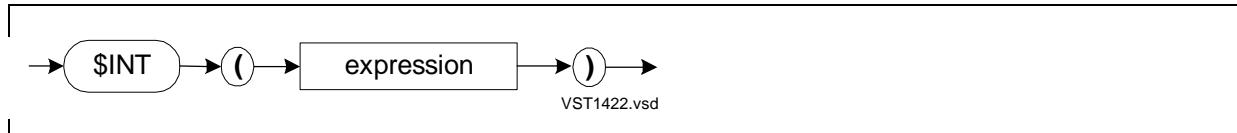
```

FIXED(2) fixnum;                                !Declare variables
INT intnum := 12345;
fixnum := $IFIX (intnum, 2);                      !Return 123.45

```

# \$INT Function

The \$INT function returns an INT value from the low-order 16 bits of an INT(32 or FIXED(0) expression. \$INT returns a fully converted INT expression from a REAL or REAL(64) expression.



*expression*

is an INT, INT(32), FIXED(0), REAL, or REAL(64) expression.

## Usage Considerations

If *expression* is INT, INT(32), or FIXED(0), \$INT returns the low-order (least significant) 16 bits and does not explicitly maintain the sign. No overflow occurs.

If *expression* is REAL or REAL(64), \$INT returns a fully converted INT value, not a truncation. If the converted value of *expression* is too large to be represented by a 16-bit two's complement integer, an overflow trap occurs.

## Examples of \$INT Function

1. In this example, \$INT returns the low-order word of an INT(32) expression:

```
INT a16;                                ! Declare variables
INT(32) a32 := 65538D;

a16 := $INT (a32);                      ! Return 2, the low-order word
                                            ! of an INT(32) expression
```

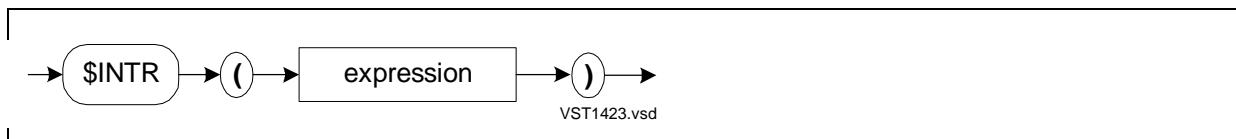
2. In this example, \$INT returns a fully converted INT value from a REAL expression:

```
INT intnum;                                ! Declare variables
REAL realnum := 20.0E-1;

intnum := $INT (realnum);                  ! Return 2, the fully converted
                                            ! INT value from a REAL
                                            ! expression
```

## \$INTR Function

The \$INTR function returns an INT value from the low-order 16 bits of an INT(32) or FIXED(0) expression. \$INTR returns a fully converted and rounded INT expression from a REAL or REAL(64) expression.



*expression*

is an INT, INT(32), FIXED(0), REAL, or REAL(64) expression.

## Usage Considerations

If *expression* is type INT, INT(32) or FIXED(0), \$INTR returns the low-order (least significant) 16 bits and does not explicitly maintain the sign. No overflow occurs.

If *expression* is type REAL or REAL(64), \$INTR returns a fully converted and rounded INT value, not a truncation. If the converted value of *expression* is too large to be represented by a 16-bit two's complement integer, an overflow trap occurs.

## Example of \$INTR Function

In this example, \$INTR returns a fully converted and rounded INT value from a REAL expression:

```
INT rndnum;                                !Declare variables
REAL realnum := 12345E-2;
rndnum := $INTR (realnum);                  !Return 123
```

## \$LADR Function

The \$LADR function returns the standard (16-bit) address of a variable that is accessed through an extended (32-bit) pointer.



*variable*

is the identifier of a variable accessed through an extended pointer.

## Usage Considerations

If *variable* is a STRING variable or a substructure, \$LADR returns a standard byte address. Otherwise, \$LADR returns a standard word address.

When \$LADR converts the extended address to a standard address, it loses the segment number in the extended address:

- If the extended address is in the user data segment, the converted address is correct.
- If the extended address is in an extended data segment, the converted address is incorrect.

(For a description of the extended address format, see Appendix B in the *TAL Programmer's Guide*.)

\$LADR is not portable to future software platforms.

## Example of \$LADR Function

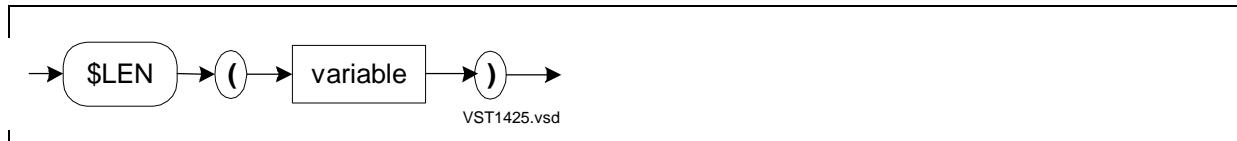
In this example, \$LADR returns a standard (16-bit) address from an extended (32-bit) address:

```
STRING .ptr;                                !Declare variables
STRING .EXT array[0:99];
STRING .EXT xptr := @array[0];

@ptr := $LADR (xptr);                      !Return 16-bit address
                                              ! from 32-bit address
```

## \$LEN Function

The \$LEN function returns the length, in bytes, of one occurrence of a variable.



**variable**

is the identifier of a simple variable, array element, pointer, structure, or structure data item.

## Usage Considerations

\$LEN returns the number of bytes contained in a single occurrence of a simple variable, array element, structure, structure data item, or item pointed to by a pointer.

To compute the total number of bytes in an entire array or substructure, multiply the value returned by \$LEN by the value returned by \$OCCURS. To compute the total number of bytes in an entire structure, first round up the value returned by \$LEN to a word boundary and then multiply the rounded value by the value returned by \$OCCURS.

If you apply \$LEN to an unfinished structure or to a substructure in an unfinished structure, the compiler emits warning 76 (cannot use \$OFFSET or \$LEN until base structure is complete).

You can use \$LEN in LITERAL expressions and global initializations, because it always returns a constant value.

## Examples of \$LEN Function

1. In this example, \$LEN returns the number of bytes in an array element:

```
INT b;                                !Declare variables
INT a [0:11];
b := $LEN (a);                         !Return 2
```

2. In this example, \$LEN returns the number of bytes in one occurrence of a structure:

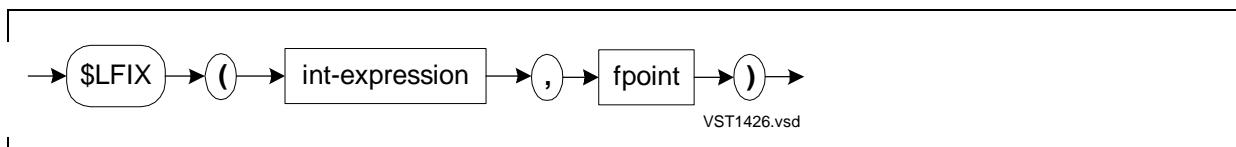
```
INT s_len;
STRUCT .s[0:99];                      !Declare 100 occurrences of
BEGIN                                     ! a structure
  INT(32) array[0:2];
END;
s_len := $LEN (s);                     !Return 12
```

3. In this example, \$OCCURS returns the number of elements in an array, and \$LEN returns the number of bytes in each element. This example multiplies 3 times 4 (the values returned by \$OCCURS and \$LEN) and yields the number of bytes in the entire array:

```
INT array_length;                      !Declare variables
INT(32) array[0:2];
array_length := $LEN (array) * $OCCURS (array);
                                         !Return 12, the length of the
                                         !entire array in bytes
```

## \$LFIX Function

The \$LFIX function returns a FIXED(*fpoint*) expression from an unsigned INT expression.



*int-expression*

is an unsigned INT expression.

*fpoint*

is a value in the range –19 through +19 that specifies the position of the implied decimal point in the result. A positive *fpoint* specifies the number of decimal places

to the right of the decimal. A negative *fpoint* specifies the number of integer places to the left of the decimal point.

## Usage Consideration

**\$LFIX** places the INT value in the low-order (least significant) word of the quadrupleword and sets the three high-order (most significant) words to 0.

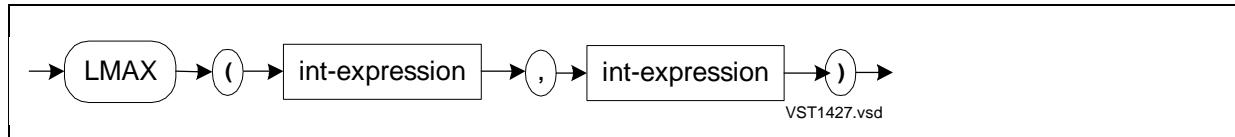
### Example of **\$LFIX** Function

In this example, **\$LFIX** returns a FIXED(2) value from an unsigned INT expression and an *fpoint* of 2:

```
FIXED(2) fixnum;                                !Declare variables
INT intnum := 125;
fixnum := $LFIX (intnum, 2);                      !Return 1.25
```

## \$LMAX Function

The **\$LMAX** function returns the maximum of two unsigned INT expressions.



int-expression  
is an unsigned INT arithmetic expression.

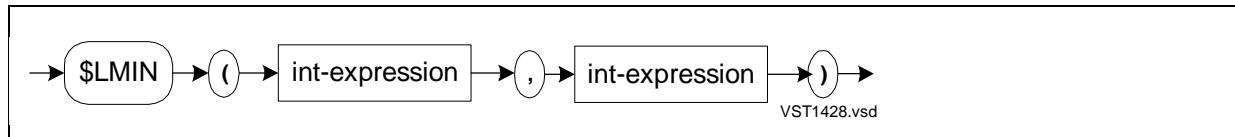
### Example of **\$LMAX** Function

In this example, **\$LMAX** compares an unsigned INT expression and a constant and returns the maximum value:

```
INT intval := 3;
max := $LMAX (intval, 5);                         !Return 5
```

## \$LMIN Function

The **\$LMIN** function returns the minimum of two unsigned INT expressions.



int-expression  
is an unsigned INT arithmetic expression.

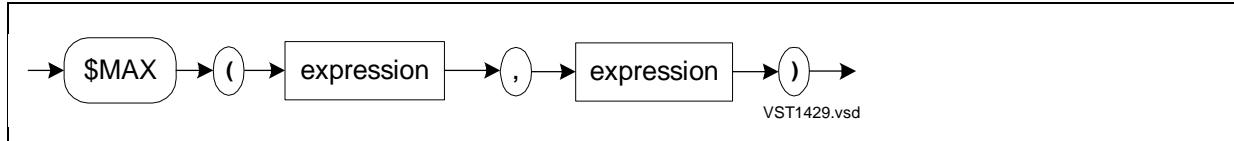
## Example of \$LMIN Function

In this example, \$LMIN returns the minimum of an unsigned INT expression and a constant:

```
INT intval := 3;
min := $LMIN (intval, 5);      !Return 3
```

## \$MAX Function

The \$MAX function returns the maximum of two signed INT, INT(32), FIXED(*fpoint*), REAL, or REAL(64) expressions.



expression  
is a signed INT, INT(32), FIXED(*fpoint*), REAL, or REAL(64) expression. Both expressions must be of the same data type.

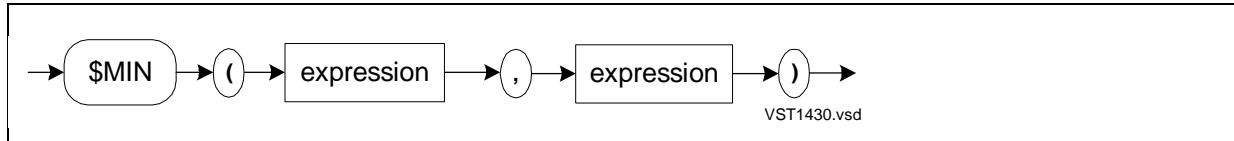
## Example of \$MAX Function

In this example, \$MAX returns the maximum of a signed REAL expression and a constant:

```
REAL realval := -3E0;
max := $MAX (realval, 5E0);      !Return 5E0
```

## \$MIN Function

The \$MIN function returns the minimum of two INT, INT(32), FIXED(*fpoint*), REAL, or REAL(64) expressions.



expression  
is an INT, INT(32), FIXED(*fpoint*), REAL, or REAL(64) expression. Both expressions must be of the same data type.

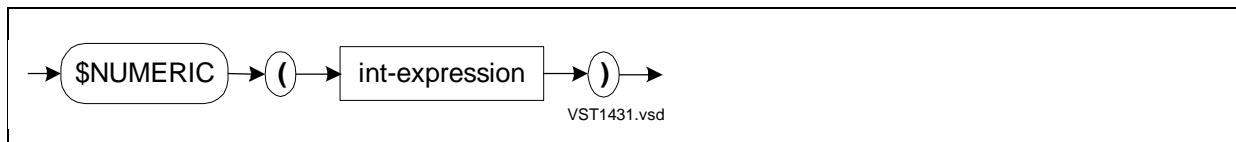
## Example of \$MIN Function

In this example, \$MIN returns the minimum of a FIXED expression and a constant:

```
FIXED fixval := -3F;
min := $MIN (fixval, 5F);                                !Return -3F
```

## \$NUMERIC Function

The \$NUMERIC function tests the right half of an INT value for the presence of an ASCII numeric character.



*int-expression*

is an INT expression.

## Usage Considerations

\$NUMERIC inspects bits <8:15> of *int-expression* and ignores bits <0:7>. It tests for a numeric character according to the criterion:

*int-expression*  $\geq$  "0" AND *int-expression*  $\leq$  "9"

If a numeric character occurs, \$NUMERIC sets the condition code to CCL (condition code less than). If you plan to test the condition code, do so before an arithmetic operation or assignment occurs.

If the character passes the test, \$NUMERIC returns a -1 (true); otherwise, it returns a 0 (false).

*int-expression* can include STRING and UNSIGNED(1–16) operands, as described in [Expression Arguments](#) on page 14-5.

## Example of \$NUMERIC Function

In this example, \$NUMERIC tests for a numeric character in a STRING argument, which the system places in the right byte of a word and treats as an INT value:

```
STRING char;                                         !Declare variable
IF $NUMERIC (char) THEN ... ;  !Test for numeric character
```

# \$OCCURS Function

The \$OCCURS function returns the number of occurrences of a variable.



*variable*

is the identifier of:

- An array, structure, or substructure (but not a template structure)
- An array declared in a structure

## Usage Considerations

\$OCCURS returns the number of:

- Elements in an array
- Occurrences of a structure or substructure

For example, if the argument is declared with the bounds [0:3], \$OCCURS returns the value 4.

You can use \$OCCURS to find the total length of an entire array or substructure:

- To find the length in bytes, multiply the values returned by \$LEN and \$OCCURS.
- To find the length in bits, multiply the values returned by \$BITLENGTH and \$OCCURS.

You can find the length of a structure in the same way, except that you must first round up the length value to the word boundary before multiplying the rounded value with the \$OCCURS value.

If *variable* is a template structure, the compiler returns error 69 (invalid template access).

If *variable* is a simple variable, pointer, or procedure parameter, the compiler returns warning 43 (a default \$OCCURS count of 1 is returned).

You can use \$OCCURS in LITERAL expressions and global initializations, because it always returns a constant value.

## Examples of \$OCCURS Function

1. In this example, \$OCCURS returns the number of occurrences of a structure, which is 6:

```
INT index;

STRUCT .job_data[0:5];                                !Declare structure
BEGIN
    INT i1;
    STRING s1;
END;

FOR index := 0 TO $OCCURS (job_data) - 1 DO ... ;
                                                !Return 6, the number of
                                                ! structure occurrences
```

2. In this example, \$OCCURS returns the number of elements in an array, and \$LEN returns the number of bytes in each element. This example multiplies 3 times 4 (the values returned by \$OCCURS and \$LEN) and yields the number of bytes in the entire array:

```
INT array_length;          !Declare variables
INT(32) array[0:2];

array_length := $LEN (array) * $OCCURS (array);
                !Return 12, the length of the entire
                ! array in bytes
```

## \$OFFSET Function

The \$OFFSET function returns the number of bytes from the address of the zeroth structure occurrence to a structure data item.



**variable**

is the fully qualified identifier of a structure data item—a substructure, simple variable, array, simple pointer, or structure pointer declared within a structure

## Usage Considerations

The zeroth structure occurrence has an offset of 0. For items other than a structure data item, \$OFFSET returns a 0.

When you qualify the identifier of *variable*, you can use constant indexes but not variable indexes; for example:

```
$OFFSET (struct1.subst[1].item) !1 is a constant index
```

For a structure pointer declared inside a structure, \$OFFSET computes the byte offset of the structure occurrence to which the pointer points, not the byte offset of the structure pointer itself. When you specify such a pointer as an argument to \$OFFSET, you must qualify its identifier with the identifier of the structure data item (see Example 2).

For UNSIGNED structure items, \$OFFSET returns the byte offset of the nearest word boundary, not the nearest byte boundary.

You can use \$OFFSET in LITERAL expressions and global initializations, because it always returns a constant value.

If you apply \$OFFSET to an unfinished structure or to a substructure in an unfinished structure, the compiler emits warning 76 (cannot use \$OFFSET or \$LEN until base structure is complete).

## Examples of \$OFFSET Function

1. In this example, \$OFFSET returns the byte offset of the third occurrence of a substructure from the address of the zeroth structure occurrence:

```
STRUCT a;                                !Declare structure
BEGIN
  INT array[0:40];
  STRUCT ab[0:9];                         !Declare substructure AB
  BEGIN                                     ! with ten occurrences
    !Lots of declarations
  END;
END;

INT c;
!Some code
c := $OFFSET (a.ab[2]);      !Return offset of third
                             ! occurrence of substructure
```

2. In this example, \$OFFSET returns the byte offset of the structure occurrence to which a structure pointer points:

```
STRUCT .tt;                                !Declare structure TT
BEGIN
  INT i;
  INT(32) d;
```

```

STRING s;
END;

STRUCT .st;                      !Declare structure ST
BEGIN
INT i;
INT j;
INT .st_ptr(tt)                  !Declare structure pointer
END;                             ! that points to structure TT

INT x;

x := $OFFSET (st.j);            !X gets 2
x := $OFFSET (tt.s);            !X gets 6
x := $OFFSET (st.st_ptr.s);     !X gets 6

```

3. This example applies \$OFFSET to an indexed template structure:

```

INT x;

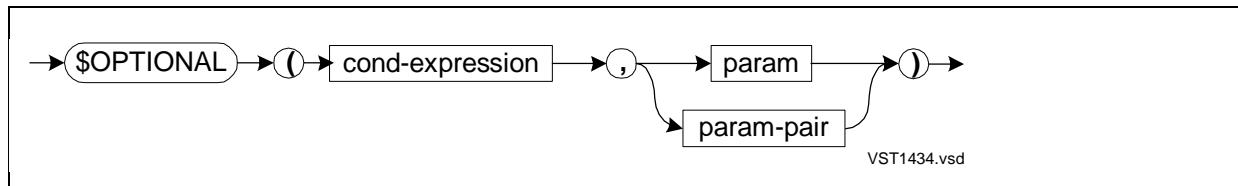
STRUCT st[-1:1];
BEGIN
INT item;
FIXED(2) price;
END;

x := $OFFSET (st[-1].item);      !X gets -10

```

## \$OPTIONAL Function

The \$OPTIONAL function controls whether a given parameter or parameter pair is passed to a VARIABLE or EXTENSIBLE procedure in a D20 or later object file.



*cond-expression*

is a conditional expression. If *cond-expression* is true, *param* or *param-pair* is passed. If *cond-expression* is false, *param* (or *param-pair*) is not passed.

*param*

is a variable identifier or an expression that defines an actual parameter to pass to a formal parameter declared in the called procedure if *cond-expression* is true.

*param-pair*

is an actual parameter pair to pass to a formal parameter pair declared in the called procedure if *cond-expression* is true. *param-pair* has the form:



**string**

is the identifier of a STRING array or simple pointer declared inside or outside a structure.

**length**

is an INT expression that specifies the length, in bytes, of *string*.

## Usage Considerations

A call to a VARIABLE or EXTENSIBLE procedure can omit some or all parameters. \$OPTIONAL lets your program pass a parameter (or parameter-pair) based on a condition at execution time. \$OPTIONAL is evaluated as follows each time the encompassing CALL statement is executed:

- If *cond-expression* is true, the parameter is passed; \$PARAM, if present, is set to true for the corresponding formal parameter.
- If *cond-expression* is false, the parameter is not passed; \$PARAM, if present, is set to false for the corresponding formal parameter.

A called procedure cannot distinguish between a parameter that is passed conditionally and one that is passed unconditionally. Passing parameters conditionally, however, is slower than passing them unconditionally. In the first case, the EXTENSIBLE mask is computed at execution time; in the second case, the mask is computed at compilation time.

## Examples of the \$OPTIONAL Function

1. This example shows that a called procedure cannot distinguish between a parameter that is passed conditionally and one that is passed unconditionally:

```

PROC p1 (i) EXTENSIBLE;
  INT i;
  BEGIN
    !Lots of code
  END;

PROC p2;
  BEGIN
    INT n := 1;
    CALL p1 ($OPTIONAL (n > 0, n) );
    CALL p1 (n);           !These two calls are
                           ! indistinguishable.
  END;
  
```

2. This example shows that a called procedure cannot distinguish between a parameter that is omitted conditionally and one that is omitted unconditionally:

```

PROC p1 (i) EXTENSIBLE;
  INT i;
  BEGIN
    !Lots of code
  END;

PROC p2;
  BEGIN
    INT n := 1;
    CALL p1 ($OPTIONAL (n < 0, n) );
    CALL p1 ( );
    !These two calls are
    ! indistinguishable.
  END;

```

3. This example shows how you can conditionally pass or not pass a parameter and a parameter pair. When P2 calls P1, S:I is passed because I equals 1, which is less than 9. 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;

```

4. You can use \$OPTIONAL 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;

```

## \$OVERFLOW Function

The \$OVERFLOW function checks the state of the overflow indicator and indicates whether an overflow occurred.



## Usage Considerations

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

To recover locally from a statement's overflow, turn off the overflow trap bit and use \$OVERFLOW in an IF statement immediately following the operation that affects the overflow indicator. If the overflow indicator is on, \$OVERFLOW is -1 (true). If the overflow indicator is off, \$OVERFLOW is 0 (false).

## Example of \$OVERFLOW Function

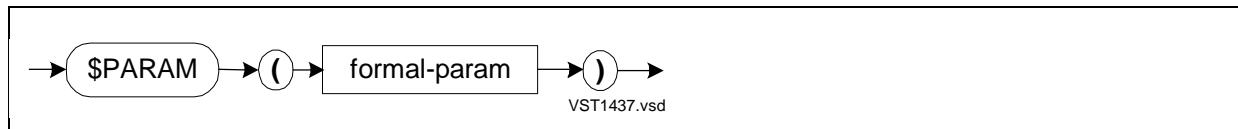
This example turns overflow trapping off, tests the overflow indicator, and turns overflow trapping back on. CODE statements, however, are not portable to future software platforms; modularize their use where possible:

```
INT i, j, k;

CODE (RDE;
      ANRI $COMP (%200);      !Turn off overflow trap bit
      SETE);
      i := j + k;
IF $OVERFLOW THEN i := 0;      !Test overflow indicator
CODE (RDE;
      ANRI $COMP (%040);      !Turn on overflow trap bit
      ORRI %200;              !Disable pending overflow
      SETE);
```

## \$PARAM Function

The \$PARAM function checks for the presence or absence of an actual parameter in the call that invoked the current procedure or subprocedure.



*formal-param*

is the identifier of a formal parameter as specified in the procedure or subprocedure declaration.

## Usage Considerations

If the actual parameter corresponding to *formal-param* is present in the CALL statement, \$PARAM returns 1. If the actual parameter is absent from the CALL statement, \$PARAM returns 0.

Only a VARIABLE procedure or subprocedure or an EXTENSIBLE procedure can use \$PARAM. If such a procedure or subprocedure has required parameters, it must check for the presence or absence of each required parameter in CALL statements. The procedure or subprocedure can also use \$PARAM to check for optional parameters.

## Example of \$PARAM Function

In this example, \$PARAM checks for the absence of each required parameter and for the presence of the optional parameter:

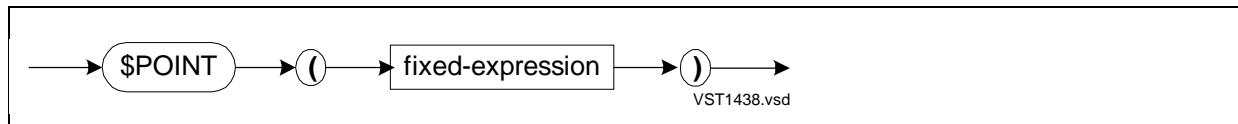
```

PROC var_proc (buffer,length,key) VARIABLE;
    INT .buffer, length,           !Required parameters
        key;                      !Optional parameter
    BEGIN
        !Some code here
        IF NOT $PARAM (buffer) OR NOT $PARAM (length) THEN RETURN;
            !Return 1 or 0 for each
            ! required parameter
        IF $PARAM (key) THEN ... ;   !Return 1 if optional
        END;                      ! parameter is present

```

## \$POINT Function

The \$POINT function returns the fpoint value, in integer form, associated with a FIXED expression.



fixed-expression  
is a FIXED expression.

## Usage Considerations

The compiler emits no instructions when evaluating *fixed-expression*. Therefore, *fixed-expression* cannot invoke a function and cannot be an assignment expression.

## Example of \$POINT Function

This example retains precision automatically when performing fixed-point division. \$POINT returns the *fpoint* value of B to \$SCALE, which then scales A by that factor:

```

FIXED(3) result;          !Declare variables
FIXED(3) a;
FIXED(3) b;

result := $SCALE (a, $POINT (b)) / b;
                    !Return fpoint of FIXED expression
                    ! and scale value by that factor

```

# \$READCLOCK Function

The \$READCLOCK function returns the current setting of the system clock.



## Usage Considerations

\$READCLOCK returns the current setting of the system clock as a FIXED(0) value.

\$READCLOCK invokes the RCLK instruction, described in the *System Description Manual* for your system.

## Example of \$READCLOCK Function

In this example, \$READCLOCK returns the current setting of the system clock:

```

FIXED the_time;           !Declare data
the_time := $READCLOCK;   !Return current clock time
  
```

# \$RP Function

The \$RP function returns the current setting of the compiler's internal RP counter. (RP is the register stack pointer.)



## Usage Consideration

\$RP returns the compile-time setting of RP. This setting is not guaranteed to be the run-time setting.

\$RP is not portable to future software platforms.

## Example of \$RP Function

In this example, \$RP returns the compiler's current RP value:

```

IF $RP <> 7 THEN ... ;           Something is on the stack
  
```

# \$SCALE Function

The \$SCALE function moves the position of the implied decimal point by adjusting the internal representation of a FIXED(*fpoint*) expression.



**fixed-expression**

is a FIXED expression.

**scale**

is an INT constant in the range –19 to +19 that defines the number of positions to move the implied decimal point to the left (*scale* > 0) or to the right (*scale* < 0) of the least significant digit.

## Usage Considerations

If the result of the scale operation exceeds the range of a FIXED(0) expression, \$SCALE sets the overflow indicator.

\$SCALE adjusts the implied decimal point of the stored value by multiplying or dividing by 10 to the *scale* power. Some precision might be lost with negative \$SCALE values.

## Example of \$SCALE Function

1. In this example, \$SCALE returns a FIXED(7) expression from a FIXED(3) expression:

```

FIXED(3) a := 9.123F;
FIXED(7) result;           ! Declare variables
result := $SCALE (a, 4);   ! Return FIXED(7) value from
                           ! FIXED(3) value
  
```

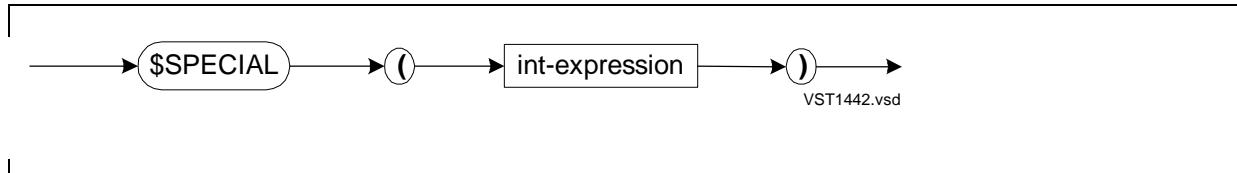
2. To retain precision when you divide operands that have nonzero *fpoint* settings, use the \$SCALE standard function to scale up the *fpoint* of the dividend by a factor equal to the *fpoint* of the divisor; for example:

```

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

# \$SPECIAL Function

The \$SPECIAL function tests the right half of an INT value for the presence of an ASCII special (non-alphanumeric) character.



*int-expression*

is an INT expression.

## Usage Considerations

\$SPECIAL inspects bits <8:15> of the *int-expression* and ignores bits <0:7>. It tests for a special character according to the following criterion:

*int-expression*.<8:15> <> alphabetic AND *int-expression*.<8:15> <> numeric

If \$SPECIAL finds a special character, it sets the condition code to CCG (condition code greater than). If you plan to check the condition code, do so before an arithmetic operation or a variable assignment occurs.

If the character passes the test, \$SPECIAL returns a –1 (true); otherwise, \$SPECIAL returns a 0 (false).

*int-expression* can include STRING and UNSIGNED(1–16) operands, as described in [Expression Arguments](#) on page 14-5.

## Example of \$SPECIAL Function

In this example, \$SPECIAL tests for the presence of a special character in a STRING argument, which the system places in the right byte of a word and treats as an INT value:

```

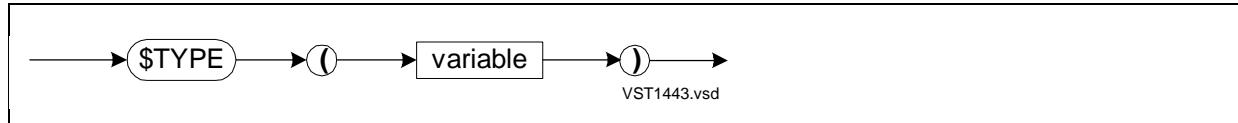
STRING char;           !Declare variable
IF $SPECIAL (char) THEN ... ; !Test for special character
  
```

# \$SWITCHES Function

The \$SWITCHES function is described in [Section 15, Privileged Procedures](#).

# \$TYPE Function

The \$TYPE function returns a value that indicates the data type of a variable.



**variable**

is the identifier of a simple variable, array, simple pointer, structure, structure data item, or structure pointer.

## Usage Considerations

\$TYPE returns an INT value that has a meaning as follows:

Value	Meaning	Value	Meaning
0	Undefined	5	REAL
1	STRING	6	REAL (64)
2	INT	7	Substructure
3	INT (32)	8	Structure
4	FIXED	9	UNSIGNED

For a structure pointer, \$TYPE returns the value 8, regardless of whether the structure pointer points to a structure or to a substructure.

You can use \$TYPE in LITERAL expressions and global initializations, because \$TYPE always returns a constant value.

## Example of \$TYPE Function

In this example, \$TYPE returns the data type of a REAL(64) variable:

```

REAL( 64 ) var1;                      ! Declare variables
INT type1;

type1 := $TYPE ( var1 );                ! Return 6 for REAL( 64 )

```

# \$UDBL Function

The \$UDBL function returns an INT(32) value from an unsigned INT expression.



int-expression  
is an unsigned INT expression.

## Usage Consideration

\$UDBL places the INT value in the low-order 16 bits of an INT(32) variable and sets the high-order 16 bits to 0.

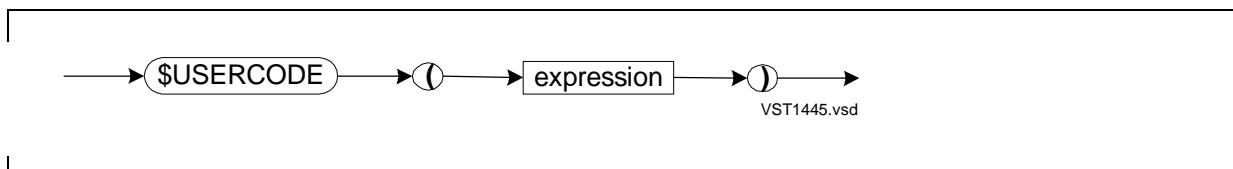
## Example of \$UDBL Function

In this example, \$UDBL returns an INT(32) value from an unsigned INT expression:

```
INT a16 := -1;                                ! Declare variables
INT(32) a32;
a32 := $UDBL (a16);                          ! Return 65535D
```

## \$USERCODE Function

The \$USERCODE function returns the content of the word at the specified location in the current user code segment.



expression

is an INT expression that specifies a word address in the current user code segment.

## Usage Considerations

\$USERCODE enables a program that is executing in system code space to obtain the content of a word location in the current user code segment.

\$USERCODE invokes the LWUC instruction, described in the *System Description Manual* for your system.

\$USERCODE is not portable to future software platforms.

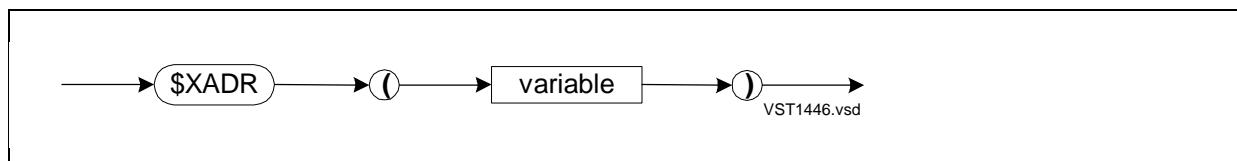
## Example of \$USERCODE Function

In this example, \$USERCODE returns the content of a word located at a label in the current user code segment:

```
PROC p;
BEGIN
  INT a, b, c;
  !Some code here
  u_code_location:           !Label a location
    a := a + b;
    !More code here
    c := $USERCODE (@u_code_location); !Get content of word
                                         ! located at label
  END;
```

## \$XADR Function

The \$XADR function converts a standard address to an extended address.



*variable*

is the identifier of a simple variable, pointer, array element, structure, or structure data item. For any other variable, the compiler issues a warning.

## Usage Considerations

\$XADR returns an extended address from the standard address of *variable*. (If *variable* is a pointer, \$XADR returns the extended address of the data to which the pointer points, not the address of the pointer itself.) If *variable* already has an extended address, \$XADR returns the extended address; in this case, \$XADR behaves like the @ operator.

If a structure spans the 32K-word boundary of the user data segment, \$XADR cannot return an extended address from the standard address of a byte-addressed structure item.

You cannot use \$XADR at the global level because \$XADR uses compiler-generated code to compute the extended address.

## Examples of \$XADR Function

1. This example initializes an extended pointer with the extended (32-bit) address that \$XADR returns from a standard (16-bit) address:

```
INT .array[0:49];           !ARRAY has 16-bit address
STRING .EXT xptr := $XADR (array);
                           !Initialize XPTR with 32-bit
                           !address returned by $XADR
```

2. In this example, \$XADR returns an extended address for an INT variable to which a standard simple pointer points, and then assigns the extended address to an extended simple pointer:

```
INT .std_ptr := %1000;      !Declare 16-bit simple pointer
INT .EXT ext_ptr;          !Declare 32-bit simple pointer
@ext_ptr := $XADR (std_ptr);
                           !Assign 32-bit address
```

3. In this example, \$XADR returns the extended address of an extended indirect array:

```
INT .EXT ext_ptr;
INT .EXT xarray[0:9];       !XARRAY has 32-bit address
ext_ptr := $XADR (ext_array[5]);
                           !Assign 32-bit address
```

## Built-in Functions

TAL supports many pTAL built-in functions. pTAL built-in functions provide the functions of required CISC instructions that standard pTAL features and millicode routines do not provide.

pTAL does not define built-ins for instructions whose functions are:

- Available through standard pTAL constructs
- Available via calls to RISC millicode routines
- Not needed to run programs on RISC processors

In a few cases, pTAL defines a built-in because the equivalent millicode routine cannot be called from pTAL. pTAL supports each built-in by a combination of RISC instructions and calls to system routines.

Many built-ins can be executed only by processes running in privileged mode. Most built-ins are executable statements rather than functions that return values, such as:

```
$ATOMIC_ADD(a, b);
```

Some built-ins are functions that return values, as in the following:

```
a := $ATOMIC_GET(b);
t := $READTIME;
```

TAL directly implements pTAL built-ins. However, the only parameter data type to some pTAL built-ins that must be correct is the parameter size. For example, the parameter for type-transfer functions is checked for size, but not for type. TAL is used only to generate TNS code on behalf of the pTAL compiler. The TAL compiler does not check the syntax of pTAL address types and type-transfer functions to the extent that the pTAL compiler does.

[Table 14-3](#) on page 14-45 describes the eight address type-transfer functions that TAL implements.

---

**Table 14-3. Address type-transfer functions**

Function	Description
\$WADDR_TO_BADDR	Converts a WADDR address to a BADDR address
\$BADDR_TO_WADDR	Converts a BADDR address to a WADDR address
\$SGWADDR_TO_SGBADDR	Converts a SGWADDR address to a SGBADDR address
\$SGBADDR_TO_SGWADDR	Converts a SGBADDR address to a SGWADDR address
\$BADDR_TO_EXTADDR	Converts a BADDR address to an EXTADDR address
\$WADDR_TO_EXTADDR	Converts a WADDR address to an EXTADDR address
\$SGBADDR_TO_EXTADDR	Converts a SGBADDR address to a EXTADDR address
\$SGWADDR_TO_EXTADDR	Converts a SGWADDR address to a EXTADDR address

---

TAL does not ensure that the data type of the type-transfer function parameter is the data type specified in the type-transfer function name. TAL ensures only that the lengths of all parameters are one word (16 bits) and that the number of words in the result is the same as the number of words in the item into which the result is stored.

For more information, refer to the *pTAL Conversion Guide* or *pTAL Reference Manual*.

[Table 14-4](#) describes the list of pTAL built-ins that TAL supports.

---

**Table 14-4. pTAL built-ins (page 1 of 3)**

Built - in	Description
\$EXTADDR_TO_WADDR	Converts an EXTADDR address to a WADDR address
\$EXTADDR_TO_BADDR	Converts an EXTADDR address to a BADDR address
\$WADDR_TO_BADDR	Converts a WADDR address to a BADDR address
\$BADDR_TO_WADDR	Converts a BADDR address to an WADDR address

---

**Table 14-4. pTAL built-ins** (page 2 of 3)

<b>Built - in</b>	<b>Description</b>
\$SGWADDR_TO_SGBADDR	Converts an SGWADDR address to an SGBADDR address
\$SGBADDR_TO_SGWADDR	Converts an SGBADDR address to an SGWADDR address
\$BADDR_TO_EXTADDR	Converts a BADDR address to an EXTADDR address
\$WADDR_TO_EXTADDR	Converts a WADDR address to an EXTADDR address
\$SGBADDR_TO_EXTADDR	Converts an SGBADDR address to an EXTADDR address
\$SGWADDR_TO_EXTADDR	Converts an SGWADDR address to an EXTADDR address
\$UDIVREM16	Divides an INT(32) dividend by an INT divisor to produce an INT quotient and INT remainder
\$UDIVREM32	Divides an INT(32) dividend by an INT divisor to produce an INT(32) quotient and INT remainder
\$STACK_ALLOCATE	Allocates space beyond the stack area of the current procedure
\$IS_CALLER_PRIV	Returns true if its callers caller is privileged else false i.e. if A calls B and B calls IS_CALLER_PRIV the function returns true if A is privileged
\$FILL8	Fills an array or structure with repetitions of an 8-bit value
\$FILL16	Fills an array or structure with repetitions of an 16-bit value
\$FILL32	Fills an array or structure with repetitions of an 32-bit value
\$ATOMIC_ADD	Increments a variable by a specified value
\$ATOMIC_AND	Performs a logical “and” operation (bitwise “and”—LAND) on a variable and a mask and stores the result back into the variable
\$ATOMIC_OR	Performs a logical “or” operation (bitwise “or”—LOR) on a variable and a mask and stores the result back into the variable
\$ATOMIC_DEP	Performs an atomic bit deposit into an INT variable
\$ATOMIC_GET	Fetches atomically the value of a 1-, 2-, or 4-byte variable
\$ATOMIC_PUT	Stores atomically a 1-, 2-, or 4-byte value into a variable
\$ASCIITOFIXED	Converts ASCII digits to a binary quadrupleword integer

**Table 14-4. pTAL built-ins** (page 3 of 3)

---

<b>Built - in</b>	<b>Description</b>
\$COUNTDUPS	Returns the number of duplicate words at the beginning of an extended memory buffer
\$EXCHANGE	Exchanges the contents of two variables
\$FIXEDTOASCII	Converts a 64-bit integer to ASCII digits
\$FIXEDTOASCIIRESIDUE	Converts a 64-bit integer to ASCII digits and stores the residue
\$MOVEANDCXSUMBYTES	Moves and accumulates an eight-bit checksum on bytes in extended memory
\$MOVENONDUP	Moves extended memory words until it encounters two adjacent identical words
\$CHECKSUM	Computes the checksum of data in extended memory
\$EXTADDR	Converts its argument to an EXTADDR address
\$PROCADDR	Converts an INT(32) expression to a PROCADDR address

---



# **15 Privileged Procedures**

This section gives information on privileged mode and operations and accessing the global data area of the system data segment. Primarily, this section describes the syntax for:

- System global pointer declarations
- 'SG'-equivalenced simple variable declarations
- 'SG'-equivalenced structure declarations
- 'SG'-equivalenced simple pointer declarations
- 'SG'-equivalenced structure pointer declarations
- Privileged standard functions—\$AXADR, \$BOUNDS, \$SWITCH
- Privileged directive—TARGET

## **Privileged Mode**

The following kinds of procedures execute in privileged mode:

- A CALLABLE or PRIV procedure
- A nonprivileged procedure that is called by a CALLABLE or PRIV procedure

Normally, only the operating system executes in privileged mode. The operating system performs privileged operations on behalf of applications programs.

### **CALLABLE Procedures**

A CALLABLE procedure is one that you declare with the CALLABLE attribute. Nonprivileged procedures can call CALLABLE procedures, but only CALLABLE procedures can call PRIV procedures.

In the following example, a CALLABLE procedure calls the PRIV procedure declared in the next example:

```
PROC callable_proc CALLABLE;  
  BEGIN  
    !Lots of code  
    CALL priv_proc;  
  END;
```

### **PRIV Procedures**

A PRIV procedure is one that you declare with the PRIV attribute. A PRIV procedure can execute privileged instructions. Only PRIV or CALLABLE procedures can call a PRIV procedure. PRIV protects the operating system from unauthorized (nonprivileged) calls to its internal procedures.

The following PRIV procedure is called by the preceding CALLABLE procedure:

```
PROC priv_proc PRIV;
BEGIN
    !Privileged instructions
END;
```

## Nonprivileged Procedures

A nonprivileged procedure that is called by a CALLABLE or PRIV procedure executes in privileged mode and can call PRIV procedures.

# Privileged Operations

Only procedures that operate in privileged mode can access system global data. Such procedures can access system data space, call other privileged procedures, and execute certain privileged instructions. Privileged procedures must be specially licensed to operate, because they might (if improperly written) adversely affect the status of the processor in which they are running.

A procedure that operates in privileged mode can use system global pointers and 'SG' equivalencing:

- To access system tables and the system data area
- To initiate certain input-output transfers
- To move and compare data between the user data area and the system data area
- To scan data in the system data area

A procedure operating in privileged mode can also:

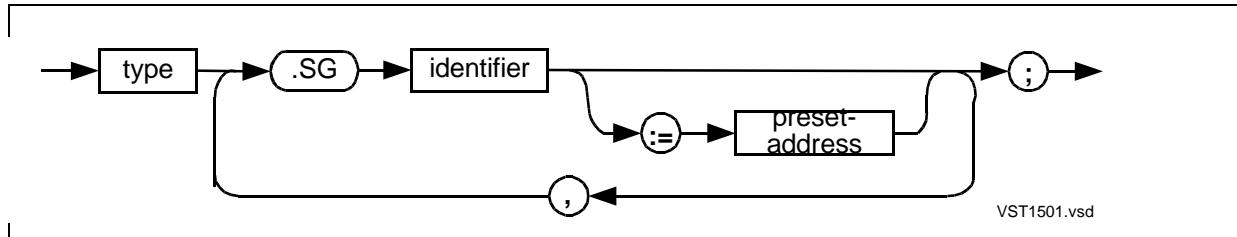
- Perform privileged operations through calls to system procedures
- Execute privileged instructions that can affect other programs or the operating system

For more information on an extended pointer pointing to a system data, see Appendix B of the *TAL Programmer's Guide*.

System tables are described in the *System Description Manual* for your system.

# System Global Pointer Declaration

The system global pointer declaration associates an identifier with a memory location at which you store the address of a variable located in the system global data area.



**type**

is any data type except UNSIGNED and specifies the data type of the value to which the pointer points.

**.SG**

is the system global indirection symbol and denotes 16-bit addressing in the system global data area.

**identifier**

is the identifier of the pointer.

**preset-address**

is the address of a variable in the system global data area. The address is determined by you or the system during system generation.

## Usage Consideration

The compiler allocates one word of primary storage for the pointer in the current user data segment. The primary storage can be global, local, or sublocal, depending on the level at which you declare the pointer.

## Example of System Global Pointer Declaration

The following example declares an INT system global pointer named NEWNAME:

```
INT .SG newname;
```

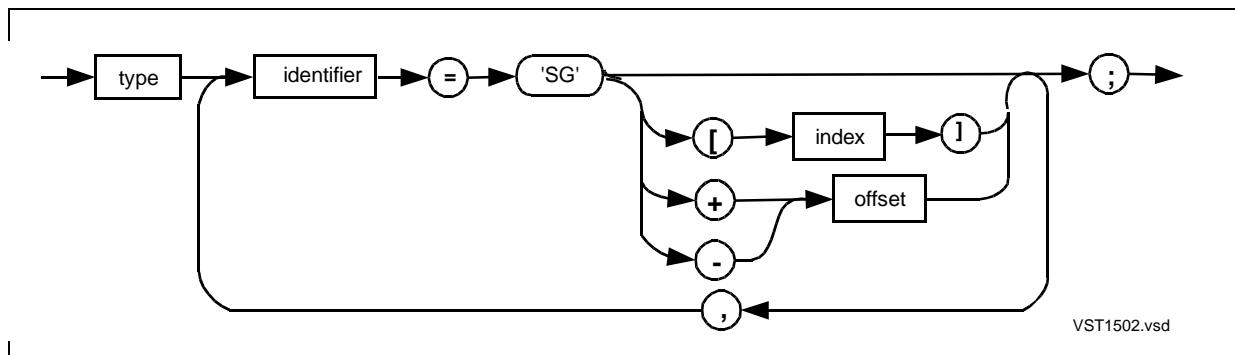
# 'SG'-Equivalenced Variable Declarations

'SG' equivalencing associates a global, local, or sublocal identifier with a location that is relative to the base address of the system global area. You can declare the following kinds of 'SG'-equivalenced variables:

- 'SG'-equivalenced simple variable
- 'SG'-equivalenced structure
- 'SG'-equivalenced simple pointer
- 'SG'-equivalenced structure pointer

## 'SG'-Equivalenced Simple Variable

The 'SG'-equivalenced simple variable declaration associates a simple variable with a location that is relative to the base address of the system global data area.



**type**

is any data type except UNSIGNED and specifies the data type of *identifier*.

**identifier**

is the identifier of a simple variable to be made equivalent to 'SG'.

**' SG '**

is the symbol that denotes the base address of the system global data area; *identifier* is addressed relative to SG[0].

**index and offset**

are equivalent INT values in the range 0 through 63.

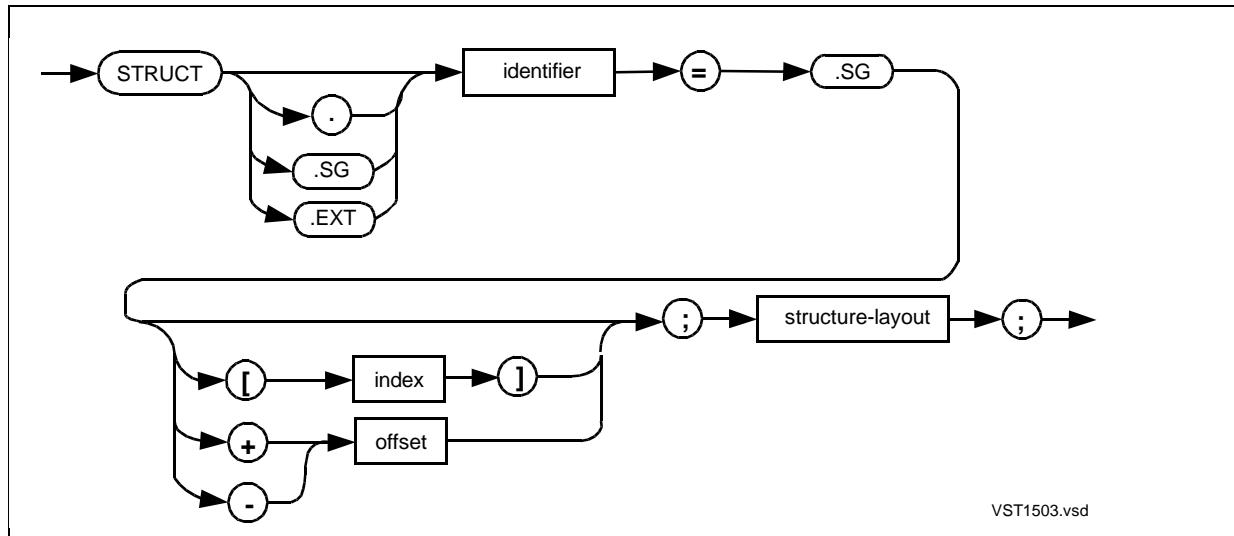
## Example of 'SG'-Equivalenced Simple Variable

This example declares a simple variable equivalent to location SG + 15 in the system global data area:

```
INT item = 'SG' + 15;
```

## 'SG'-Equivalenced Definition Structure

The 'SG'-equivalenced definition structure declaration associates a definition structure with a location relative to the base address of the system global data area.



- . (period)

is the standard indirection symbol and denotes 16-bit addressing, in this case in the system global data area.

- .SG

is the system global indirection symbol and denotes 16-bit addressing in the system global data area.

- .EXT

is the extended indirection symbol and denotes 32-bit addressing.

- identifier

is the identifier of a definition structure to be made equivalent to 'SG'.

- ' SG '

is the symbol that denotes the base address of the system global data area; *identifier* is addressed relative to SG[0].

index and offset  
are equivalent INT values in the range 0 through 63.

structure-layout  
is a BEGIN-END construct that contains declarations for structure items.

## Usage Consideration

If you specify an indirection symbol (., .SG, or .EXT), the structure behaves like a structure pointer. If you do not specify an indirection symbol, the structure has direct addressing mode.

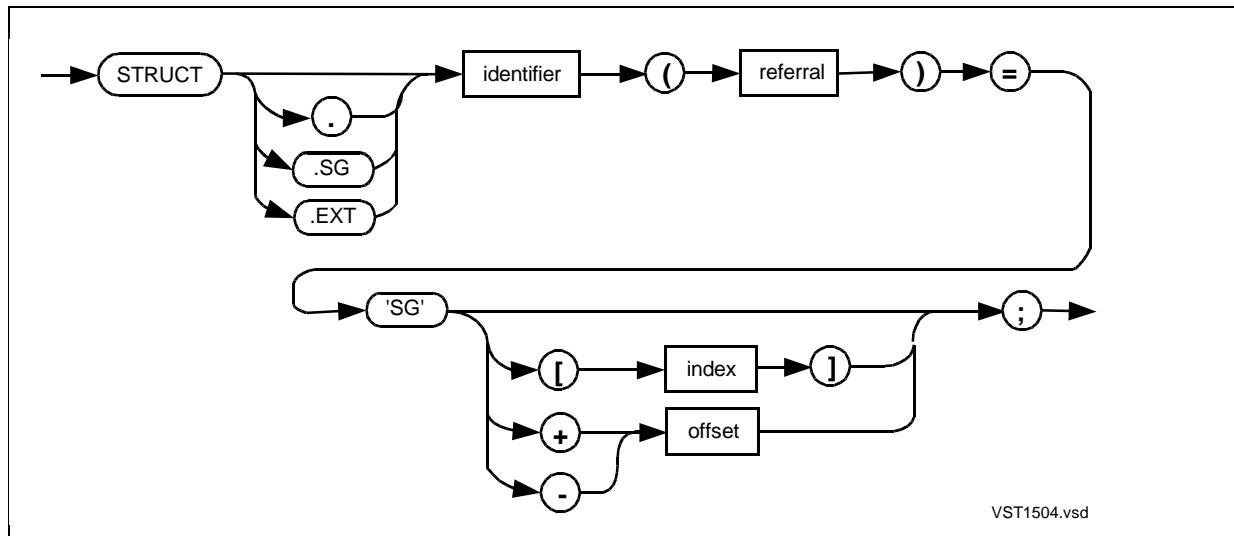
### Example of 'SG'-Equivalenced Definition Structure

The following example declares a definition structure equivalent to location SG[10] in the system global data area:

```
STRUCT def_struct = 'SG'[10];
BEGIN
  STRING out;
  FIXED up;
  REAL in;
END;
```

### 'SG'-Equivalenced Referral Structure

The 'SG'-equivalenced referral structure declaration associates a referral structure with a location relative to the base address of the system global data area.



- . (period)

is the standard indirection symbol and denotes 16-bit addressing, in this case in the system global data area.

.SG

is the system global indirection symbol and denotes 16-bit addressing in the system global data area.

.EXT

is the extended indirection symbol and denotes 32-bit addressing.

*identifier*

is the identifier of a referral structure to be made equivalent to 'SG'.

*referral*

is the identifier of a previously declared structure or structure pointer that is to provide the layout for this structure.

'SG'

is the symbol that denotes the base address of the system global data area; *identifier* is addressed relative to SG[0].

*index* and *offset*

are equivalent INT values in the range 0 through 63.

## Usage Considerations

If you specify an indirection symbol (., .SG, or .EXT), the structure behaves like a structure pointer. If you do not specify an indirection symbol, the structure has direct addressing mode.

## Example of 'SG'-Equivalenced Referral Structure

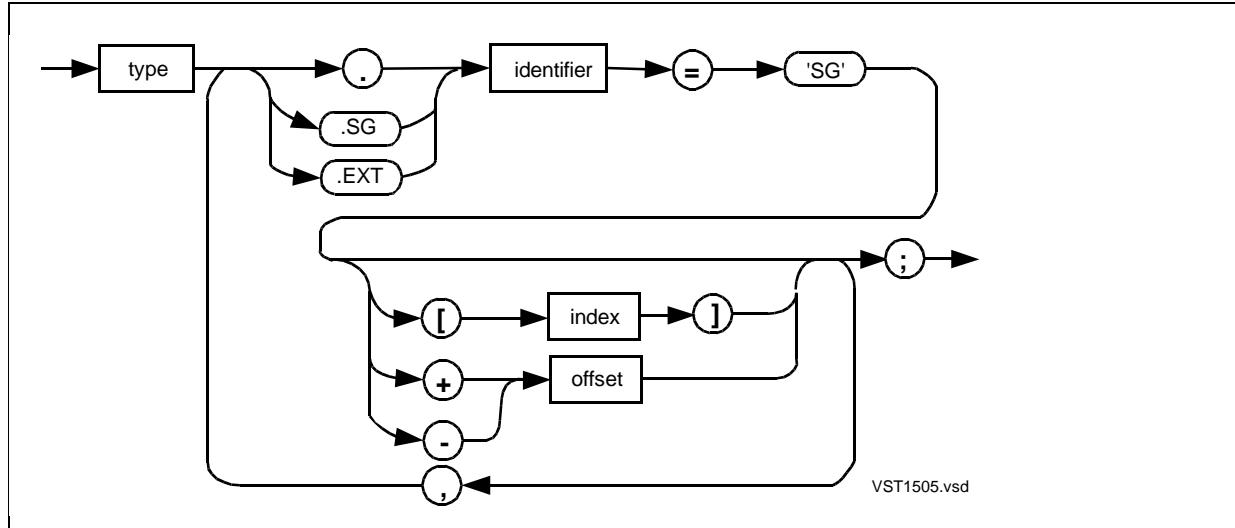
This example declares a referral structure equivalent to location SG[30] in the system global data area:

```
STRUCT def_struct;
BEGIN
  STRING a[0:99];
  REAL b[0:9];
END;

STRUCT ref_struct (def_struct) = 'SG'[30];
```

# 'SG'-Equivalenced Simple Pointer

The 'SG'-equivalenced simple pointer declaration associates a simple pointer with a location relative to the base address of the system global data area.



**type**

is any data type except UNSIGNED and specifies the data type of the value to which the pointer points.

. (period)

is the standard indirection symbol and denotes 16-bit addressing, in this case in the system global data area.

. SG

is the system global indirection symbol and denotes 16-bit addressing in the system global data area.

. EXT

is the extended indirection symbol and denotes 32-bit addressing.

**identifier**

is the identifier of a simple pointer to be made equivalent to 'SG'.

' SG '

is the symbol that denotes the base address of the system global data area; *identifier* is addressed relative to SG[0].

index and offset  
are equivalent INT values in the range 0 through 63.

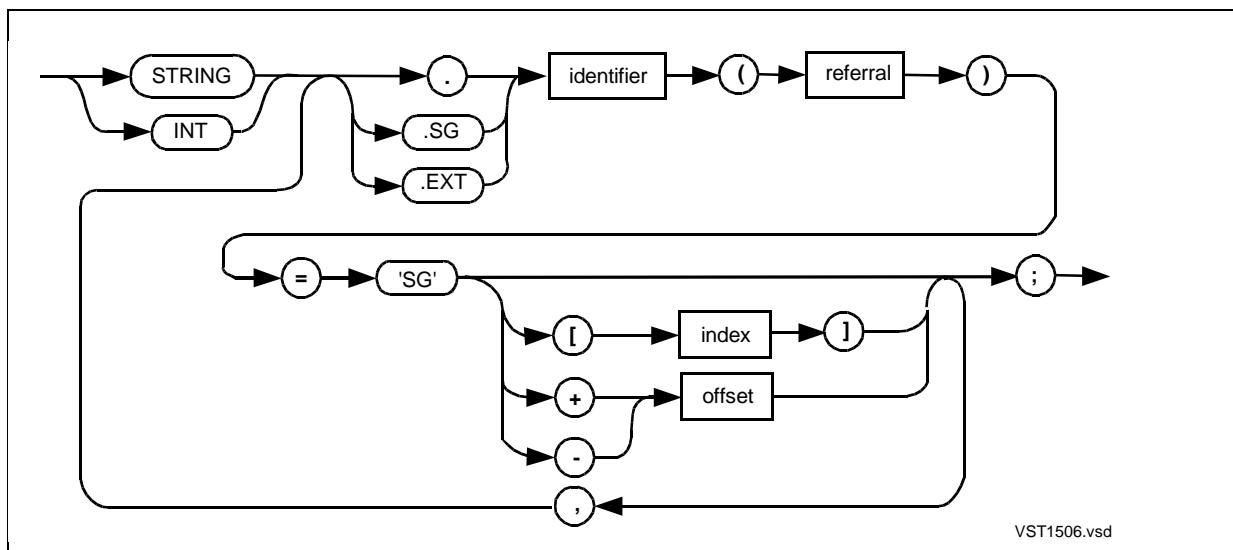
## Example of 'SG'-Equivalenced Simple Pointer

This example declares a simple pointer equivalent to location SG + 2 in the system global data area:

```
INT .ptr = 'SG' + 2;
```

## 'SG'-Equivalenced Structure Pointer

The 'SG'-equivalenced structure pointer declaration associates a structure pointer with a location relative to the base address of the system global data area.



STRING

denotes the STRING attribute.

INT

denotes the INT attribute.

. (period)

is the standard indirection symbol and denotes 16-bit addressing, in this case in the system global data area.

.SG

is the system global indirection symbol and denotes 16-bit addressing in the system global data area.

.EXT

is the extended indirection symbol and denotes 32-bit addressing.

identifier

is the identifier of a structure pointer to be made equivalent to 'SG'.

referral

is the identifier of a previously declared structure or structure pointer that is to provide the layout for *identifier*.

' SG '

is the symbol that denotes the base address of the system global data area; *identifier* is addressed relative to SG[0].

index and offset

are equivalent INT values in the range 0 through 63.

## Usage Considerations

[Table 9-3](#) on page 9-7 describes the kind of addresses a structure pointer can contain depending on the STRING or INT attribute and addressing symbol.

## Example of 'SG'-Equivalenced Simple Pointer

This example declares a structure that provides the structure layout for a structure pointer and declares the structure pointer equivalent to location SG[30] in the system global data area:

```
STRUCT .some_struct;                                !Declare structure
BEGIN
    INT a;
    INT b[ 0:5 ];
END;

INT .struct_ptr (some_struct) = 'SG' + 30;
```

# Functions for Privileged Operations

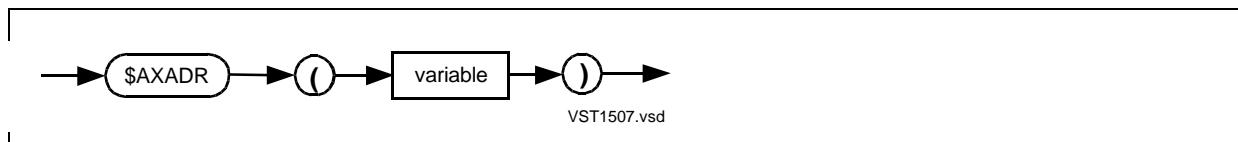
The following privileged functions perform operations that are restricted to procedures running in privileged mode:

Standard Function	Operation
\$AXADR	Converts a standard address or a relative extended address to an absolute extended address
\$BOUNDS	Checks the locations of parameters passed to system procedures
\$SWITCHES	Returns the current setting of the switch register

The following pages describe each privileged standard function in alphabetic order.

## \$AXADR Function

The \$AXADR function returns an absolute extended address.



*variable*

is the identifier of a simple variable, pointer, array element, structure, or structure data item.

## Usage Considerations

\$AXADR converts a standard or relative extended address to an absolute extended address. If *variable* is a pointer, \$AXADR returns the absolute extended address of the item to which the pointer points, not the address of the pointer itself.

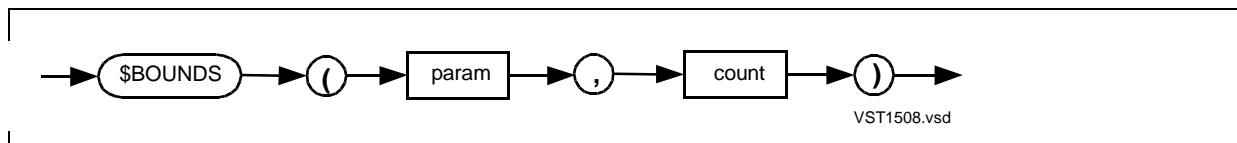
## Example of \$AXADR Function

This example converts the standard address of INTR to an absolute extended address:

```
PROC myproc PRIV;
BEGIN
  STRING .EXT str;
  INT intr;
  !Lots of code
  @str := $AXADR (intr);
  !More code
END;
```

## \$BOUNDS Function

The \$BOUNDS function checks the location of a parameter passed to a system procedure to prevent a pointer that contains an incorrect address from overlaying the stack (S) register with data.



param

is the identifier of an INT reference parameter of the procedure from which the \$BOUNDS function is called. *param* must be declared at the global level. If you specify a value parameter or a subprocedure parameter, an error results.

count

is an INT value that represents the word count.

# Usage Considerations

`$BOUNDS` checks to see whether the stack space—represented by a starting address and word count—causes the `S` register to overflow.

`$BOUNDS` returns an INT result. If no bounds error occurs, `$BOUNDS` returns a 0 (false). If a bounds error occurs, `$BOUNDS` returns a -1 (true).

**\$BOUNDS** is not portable to future software platforms; it is described here only to support existing programs. Instead, use the **XBNDSTEST** or **XSTACKTEST** system procedure as described in the *Guardian Procedure Calls Reference Manual*.

## Example of \$BOUNDS Function

This example checks the location of the parameter BUF. Before writing three words of information to the location pointed to by BUF, the example calls \$BOUNDS to make sure that the new information does not accidentally overwrite existing information in the system data segment:

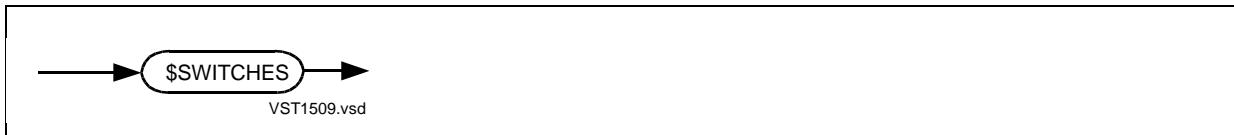
```

PROC example (buf) PRIV;
    INT .SG buf;
BEGIN
    !Lots of code
    IF $BOUNDS (buf, 3) THEN
        CALL error
    ELSE buf ':=' [1, 2, 3];
    !More code
END;

```

# \$SWITCHES Function

The \$SWITCHES function returns the current content of the switch register.



## Usage Considerations

For NonStop and TXP processors, the switch register stores the current setting of the physical switches on the processor.

\$SWITCHES is not portable to future software platforms.

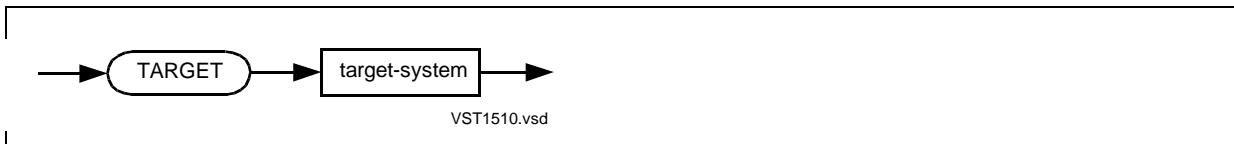
## Example of \$SWITCHES Function

The following example stores the current content of the switch register into N:

```
INT n;
n := $SWITCHES;
```

# TARGET Directive

The TARGET directive lets you specify the target system for conditional code. TARGET works in conjunction with the IF and ENDIF directives. TARGET is a D20 or later feature.



target-system

is the system for which conditional code is written. target-system can be one of:

ANY	This value causes IF ANY conditional code to be compiled. ANY specifies that the conditional code as written is not system-dependent.
TNS_ARCH	This value causes IF TNS_ARCH conditional code to be compiled.
TNS_R_ARCH	This value causes IF TNS_R_ARCH conditional code to be compiled.

## Usage Considerations

You can specify TARGET in the compilation command or anywhere in the source code any number of times. Do not, however, specify both TARGET TNS\_ARCH and TARGET TNS\_R\_ARCH in the same compilation unit.

You can use TARGET to source in system-specific blocks of declarations from library files, such as operating system declaration files.

### TARGET ANY Directive

A TARGET ANY can follow (and inherit the attributes of) a TARGET TNS\_ARCH or TARGET TNS\_R\_ARCH. If either IF TNS\_ARCH or IF TNS\_R\_ARCH follows a TARGET ANY, however, the compiler issues a warning and does not compile the corresponding system-specific conditional code.

If TARGET ANY is in effect, the compiler:

- Compiles the code between IF ANY and ENDIF ANY
- Skips the code between IFNOT ANY and ENDIF ANY

If TARGET ANY is not in effect, the compiler:

- Compiles the code between IFNOT ANY and ENDIF ANY
- Skips the code between IF ANY and ENDIF ANY

### TARGET TNS\_ARCH Directive

If TARGET TNS\_ARCH is in effect, the compiler:

- Compiles the code between IF TNS\_ARCH and ENDIF TNS\_ARCH
- Skips the code between IFNOT TNS\_ARCH and ENDIF TNS\_ARCH

If TARGET TNS\_ARCH is not in effect, the compiler:

- Compiles the code between IFNOT TNS\_ARCH and ENDIF TNS\_ARCH
- Skips the code between IF TNS\_ARCH and ENDIF TNS\_ARCH

### TARGET TNS\_R\_ARCH Directive

If TARGET TNS\_R\_ARCH is in effect, the compiler:

- Compiles the code between IF TNS\_R\_ARCH and ENDIF TNS\_R\_ARCH
- Skips the code between IFNOT TNS\_R\_ARCH and ENDIF TNS\_R\_ARCH

If TARGET TNS\_R\_ARCH is not in effect, the compiler:

- Compiles the code between IFNOT TNS\_R\_ARCH and ENDIF TNS\_R\_ARCH
- Skips the code between IF TNS\_R\_ARCH and ENDIF TNS\_R\_ARCH

## Any TARGET Directive

If any TARGET directive is in effect, the compiler:

- Compiles code between IF TARGETSPECIFIED and ENDIF TARGETSPECIFIED
- Skips code between IFNOT TARGETSPECIFIED and ENDIF TARGETSPECIFIED

If no TARGET directive is in effect, the compiler:

- Compiles code between IFNOT TARGETSPECIFIED and ENDIF TARGETSPECIFIED
- Skips code between IF TARGETSPECIFIED and ENDIF TARGETSPECIFIED

## Examples of TARGET Directive

1. Use TARGET to source in system-specific declarations from a library file. The following fragment from a library file applies to Examples 2, 3, and 4:

```
?SECTION part1
?IF TNS_ARCH
    LITERAL pagesize = 2048;
    LITERAL pages_in_seg = 64;
?ENDIF TNS_ARCH
?IF TNS_R_ARCH
    LITERAL pagesize = 4069;
    LITERAL pages_in_seg = 32;
?ENDIF TNS_R_ARCH

    LITERAL bytes_in_seg = pages_in_seg * pagesize;
                                !System-specific

    LITERAL max_file_size = 12;           !Not system-specific

?SECTION part2
    LITERAL max_buffer_size = 1024;      !Not system-specific
```

2. If your program uses MAX\_BUFFER\_SIZE, source in section PART2. Do not specify a TARGET directive when you compile your program.
3. If your program uses BYTES\_IN\_SEG, produce two object files by compiling once with TARGET TNS\_ARCH and then again with TARGET TNS\_R\_ARCH. Specify each TARGET directive in a compilation command or at the beginning of the source file. For example, you can specify TARGET in compilation commands as follows:

```
TAL /IN mysource /objtns; TARGET TNS_ARCH
TAL /IN mysource /objtnsr; TARGET TNS_R_ARCH
```

4. If your program uses MAX\_FILE\_SIZE, you must specify either TARGET option to avoid compilation errors on the BYTES\_IN\_SEG declaration. The object file does not rely on system-specific declarations, so you can run the object file on both systems by using the stand-alone Binder command SET TARGET ANY.

# **16 Compiler Directives**

This section describes compiler directives provided by the TAL compiler. Compiler directives let you:

- Specify input source code
- Select options that control listings, generate object code, and build the object file
- Perform conditional compilation
- Allocate and delete the compiler's internal data structures

## **Specifying Compiler Directives**

You can specify compiler directives either in a compilation command or in a directive line, unless otherwise noted in this section. The compiler interprets and processes each directive at the point of occurrence.

### **Compilation Command**

When you issue the compilation command at the TACL prompt, you can specify compiler directives following the semicolon. In the following example, NOMAP suppresses the symbol map and CROSSREF produces a cross-reference listing:

```
TAL /IN mysrc, OUT $s.#lists/ myobj; 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  
DEFINETOG  
DUMPCONS  
ENDIF  
IF  
IFNOT  
PAGE  
RP  
SECTION  
SOURCE
```

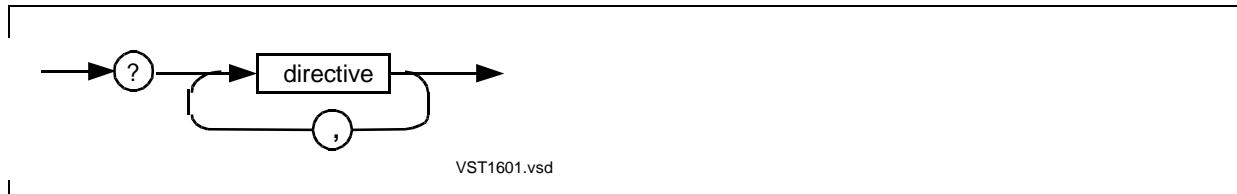
The following directives can appear only in the compilation command:

```
EXTENDTALHEAP  
SQL with the PAGES option  
SYMBOLPAGES
```

For information on compilation command options, see Section 14, Compiling Programs, in the *TAL Programmer's Guide*.

## Directive Line

A directive line in your source code can contain one or more compiler directives unless otherwise noted in the directive descriptions.



?

indicates a directive line. ? can appear only in column 1.

directive

is the name of a compiler directive described in this section.

## Usage Considerations

- Begin each directive line and each continuation directive line by specifying ? in column 1. (?) is not part of the directive name.)
- Place the name of a directive and its arguments on the same line unless the directive description says you can use continuation lines. For example, SEARCH and SOURCE can continue on subsequent lines.
- When you use continuation lines, place the leading parenthesis of the argument list on the same line as the directive name. Otherwise, the compiler issues a warning and ignores the argument list.
- Do not put stray characters such as semicolons at the end of a directive line. If you do, the compiler issues a warning.
- You can use an equal sign (=) as a separator in directives that accept numeric values, but no other NonStop compiler allows an equal sign in directives.

## Examples of Directive Lines

1. This directive line uses a continuation line for multiple directives:

```
?NOLIST, NOCODE, INSPECT, SYMBOLS, NOMAP, NOLMAP, GMAP  
?CROSSREF, INNERLIST
```

2. This directive line shows proper placement of the leading parenthesis of the argument list:

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

## 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 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 specify partial file names as described in Appendix E in the *TAL Programmer's Guide*. If you specify a partial file name, the compiler uses default values as described in Appendix E.

For the SEARCH, SOURCE, and USEGLOBALS directives, the compiler can use the node (system), volume, and subvolume specified in TACL ASSIGN SSV (Search SubVolume) commands.

## Logical File Names

A logical file name is a TACL DEFINE name or a TACL ASSIGN name. The following directives accept a logical file name in place of a file name:

ERRORFILE  
SAVEGLOBALS  
SEARCH  
SOURCE  
USEGLOBALS

Appendix E in the *TAL Programmer's Guide* gives more information on disk file names and TACL DEFINE and ASSIGN commands.

## Summary of Compiler Directives

[Table 16-1](#) on page 16-6 summarizes directives grouped in the following categories. In the table, boldface type indicates a default directive.

- Compiler input
- Compiler listing
- Diagnostic output
- Object-file content
- Conditional compilation
- Compiler's internal data structures
- Object-file run-time environment

---

**Table 16-1. Summary of Compiler Directives** (page 1 of 6)

<b>Category</b>	<b>Directive</b>	<b>Alternate</b>	<b>Operation</b>
Compiler Input	ABORT	NOABORT	Terminates the compilation if the compiler cannot open a source file
	BEGINCOMPILATION		Starts compiling source code
	COLUMNS		Treats as comments any text that appears beyond the specified column
	SAVEGLOBALS		Saves global data declarations in a file
	SECTION		Names a section of the source file
	SOURCE		Sources in code from another input file
	USEGLOBALS		Retrieves saved global data declarations

---

**Table 16-1. Summary of Compiler Directives** (page 2 of 6)

<b>Category</b>	<b>Directive</b>	<b>Alternate</b>	<b>Operation</b>
Compiler Listing	ABSLIST	<b>NOABSLIST</b>	Lists addresses relative to the code area base
	<b>CODE</b>	NOCODE POPCODE PUSHCODE	Lists instruction codes in octal format after each procedure
	CROSSREF	<b>NONCROSSREF</b>	Cross-references source identifier classes
	DEFEXPAND	<b>NODEFEXPAND</b> PUSHDEFEXPAND POPDEFEXPAND	Lists the text of invoked DEFINE macros
	FMAP	<b>NOFMAP</b>	Lists the file map of source files used
	<b>GMAP</b>	NOGMAP	Lists the global map
	ICODE	<b>NOICODE</b> POPICODE PUSHICODE	Lists mnemonics after each procedure
	INNERLIST	<b>NOINNERLIST</b> PUSHINNERLIST POPINNERLIST	Lists mnemonics after each source statement
	LINES		Skips to the top of form after a specified number of lines if the list file is a line printer or a process
	<b>LIST</b>	NOLIST PUSHLIST POPLIST	Lists the source code
	<b>LMAP</b> <b>MAP</b>	NOLMAP NOMAP PUSHMAP POPMAP	Lists the load maps Lists the identifier map
	PAGE		Ejects the page if the list file is a line printer or a process
	<b>PRINTSYM</b>	NOPRINTSYM	Selectively lists symbols
	<b>SUPPRESS</b>	<b>NOSUPPRESS</b>	Suppresses all listings but the header, diagnostics, and trailer

**Table 16-1. Summary of Compiler Directives** (page 3 of 6)

<b>Category</b>	<b>Directive</b>	<b>Alternate</b>	<b>Operation</b>
Diagnostics	ERRORFILE		Logs error and warning messages to an error file
	ERRORS		Terminates compilation if the specified number of error messages occur
	RELOCATE		Issues warnings for nonrelocatable global variables
Object-File Content	WARN	NOWARN	Selectively suppresses warnings
	ASSERTION		Conditionally invokes a procedure for debugging purposes
	CHECK	<b>NOCHECK</b> PUSHCHECK POPCHECK	Generates range-checking code
	COMPACT	NOCOMPACT	Fills gaps in the lower code area
	CPU		Generates a TNS object file; this directive is now superfluous
	DUMPCONS FIXUP	NOFIXUP	Emits constants and labels to the object code Performs fixup steps

**Table 16-1. Summary of Compiler Directives** (page 4 of 6)

<b>Category</b>	<b>Directive</b>	<b>Alternate</b>	<b>Operation</b>
Object-File Content	INHIBITXX	NOINHIBITXX	Suppresses extended indexed (XX) instructions
	INT32INDEX	NOINT32INDEX	Generates correct extended addresses for structure items accessed by INT indexes in D-series programs
	OLDFLTSTDFUNC		Treats \$FLT, \$FLTR, \$EFLT, and \$EFLTR arguments as FIXED(0) values
	OPTIMIZE		Optimizes the object code at the specified level
PEP			Sets the size of the procedure entry point (PEP) table
ROUND		NOROUND	Performs scalar rounding
SEARCH			Resolves external references using the specified object files
SQL			Prepares for processing of NonStop SQL statements
SQLMEM			Controls SQL parameter placement in run-time memory
SYNTAX			Checks the syntax; suppresses the object code

**Table 16-1. Summary of Compiler Directives** (page 5 of 6)

<b>Category</b>	<b>Directive</b>	<b>Alternate</b>	<b>Operation</b>
Conditional Compilation	DEFINETOG		Specifies toggles without changing their settings (D20 RVU or later)
	ENDIF		Stops conditional compilation
	IF	IFNOT	Begins conditional compilation
	RESETTOG		Turns toggles (that were turned on by SETTOG) off
	SETTOG		Specifies toggles and turns them on
	TARGET		Specifies target system for D20 or later conditional code
Compiler Data Structures	DECS		Decrements the compiler's internal S-register value
	EXTENDTALHEAP		Increases the size of the compiler's extended heap (D-series system)
	RP		Sets the compiler's internal register pointer (RP) counter
	SYMBOLPAGES		Increases the size of the compiler's internal symbol table

**Table 16-1. Summary of Compiler Directives** (page 6 of 6)

<b>Category</b>	<b>Directive</b>	<b>Alternate</b>	<b>Operation</b>
Run-time Environment	DATAPAGES		Sets the size of the data area in the user data segment
	ENV		Specifies a run-time environment such as the CRE for a D-series object file
	EXTENDSTACK		Increases the size of the data stack
	HEAP		Sets the size of the CRE user heap for a D-series object file
	HIGHPIN		Sets the HIGHPIN attribute in a D-series object file
	HIGHREQUESTERS		HIGHREQUESTERS attribute in a D-series object file
	INSPECT	<b>NOINSPECT</b>	Sets the Inspect product as the default debugger
	LARGESTACK		Sets the size of the extended stack
	LIBRARY		Uses the specified user library file for resolving external references
	RUNNAMED		Causes a D-series object file to run as a named process
Run-time Environment	SAVEABEND	<b>NOSAVEABEND</b>	Creates a process-state save file if the program ends abnormally
	STACK		Sets the size of the data stack
	SUBTYPE		Saves the specified process subtype in the object file header
	SYMBOLS	<b>NOSYMBOLS</b>	Generates the symbol table for the Inspect product

# ABORT Directive

With ABORT, the compiler terminates compilation if it cannot open a file specified in a SOURCE directive.

The default is ABORT.



## Usage Considerations

This directive can appear in the compilation command or anywhere in the source code.

### ABORT

If ABORT is in effect and the compiler cannot open a file listed in a SOURCE directive, the compilation session terminates. ABORT issues an error message to the output file, stating the name of the file that cannot be opened.

### NOABORT

If NOABORT is in effect and the compiler cannot open a file listed in a SOURCE directive, the compiler prompts the home terminal for the name of a file. You can then take any of the following actions:

- Retry the unopened file after moving it to the required location.
- Skip the unopened file and continue compilation.
- Substitute the name of another file.
- Abort the compilation.

If you choose to either skip the unopened file or abort the compilation, the compiler issues an error message.

If SUPPRESS is also in effect, the compiler prints the SOURCE directive before the error message.

## Example of ABORT Directive

In this example, NOABORT prompts you for a file name when the compiler cannot open a file listed in a SOURCE directive:

```

!This is MYSOURCE file
!Global declarations
?NOABORT
?SOURCE somefile (proc1, proc2, proc3)
?SOURCE anyfile
!Procedure declarations

```

## ABSLIST Directive

ABSLIST lists code addresses relative to the code area base.

The default is NOABSLIST.



## Usage Considerations

This directive can appear in the compilation command or anywhere in the source code.

ABSLIST lists the code address for each source line relative to location C[0], the base of the code area. The code address for each line applies to the first instruction generated from the source statement on that line. ABSLIST has no effect if NOLIST or SUPPRESS is in effect.

If you use ABSLIST, you must also use a PEP directive to specify the size of the PEP table. Place the PEP directive before the first procedure declaration that is not a FORWARD or EXTERNAL procedure declaration.

NOABSLIST lists code addresses relative to the base of the procedure.

## Limitations

General use of ABSLIST is not recommended because some addresses listed by ABSLIST are incorrect, for example, if the file:

- Has more than 32K words of code
- Has RESIDENT procedures following non-RESIDENT procedures

- Does not supply enough PEP table space in the PEP directive
- Does not declare all procedures FORWARD

Also, if the file reaches the 64K-word limit, the compiler disables ABSLIST, starts printing offsets from the procedure base, and emits a warning.

## Example of ABSLIST Considerations

This example shows placement of the ABSLIST and PEP directives before the first procedure declaration:

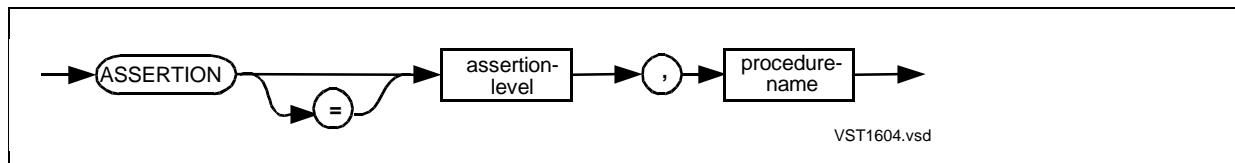
```

!Global declarations
?ABSLIST; PEP 60
PROC a;
    BEGIN
        !Lots of code
    END;

```

## ASSERTION Directive

ASSERTION invokes a procedure when a condition defined in an ASSERT statement is true.



**assertion-level**

is an unsigned decimal constant in the range 0 through 32,767 that defines a numeric relationship to an ASSERT statement *assert-level*.

**procedure-name**

is the name of the procedure to invoke if the condition defined in an ASSERT statement is true and the ASSERTION directive *assertion-level* is less than the ASSERT statement *assert-level*. The named procedure must not have parameters.

## Usage Considerations

ASSERTION can appear anywhere in the source code but not in the compilation command. If other directives appear on the same directive line, ASSERTION must be last on that line.

You use the ASSERTION directive with the ASSERT statement 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 procedure:

```
?ASSERTION 5, PROCESS_DEBUG_           !Assertion-level is 5
```

2. Place an ASSERT statement at the point where you want to invoke the error-handling procedure when some condition occurs. In the ASSERT statement, specify an assert-level that is equal to or higher than the assertion-level and specify an expression that tests a condition. For example, the standard function \$CARRY returns true if the carry indicator is on and false if it is off:

```
ASSERT 10 : $CARRY;                  !Assert-level is 10
```

3. During program execution, if an ASSERT statement assert-level is equal to or higher than the current ASSERTION directive assertion-level and the associated condition is true, the compiler invokes 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 an assertion-level that is higher than the highest ASSERT statement assert-level you want to nullify:

```
?ASSERTION 11, PROCESS_DEBUG_         !Assertion-level nullifies
                                         ! assert-level 10 and below
```

## Example of ASSERTION Directive

This example invokes PROCESS\_DEBUG\_ whenever a carry or overflow 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
```

If you change the *assertion-level* in the ASSERTION directive to 15, you nullify the two ASSERT statements that specify *assert-level* 10 and the \$CARRY condition.

If you change the *assertion-level* to 30, you nullify all the ASSERT statements. Thus, if ASSERT statements that cover a particular condition all have the same *assert-level*, it is easier to nullify specific levels of ASSERT statements.

## BEGINCOMPILATION Directive

BEGINCOMPILATION marks the point in the source file where compilation is to begin if the USEGLOBALS directive is in effect.



### Usage Considerations

If you have saved global data declarations in a previous compilation (by specifying SAVEGLOBALS), you can use the saved declarations in subsequent compilations (by specifying USEGLOBALS, BEGINCOMPILATION, and SEARCH).

- USEGLOBALS suppresses compilation of text lines and SOURCE directives but not other directives.
- SEARCH specifies the object file that provides global initializations and template structure declarations for the USEGLOBALS compilation.
- BEGINCOMPILATION begins compilation of text lines and SOURCE directives.

BEGINCOMPILATION, if present, must appear in the source code between the last global data declaration or SEARCH directive and the first procedure declaration, including any EXTERNAL and FORWARD declarations. BEGINCOMPILATION takes effect only if USEGLOBALS is in effect.

To use BEGINCOMPILATION, you must not have changed the global data declarations after you compiled with SAVEGLOBALS to produce the object file containing the initializations.

For more information, see [SAVEGLOBALS Directive](#) on page 16-75 and [USEGLOBALS Directive](#) on page 16-93.

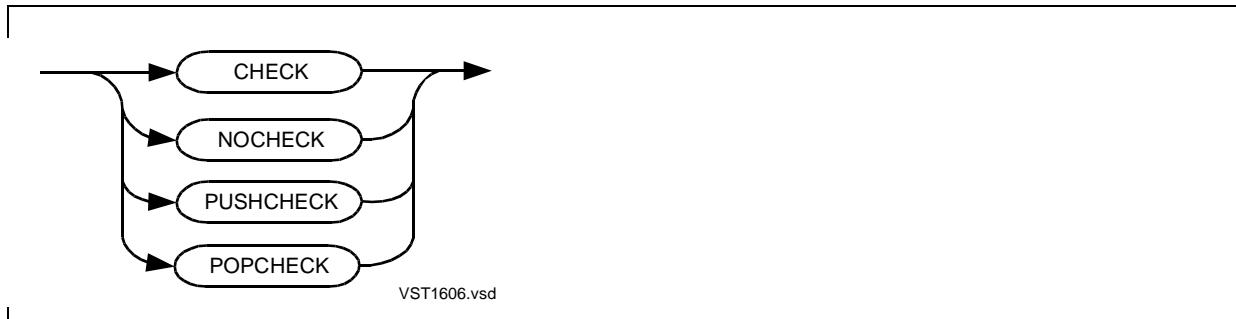
### Example of BEGINCOMPILATION Directive

For an example of how SAVEGLOBALS, USEGLOBALS, BEGINCOMPILATION, and SOURCE interact, see SAVEGLOBALS in this section.

# CHECK Directive

**CHECK** generates range-checking code for certain features.

The default is NOCHECK.



# Usage Considerations

You can specify the **CHECK** directive in the compilation command or anywhere in the source text.

**CHECK** turns the checking setting on for subsequent code.

`NOCHECK` turns the checking setting off for subsequent code.

**PUSHCHECK** pushes the current checking setting onto the directive stack without changing the current setting.

**POPCHECK** removes the top value from the directive stack and sets the current checking setting to that value.

## Extended Stack

If the source code includes extended local arrays or structures, the compiler allocates an extended stack in the automatic extended data segment. The compiler also allocates two pointers (#SX and #MX) to the extended stack. #SX points to the first free location in the current stack frame, and #MX contains the maximum allowable value for #SX, less eight bytes.

If the value of #SX is greater than or equal to that of #MX, the extended stack overflows. If CHECK is in effect, the compiler sets the S-register to %177777, causing a stack overflow trap.

## **Unlabeled CASE Statement**

If you omit the OTHERWISE clause of an unlabeled CASE statement and the selector value is out of range (negative or greater than n), the compiler behaves as follows:

- If the CHECK directive is in effect and if your program has enabled arithmetic traps, a divide-by-zero instruction trap occurs.

- If NOCHECK is in effect or if your program has disabled arithmetic traps, control passes to the statement following the unlabeled CASE statement and program results are unpredictable.

## Labeled CASE Statement

CHECK and NOCHECK do not affect the labeled CASE statement. If you omit the OTHERWISE clause in a labeled CASE statement and the selector value does not select an alternative, a run-time error occurs.

## Example of CHECK Directive

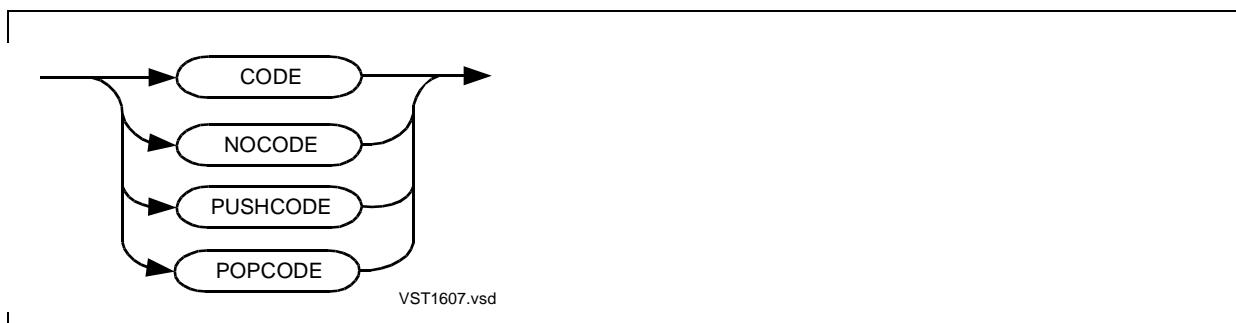
This example uses CHECK to provide a stack overflow trap if the extended stack overflows:

```
!This is MYSOURCE file
?CHECK
PROC a;
    BEGIN
        !Lots of extended local data declarations
        !Lots of code
        CALL a;
        !More code
    END;
```

## CODE Directive

CODE lists instruction codes and constants in octal format after each procedure.

The default is CODE.



## Usage Considerations

This directive can appear in the compilation command or anywhere in the source code.

CODE turns the code-listing setting on for subsequent code. CODE has no effect if NOLIST or SUPPRESS is in effect.

NOCODE turns the code-listing setting off for subsequent code.

PUSHCODE pushes the current code-listing setting onto the directive stack without changing the current setting.

POPCODE removes the top value from the directive stack and sets the current code-listing setting to that value.

### CODE Listing

The code listing for each procedure follows the local map, if any, for the procedure. Each line lists an octal address (the offset from the procedure base), followed by eight words of instructions in octal.

For global variables declared within a named data block, the G+ addresses shown are relative to the start of the data block and are not the final addresses. At the end of compilation, BINSERV determines the final G+ addresses for such global variables and the code locations for items such as PCAL instructions and global read-only arrays.

After compilation, you can display the final addresses by using Binder and Inspect commands.

## Example of CODE Directive

This compilation command specifies NOCODE to suppress the code listing:

```
TAL /IN mysrc, OUT $s.#lists/ myobj; NOCODE
```

## COLUMNS Directive

COLUMNS directs the compiler to treat any text beyond the specified column as comments.



columns-value

is the column beyond which the compiler is to treat text as comments. Specify an unsigned decimal constant in the range 12 through 132. The default value is 132. If

you specify a value smaller than 12, the compiler issues a warning and uses 12. If you specify a value larger than 132, the compiler issues a warning and uses 132.

## Usage Considerations

COLUMNS can appear in the compilation command or anywhere in the source code.

In the source code, COLUMNS if present must be the first or only directive in the directive line. If it is not the first or only directive in the directive line, the compiler issues a diagnostic message and ignores COLUMNS and any remaining directives on that line. This ordering is not enforced if COLUMNS appears in the compilation command.

## Recommended Uses

Normally, you specify COLUMNS at the beginning of the source code preceding any SECTION directive. You can set the *columns-value* as follows:

- If a source file has no unprefixed comments at the ends of lines, specify a *columns-value* of 132 to prevent lines from being truncated in the event this file is sourced in by a file that has a smaller *columns-value*. Unprefixed comments begin without the dash (--) or exclamation point (!) prefix.
- If a source file has unprefixed comments at the ends of lines, specify a *columns-value* that allows the compiler to ignore the comments.

Although a source file can include any number of COLUMNS directives, varying the *columns-value* throughout the file is not recommended.

## Context of the Columns Value

The *columns-value* in effect at any given time depends on the context, as follows:

- The main input file initially assumes the *columns-value* set by the last COLUMNS directive in the compilation command. If no COLUMNS directive appears, the main input file initially assumes a *columns-value* of 132.
- At each SOURCE directive, each sourced-in file initially assumes the *columns-value* in effect when the SOURCE directive appeared.
- At each SECTION directive, the *columns-value* is set by the last COLUMNS directive before the first SECTION directive in the sourced-in file. If no such COLUMNS directive appears, each SECTION initially assumes the *columns-value* in effect at the beginning of the sourced-in file.
- Within a section, a COLUMNS directive sets the *columns-value* only until the next COLUMNS or SECTION directive or the end of the file.
- After a SOURCE directive completes execution (that is, after all sections listed in the SOURCE directive are read or the end of the file is reached), the compiler restores the *columns-value* to what it was when the SOURCE directive appeared.

- In all other cases, the *columns-value* is set by the most recently processed COLUMNS directive.

If a SOURCE directive lists sections, the compiler processes no source code outside the listed sections except any COLUMNS directives that appear before the first SECTION directive in the sourced-in file. For more information on sourcing in files or sections, see the SOURCE and SECTION directives.

## Examples of COLUMNS Directive

1. This directive appears at the beginning of a source file that has no unprefixed comments at the ends of lines. It prevents truncation of lines in the event this file is sourced in by a file that has narrow lines specified.

```
?COLUMNS 132
```

2. This directive appears at the beginning of a source file that starts unprefixed comments in column 81 of each line. It constrains the source file to 80 columns:

```
?COLUMNS 80
```

## COMPACT Directive

COMPACT moves procedures into gaps below the 32K-word boundary of the code area if they fit.

The default is COMPACT.



## Usage Considerations

This directive can appear in the compilation command or anywhere in the compilation unit. The last COMPACT or NOCOMPACT directive sets the option for the compilation unit.

By default, COMPACT is in effect. BINSERV moves procedures into any gap (if they fit) in the lower 32K-word area of the code segment.

If NOCOMPACT is in effect, BINSERV does not fill the gaps in the lower 32K-word area of the code segment.

## Example of COMPACT Directive

This example specifies the COMPACT directive at the end of the source file in case sourced-in files specify NOCOMPACT. This is to ensure that BINSERV fills in gaps in the lower 32K-word area of the code segment:

```
!This is MYSOURCE file  
!Lots of code  
!Last procedure  
?COMPACT  
!End of source file
```

## CPU Directive

CPU specifies that the object file runs on a TNS system. (The need for this directive no longer exists. This directive has no effect on the object file and is retained only for compatibility with programs that still specify it.)



cpu-type

indicates the object file is to run on a TNS system—that is, a NonStop, NonStop TXP, NonStop VLX, NonStop CLX, or NonStop Cyclone system. *cpu-type* can be one of these equivalent keywords:

TNS/II

NONSTOP

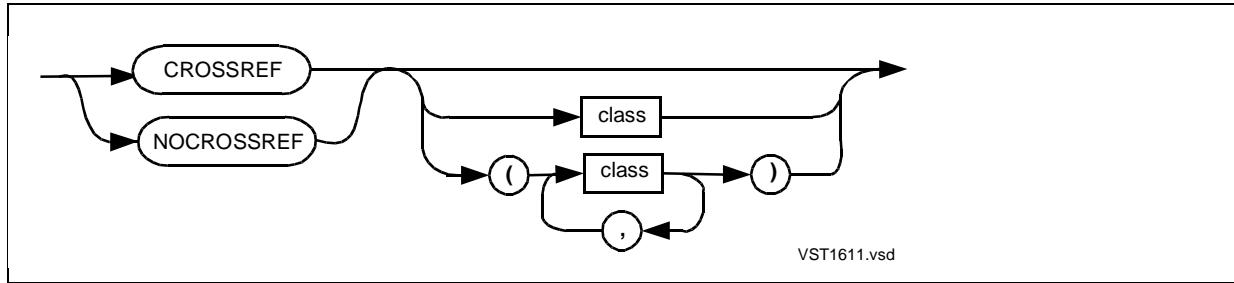
## Usage Considerations

This directive can appear in the compilation command or in the source code before the first declaration.

## CROSSREF Directive

CROSSREF collects source-level declarations and cross-reference information or specifies CROSSREF classes.

The default is NOCROSSREF.



class

is an identifier class. The default list of classes includes all classes except UNREF. You can specify any of the following classes:

Class	Meaning
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 CONSTANTS class is available in the stand-alone Crossref product, but not in the CROSSREF directive.

## Usage Considerations

CROSSREF and NOCROSSREF can appear on the compilation command or any number of times anywhere in the source code. To list the cross-references, LIST and NOSUPPRESS must be in effect at the end of the source file. The cross-reference information appears between the global maps and the load maps.

---

**Note.** If you use USEGLOBALS, do not use CROSSREF. If you do, the compiler will not pass Inspect and cross-reference information to SYMSERV.

---

## Selecting Classes

You can make changes to the current class list at any point in the compilation unit. The compiler collects cross-references for the class list in effect at the end of the compilation:

- To add classes, specify CROSSREF and list the classes you want to add.
- To delete classes, specify NOCROSSREF and list the classes you want to delete.

With parameters, CROSSREF and NOCROSSREF affect the class list, but do not start and stop the collection of cross-references.

## Collecting Cross-References

For each identifier class in the list, the compiler collects the following cross-reference information:

- Identifier qualifiers—structure, subprocedure, and procedure identifiers
- Compiler attributes—class and type modifiers
- Host source file name
- Type of reference—definition, read, write, invocation, parameter, or other

You can collect cross-references for individual procedures or data blocks:

- To start collection of cross-references, specify CROSSREF without parameters.
- To stop collection of cross-references, specify NOCROSSREF with no parameters.

Without parameters, CROSSREF and NOCROSSREF start and stop the collection of cross-references, but do not affect the class list.

CROSSREF without parameters takes effect at the beginning of the next procedure or data block and remains in effect for the entire program or until a NOCROSSREF appears.

NOCROSSREF without parameters takes effect at the beginning of the next procedure or data block and remains in effect for the entire program or until a CROSSREF appears.

For other cross-reference options, use the stand-alone Crossref product described in the *CROSSREF Manual*. For example, the Crossref product can collect cross-references from source files written in one or more languages.

## Example of CROSSREF Directive

This example makes changes to the current class list and turns the collection and listing of cross-references on and off:

```

!Default LIST and NOSUPPRESS are in effect;
! list cross-references.

?CROSSREF, CROSSREF UNREF, NOCROSSREF VARIABLES
                                         !Collect cross-references

NAME test;
    INT i;

?NOCROSSREF                               !Do not collect cross-
BLOCK PRIVATE;                            ! references
    INT j;
END BLOCK;

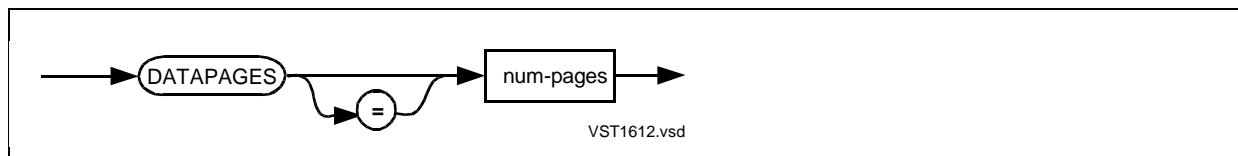
?CROSSREF, CROSSREF VARIABLES !Collect cross-references
PROC p MAIN;
    BEGIN
        !Lots of code
    END;

?SUPPRESS                                 !Do not list cross-
PROC q;                                    ! references
    BEGIN
        !More code
    END;
?NOSUPPRESS                                !List cross-references

```

## DATAPAGES Directive

DATAPAGES sets the size of the data area in the user data segment.



*num-pages*

is the number of 2048-byte memory pages to allocate for the object file's data area in the user data segment. Specify an unsigned decimal constant in the range 0 through 64.

If you specify an out-of-range value, BINSERV sets *num-pages* to 64.

If you specify an insufficient amount or omit DATAPAGES, BINSERV allocates enough memory pages for global data and two times the data stack space needed for local data.

## Usage Considerations

Before compilation, you can specify the number of data area memory pages as follows:

- You can specify the DATAPAGES directive in the compilation command or anywhere in the source code. If you specify a *num-pages* value greater than 32, DATAPAGES overrides the STACK and EXTENDSTACK directives. In a compilation unit that contains a MAIN procedure, the ENV COMMON directive overrides DATAPAGES
- You can specify the memory pages parameter of the C-series NEWPROCESS procedure or the D-series PROCESS\_CREATE\_ procedure in your compilation unit

After compilation, you can set the number of data area memory pages as follows:

- In Binder, you can use the DATA, STACK, and EXTENDSTACK options of the SET command
- At the TACL prompt, you can use the MEM option of the TACL RUN command

Programs using SQL instructions require extra data area memory pages.

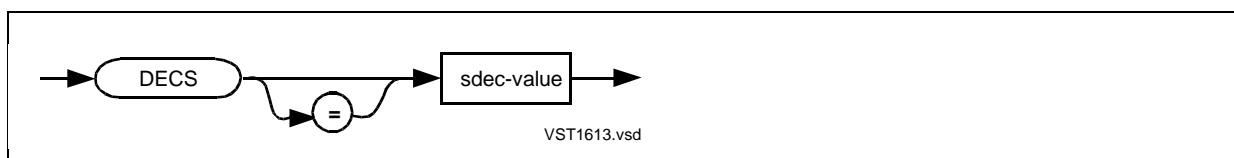
## Example of DATAPAGES Directive

This compilation command starts the compiler and sets the data area size to 64 memory pages:

```
TAL / IN mysrc, OUT $s.#lists / myobj; DATAPAGES 64
```

## DECS Directive

DECS decrements the compiler's internal S-register counter.



**sdec-value**

is the amount by which to decrement the compiler's S-register counter. Specify a signed decimal constant in the range –32,768 through 32,767.

## Usage Considerations

DECS can appear anywhere in the source code outside the global declarations. It cannot appear in the compilation command.

If you manipulate data stack contents without the compiler's knowledge (with CODE statements, for example), you should use DECS to calibrate the compiler's internal S-register counter.

If the compiler decrements (or increments) the S register below (or above) its initial value in a procedure or subprocedure, the compiler issues a warning of S-register underflow (or overflow). If the source program is correct, use DECS to calibrate the compiler's S-register counter, particularly in subprocedures because sublocal variables have S-relative addresses.

Modularize any use of the DECS directive and CODE statement as much as possible; they are not portable to future software platforms.

## Example of DECS Directive

The following example shows the following actions:

1. A CODE (PUSH ...) statement (not a CALL statement) places the parameters for PROC\_NAME onto the data stack.
2. A CODE (PCAL ...) statement calls PROC\_NAME.
3. PROC\_NAME decrements the data stack by three words without the compiler's knowledge.

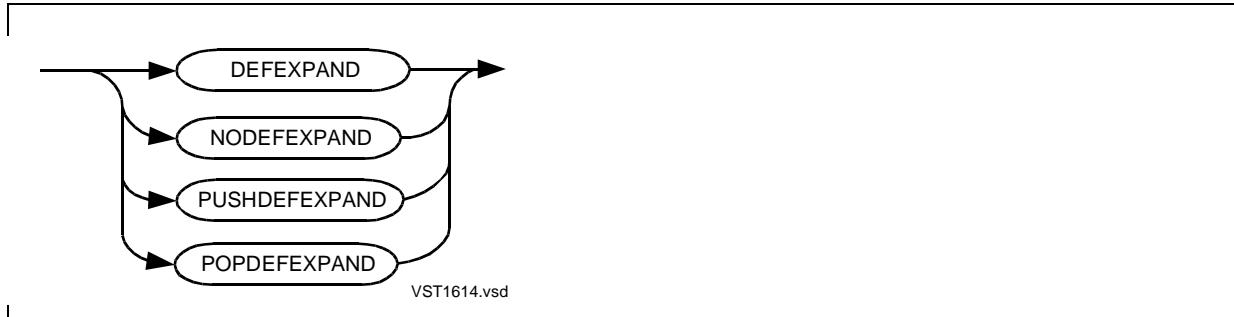
A DECS directive calibrates the compiler's internal S-register setting; DECS decrements the register setting by three words.

```
SUBPROC sp;
BEGIN
    !Lots of code
    STACK param1, param2, param3;      !Load parameters onto
                                         !register stack
    CODE (PUSH %722);                 !Push parameters onto
                                         !data stack
    CODE (PCAL proc_name);            !Call the procedure
?DECS 3                                !Decrement compiler's
                                         !S-register counter by
    !More code                               !three words
END;
```

## DEFEXPAND Directive

DEFEXPAND lists expanded DEFINEs and SQL-TAL code in the compiler listing.

The default is NODEFEXPAND.



## Usage Considerations

This directive can appear in the compilation command or anywhere in the source code.

**DEFEXPAND** turns the define-listing setting on for subsequent code. **DEFEXPAND** has no effect if **NOLIST** or **SUPPRESS** is in effect.

**NODEFEXPAND** turns the define-listing setting off for subsequent code.

**PUSHDEFEXPAND** pushes the current define-listing setting onto the directive stack without changing the current setting.

**POPDEFEXPAND** removes the top value from the directive stack and sets the current define-listing setting to that value.

## DEFEXPAND Listing

In the **DEFEXPAND** listing, the **DEFINE** body appears on lines following the **DEFINE** identifier. In the listing:

- Lowercase letters appear in uppercase.
- No comments, line boundaries, or extra blanks appear.
- The lexical level of the **DEFINE** appears in the left margin, starting at 1.
- Parameters to the **DEFINE** appear as **\$n** (C-series system) or **#n** (D-series system), where **n** is the sequence number of the parameter, starting at 1.

A single SQL statement might generate code that contains many TAL declarations or statements. **DEFEXPAND** includes such code in the listing.

## Example of DEFEXPAND Directive

This example requests that the compiler list the expanded **DEFINE**:

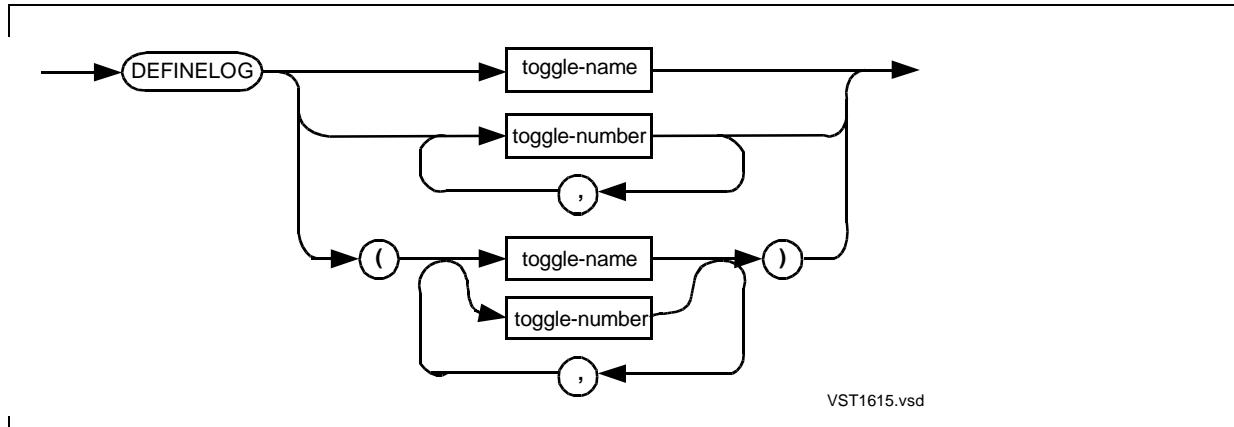
```

?DEFEXPAND                                !List expanded DEFINES
DEFINE increment (x) = x := x + 1#;        !Expanded DEFINE
DEFINE decrement (y) = y := y - 1#;        !Expanded DEFINE
                                                !Other global data declarations

```

# DEFINETOГ Directive

DEFINETOГ specifies named or numeric toggles, without changing any prior settings, for use in conditional compilation. DEFINETOГ is a D20 or later feature.



**toggle-name**

is a user-defined name that conforms to the TAL identifier format.

**toggle-number**

is an unsigned decimal constant in the range 1 through 15. Leading zeros are ignored.

## Usage Considerations

DEFINETOГ can appear anywhere in the source code but not in the compilation command. DEFINETOГ without a parenthesized list must be the last directive on the line. When DEFINETOГ has a parenthesized list, other directives can follow on the same line, with a comma separating the closing parenthesis from the next directive.

DEFINETOГ interacts with the IF, IFNOT, and ENDIF directives. IF and IFNOT test the setting of toggles and mark the beginning of conditional compilation. ENDIF marks the end of conditional compilation.

## Named Toggles

Before you use a named toggle in an IF or IFNOT directive, you must specify that name in a DEFINETOГ, SETTOГ, or RESETTOГ directive. Which of these directives

you use depends on whether you want settings of named toggles unchanged or turned on or off:

Directive	Setting of New Toggle	Setting of Specified Existing Toggle
DEFINETOГ	Off	Unchanged
SETTOГ	On	On
RESETTOГ	Off	Off

You can use DEFINETOГ if you are not sure the toggles were created earlier in the compilation, possibly in a file that you sourced in. If you specify toggles that already exist, DEFINETOГ does not change their settings (as SETTOГ and RESETTOГ do).

## Numeric Toggles

You can use a numeric toggle in an IF or IFNOT directive even if that number has not been specified in a DEFINETOГ, SETTOГ, or RESETTOГ directive.

By default, all numeric toggles not turned on by SETTOГ are turned off. To turn off numeric toggles turned on by SETTOГ, use RESETTOГ.

## Examples of DEFINETOГ Directive

1. This example specifies named and numeric toggles:

```
?DEFINETOГ (debug_version, new_version, 4, 7, 11)
```

2. In this example, IF finds the toggle SCANNER is off and causes the compiler to skip over the source text between IF SCANNER and ENDIF SCANNER:

```
?DEFINETOГ scanner           ! Specify toggle SCANNER
!Some code here

?IF scanner                 !Test toggle for on state
PROC skipped;               !Find it off; skip procedure
    BEGIN
        !More code here
    END;
?ENDIF scanner              !End of skipped part
```

3. In this example, IFNOT finds toggle EMITTER is off and causes the source text between IFNOT EMITTER and ENDIF EMITTER to be compiled:

```
?DEFINETOGL emitter           !Specify toggle EMITTER
!Some code here

?IFNOT emitter               !Test toggle for off state
PROC kept;
BEGIN
    !More code here
END;
?ENDIF emitter               !End of compiled part
```

4. In this example, SETTOG turns on toggle ON\_TOG. DEFINETOGL then specifies ON\_TOG but does not change its setting. IF finds the toggle is on and causes the source text between IF ON\_TOG and ENDIF ON\_TOG to be compiled:

```
?SETTOG on_tog              !Turn on toggle ON_TOG
!Lots of code here
?DEFINETOGL on_tog          !Specify toggle ON_TOG without
                            ! changing its setting

!Some code here

?IF on_tog                  !Test toggle for on state
PROC kept;
BEGIN
    !More code here
END;
?ENDIF on_tog               !End of compiled part
```

## DUMPCONS Directive

DUMPCONS inserts the contents of the compiler's constant table into the object code.



### Usage Considerations

DUMPCONS can appear any number of times anywhere in the source code. It cannot appear in the compilation command.

Each time DUMPCONS appears, the compiler immediately inserts the content of the compiler's constant table into the object code. (The constant table contains constants and labels.)

If DUMPCONS does not appear, the compiler normally inserts the content of the compiler's constant table into the object code at the end of a procedure. If an instruction can only reach constants located within 256 words forward or backward, however, the compiler inserts the constants accordingly.

The compiler generates an unconditional branch around each insertion of constants and labels into the object code.

## Use With CODE Statements

If you use CODE statements to create a block of data in your code, the compiler might insert constants or branch labels in the middle of your block of data. If you need to keep the data generated by your CODE statements in one contiguous sequence, put a DUMPCONS directive immediately before the CODE statements. The compiler then inserts all pending constants and branch labels into the object code before it compiles the CODE statements.

## Example of DUMPCONS Directive

This example emits collected constants and labels into the object code before building a doubleword, to ensure that the two single words are adjacent:

```
PROC m;
BEGIN
  !Lots of code
?DUMPCONS
  CODE (CON, 0);           !Build a doubleword value of 1
  CODE (CON, 1);           ! in the user code segment with
END;                      ! no intervening constants
```

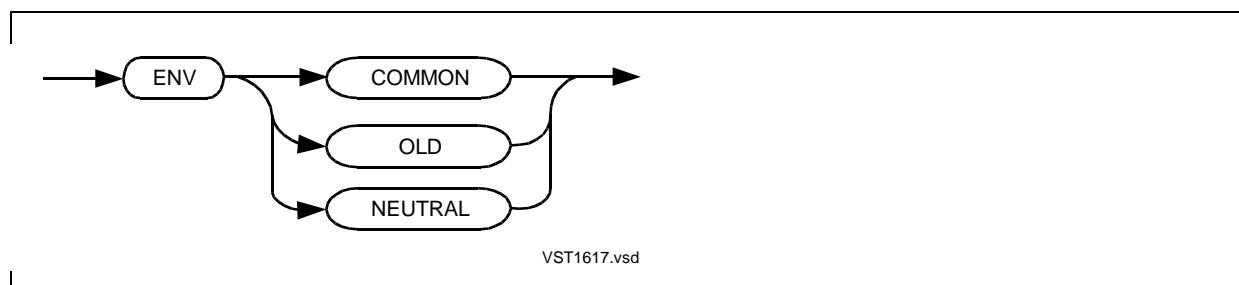
## ENDIF Directive

ENDIF is described under the IF directive in this section.

## ENV Directive

ENV specifies the intended run-time environment of a D-series object file.

The default is ENV NEUTRAL.



The meaning of the ENV attributes are:

<b>ENV Attribute</b>	<b>Intended Run-Time Environment</b>
COMMON	The CRE
OLD	A COBOL or FORTRAN run-time environment outside the CRE
NEUTRAL	None; the program relies primarily on system procedures

## Usage Considerations

ENV can appear only once in a compilation unit, either in the compilation command or in the source code before any declarations. Use ENV only with a D-series compilation unit.

ENV lets you specify the intended run-time environment of a D-series object file. To execute successfully, however, the object file must meet the requirements of the run-time environment.

If ENV is not in effect, all procedures 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.

When ENV COMMON is in effect, all procedures in the compilation unit (except procedures 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 procedure in a SEARCH file retains its original ENV attribute.

Using Binder, you can bind an object file compiled with ENV COMMON to any object file except those compiled with ENV OLD. Each procedure in the new object file retains its original ENV attribute.

When a compilation unit contains a MAIN procedure and ENV COMMON is in effect, the compiler allocates and initializes special data blocks as required by the CRE. For information on these data blocks and on the requirements of the CRE, see Section 17, Mixed-Language Programming, in the *TAL Programmer's Guide*.

### ENV OLD Directive

An object file compiled 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 procedures in the compilation unit (except procedures 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 procedure in a SEARCH file retains its original ENV attribute.

Using Binder, you can bind an object file compiled with ENV OLD to any object file except those compiled with ENV COMMON. Each procedure 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 procedures in the compilation unit (except procedures 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. All procedures in the new object file have the ENV NEUTRAL attribute.

Using Binder, you can bind an object file compiled with ENV NEUTRAL to any object file. Each procedure 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 procedures in the compilation unit (except procedures in object files listed in SEARCH directives) have the ENV NEUTRAL attribute. SEARCH directives can list object files compiled with any ENV attribute. Each procedure 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 procedure in the new object file retains its original ENV attribute.

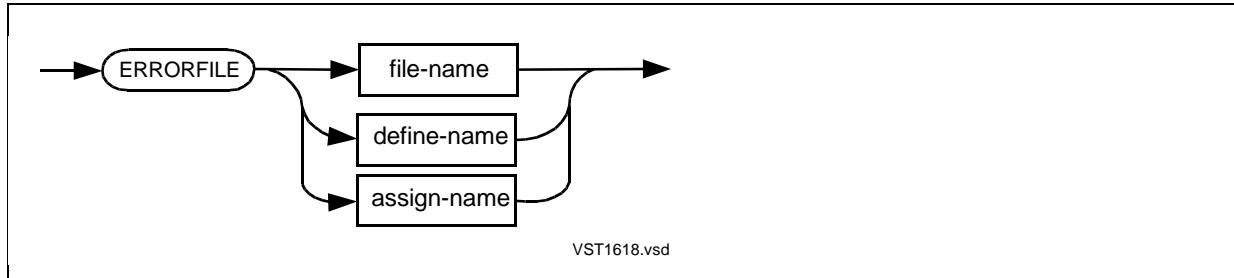
## **Examples of ENV Directive**

This example specifies the COMMON option of the ENV directive:

```
?ENV COMMON
```

## **ERRORFILE Directive**

ERRORFILE logs compilation errors and warnings to an error file so you can use the TACL FIXERRS macro to view the diagnostic messages in one PS Text Edit window and correct the source file in another window.



**file-name**

is the name of either:

- An existing error file created by ERRORFILE. Such a file has file code 106 (an entry-sequenced disk file used only with the TACL FIXERRS macro). The compiler purges any data in it before logging errors and warnings.
- A new error file to be created by ERRORFILE if errors occur.

If a file with the same name exists but the file code is not 106, the compiler terminates compilation to prevent overwriting the file.

You can specify partial file names as described in Appendix E in the *TAL Programmer's Guide*. The compiler uses the current default volume and subvolume names as needed. For this directive, the compiler does not use TACL ASSIGN SSV information to complete the file name.

**define-name**

is the name of a TACL MAP DEFINE that refers to an error file.

**assign-name**

is a logical file name you have equated with an error file by issuing a TACL ASSIGN command.

## Usage Considerations

ERRORFILE can appear in the compilation command or in the source code before any declarations.

The compiler writes a header record to the error file and then writes a record for each error or warning. Each record contains information such as:

- The location of the error or warning—source file name, edit line number, and column number
- The message text of the error or warning

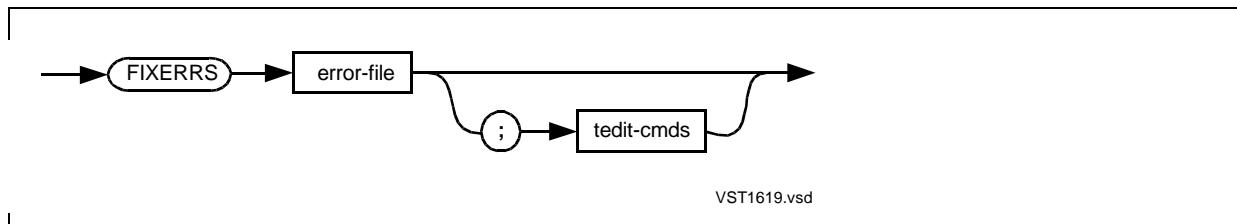
At the end of the compilation, the compiler prints the complete name of the error file in the trailer message of the compilation listing.

## FIXERRS Macro

After the compiler logs messages to the error file, you can invoke the TACL FIXERRS macro and correct the source file. FIXERRS uses the PS Text Edit ANYHOW option to open the source file in a two-window session. One window displays a diagnostic message. The other window displays the source code to which the message applies. If you have write access to the file, you can correct the source code. If you have only read access, you can view the source code, but you cannot correct it.

Initially, the edit cursor is located in the source code at the first diagnostic. To move the cursor to the next or previous diagnostic, use the PS Text Edit NEXTERR or PREVERR command.

The TACL command for invoking FIXERRS is:



**error-file**

is the name of the error file specified in the ERRORFILE directive.

**tedit-cmds**

are any PS Text Edit commands that are allowed on the PS Text Edit run line.

The following example issues a TACL DEFINE command that invokes FIXERRS and defines PS Text Edit function keys for NEXTERR and PREVERR:

```
[ #DEF MYFIXERRS MACRO |BODY|
  FIXERRS %1%; SET <F9>, NEXTERR; SET <SF9>, PREVERR] ]
```

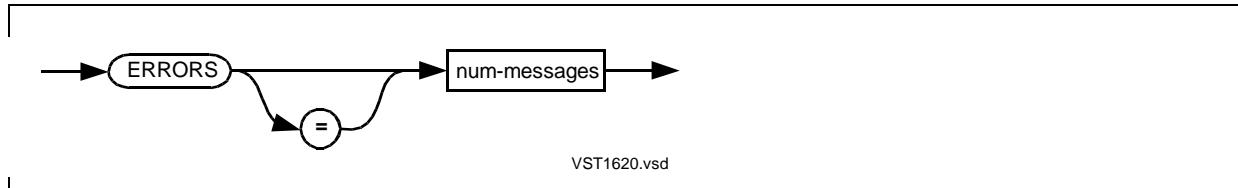
## Example of ERRORFILE Directive

This example specifies MYERRORS as the file to which the compiler is to report compilation errors or warnings:

```
!This is MYSOURCE file
?ERRORFILE myerrors
!Global declarations
```

# ERRORS Directive

ERRORS sets the maximum number of error messages to allow before the compiler terminates the compilation.



**num-messages**

is the maximum number of error messages to allow before the compilation terminates. Specify an unsigned decimal constant in the range 0 through 32,767.

## Usage Considerations

ERRORS can appear in the compilation command or anywhere in the source code. A single error can cause many error messages. The compiler counts each error message separately. If the compiler's count exceeds the maximum you specify, the compiler terminates the compilation. (Warning messages do not affect the count.)

If you do not specify ERRORS, the compiler does not terminate the compilation because of the number of errors.

## Example of ERRORS Directive

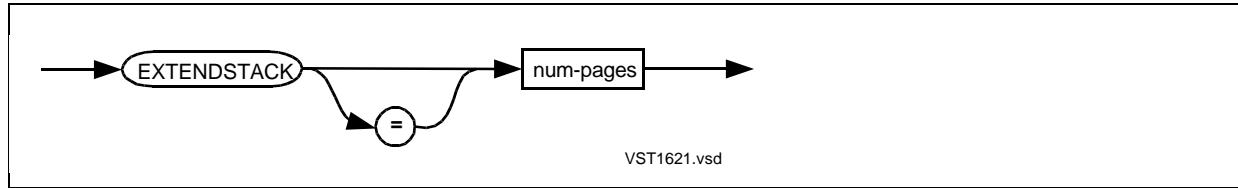
This example specifies that the compiler terminate the compilation when 10 error messages are emitted:

```

!This is MYSOURCE file
?ERRORS 10
!Global declarations
  
```

# EXTENDSTACK Directive

EXTENDSTACK increases the size of the data stack in the user data segment.



`num-pages`

is the number of 2048-byte memory pages to add to Binder's estimate of the data stack size. Specify an unsigned decimal constant in the range 0 through 32.

## Usage Considerations

`EXTENDSTACK` can appear in the compilation command or anywhere in the source code. If you omit this directive, the default is the data stack size estimated by Binder.

The following directives override the `EXTENDSTACK` directive:

- A `DATAPAGES` directive that specifies a `num-pages` value greater than 32.
- An `ENV COMMON` directive in a compilation unit that contains a `MAIN` procedure. The compiler allocates 64K words for the user data segment.

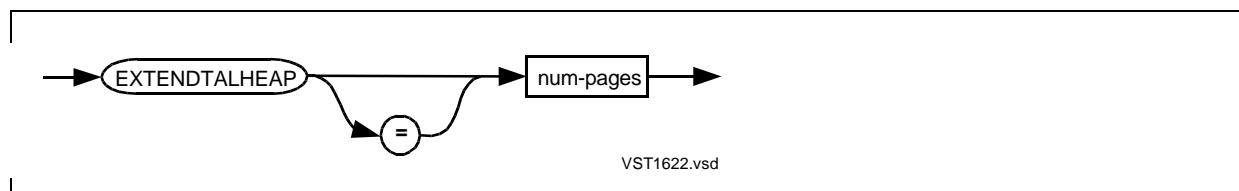
## Example of `EXTENDSTACK` Directive

This example adds 20 memory pages to Binder's estimate of the data stack in the user data segment:

```
?EXTENDSTACK 20
```

## `EXTENDTALHEAP` Directive

`EXTENDTALHEAP` increases the size of the compiler's internal heap for a D-series compilation unit.



`num-pages`

is the number of 2048-byte memory pages to add to the compiler's internal heap. Specify an unsigned decimal constant in the range 0 through 32,767. The default value is 0.

## Usage Considerations

`EXTENDTALHEAP` can appear only in the compilation command and should appear before any directives except `SQL` or `SYMBOLPAGES`. It can appear only once in a compilation unit. Use `EXTENDTALHEAP` only with D-series compilation units.

If your program requires a larger compiler heap than is currently available, the compiler issues error 169 (increase the size of the internal heap of the compiler by recompiling with the `EXTENDTALHEAP` directive).

If a heap overflow terminates your compilation, you must specify the EXTENDTALHEAP directive in all subsequent compilations.

## Example of EXTENDTALHEAP Directive

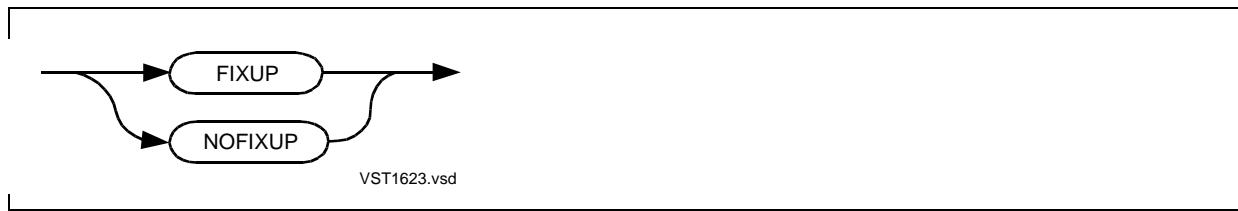
This compilation command starts the compiler and adds 120 memory pages to the compiler's internal heap:

```
TAL /in mysrc/ myobj; EXTENDTALHEAP 120
```

## FIXUP Directive

FIXUP directs BINSERV to perform its fixup step.

The default is FIXUP.



## Usage Considerations

This directive can appear in the compilation command or anywhere in the source code. The last instance of this directive determines whether BINSERV does fixups.

If FIXUP is in effect, BINSERV:

- Assigns values for primary global pointers that point to secondary global variables or extended global variables
- Fixes code references to data space (for example, LOAD G+...)
- Fixes code references to code space (for example, PCAL instructions or references to global read-only arrays)

NOFIXUP suppresses the fixup step. Use NOFIXUP if you are creating a file for later binding or for use as a code resource in a later compilation. If the resulting object file is not executed, BINSERV need not do fixups.

## Example of FIXUP Directive

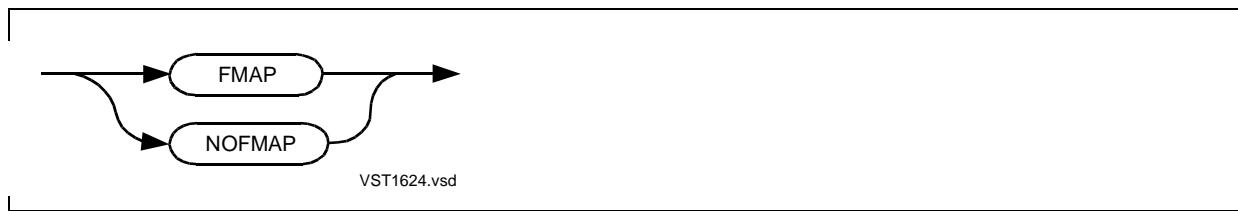
This example requests that BINSERV not perform its fixup step because the file is created for later binding:

```
PROC last_proc;
    BEGIN
        !Lots of code
    END;
?NOFIXUP
```

## FMAP Directive

FMAP lists the file map.

The default is NOFMAP.



## Usage Considerations

FMAP and NOFMAP can appear in the compilation command or anywhere in the source code. The last FMAP or NOFMAP encountered in the compilation unit determines whether the compiler lists the file map or not.

FMAP has no effect if either NOLIST or SUPPRESS is in effect.

NOFMAP suppresses the file map.

## File Map Listing

The file map starts with the first file the compiler encounters and includes 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.

In the compilation listing, the file map appears after the map of global identifiers.

## Examples of FMAP Directive

1. This example enables printing of the file map, which shows the fully qualified name of each file from which the compilation read source text:

?FMAP

2. This example disables printing of the file map:

?NOFMAP

## GMAP Directive

GMAP lists the global map.

The default is GMAP.



## Usage Considerations

You can specify GMAP or NOGMAP in the compilation command or anywhere in the source code. The last GMAP or NOGMAP encountered in the compilation unit determines whether the compiler lists the global map or not.

GMAP has no effect if NOLIST, NOMAP, or SUPPRESS is in effect.

NOGMAP suppresses the global map even if MAP is in effect.

## Global Map Listing

The global map lists all identifiers in the compilation unit and tells what kind of object they are, including identifier class and type.

The global map appears at the end of the compilation listing.

## Examples of GMAP Directive

1. This example enables printing of the global map, which lists all identifiers and gives their class and type:

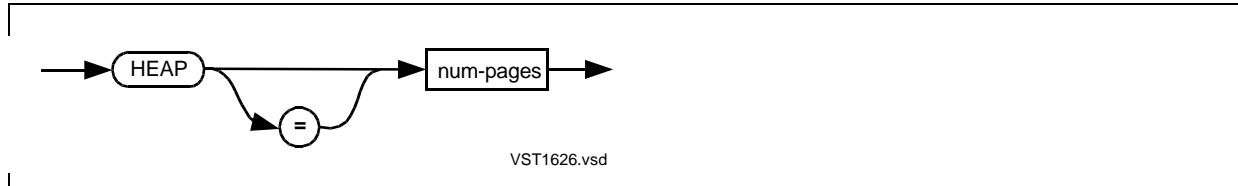
?GMAP

2. This example disables printing of the global map:

```
?NOGMAP
```

## HEAP Directive

HEAP sets the size of the CRE user heap for a D-series compilation unit if the ENV COMMON directive is in effect.



`num-pages`

is the number of 2048-byte memory pages to allocate for the CRE user heap (named #HEAP). Specify an unsigned decimal constant in either of the following ranges:

- 0 through 31—if your program contains C or Pascal small-memory-model routines
- 0 through 32,767—if your program either:
  - Contains C or Pascal large-memory-model routines
  - Contains no C or Pascal routines

## Usage Considerations

HEAP can appear in the compilation command or anywhere in the source code. You can specify HEAP any number of times. The compiler uses the size specified in the last HEAP directive encountered in the compilation unit. Use HEAP only with D-series compilation units that include the ENV COMMON directive. If your program invokes a routine that needs the user run-time heap, you must use HEAP.

For the small-memory model, Binder allocates the CRE user heap as the last global data block (just below the data stack) in the user data segment. Your use of the lower 32K-word area of the user data segment determines what heap size you can specify.

For the large-memory model, Binder allocates the CRE user heap as the last data block in the automatic extended data segment. If you must also allocate an extended data segment yourself, follow the directions in Appendix B, “Managing Addressing,” in the *TAL Programmer’s Guide*.

For information on using the CRE user heap, see Section 17, Mixed-Language Programming, in the *TAL Programmer’s Guide*.

## Example of HEAP Directive

This example sets the size of the user run-time heap for the large memory model:

```
?HEAP 200
```

## HIGHPIN Directive

HIGHPIN sets the HIGHPIN attribute in a D-series object file.



## Usage Considerations

HIGHPIN can appear in the compilation command or anywhere in the source code. HIGHPIN can appear any number of times in a compilation. It need not appear in each source file, just once in the compilation unit.

When the operating system creates a process, it assigns a process identification number (PIN) to the process. D-series systems support the following ranges of PINs:

Low-PIN range	0 through 254
High-PIN range	256 through the maximum number supported for the processor in which the process runs

To run an object file at high PIN from the TACL prompt, the following conditions must be met:

- Your processor is configured for more than 256 process control blocks (PCBs).
- High pins are available in your processor.
- Your object file and user library, if any, have the HIGHPIN attribute set.
- The TACL HIGHPIN built-in variable or the HIGHPIN run-time parameter is set.

If the HIGHPIN attribute of the object file is set, the operating system assigns a high PIN, if available. If no high PINs are available, the operating system assigns a low PIN.

Each object file in the target file must have the HIGHPIN attribute set. You can set the HIGHPIN attribute of an object file either:

- During compilation by using the HIGHPIN directive
- After compilation by using a Binder command

If the preceding conditions are met, your object file can create another process to run at high PIN by specifying the PROCESS\_CREATE\_ system procedure with create-options bit 15 set to 0 and bit 10 set to 1.

The following sequence of examples show how to run an object file at high PIN from the TACL prompt. The examples show how to check your processor configuration and high-PIN availability, set the HIGHPIN attribute, and override the TACL HIGHPIN setting if it is off.

## Examples of Running Object Files at HIGHPIN

1. To check the number of PCBs configured in your processor and to see if high PINs are available, run the Peek product. For example, suppose you want to run your object file on processor 1:

```
PEEK / CPU 1 /
```

The following display excerpt shows example values for the information you need to check:

	... CURRENT USAGE	# CONFIGURED ...
PCB	127: 48	255: 244

The processor is configured for high PINS if the sum of the two values displayed for PCBs under # CONFIGURED is 256 or greater.

The processor has high PINs available if the right-hand value under CURRENT USAGE is less than the right-hand value under # CONFIGURED.

2. You can set the HIGHPIN attribute of an object file during compilation by including the HIGHPIN directive in the compilation command:

```
TAL /IN talsrc, OUT $S.#tallst, NOWAIT/ talobj; HIGHPIN
```

3. Alternatively, you can set the HIGHPIN attribute of an object file after compilation by typing the following Binder command:

```
BIND CHANGE HIGHPIN ON IN talobj
```

4. Before you run the object file, you can check the current setting of the TACL HIGHPIN built-in variable by typing:

```
#HIGHPIN
```

5. If #HIGHPIN returns a NO value, you can set the HIGHPIN run-time parameter (and run your object file at high PIN):

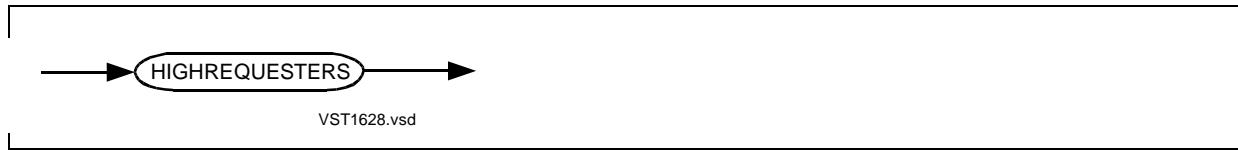
```
RUN talobj / HIGHPIN ON /
```

6. If #HIGHPIN returns a YES value, you can simply run your object file at high PIN:

```
RUN talobj
```

# HIGHREQUESTERS Directive

HIGHREQUESTERS sets the HIGHREQUESTERS attribute in a D-series object file.



## Usage Considerations

HIGHREQUESTERS can appear in the compilation command or anywhere in the source code. HIGHREQUESTERS can appear any number of times in a compilation. It need not appear in each source file, just once in the compilation unit.

HIGHREQUESTERS sets the HIGHREQUESTERS attribute in the object file, which means the object file supports HIGHPIN requesters. If you do not specify HIGHREQUESTERS, the object file cannot support HIGHPIN requesters (the default).

An object file can be called by D-series HIGHPIN requesters if the object file:

- Is compiled with HIGHREQUESTERS in effect
- Contains a MAIN procedure
- Fulfills other requirements described in the *Guardian Application Conversion Guide*

As an alternative, you can set the HIGHREQUESTERS attribute after compilation by using Binder commands.

## Examples of HIGHREQUESTERS Directive

1. This example specifies the HIGHREQUESTERS directive in a directive line in a D-series source file:

```
?HIGHREQUESTERS
```

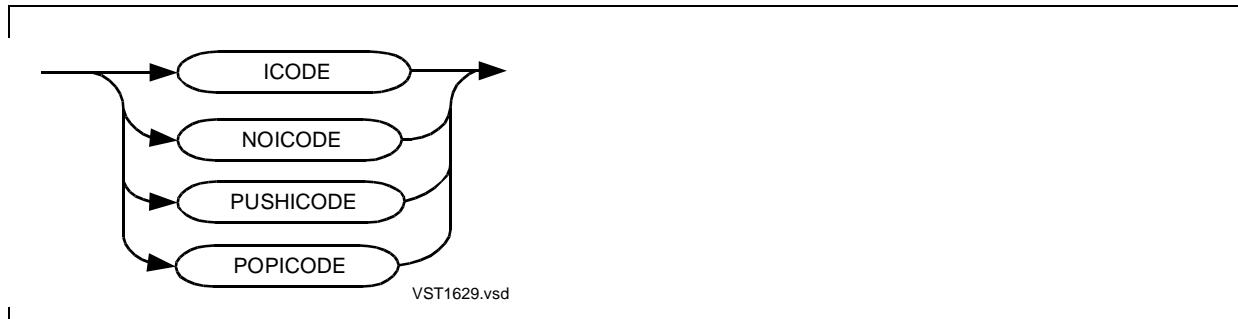
2. This example specifies the HIGHREQUESTERS directive in a compilation command to compile a D-series source file:

```
TAL /IN tsrc, OUT $S.#lst, NOWAIT/ tobj; HIGHREQUESTERS
```

# ICODE Directive

ICODE lists the instruction-code (icode) mnemonics for subsequent procedures.

The default is NOICODE.



## Usage Considerations

This directive can appear in the compilation command or anywhere in the source code. ICODE lists mnemonics for entire procedures and cannot be turned on and off within a procedure. (By contrast, INNERLIST lists mnemonics for statements and can be turned on and off within procedures.)

ICODE turns the icode-listing setting on for subsequent procedures. ICODE has no effect if NOLIST or SUPPRESS is in effect.

NOICODE turns the icode-listing setting off for subsequent procedures.

PUSHICODE pushes the current icode-listing setting onto the directive stack without changing the current setting.

POPICODE removes the top value from the directive stack and sets the current icode-listing setting to that value.

## ICODE Listing

For each procedure for which ICODE is in effect, the icode listing follows the code listing if any.

For global variables declared within a named data block, the G+ addresses shown are relative to the start of the data block and are not the final addresses. At the end of compilation, BINSERV determines the final G+ addresses of such global variables and the code locations of items such as PCAL instructions and global read-only arrays.

After compilation, you can display the final addresses by using Binder and Inspect commands.

## Example of ICODE Directive

This compilation command lists the icode mnemonics starting with the first procedure to be compiled:

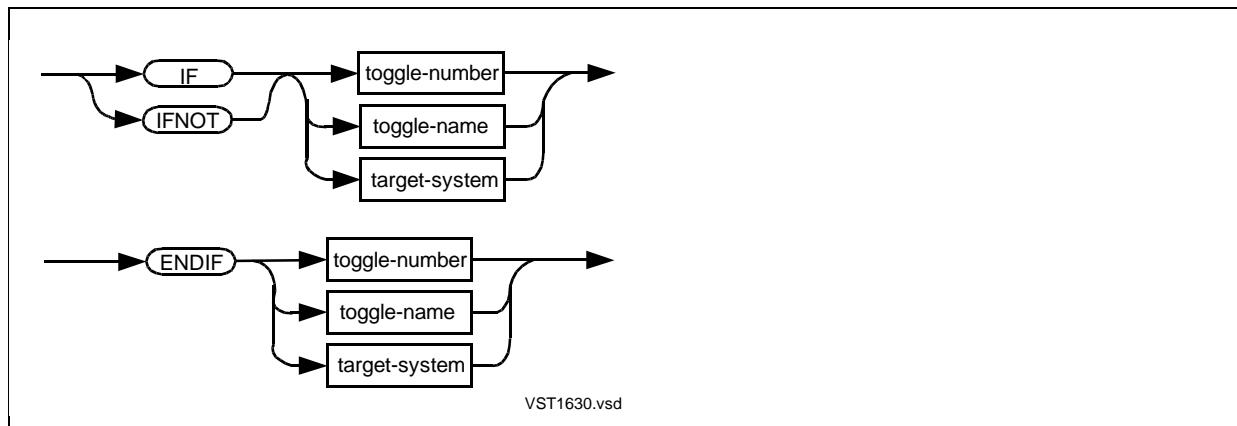
```
TAL /IN mysrc, OUT $s.#lists/ myobj; ICODE
```

## IF and ENDIF Directives

IF and IFNOT control conditional compilation based on a condition.

The ENDIF directive terminates the range of the matching IF or IFNOT directive.

The D20 or later release supports named toggles and target-system toggles in addition to numeric toggles.



**toggle-number**

is an integer value from 1 through 15 specified in a previous DEFINETOG, SETTOG, or RESETTOG directive. Leading zeros are ignored.

**toggle-name**

is a toggle name specified in a previous DEFINETOG, SETTOG, or RESETTOG directive.

**target-system**

is one of:

ANY	ANY is specified in a TARGET directive in this compilation.
TARGETSPECIFIED	A TARGET directive appears in this compilation.
TNS_ARCH	TNS_ARCH is specified in a TARGET directive in this compilation.
TNS_R_ARCH	TNS_R_ARCH is specified in a TARGET directive in this compilation.

## Usage Considerations

IF, IFNOT, and ENDIF can appear in a directive line anywhere in the source code but not in the compilation command. IF and IFNOT must appear last in any directive line. ENDIF must be the only directive on the directive line.

For ENDIF, always specify a toggle that matches a previous IF or IFNOT toggle. ENDIF terminates the compilation begun by the IF or IFNOT that specifies the matching toggle.

### Named or Numeric Toggles

For named or numeric toggles, IF causes compilation of subsequent text only if you turn the same toggle on with SETTOG.

IFNOT causes compilation of subsequent text only if the same toggle is turned off. A toggle is turned off if you create it with DEFINETOG or RESETTOG, or if you create it with SETTOG and then turn it off with RESETTOG.

For both IF and IFNOT, the skipping of text, once begun, continues until the matching ENDIF directive appears. In the following fragment, the compiler skips both parts if the toggle is off, because no ENDIF appears for IF. Avoid this error:

```
? RESETTOG flag           !Create FLAG in off state
?IF flag
    !Statements for true condition
    ! (skipped because FLAG is off)
?IFNOT flag
    !Statements for false condition
    ! (also skipped because no ENDIF appears for IF FLAG)
?ENDIF flag
```

If you insert an ENDIF for IF in the preceding fragment, the compiler skips only the first part:

```
? RESETTOG flag           !Create FLAG in off state
?IF flag
    !Statements for true condition
    ! (skipped because FLAG is off)
?ENDIF flag           !ENDIF stops the skipping of statements
?IFNOT flag
    !Statements for false condition
    ! (compiled because ENDIF appears for IF FLAG)
?ENDIF flag
```

## Target-System Toggle

For target-system toggles, IF causes compilation of subsequent text only if you turned the same toggle on with TARGET.

You can specify *target-system*, for example, when the code deals with the processor state or memory management tables or other processor-specific areas.

For information on how IF, IFNOT, and ENDIF work with the TARGET directive, see the TARGET directive, described in [Section 15, Privileged Procedures](#).

## Compiler Listing

An asterisk (\*) appears in column 10 of the listing for any statements not compiled because of the IF or IFNOT directive.

## Examples of IF and ENDIF Directives

1. In this example, DEFINETOG creates toggle OMIT and leaves it turned off. IF tests the toggle for the on state, finds it is off, and causes the compiler to skip the source text between IF OMIT and ENDIF OMIT:

```
?DEFINETO omit          !Create toggle OMIT in off state
!Some code here

?IF omit               !Test toggle for on state
PROC lost;             !Find it off; skip procedure
BEGIN
    !More code here
END;
?ENDIF omit           !End of skipped portion
```

2. In this example, SETTOG creates toggle KEEP and turns it on. IF tests the toggle for the on state, finds it is on, and causes the compiler to compile the source text between IF KEEP and ENDIF KEEP:

```
?SETTOG keep           !Create toggle KEEP; turn it on
!Some code here

?IF keep               !Test toggle for on state
PROC kept;             !Find it on; compile procedure
BEGIN
    !More code here
END;
?ENDIF keep            !End of compiled portion
```

3. In this example, SETTOG creates toggle DONE and turns it on. IFNOT tests the toggle for the off state, finds it is on, and causes the compiler to skip the source text between IF DONE and ENDIF DONE:

```
?SETTOG done           !Create toggle DONE; turn it on
!Some code here

?IFNOT done           !Test toggle for off state
PROC skipped;         !Find it on; skip procedure
BEGIN
  !Lots of code
END;
?ENDIF done           !End of skipped portion
```

4. In this example, SETTOG turns on toggle 1. IF tests the toggle, finds it is on, and causes the compiler to compile the source text between IF 1 and ENDIF 1:

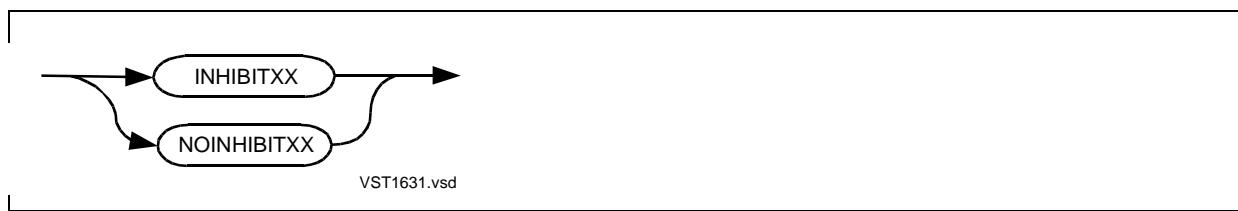
```
?SETTOG 1             !Turn toggle 1 on
!Some code here

?IF 1                 !Test toggle for on state;
PROC kept;            !Find it on; compile procedure
BEGIN
  !More code
END;
?ENDIF 1              !End of compiled portion
```

## INHIBITXX Directive

INHIBITXX generates inefficient but correct code for extended global declarations in relocatable blocks that Binder might locate after the first 64 words of the primary global area of the user data segment.

The default is NOINHIBITXX.



## Usage Considerations

INHIBITXX or NOINHIBITXX can appear in the compilation command or any number of times in the compilation unit before the first procedure declaration.

INHIBITXX turns the INHIBITXX setting on for subsequent declarations.

NOINHIBITXX turns the INHIBITXX setting off for subsequent declarations. Such declarations retain the INHIBITXX or NOINHIBITXX attribute throughout the compilation.

NOINHIBITXX enables compiler use of XX instructions (LWXX, SWXX, LBXX, and SBXX), which generate efficient indexing for extended declarations located between G[0] and G[63]. Binder, however, might relocate such extended declarations beyond G[63] when you bind separately compiled modules or bind TAL code with code written in other languages. The indexing code then becomes incorrect and Binder issues error 20 (data reference failed due to relocation).

INHIBITXX suppresses compiler use of XX instructions and generates inefficient but correct indexing for extended declarations located between G[0] and G[63] even if Binder relocates such declarations after G[63]. INHIBITXX does not generate correct indexing if the offset of a structure data item from the zeroth occurrence is out of the INT range (-32,768 through 32,767). For more information, see the INT32INDEX directive in this section.

If you specify INHIBITXX before one or more data declarations, the INHIBITXX attribute applies to those particular data items throughout the compilation.

INHIBITXX and NOINHIBITXX have no effect on:

- Extended declarations above G[63] of the user data segment. The compiler never uses XX instructions for such declarations.
- Extended declarations within BLOCK declarations with the AT (0) or BELOW (64) option. The compiler automatically generates efficient code for such declarations. Binder does not relocate such declarations after G[63].

The INT32INDEX directive overrides INHIBITXX or NOINHIBITXX, whichever is in effect.

For information on the XX instructions, see the *System Description Manual* for your system.

## Example of INHIBITXX Directive

This example shows use of NOINHIBITXX, which generates efficient addressing, and INHIBITXX, which suppresses efficient addressing:

```

        !Default NOINHIBITXX in effect

STRUCT .EXT xstruct[0:9];!XSTRUCT has NOINHIBITXX attribute
BEGIN
STRING array[0:9];
END;

INT index;
STRING var;
?INHIBITXX           !Set INHIBITXX attribute for}
                      ! for next declaration

STRING .EXT xstruct2 (xstruct);
                           !XSTRUCT2 has INHIBITXX attribute

PROC my_proc MAIN;
BEGIN
@xstruct2 := @xstruct;
var := xstruct[index].array[0];
                           !Generate efficient addressing
                           ! because XSTRUCT has NOINHIBITXX
                           ! attribute, but if Binder
                           ! relocates this declaration
                           ! beyond G[63], the addressing
                           ! code is incorrect

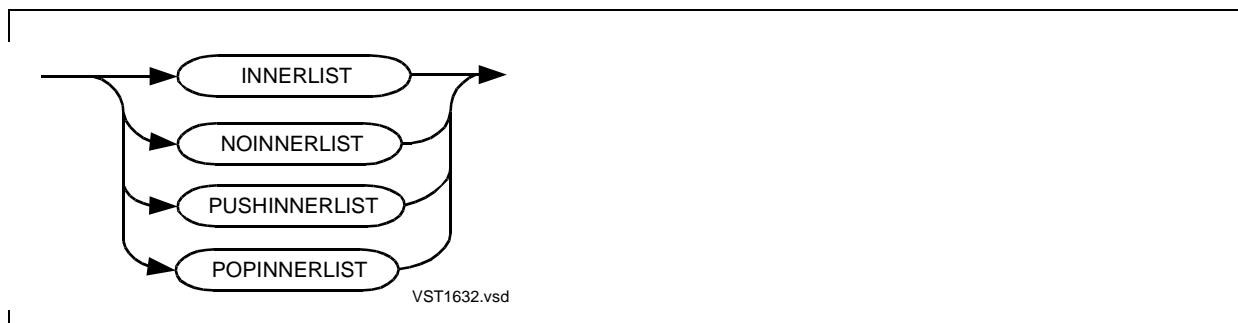
var := xstruct2[index].array[0];
                           !Generate inefficient addressing
                           ! because XSTRUCT2 has INHIBITXX
                           ! attribute; the addressing is
                           ! correct even when the
                           ! declaration is relocated beyond
                           ! G[63]
END;

```

## INNERLIST Directive

INNERLIST lists the instruction code mnemonics (and the compiler's RP setting) for each statement.

The default is NOINNERLIST.



## Usage Considerations

This directive can appear in the compilation command or anywhere in the source code.

INNERLIST turns the innerlisting setting on and lists mnemonics for subsequent statements and can be turned on and off within procedures. (By contrast, ICODE lists mnemonics for entire procedures and cannot be turned on and off within procedures.) It has no effect if NOLIST or SUPPRESS is in effect.

NOINNERLIST turns the innerlisting setting off for subsequent statements.

PUSHINNERLIST pushes the current innerlisting setting onto the directive stack without changing the current setting.

POPINNERLIST removes the top value from the directive stack and sets the current innerlisting setting to that value.

### INNERLIST Listing

INNERLIST lists mnemonics after each statement for which INNERLIST is in effect, rather than at the end of the procedure.

For global variables declared within a named data block, the G+ addresses shown are relative to the start of the data block and are not the final addresses. At the end of compilation, BINSERV determines the final G+ addresses of such global variables and the code locations of items such as PCAL instructions and global read-only arrays.

The INNERLIST listing is less complete than the ICODE listing. Because the compiler is a one-pass compiler, many instructions appear with skeleton or space-holder images that the compiler or BINSERV modifies later.

If OPTIMIZE and INNERLIST are in effect, the compiler first lists the original code, then reports “Optimizer replacing the last n instructions,” and lists the optimized code. If INNERLIST is switched off too soon, only the original code might show.

### Example of INNERLIST Considerations

This example lists instruction code mnemonics after certain statements:

```
PROC any;
BEGIN
    INT x, y, z;                      !No innerlisting here
    !Statements that initialize variables
?INNERLIST                         !Start innerlisting here
    !Statements that manipulate variables
?NOINNERLIST                        !Stop innerlisting here
END;
```

# INSPECT Directive

INSPECT sets the Inspect product as the default debugger for the object file.

The default is NOINSPECT.



## Usage Considerations

This directive can appear in the compilation command or anywhere in the source code. The last INSPECT or NOINSPECT directive in a source file takes effect for the object file.

If you omit the INSPECT directive, Debug is the default debugger for the object file.

After compilation, you can select the Inspect product as the default debugger when you use Binder or run the object code. If you set the Inspect product in the object file, you cannot override it at run time.

## Effect of Other Directives

If SAVEABEND is in effect, BINSERV and Binder automatically turn the INSPECT setting on. If NOINSPECT is in effect, BINSERV and Binder turn SAVEABEND off.

If you want to reference variables symbolically in an Inspect session, you must include one or more SYMBOLS directives in your source code. SYMBOLS generates the symbol table in the object file. You can generate symbols for the entire source file or for individual procedures or data blocks. (If you do not specify SYMBOLS, the Inspect product only recognizes procedure names and code locations.)

## Example of INSPECT Directive

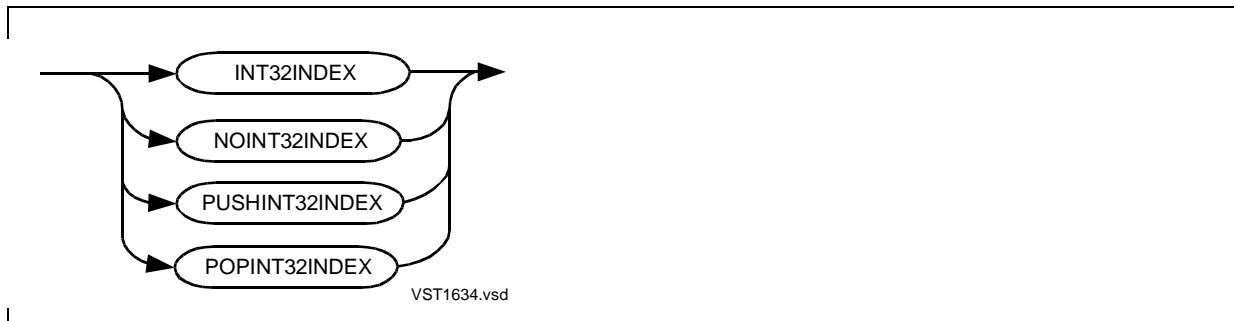
This example requests the Inspect product and saves symbols for the object file for use with the Inspect product:

```
?INSPECT, SYMBOLS
PROC a;
BEGIN
  !Lots of code
END;
```

# INT32INDEX Directive

INT32INDEX generates INT(32) indexes from INT indexes for accessing items in an extended indirect structure in a D-series program.

The default is NOINT32INDEX.



## Usage Considerations

This directive can appear in the compilation command or any number of times in a D-series compilation unit.

INT32INDEX turns the INT32INDEX setting on for subsequent declarations.

NOINT32INDEX turns the INT32INDEX setting off for subsequent declarations. Such declarations retain their INT32INDEX or NOINT32INDEX attribute throughout the compilation.

PUSHINT32INDEX pushes the current INT32INDEX setting onto the directive stack without changing the current setting.

POPINT32INDEX removes the top value from the directive stack and sets the current INT32INDEX setting to that value.

## INT32INDEX

INT32INDEX generates an INT(32) index from an INT index for a structure item in an extended indirect structure. INT32INDEX thus produces a correct extended offset even when the word offset of the structure field (from the zeroth structure occurrence) is greater than the signed INT range (-32,768 through 32,767). (For a STRING item or a substructure, the byte offset must be in this range.)

INT32INDEX overrides the INHIBITXX or NOINHIBITXX directive.

## NOINT32INDEX

NOINT32INDEX produces code that is faster than that produced by INT32INDEX, but it produces an incorrect extended offset when the offset of the structure field is outside the signed INT range. In such cases, you must use INT(32) indexing as discussed in Section 8, “Using Structures,” in the *TAL Programmer’s Guide*.

NOINT32INDEX does not override INHIBITXX or NOINHIBITXX.

## Example of INT32INDEX Directive

The following example shows the use of INT32INDEX, which always generates correct offsets, and NOINT32INDEX, which generates incorrect offsets in certain cases:

```
?INT32INDEX                                ! Set INT32INDEX attribute
                                             ! for next declaration

STRUCT .EXT xstruct[0:9999];!XSTRUCT has INT32INDEX
  BEGIN                                     ! attribute
    STRING array[0:9];
  END;

INT index;

PROC my_proc MAIN;
  BEGIN
?NOINT32INDEX                                ! Set NOINT32INDEX
                                             ! attribute for next
                                             ! declaration

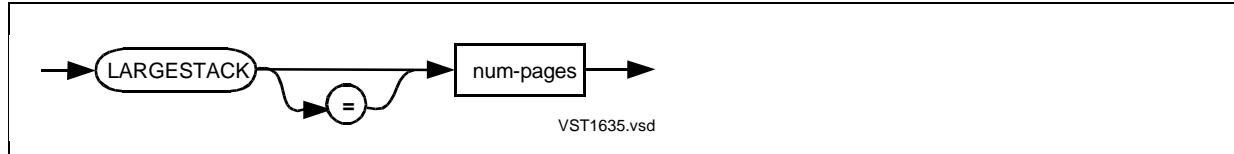
  INT .EXT xstruct2 (xstruct) := @xstruct[0];
                                             !XSTRUCT2 has NOINT32INDEX
                                             ! attribute

  xstruct[index].array[0]:= 1;!Generate correct offset even
                            ! when offset is > 32,767,
                            ! because XSTRUCT has
                            ! INT32INDEX attribute

  xstruct2[index].array[0] := 1;
                            !Generate incorrect offset
                            ! if offset is > 32,767,
                            ! because XSTRUCT2 has
                            ! NOINT32INDEX attribute
END;
```

# LARGESTACK Directive

LARGESTACK sets the size of the extended stack in the automatic extended data segment.



*num-pages*

is the number of 2048-byte memory pages to allocate for the extended stack.

Specify an unsigned decimal constant in the range 0 through 32,767. The default is the maximum extended stack space required by any single procedure or 64K bytes, whichever is greater.

## Usage Considerations

LARGESTACK can appear in the compilation command or anywhere in the source code. The last instance of LARGESTACK sets the extended stack size.

If you compile several files separately and then bind them to produce a new object file, Binder chooses the extended stack size as follows:

- On a D-series system, Binder chooses the extended stack size provided by the object file that contains the MAIN procedure.
- On a C-series system, Binder chooses the largest extended stack size provided among the object files.

After compilation, you can set the extended stack size by using the SET LARGESTACK command of Binder. The block name of the extended stack is \$EXTENDED#STACK.

## Example of LARGESTACK Directives

This example sets the size of the extended stack to 128 memory pages:

```

PROC last_proc;
    BEGIN
        !Lots of code
    END;
?LARGESTACK 128

```

# LIBRARY Directive

LIBRARY specifies the name of the TNS software user run-time library to be associated with the object file at run time.



*file-name*

specifies a user run-time library to search before searching the system library to satisfy external references. *file-name* must be the name of a disk file. It cannot be the name of a TACL DEFINE or a TACL ASSIGN logical file.

You can specify partial file names as described in Appendix E in the *TAL Programmer's Guide*. The compiler uses the current default volume and subvolume names as needed. For this directive, the compiler does not use TACL ASSIGN SSV information to complete the file name.

## Usage Considerations

This directive can appear in the compilation command or anywhere in the source code.

After compilation, you can change the library name by using Binder commands or by using the LIB option of the TACL RUN command.

## Example of LIBRARY Directive

This example specifies that a user library named MYLIB be associated with the object file at run time:

```

!Lots of code
?LIBRARY mylib
!More code

```

## About User Libraries

A user library is a set of procedures that the operating system can link to a program file at run time. User libraries are available in TAL and FORTRAN and COBOL85 programs, but not in Pascal.

User libraries provide many benefits to programs. You can place commonly used procedures in a user library:

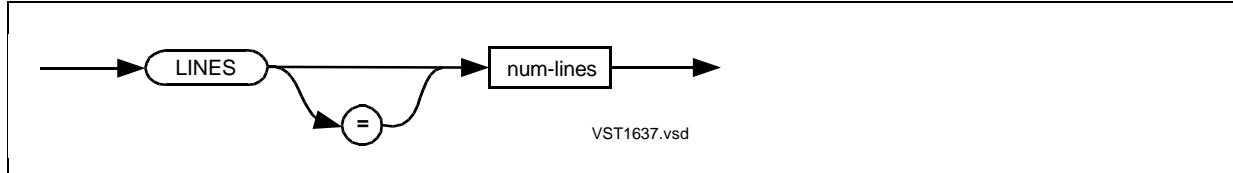
- To reduce the storage required for object code on disk and in main memory
- To share a set of common procedures among applications

- To extend a single application's code space

To build user libraries, there are certain restrictions that user library files need to follow. For more information on user library restrictions, refer to the *Binder Manual*.

## LINES Directive

LINES sets the maximum number of output lines per page.



`num-lines`

is the maximum number of lines per page. Specify an unsigned decimal constant in the range 10 through 32,767. The default is 60 lines per page.

## Usage Considerations

LINES can appear in the compilation command or anywhere in the source code.

If the list file is a terminal, the compiler ignores the LINES directive. If the list file is a line printer or a process, the compiler skips to the top of form.

## Examples of LINES Directive

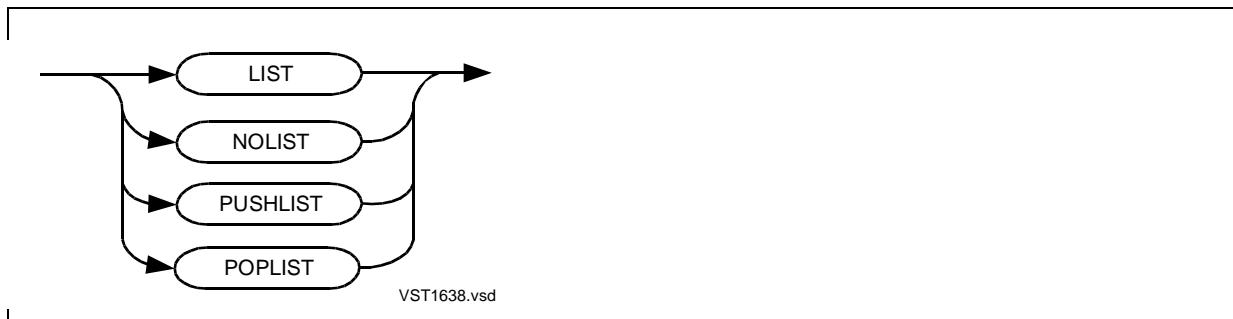
This example sets the maximum number of lines per page in the output listing to 66 lines per page:

```
?LINES 66
```

## LIST Directive

LIST lists the source text for subsequent source code if NOSUPPRESS is in effect.

The default is LIST.



## Usage Consideration

This directive can appear in the compilation command or anywhere in the source code.

LIST turns the listing setting on for subsequent code. LIST has no effect if SUPPRESS is in effect. LIST is required by the ABSLIST, CODE, CROSSREF, DEFEXPAND, FMAP, GMAP, ICODE, INNERLIST, MAP, PAGE, and PRINTSYM directives. To list cross-references, LIST and NOSUPPRESS must be in effect at the end of the source file.

NOLIST turns the listing setting off for subsequent code (the LMAP maps, compiler leader text, error messages, and trailer text still get listed).

PUSHLIST pushes the current listing setting onto the directive stack without changing the current setting.

POPLIST removes the top value from the directive stack and sets the current listing setting to that value.

## Source Listing

Each line in the source listing consists of:

- An edit file number
- A 6-digit octal code address—an instruction offset or a secondary global count
- One of the following lexical-level values:

Lexical Level	Meaning
0	Global Level
1	Procedure Level
2	Subprocedure Level

- Nesting level of BEGIN-END items such as structures, substructures, IF statements, and CASE statements

## Examples of LIST Directive

1. This example shows how to suppress listings of system declarations, but not your own source code, by using NOLIST and LIST:

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

?LIST
```

2. In this example, PUSHLIST pushes the current listing setting (LIST) onto the directive stack, NOLIST suppresses listing of system declarations, and POPLIST resumes the listing:

```

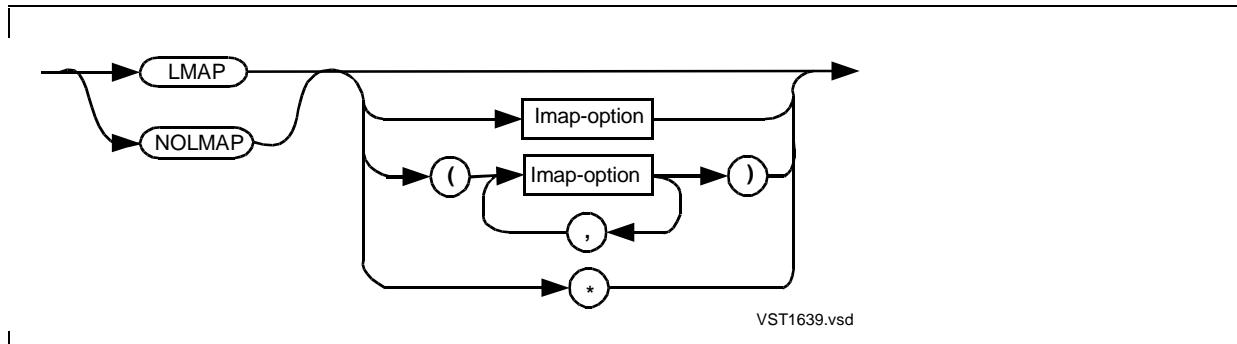
!Default listing setting is LIST
?PUSHLIST, NOLIST, SOURCE $SYSTEM.SYSTEM.EXTDECS (
? PROCESS_GETINFO_, FILE_OPEN_, WRITEREADX, READX)
?POPLIST

```

## LMAP Directive

LMAP lists load-map and cross-reference information.

The default is LMAP ALPHA.



**lmap-option**

specifies the type of load map to list; it is one of:

ALPHA	List load maps of procedures and data blocks sorted by name
LOC	List load maps of procedures and data blocks sorted by starting address, in addition to the ALPHA listing
XREF	List entry-point and common data-block cross-references for the object file (not the source-level cross-references produced by CROSSREF), in addition to the ALPHA listing
* (asterisk)	specifies ALPHA, LOC, and XREF maps

## Usage Considerations

This directive can appear in the compilation command or anywhere in the source code.

If you specify LMAP with no options, LMAP ALPHA is the default; it lists load maps of procedures and data blocks sorted by name.

If you specify LMAP with the LOC and XREF options, these options are added to the ALPHA default.

If LMAP is in effect, NOLMAP with options suppresses the specified load maps.  
 NOLMAP without options suppresses all load maps.  
 LMAP has no effect if SUPPRESS is in effect.

## Example of LMAP Directive

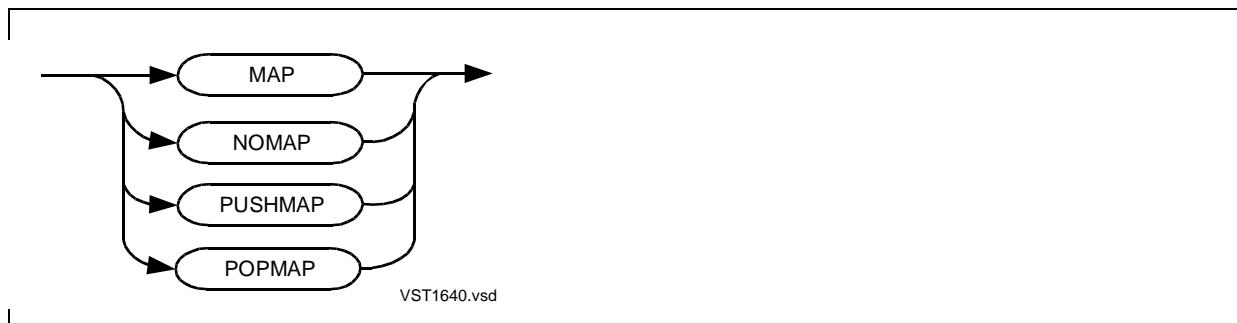
This example illustrates the LMAP directive:

```
?LMAP (LOC, XREF)           !Adds LOC and XREF to ALPHA default
!Some code here
?NOLMAP (XREF)             !Deletes only XREF from the listing
```

## MAP Directive

MAP lists the identifier maps.

The default is MAP.



## Usage Considerations

This directive can appear in the compilation command or anywhere in the source code.

MAP turns the map-listing setting on for subsequent code. MAP has no effect if NOLIST or SUPPRESS is in effect. MAP is required by GMAP.

NOMAP turns the map-listing setting off for subsequent code.

PUSHMAP pushes the current map-listing setting onto the directive stack without changing the current setting.

POPMAP removes the top value from the directive stack and sets the current map-listing setting to that value.

## Map Listing

MAP lists sublocal identifiers after each subprocedure, local identifiers after each procedure, and global identifiers after the last procedure in the source program.

The map listing consists of information including:

- Identifier class—VAR, SUBPROC, ENTRY, LABEL, DEFINE, LITERAL
- Type—data type, structure, substructure, or structure pointer
- Addressing mode—direct or indirect
- Subprocedure, entry, or label offset
- Text of LITERALS and DEFINES

## Example of MAP Directive

This compilation command starts the compiler and suppresses the identifier maps:

```
TAL /IN mysrc, OUT $s.#lists/ myobj; NOMAP
```

## OLDFLTSTDFUNC Directive

OLDFLTSTDFUNC treats arguments to the \$FLT, \$FLTR, \$EFLT, and \$EFLTR standard functions as if they were FIXED(0) values.



## Usage Considerations

OLDFLTSTDFUNC can appear in the compilation command or anywhere in the source code.

OLDFLTSTDFUNC prevents the scaling of the FIXED(*n*) argument of the \$FLT, \$FLTR, \$EFLT, and \$EFLTR standard functions, where *n* is the fixed-point position.

For instance, if OLDFLTSTDFUNC is in effect, \$FLT(1f), \$FLTR(0.1f), and \$EFLT(0.00001f) all yield the same floating-point value.

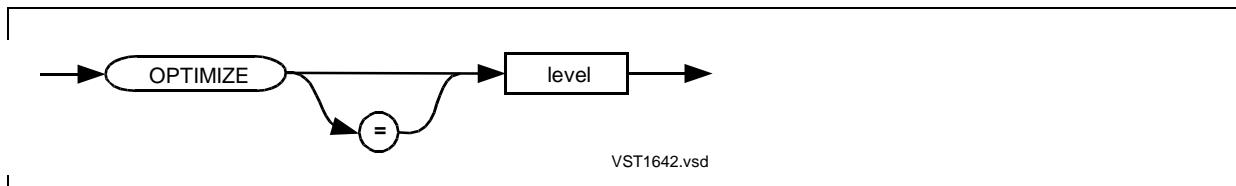
## Example of OLDFLTSTDFUNC Directive

This compilation command starts the compiler and invokes the OLDFLTSTDFUNC directive:

```
TAL /IN mysrc, OUT $s.#lists/ myobj; OLDFLTSTDFUNC
```

# OPTIMIZE Directive

OPTIMIZE specifies the level at which the compiler optimizes the object code.



**level**

is the level of optimization, specified as 0, 1, or 2. The default level is 1.

Level	Optimization	When to use:
0	None	When an internal compiler error is exposed by another optimization level
1	Within a statement	When you are developing and testing a program
2	Within and across statement boundaries	When you want to produce the most efficient code

## Usage Considerations

OPTIMIZE can appear in a compilation command or anywhere in the source code.

If OPTIMIZE is in effect, the compiler replaces short instruction sequences with equivalent but more efficient instruction sequences. The compiler does not perform global code optimizations such as removing invariant expressions from loops.

Specifying higher optimization levels does not appreciably increase compilation time nor affect compiler use of resources such as memory and disk space. Usually, optimization slightly decreases the object code size and increases the execution speed by some small amount.

The listing generated by the INNERLIST directive reports the optimizations, showing first the original instructions and then the revised ones.

## Examples of OPTIMIZE Directive

1. Level 1 transforms the code for loading an index register with a constant value from two instructions into one:

```

LDI 1 !Transform to LDXI 1,7
STAR 7

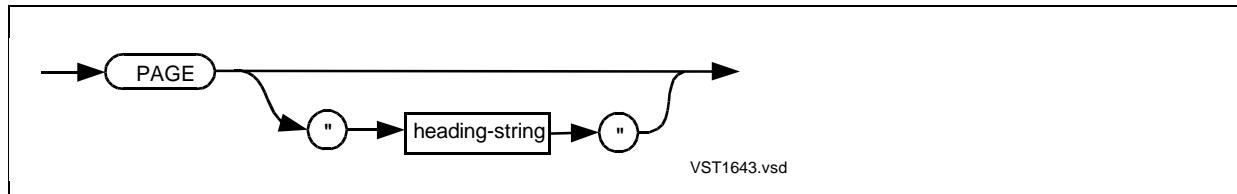
```

2. When a statement references a variable that was assigned a value by the previous statement, level 2 transforms a store and a load into a nondestructive store:

```
STOR L+1 !Transform to NSTO L+1
LOAD L+1
```

## PAGE Directive

PAGE optionally prints a heading and causes a page eject.



*heading-string*

is a character string of up to 82 characters, specified on a single line and enclosed in quotation marks. The quotation marks are required delimiters and are not printed. If the string is longer than 82 characters, the compiler truncates the extra characters.

### Usage Considerations

PAGE can appear anywhere in the source code but not in the compilation command.

PAGE has no effect if NOLIST or SUPPRESS is in effect.

The first PAGE prints the heading, skips a line, and then continues printing but does not cause a page eject.

A subsequent *heading-string* replaces the previous header.

If the list file is a terminal, the compiler ignores the PAGE directive. If the list file is a line printer or a process, the compiler skips to the top of form.

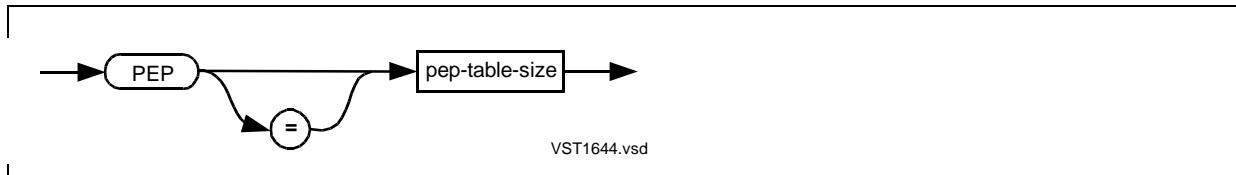
### Example of PAGE Directive

This example prints headings at the top of pages in the compiler listings:

```
!This is MYSOURCE file
?PAGE "Here are global declarations for MYSOURCE"
!Global declarations
?PAGE "Here are procedure declarations for MYSOURCE "
!Procedure declarations
```

# PEP Directive

PEP specifies the size of the procedure entry-point (PEP) table.



**pep-table-size**

is the number of words to allocate for the PEP table. Specify an unsigned decimal constant in the range 3 through 512.

## Usage Considerations

The PEP directive can appear in the compilation command or anywhere in the source code.

If you use ABSLIST, which lists code-relative addresses for instruction locations, you must define the size of the PEP table before the first procedure declaration appears.

Specify PEP before the first procedure declaration or declare all procedures that precede PEP as FORWARD or EXTERNAL.

## PEP Table

The PEP table must be at least large enough to contain one word per nonexternal entry point.

You can respecify PEP at any time without causing a warning from the compiler. If you respecify the PEP table size or if the size is insufficient for the program, the ABSLIST addresses are invalid.

## Example of PEP Directive

The following example sets the size of the PEP table at 60 words before the first procedure declaration so that ABSLIST has effect:

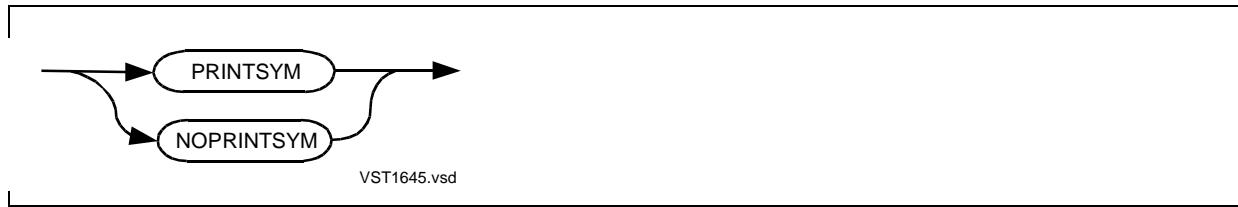
```

!This is MYSOURCE file.
!Global declarations
?ABSLIST, PEP 60
PROC mymain MAIN;
  BEGIN
    !Lots of code
  END;
  
```

# PRINTSYM Directive

PRINTSYM lists symbols.

The default is PRINTSYM.



## Usage Considerations

This directive can appear in the compilation command or anywhere in the source code.

You can use PRINTSYM and NOPRINTSYM to list individual symbols or groups of symbols, such as global, local, or sublocal declarations.

PRINTSYM turns the symbol-listing setting on for subsequent declarations.

PRINTSYM has no effect if NOLIST or SUPPRESS is in effect.

NOPRINTSYM turns the symbol-listing setting off for subsequent declarations.

## Example of PRINTSYM Directive

This example suppresses printing in the global map of variables I and J, which are declared between the NOPRINTSYM directive and the PRINTSYM directive:

```
?NOPRINTSYM  
INT i;  
INT j;  
?PRINTSYM  
INT k;
```

# RELOCATE Directive

RELOCATE lists BINSERV warnings for declarations that depend on absolute addresses in the primary global data area of the user data segment.



## Usage Considerations

RELOCATE can appear in the compilation command or in the source code.

RELOCATE affects only the source code that follows it, however, so it is safest to specify it at the beginning of the compilation.

The compiler checks for nonrelocatable data only if RELOCATE appears.

Use RELOCATE when the primary global data area (the area below word address G[256]) is relocatable. If you are compiling the modules of a program separately or binding TAL code with code written in other languages, the primary global data must be relocatable.

When you build the target file, Binder issues warnings for references to nonrelocatable data.

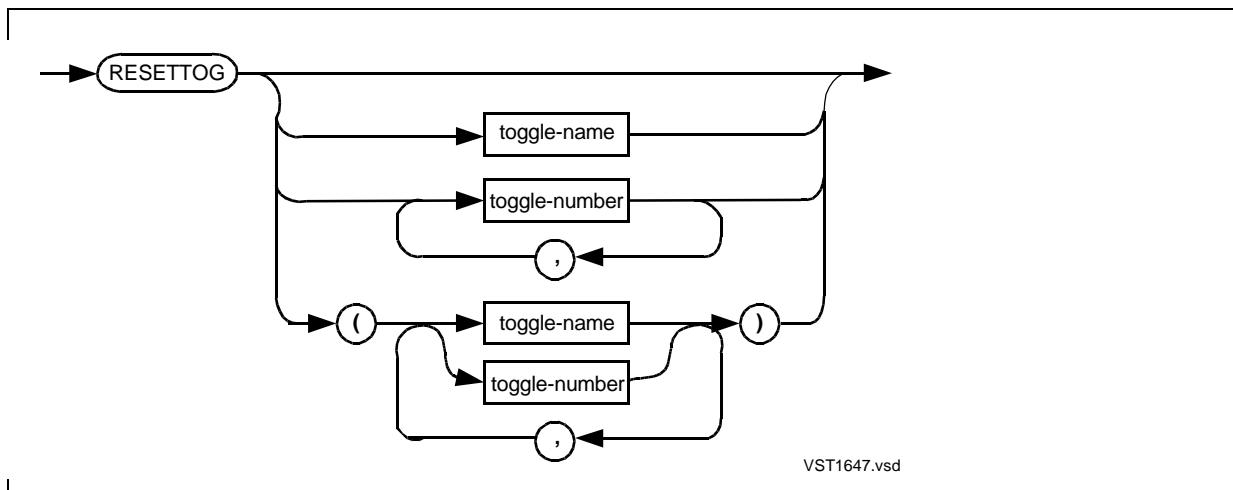
## Example of RELOCATE Directive

In this example, because RELOCATE is in effect, any reference to I (a nonrelocatable global declaration) produces a warning:

```
?RELOCATE
!Lots of code
INT i = 'G' + 22;           !Nonrelocatable global declaration I
!Lots more code
i := 25;                   !Reference to I
```

## RESETTOG Directive

RESETTOG creates new toggles in the off state and turns off toggles created by SETTOG. The RESETTOG directive supports named toggles in addition to numeric toggles.



`toggle-name`

is a named toggle to turn off. You must specify each named toggle you want to turn off.

`toggle-number`

is a numeric toggle to turn off. Specify an unsigned decimal constant in the range 1 through 15. Leading zeros are ignored. RESETTOG with no arguments turns off all numeric toggles but does not affect named toggles.

## Usage Considerations

RESETTOG can appear anywhere in the source code and in the compilation command.

RESETTOG without a parenthesized list must be the last directive on the line. When RESETTOG has a parenthesized list, other directives can follow on the same line, with a comma separating the closing parenthesis from the next directive.

RESETTOG interacts with the IF, IFNOT, and ENDIF directives. IF and IFNOT test the setting of toggles and mark the beginning of conditional compilation. ENDIF marks the end of conditional compilation.

## Named Toggles

Before you use a named toggle in an IF or IFNOT directive, you must specify that name in a DEFINETOG, SETTOG, or RESETTOG directive. Which of these directives you use depends on whether you want settings of named toggles unchanged or turned on or off:

Directive	Setting of New Toggle	Setting of Specified Existing Toggle
DEFINETOGL	Off	Unchanged
SETTOG	On	On
RESETTOG	Off	Off

## Numeric Toggles

You can use a numeric toggle in an IF or IFNOT directive, even if that number has not been specified in a RESETTOG, SETTOG, or DEFINETOGL directive.

By default, all numeric toggles not turned on by SETTOG are turned off. To turn off numeric toggles turned on by SETTOG, use RESETTOG.

## Example of RESETTOG Directive

In this example, RESETTOG turns off two of six toggles that were turned on by SETTOG. IF tests a toggle, finds it is off, and causes the compiler to skip over the source text between IF VERSN2 and ENDIF VERSN2 :

```
?SETTOG (versn1, versn2, 7, 4, 11) !Turn on toggles
?SETTOG versn3                                !Turn on toggle
?RESETTOG (versn2, 7)                         !Turn off toggles
!Lots of code
?IF versn2                                     !Test toggle for on state
PROC version_2;                                 !Find it off; skip
BEGIN                                         ! procedure
!More code
END;                                           !End of skipped portion
?ENDIF versn2
```

## ROUND Directive

ROUND rounds FIXED values assigned to FIXED variables that have smaller *fpoint* values than the values you are assigning.

The default is NOROUND.



## Usage Considerations

This directive can appear in the compilation command or anywhere in the source code.

ROUND turns the rounding setting on. If the *fpoint* of the assignment value is greater than that of the variable, ROUND first truncates the assignment value so that its *fpoint* is one greater than that of the destination variable. The truncated assignment value is then rounded away from zero as follows:

$$\text{value} = (\text{IF } \text{value} < 0 \text{ THEN } \text{value} - 5 \text{ ELSE } \text{value} + 5) / 10$$

In other words, if the truncated assignment value is negative, 5 is subtracted; if positive, 5 is added. Then, an integer division by 10 and truncates it again, this time by a factor of 10. Thus, if the absolute value of the least significant digit of the initially

truncated assignment value is 5 or more, a one is added to the absolute value of the final least significant digit.

NOROUND turns the rounding setting off. That is, rounding does not occur when a FIXED value is assigned to a FIXED variable that has a smaller *fpoint*. If the *fpoint* of the assignment value is greater than that of the variable, the assignment value is truncated and some precision is lost.

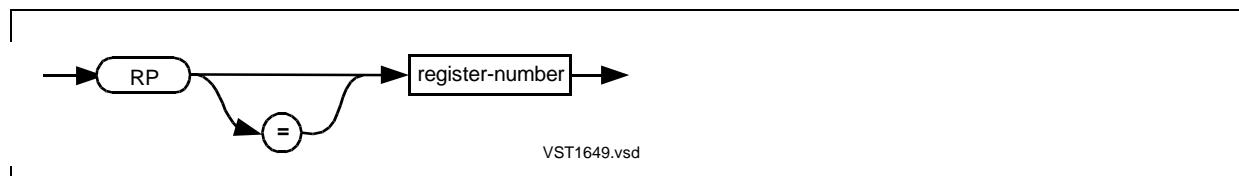
## Example of ROUND Directive

This example rounds all FIXED values:

```
?ROUND                                !Request rounding
!Global declarations
PROC a;
    BEGIN
        FIXED(2) f1;
        FIXED(3) f2;
        f1 := f2;
    END;
```

## RP Directive

RP sets the compiler's internal register pointer (RP) count. RP tells the compiler how many registers are currently in use on the register stack.



register-number

is the value to which the compiler is to set its internal RP count. Specify an unsigned decimal constant in the range 0 through 7. If you specify 7, the compiler considers the register stack to be empty.

## Usage Considerations

The RP directive can appear only within a procedure.

The register pointer (RP) is a field in the environment register (ENV) that points to the top of the register stack.

If you manipulate data stack contents without the compiler's knowledge (with CODE statements, for example), you should use the RP directive to calibrate the compiler's internal RP count.

If the compiler decrements (or increments) the RP register below (or above) its initial value in a procedure or subprocedure, the compiler issues a warning of RP register underflow (or overflow). If the source program is correct, use the RP directive to calibrate the compiler's internal RP count.

After each high-level statement (not CODE, STACK, or STORE), the compiler's internal RP setting is always 7 (empty).

Modularize use of the RP directive and CODE, STACK, and STORE statements where possible; they are not portable to future software platforms.

## Example of RP Directive

This example sets the compiler's internal RP count:

```

INT c;

INT PROC a;
BEGIN
  !Lots of code
  RETURN 1;
END;

PROC b (g);
  INT g;
BEGIN
  INT i;
  IF g THEN
    STACK 0
  ELSE
    BEGIN
    STACK c;
    CODE (DPCL);      !Called routine returns a 16-bit
                      ! integer by way of the register stack
                      ! without knowledge of the compiler
? RP 0              !Set compiler's internal RP count to 0
END;
STORE i;
!More code
END;

PROC m MAIN;
BEGIN
c := @a;
CALL b (0);
END;

```

# RUNNAMED Directive

RUNNAMED causes a D-series object file to run on a D-series system as a named process even if you do not provide a name for it.



## Usage Considerations

The RUNNAMED directive can appear in the compilation command or anywhere in the source code. It can appear any number of times in a compilation unit, but need appear just once in the compilation unit. Use RUNNAMED only with D-series compilation units.

If RUNNAMED and HIGHPIN are in effect, a process can run at high PIN on a D-series system and be opened by a C-series process using the OPEN procedure.

RUNNAMED sets the RUNNAMED attribute in the object file. If the RUNNAMED attribute is set for any object file in a target file, the RUNNAMED attribute is set for all the object files in the target file.

If you do not specify RUNNAMED in the compilation unit and you want the process to run as a named process, you can use the CHANGE command of Binder or the NAME option of the TACL RUN command. To avoid possible complications resulting from any unnamed processes running at a high PIN on a D-series system, however, specify RUNNAMED in all D-series compilation units.

## Examples of RUNNAMED Directive

1. This example shows the RUNNAMED directive in a directive line in the source code:

```
?HIGHPIN, RUNNAMED
```

2. This example shows the RUNNAMED directive in a compilation command:

```
TAL /IN mysrc, OUT $S.#mylst/ myobj; HIGHPIN, RUNNAMED
```

# SAVEABEND Directive

SAVEABEND directs the Inspect product to generate a save file if your process terminates abnormally during execution.

The default is NOSAVEABEND.



## Usage Considerations

SAVEABEND and NOSAVEABEND can appear in the compilation command or anywhere in the source program. The compiler uses the last specification when it builds the object file.

SAVEABEND requests a save file and sets the INSPECT directive on; at run time the Inspect product must be available on the system that runs the process.

NOSAVEABEND suppresses the save file but does not affect the INSPECT directive setting.

After compilation, you can specify the SAVEABEND option by using Binder or TACL RUN options.

## Save File

The save file contains data-area and file-status information at the time of process failure. You can examine the save file by using Inspect commands. The Inspect product assigns the save file a name of the form ZZSA $nnnn$ , where  $nnnn$  is an integer. The defaults for volume and subvolume are the object file's volume and subvolume. (You can specify a name for the save file by using the Inspect product.) For more information, see the *Inspect Manual*.

## Example of SAVEABEND Directive

This example generates a save file for the object file if execution terminates abnormally in procedure A:

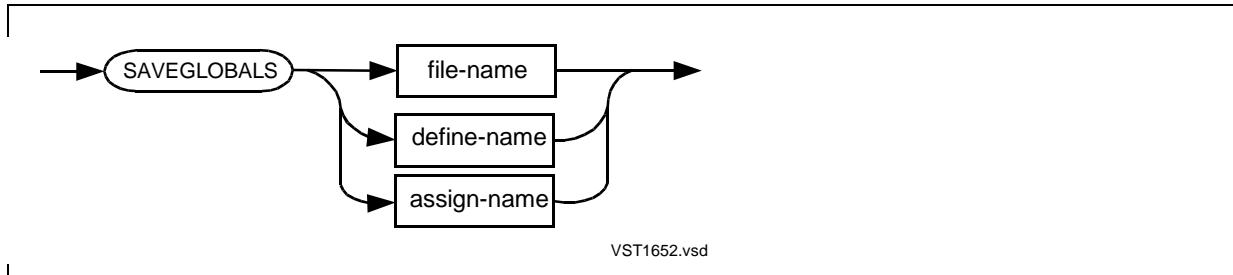
```

?SAVEABEND           ! Set Inspect product; generate save file
PROC a;
BEGIN
  !Lots of code
END;

?NOSAVEABEND        ! Suppress save file
PROC b;
BEGIN
  !Lots of code
END;                ! INSPECT and NOSAVEABEND still in effect
  
```

# SAVEGLOBALS Directive

SAVEGLOBALS saves all global data declarations in a file for use in subsequent compilations that specify the USEGLOBALS directive.



**file-name**

is the name of a disk file to which the compiler is to write the global data declarations.

If *file-name* already exists, the compiler purges the existing file and creates an unstructured global declarations file that has a file code of 105.

If the existing file is secured so that the compiler cannot purge it, the compilation terminates.

You can specify partial file names as described in Appendix E in the *TAL Programmer's Guide*. The compiler uses the current default volume and subvolume names as needed and lists the complete file name in the trailer message at the end of compilation. For this directive, the compiler does not use TACL ASSIGN SSV information to complete the file name.

**define-name**

is the name of a TACL MAP DEFINE that refers to the disk file to which you want the compiler to write the global data declarations.

**assign-name**

is a logical file name you have equated to the actual disk file (to which you want the compiler to write the global data declarations) by issuing a TACL ASSIGN command.

## Usage Considerations

SAVEGLOBALS can appear in the compilation command or in the source code before any global data declarations. If it appears anywhere else, the compiler issues a warning message and ignores the directive.

## Saving Global Data Declarations

When you compile with SAVEGLOBALS, the compiler saves the global data declarations in a file. If you make no changes in the global data declarations, you can use the saved declarations in subsequent compilations and reduce the compilation time.

Other guidelines for saving global data declarations include the following:

- If the global data declarations contain any syntax errors, correct the errors and recreate the global declarations file by using SAVEGLOBALS.
- SAVEGLOBALS does not save EXTERNAL or FORWARD procedure declarations.
- You must recompile these declarations in the USEGLOBALS compilation.
- If SAVEGLOBALS and USEGLOBALS appear in the same compilation unit, the compiler issues an error message and complies with only the first of the two directives.
- If SAVEGLOBALS appears in a compilation unit you submit to the stand-alone Crossref product, the compiler ignores the SAVEGLOBALS directive.
- 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 in the USEGLOBALS compilation. C20, D10, and D20, for example, are different versions of the compiler.

If SAVEGLOBALS is in effect, the compiler takes the following actions when it encounters the first procedure declaration:

- The compiler stores the global data declarations in the specified file. Global data declarations include all global data identifiers and their attributes (such as data type and kind of variable) but not their initializations.
- The compiler stores the initialization values in the object code. Initialization values include addresses and constant lists.

## Retrieving Global Data Declarations

After a SAVEGLOBALS compilation completes successfully, you can specify the following directives in a USEGLOBALS compilation to retrieve the global data declarations and initializations:

<b>Directive in USEGLOBALS Compilation</b>	<b>Effect in Same USEGLOBALS Compilation</b>
USEGLOBALS	Retrieves global data declarations; suppresses compilation of text lines and SOURCE directives (but not other directives) until BEGINCOMPILATION appears
SEARCH	Retrieves global initializations and template structure declarations
BEGINCOMPILATION	Begins compilation of text lines and SOURCE directives

Specify BEGINCOMPILATION after the last global declaration or SEARCH directive and before the first procedure declaration.

## Effect of Other Directives

When you use the following directives in the SAVEGLOBALS compilation, they affect subsequent USEGLOBALS compilations as follows:

<b>Directive in USEGLOBALS Compilation</b>	<b>Effect in Subsequent USEGLOBALS Compilations</b>
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 (D-series system only)
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

## Examples of SAVEGLOBALS Directive

- The following source file (MYPROG) is compiled in examples 2 through 5, which show how the SAVEGLOBALS, USEGLOBALS, BEGINCOMPILATION, SEARCH, and SYNTAX directives interact:

```

!Source file MYPROG
!Unless USEGLOBALS (examples 3 and 5) is in effect,
! compile the entire source file.

?SOURCE glbfile1 (section1, section2)
?SOURCE moreglbs
    INT ignore_me1;
    INT ignore_me2;

?BEGINCOMPILATION           !When USEGLOBALS is in effect,
                            ! 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
    !More code
END;

```

- A SAVEGLOBALS compilation compiles MYPROG and saves global data declarations in file TALSYM and global initializations in object file MYOBJ:

```
TAL /IN myprog/ myobj; SAVEGLOBALS talsym
```

- A USEGLOBALS compilation then produces object file NEWOBJ and retrieves global data declarations from TALSYM and global initializations from MYOBJ. When USEGLOBALS is in effect, the compiler ignores text lines and SOURCE directives until BEGINCOMPILATION appears in the source file:

```
TAL /IN myprog/ newobj; USEGLOBALS talsym, SEARCH myobj
```

- Alternatively, you can place the preceding SEARCH directive in the source file anywhere before the BEGINCOMPILATION directive. You can check the syntax of global data declarations before saving them:

```
TAL /IN myprog/; SAVEGLOBALS talsym, SYNTAX
```

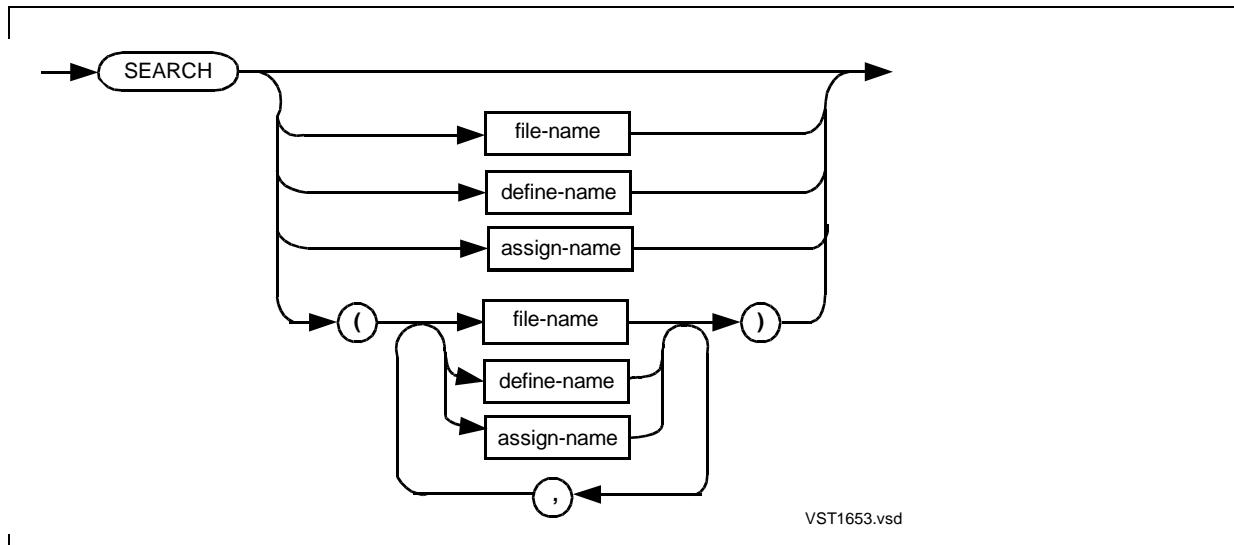
- After you correct any errors, you can recompile MYPROG as follows, assuming there are no global initializations:

```
TAL /IN myprog/; USEGLOBALS talsym
```

## SEARCH Directive

SEARCH specifies object files from which BINSERV can resolve unsatisfied external references and validate parameter lists at the end of compilation.

By default, BINSERV does not attempt to resolve unsatisfied external references.



**file-name**

is the name of a disk object file from which BINSERV can resolve unsatisfied external references. Specify the file names in the order in which you want the search to take place.

You can specify partial file names as described in Appendix E in the *TAL Programmer's Guide*. The compiler uses the current default volume and subvolume names as needed to complete file names. The compiler can also use TACL ASSIGN SSV information for file names.

**define-name**

is the name of a TACL MAP DEFINE that refers to a disk object file from which the compiler is to resolve unsatisfied external references.

**assign-name**

is a logical file name you have equated to a disk object file (from which the compiler is to resolve unsatisfied external references) by issuing a TACL ASSIGN command.

## Usage Considerations

SEARCH can appear in the compilation command or anywhere in the source code. A SEARCH directive can extend to continuation lines, each line beginning with ? in column 1.

### Search List

The compiler sends the list of object files from each SEARCH directive to BINSERV. BINSERV appends the file names in the order specified to the master search list for the current source file.

You can clear the search list at any point in the source file by specifying SEARCH with no file names.

At the end of compilation, BINSERV uses the files that remain on the search list to resolve external references. BINSERV searches the files in the order in which they appear in the search list. If a procedure or entry-point name that resolves an external reference appears in more than one file, BINSERV binds only the first occurrence, so the order in which you specify the files is important.

After a successful compilation, BINSERV binds the new object file. During binding, BINSERV uses procedures from object files in the master search list to resolve any unsatisfied external references in your program. If the external procedures also contain references to other external procedures or to data blocks, BINSERV resolves them from object files in the master search list.

### Retrieving Global Initializations and Template Structures

You also use SEARCH to retrieve previously saved global initializations and template structure declarations. For example, in the following compilation command, USEGLOBALS retrieves global data declarations saved in file TALSYM by SAVEGLOBALS in a previous compilation, and SEARCH retrieves the initialization values from object file MYOBJ, which was also created in the SAVEGLOBALS compilation:

```
TAL /IN myprog/ newobj; USEGLOBALS talsym, SEARCH myobj
```

As an alternative, you can place the SEARCH directive in the source code anywhere before the BEGINCOMPILATION directive.

For more information on saving global data declarations, see the [SAVEGLOBALS Directive](#) on page 16-75.

## Examples of SEARCH Directive

1. This example shows SEARCH directives for a search in the order FILE1, FILE2, FILE3, and FILE4:

```
?SEARCH (file1, file2)
?SEARCH (file3, file4)
```

2. 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;
```

## SECTION Directive

SECTION gives a name to a section of a source file for use in a SOURCE directive.



section-name

is an identifier to associate with all source text that follows the SECTION directive until another SECTION directive or the end of the source file occurs.

## Usage Considerations

SECTION can appear anywhere in the source code but not in the compilation command. SECTION must be the only directive on the directive line.

# Example of SECTION Directive

1. This example gives a different section name, such as SORT\_PROC and NEXT\_PROC, to each procedure in a source library:

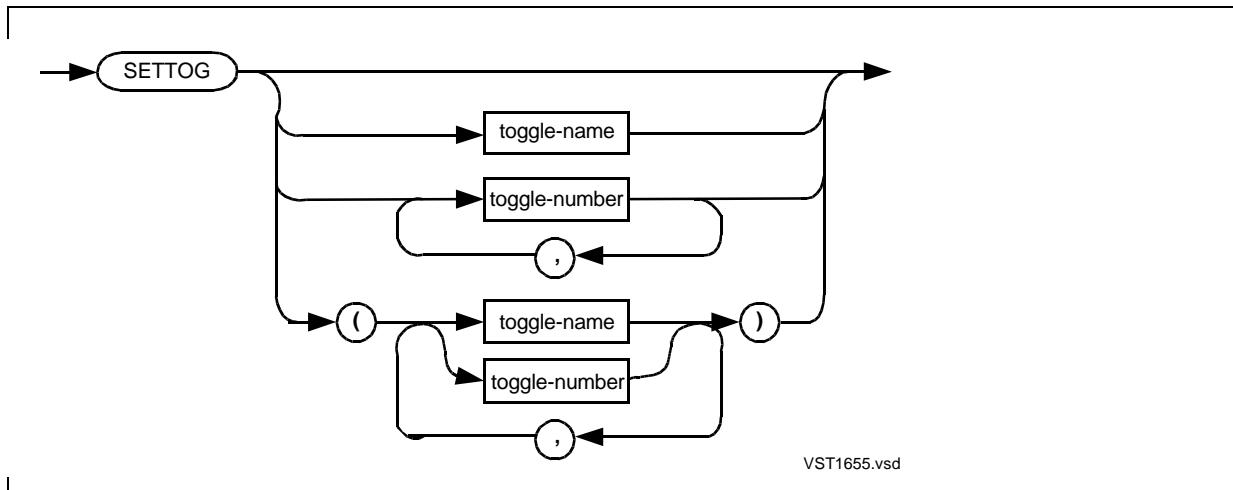
```
!File ID APPLIB
?SECTION sort_proc
PROC sort_on_key(key1, key2, key3, length);
    INT .key1, .key2, .key3, length;
BEGIN
    !Lots of code
END;
?SECTION next_proc
```

2. Another source file includes the previous file name and a section name in a SOURCE directive:

?SOURCE applib (sort\_proc)

# SETTOG Directive

**SETTOG** turns the specified toggles on for use in conditional compilations. The **SETTOG** directive supports named toggles in addition to numeric toggles.



toggle-name

is a user-defined name that conforms to the TAL identifier format. You must specify each toggle name you want turned on.

## toggle-number

is an unsigned decimal constant in the range 1 through 15. Leading zeros are ignored. SETTOG with no arguments turns on all numeric toggles but does not affect named toggles.

## Usage Considerations

SETTOG can appear anywhere in the source code and in the compilation command. SETTOG without a parenthesized list must be the last directive on the line. When SETTOG has a parenthesized list, other directives can follow on the same line, with a comma separating the closing parenthesis from the next directive.

SETTOG interacts with the IF, IFNOT, and ENDIF directives. IF and IFNOT test the setting of toggles and mark the beginning of conditional compilation. ENDIF marks the end of conditional compilation.

### Named Toggles

Before you use a named toggle in an IF or IFNOT directive, you must specify that name in a DEFINETOG, SETTOG, or RESETTOG directive. Which of these directives you use depends on whether you want settings of named toggles unchanged or turned on or off:

Directive	Setting of New Toggle	Setting of Specified Existing Toggle
DEFINETOGL	Off	Unchanged
SETTOGL	On	On
RESETTOGL	Off	Off

### Numeric Toggles

You can use a numeric toggle in an IF or IFNOT directive, even if that number has not been specified in a SETTOG, RESETTOG, or DEFINETOGL directive.

By default, all numeric toggles not turned on by SETTOG are turned off. To turn off numeric toggles turned on by SETTOG, use RESETTOG.

### Examples of SETTOG Directive

1. In this example, SETTOG turns on six toggles and RESETTOG turns off two of them. IF tests a toggle, finds it is turned on, and causes the compiler to compile the source text between IF VERSN2 and ENDIF VERSN2 :

```
?SETTOG (versn1, versn2, 7, 4, 11)
?SETTOG versn3          !Turn on toggles
?RESETTOG (versn1, 7)    !Turn off toggles
!Lots of code

?IF versn2                !Test toggle for on state
PROC version_2;            !Find it on; compile procedure
  BEGIN
    !More code
  END;
?ENDIF versn2             !End of compiled portion
```

2. In this example, IFNOT tests a toggle for the off state, finds it is turned on, and causes the compiler to skip the source text between IFNOT SCANNER and ENDIF SCANNER:

```
?SETTOG scanner           !Turn toggle SCANNER on
!Some code here

?IFNOT scanner           !Test toggle for off state
PROC skipped;            !Find it on; skip procedure
BEGIN
  !More code here
END;
?ENDIF scanner           !End of skipped procedure
```

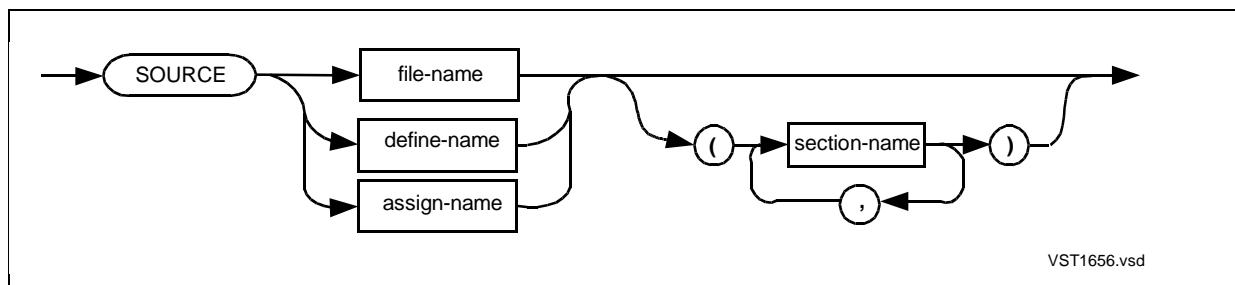
3. In this example, SETTOG turns on toggle 1. IF tests the toggle, finds it is turned on, and causes the compiler to compile the source text between IF 1 and ENDIF 1:

```
?SETTOG 1                 !Turn toggle 1 on
!Some code here

?IF 1                     !Test the toggle for on state
PROC some_proc;           !Find it on; compile procedure
BEGIN
  !More code here
END;
?ENDIF 1                  !End of compiled portion
```

## SOURCE Directive

SOURCE specifies source code to include from another source file.



**file-name**

specifies the name of a disk file from which the compiler is to read source code. You can specify partial file names as described in Appendix E in the *TAL Programmer's Guide*. The compiler uses TACL ASSIGN SSV information, if specified, to complete the file name. Otherwise, the compiler uses the current default volume and subvolume names as needed.

define-name

is the name of a TACL MAP DEFINE that refers to a disk file from which the compiler is to read source code.

assign-name

is a logical file name you have equated to a disk file (from which the compiler is to read source code) by issuing a TACL ASSIGN command.

section-name

is an identifier specified in a SECTION directive within the sourced-in file. If the compiler does not find *section-name* in the specified file, it issues a warning.

## Usage Considerations

SOURCE can appear anywhere in the source code but not in the compilation command.

If other directives appear on the same line, SOURCE must be last in the line. The list of section names can extend to continuation lines, each line beginning with ? in column 1. The leading parenthesis, if present, must appear on the same line as SOURCE.

## 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.

If USEGLOBALS is in effect, the compiler ignores all SOURCE directives until it encounters BEGINCOMPILATION. For more information on how these directives interact, see the [SAVEGLOBALS Directive](#) on page 16-75.

## Examples of SOURCE Directive

1. This SOURCE directive instructs the compiler to process the file until an end of file occurs:

```
?SOURCE $src.current.routines
```

(Any SECTION directives in the file ROUTINES are treated as comments.)

2. This SOURCE directive reads three sections from the source file. It reads the files in the order in which they appear in the source file, not in the order specified in the SOURCE directive.

```
?SOURCE $src.current.routines (sec1, sec2, sec3)
```

(The specified files appear in the source file in the order SEC3, SEC2, and SEC1, so they are read in that order.)

3. This example shows how you can specify the order in which the compiler is to read the sections, regardless of their order in the source file:

```
?SOURCE $src.current.routines (sec1)
?SOURCE $src.current.routines (sec2)
?SOURCE $src.current.routines (sec3)
```

## SQL Directive

SQL instructs the compiler to make preparations for the processing of SQL commands.

The syntax for this directive is described in the *NonStop SQL Programming Manual for TAL*.

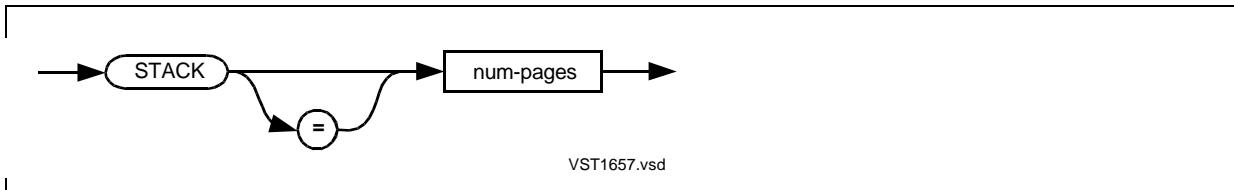
## SQLMEM Directive

SQLMEM specifies where in memory the compiler is to place internal SQL data structures that describe SQL statements and host variables.

The syntax for SQLMEM is described in the *NonStop SQL Programming Manual for TAL*.

# STACK Directive

STACK sets the size of the data stack in the user data segment.



*num-pages*

is the number of 2048-byte memory pages to allocate for the data stack (storage area for local and sublocal data). Specify an unsigned decimal constant in the range 0 through 32.

## Usage Considerations

STACK can appear in the compilation command or anywhere in the source code. If you omit this directive, the default is the space estimated by BINSERV for local storage.

The combined number of memory pages allocated for the data area is equal to *num-pages* plus the space required for global data blocks. That combined area can be at most 32,768 words (65,536 bytes).

The following directives override the STACK directive:

- A DATAPAGES directive that specifies more than 32 memory pages
- An ENV COMMON directive in a compilation unit that contains a MAIN procedure

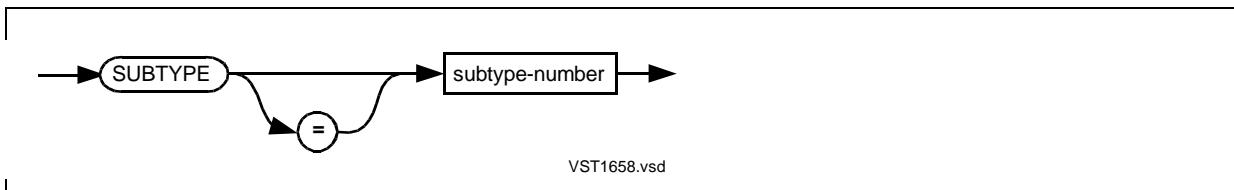
## Example of STACK Directive

This example sets the size of the data stack in the user data segment to 20 memory pages:

```
?STACK 20
```

# SUBTYPE Directive

SUBTYPE specifies that the object file is to execute as a process of a specified subtype.



subtype-number

specifies the process subtype number by which the object file is to identify itself to other processes. Specify an unsigned decimal constant in the range 0 through 63. The default process subtype is 0.

## Usage Considerations

SUBTYPE can appear in the compilation command or anywhere in the source code. If SUBTYPE appears more than once in a compilation, the compiler uses the subtype number specified in the last SUBTYPE directive it encounters.

The compiler stores the specified process subtype in the object file header. At run time, the system creates a process from the object file and assigns the saved process subtype to the process.

NonStop defines the meaning and behavior of subtypes 1 through 47. Nonprivileged subtypes you can use from this group are:

- |    |                             |
|----|-----------------------------|
| 30 | A device-simulating process |
| 31 | A spooler collector process |

To allow a terminal simulation process to specify its own device type, you must specify subtype 30 for the process. You can either specify SUBTYPE 30 at the beginning of your program or use a Binder command after compilation.

You can define the meaning and behavior of subtypes 48 through 63 and then specify them in SUBTYPE directives.

For more information on subtypes, see the *Guardian Programmer's Guide*.

## Example of SUBTYPE Directive

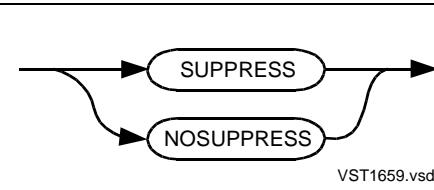
This example shows a SUBTYPE 30 directive at the beginning of a terminal simulation program:

```
!This is MYSOURCE file.  
?SUBTYPE 30  
!Global data declarations
```

## SUPPRESS Directive

SUPPRESS overrides all the listing directives.

The default is NOSUPPRESS.



## Usage Considerations

This directive can appear in the compilation command or anywhere in the source code. Both SUPPRESS and NOSUPPRESS can appear in the source code.

SUPPRESS suppresses all compilation listings except the compiler leader text, diagnostic messages, and the trailer text. That is, the compiler and BINSERV produce diagnostic and trailer text, but BINSERV does not produce the load maps.

SUPPRESS overrides all the listing directives—ABSLIST, CODE, CROSSREF, DEFEXPAND, FMAP, GMAP, ICODE, INNERLIST, LIST, LMAP, MAP, PAGE, and PRINTSYM.

SUPPRESS does not alter the source code.

NOSUPPRESS lets the listing directives take effect.

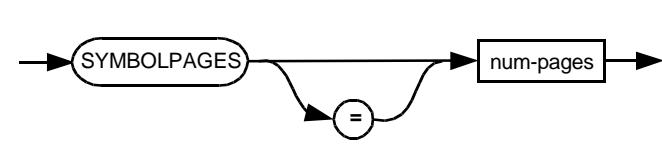
## Example of SUPPRESS Directive

This compilation command starts the compilation and suppresses all source code listings and maps from printing in the compiler output:

```
TAL /IN mysrc, OUT $s.#lists/ myobj; SUPPRESS
```

## SYMBOLPAGES Directive

SYMBOLPAGES sets the size of the internal symbol table the compiler uses as a temporary storage area for processing variables and SQL statements.



num-pages

specifies the number of 2048-byte memory pages to allocate for the symbol table. Specify an unsigned decimal constant in the range 512 through 32,767. The default is 512 pages (1 megabyte).

## Usage Considerations

**SYMBOLPAGES** can appear only in the compilation command, not in the source code.

**SYMBOLPAGES** sets the size of the compiler's internal symbol table, which resides in the automatic extended data segment. The compiler uses the symbol table as a temporary storage area, for example, when processing variables and SQL statements.

For more information on SQL, see the *NonStop SQL Programming Manual for TAL*.

If the symbol table overflows, the compiler issues error 57 (symbol table overflow). Use **SYMBOLPAGES** to specify a larger table size.

## Example of **SYMBOLPAGES** Directive

This compilation command starts the compiler and sets the size of the compiler's internal symbol table at 4096 memory pages:

```
TAL /IN mysrc, OUT $s.#lists/ myobj; SYMBOLPAGES 4096
```

## SYMBOLS Directive

**SYMBOLS** saves symbols in a symbol table (for Inspect symbolic debugging) in the object file.

The default is **NOSYMBOLS**.



## Usage Considerations

This directive can appear in the compilation command or anywhere in the source code.

**SYMBOLS** turns the symbol-saving attribute on for subsequent code.

**NOSYMBOLS** turns the symbol-saving attribute off for subsequent code.

## Saving Symbols

Normally you save symbols for the entire compilation by specifying **SYMBOLS** once at the beginning of the compilation unit. The symbol table then contains all the symbols generated by the source code.

Alternatively, you can save symbols for specific procedures and global data blocks, although this is not the common practice. If you save symbols for a specific procedure or data block and a referral structure in that procedure or data block references a

structure definition that was not saved, the compilation terminates with an error message from SYMSERV.

## Deleting Symbols

After debugging the program, you can use Binder to:

- Create a new object file without symbols. The old object file remains intact, but you drastically reduce what you can do with the Inspect product.

```
ADD * FROM oldobj
SET SYMBOLS OFF
BUILD newobj
```

- Delete both symbol and Binder tables from the old object file and omit them from the new object file. Before you do so, make sure you no longer need to examine or modify the file. Once you strip these tables, you can no longer use the Inspect product nor bind in new procedures into the object code.

```
STRIP oldobj
```

## Examples of SYMBOLS Directive

1. This example requests the Inspect product and saves symbols for the entire compilation unit:

```
!This is MYSOURCE file
?INSPECT, SYMBOLS

!Declare global data here
!Declare procedures and so forth
```

2. This example requests the Inspect product and saves symbols for a procedure and a global data block. This use is not the common practice; usually you specify SYMBOLS once at the beginning of the program:

```
!This is OURSOURCE file
?INSPECT
?SYMBOLS           ! Save symbols for implicit block
    INT a;
    STRING b;
?NOSYMBOLS         ! Stop saving symbols

BLOCK global_data;
    FIXED c;
    STRING d;
END BLOCK;

?SYMBOLS           ! Save symbols for procedure
PROC uxb;
BEGIN
    !Lots of code
```

```
END;  
?NOSYMBOLS           !Stop saving symbols
```

## SYNTAX Directive

SYNTAX checks the syntax of the source text without producing an object file.



### Usage Considerations

SYNTAX can appear in the compilation command or anywhere in the source code.

You can use the compiler to detect syntax errors before you code large portions of the source code. If SYNTAX is in effect, the compiler checks the syntax of the source text without producing an object file.

BINSERV is not needed if no object file is produced.

- SYNTAX in the compilation command prevents BINSERV from starting.
- SYNTAX early in the source text stops BINSERV after it starts.

SYNTAX does not affect the CROSSREF directive. The compiler can generate a cross-reference listing even if it produces no object file.

### Checking Saved Global Data Declarations

To check the syntax of saved global data declarations, you can use the SYNTAX, SAVEGLOBALS, and USEGLOBALS directives. If the syntax check finds errors in the global data declarations, you can correct them and recompile the source file using SAVEGLOBALS. For more information on how these directives interact, see the [SAVEGLOBALS Directive](#) on page 16-75.

### Examples of SYNTAX Directive

1. This compilation command checks the syntax of global data declarations in source file MYPROG and saves the declarations in file TALSYM for use in subsequent compilations:

```
TAL /IN myprog/; SAVEGLOBALS talsym, SYNTAX
```

2. This compilation command checks for the syntax of the code or data in source file MYPROG. In this compilation, USEGLOBALS retrieves global data declarations saved in the compilation shown in Example 1. (Because the previous compilation

produced no object file, you need not use SEARCH to retrieve global initializations as you normally would when you use USEGLOBALS.)

```
TAL /IN myprog/; USEGLOBALS talsym, SYNTAX
```

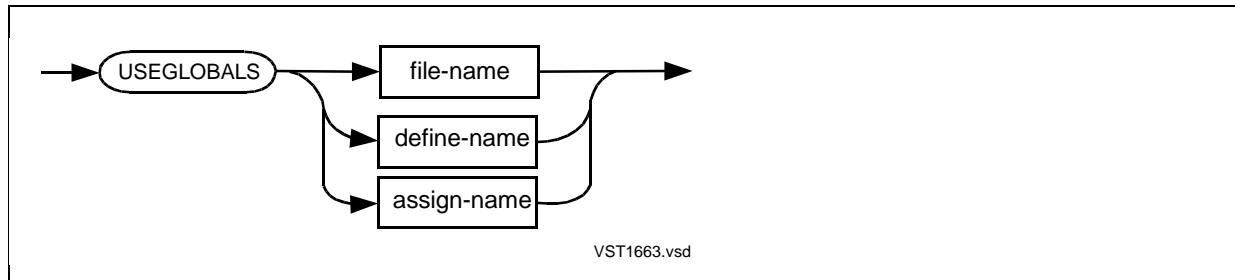
## TARGET Directive

TARGET lets you specify the target system for which conditional code is written.

For more information on the syntax for TARGET, see [Section 15, Privileged Procedures](#).

## USEGLOBALS Directive

USEGLOBALS retrieves the global data declarations saved in a file by SAVEGLOBALS during a previous compilation.



**file-name**

is the name of the global declarations disk file created by SAVEGLOBALS in a previous compilation.

You can specify partial file names as described in Appendix E in the *TAL Programmer's Guide*. The compiler uses TACL ASSIGN SSV information, if specified, to complete the file name. Otherwise, the compiler uses the current default volume and subvolume names as needed.

**define-name**

is the name of a TACL MAP DEFINE that refers to the global declarations file.

**assign-name**

is a logical file name you have equated to a disk file (that refers to the global declarations file) by issuing a TACL ASSIGN command.

## Usage Considerations

USEGLOBALS can appear either in the compilation command or in the source code before any global data declarations. If USEGLOBALS appears anywhere else, the compiler issues a warning and ignores the directive.

---

**Note.** Do not use USEGLOBALS and SAVEGLOBALS in the same compilation unit. If you do, the compiler issues an error message and uses only the first of the two directives.

---

Do not use USEGLOBALS and CROSSREF in the same compilation unit. If you do, the compiler does not pass Inspect and cross-reference information to SYMSERV.

## Saving Global Data Declarations

When you compile with SAVEGLOBALS, the compiler saves global data declarations as follows:

- It stores the identifiers and data characteristics (data type and kind of variable) in a global declarations file.
- It stores the initialization values (addresses and constant lists) in the object file.

## Retrieving Saved Global Data Declarations

After the SAVEGLOBALS compilation completes, you can subsequently compile the source file and retrieve the saved global data declarations and initializations by using the following directives:

- USEGLOBALS retrieves the saved global data declarations and suppresses compilation of text lines and SOURCE directives until a BEGINCOMPILATION appears.
- BEGINCOMPILATION marks the point at which compilation is to begin. BEGINCOMPILATION, if present, must appear after the last global data declaration or SEARCH directive and before the first procedure declaration, including EXTERNAL and FORWARD declarations.
- SEARCH retrieves global initializations and template structure declarations (unless you used SYNTAX in the SAVEGLOBALS compilation).

Make sure the global data declarations in both the SAVEGLOBALS and USEGLOBALS compilations are identical. If you include new or changed data declarations anywhere in the USEGLOBALS source file, results are unpredictable.

The USEGLOBALS compilation terminates if the global declarations file:

- Cannot be found or opened by the compiler
- Is not file code 105
- Was created using a different version of the compiler

Whenever you switch to a new version of the compiler, you must recompile the source code using SAVEGLOBALS to create a new global declarations file.

## Effect of Other Directives

When you use the following directives in the SAVEGLOBALS compilation, they affect the USEGLOBALS compilation as follows:

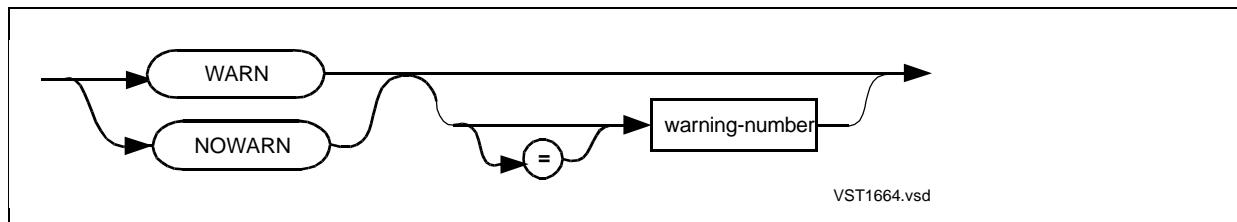
<b>Directive Used in SAVEGLOBALS Compilation</b>	<b>Effect in Subsequent USEGLOBALS Compilations</b>
SYNTAX	Negates need for using SEARCH in the USEGLOBALS compilation because no object file was produced by the SAVEGLOBALS compilation. You must use SYNTAX in the USEGLOBALS compilation, however.
INHIBITXX	Continues to generate inefficient but correct indexing for global extended declarations located in the first 64 words of primary global area.
PRINTSYM	Continues to print symbols in listings.
SYMBOLS	Continues making symbols available for all data blocks.
INT32INDEX	Continues to produce INT(32) indexes from INT indexes.

## Example of USEGLOBALS Directive

For an example and information on how the SAVEGLOBALS, USEGLOBALS, BEGINCOMPILATION, SEARCH, and SYNTAX directives interact, see the [SAVEGLOBALS Directive](#) on page 16-75.

## WARN Directive

For an example and information on how the SAVEGLOBALS, USEGLOBALS, BEGINCOMPILATION, =, and SYNTAX directives interact, see the [SAVEGLOBALS Directive](#) on page 16-75.



**warning-number**

is the number of a warning message. If specified with **WARN**, the message is printed. If specified with **NOWARN**, the message is suppressed.

## Usage Considerations

WARN or NOWARN can appear in the compilation command or anywhere in the source code.

To print selected warnings, you must specify WARN before any NOWARN directives. If you specify NOWARN first, any subsequent WARN warning-number directives have no effect.

You can use NOWARN when a compilation produces a warning and you have determined that no real problem exists. Before the source line that produces the warning, specify NOWARN and the number of the warning you want suppressed. Following that source line, specify a WARN directive.

### NOWARN Statistics

If NOWARN is in effect, the compiler records the number of suppressed and un-suppressed warnings. The compilation statistics at the end of the compiler listing include the following counts:

Number of un-suppressed compiler warnings = *count*

Number of warnings suppressed by NOWARN = *count*

Unsuppressed compiler warnings are compiler warnings that are not suppressed by NOWARN directives. The summary does not report the location of the last compiler warning.

If no compiler errors and no un-suppressed compiler warnings occur, the completion code is zero.

### Example of WARN Directive

This example specifies that the compiler does not print warning message 12:

```
?NOWARN 12
```

# A Error Messages

This appendix describes compiler messages:

- Compiler initialization messages
- Error messages
- Warning messages

## Compiler Initialization Messages

Initialization messages are unnumbered diagnostic messages that can appear during compiler initialization. If the OUT file is available, the compiler sends the messages there; otherwise, it sends them to the home terminal.

An initialization error terminates the compilation with a completion code of 3; that is, the compiler could not access a file. No object file is created.

Initialization messages are:

CREATION OF TEMPORARY FILE FAILED  
ILLEGAL INPUT DEVICE  
LIST DEVICE NOT AVAILABLE  
LIST FILE CANNOT BE AN EDIT FILE  
OPEN OF INITIAL SOURCE FILE FAILED  
OPEN OF TEMPORARY FILE FAILED  
REWIND OF SOURCE FILE FAILED

Any of these self-explanatory messages can be followed by a line of the format:

FILE MANAGEMENT ERROR # *error-number* ON FILE : *file-name*

## About Error and Warning Messages

The compiler scans each line of the source code and notifies you of an error or potential error by displaying one of two types of messages:

Message	Meaning	User Action
Error	Indicates a source error that prevents compilation of the source file into an object file.	Correct the error and recompile the source code.
Warning	Indicates a potential error condition that might affect program compilation or execution.	Check the source code carefully. If your program is adversely affected, make corrections and recompile the source code.

To indicate the location of the error or potential error, the compiler prints a circumflex symbol (^) in the source listing. The circumflex usually appears under the first character position following the detection of the error. (However, if the error involves the relationship of the current source line with a previous line, the circumflex does not always point to the actual error.)

On the next line, the compiler displays a message describing the nature of the error. The forms of error and warning messages are:

```
***** ERROR ***** message-number -- message-text  
***** WARNING ***** message-number -- message-text
```

Occasionally, the compiler adds a third line for supplemental information. For example, the following message refers you to an earlier procedure that contains an error:

```
IN PROC proc-name
```

As another example, the following line refers you to a previous page that contains an error:

```
PREVIOUS ON PAGE #page-num
```

Error messages are described on the following pages in ascending numeric order, followed by warning messages. Although the compiler prints each message on a single line, some messages here are continued on additional lines because of line limitations.

Messages no longer in use are not shown in the list. Thus, a few numbers are omitted from the numeric sequence.

## Error Messages

Error diagnostic messages identify source errors that prevent correct compilation. No object file is produced for the compilation.

### 0

#### Compiler Error

This error means that the compiler's data is no longer correct.

If the IN and OUT file numbers are incorrect when the compiler tries to send error 0 to the OUT file, the following file error appears at the home terminal. This error means the file has not been opened.

```
??: 016
```

Error 0 can occur after syntax errors or after a logic error. Error 0 is sometimes preceded by other error messages.

- Syntax error. Correct all syntax errors and recompile. If error 0 persists, contact your service provider. The following syntax errors, for example, cause error 0:

- A substructure that has an odd length, starts with an INT item, and has the same nonzero upper and lower bounds. The compiler does not pad this substructure properly.
  - Syntax errors within a structure declaration.
  - Nesting of procedure declarations or the appearance of such nesting, such as a formal parameter declared as a procedure but not included in the formal parameter list.
  - Parameter identifiers not separated by commas in the formal parameter list of a procedure declaration.
  - A formal parameter identifier that is a reserved keyword.
  - A BLOCK or NAME identifier that appears outside its declaration.
  - Too many actual parameters in a dynamic procedure call.
  - DEFINE text that begins with, but does not close with, a single quotation mark.
  - An invalid conditional expression in an IF statement or IF expression.
  - An invalid @ on the left side of an assignment.
  - Use of FILLER or BIT\_FILLER as variable identifiers within a structure.
- Logic error in compiler operation. A number follows the message for the use of NonStop development personnel. Report this occurrence to HP and include a copy of the complete compilation listing (and source, if possible).

## 1

### Parameter Mismatch

A parameter mismatch, such as the following, has occurred:

- The parameter type of an actual parameter is not the parameter type expected by that procedure. Pass an actual parameter that has the expected parameter type.
- The addressing mode (standard versus extended) of a parameter declaration does not match the addressing mode of its FORWARD or EXTERNAL declaration. Correct the addressing mode in the parameter declaration to match its FORWARD or EXTERNAL declaration.

**2**

Identifier declared more than once

Duplicate identifiers are declared within this scope. Replace one of the identifiers with an identifier that is unique within this scope. For example, identifier RESULT is declared twice within a procedure:

```
PROC p;  
  BEGIN  
    LABEL result;  
    INT result;  
!Duplicate identifier  
  END;
```

**3**

Recursive DEFINE invocation

A reference to a DEFINE declaration that is recursive appears in your source code. The message appears when the compiler expands the DEFINE. In the following example, the compiler expands A, which expands B, which expands A, and so on:

```
DEFINE a = b#, b = a#;
```

Rewrite the DEFINE so that it does not call itself.

**4**

Illegal MOVE statement or group comparison

An incorrect move statement or group comparison expression appears, for which the compiler cannot generate code. Correct the statement or expression.

**5**

Global primary area exceeds 256 words

The space required for your global variables exceeds the 256-word global primary area. The compiler allocates the following kinds of global variables in this area:

- Directly addressed simple variables, arrays, and structures
- 16-bit or 32-bit pointers you declare and initialize yourself
- 16-bit or 32-bit implicit pointers for indirect arrays and indirect structures
- Two 32-bit extended-stack pointers if you declared extended local variables

Declare most global arrays and structures by using indirection. Declare very large global arrays and structures by using extended indirection.

## 6

Illegal digit

A numeric constant contains a digit that is incorrect in the stated base of the constant. For example, an octal constant contains the digit 9. Correct the constant in accordance with [Section 3, Data Representation](#).

## 7

String overflow

A character string appears that:

- Contains more than 128 characters. Reduce the length of the character string.
- Does not terminate in the line in which it begins. Specify a constant list of smaller character strings; for example:

```
STRING a[0:99] := [ "These two constant strings will "
    "appear as if they were one character string." ]
```

## 8

Not defined for INT(32), FIXED, or REAL

An arithmetic operation occurs that is not permissible for operands of the listed data types. Correct the expression in accordance with [Section 4, Expressions](#).

## 9

Compiler does not allocate space for .SG STRUCT

A structure declaration incorrectly includes .SG (the system global indirection symbol). To access the system global area, you can:

- Equivalence a structure to a location relative to the base address of the system global area. (See [Section 15, Privileged Procedures](#))
- Declare a system global pointer (using the .SG symbol) and assign a structure address to it.

Otherwise, declare a global or local structure using the standard or extended indirection symbol (. or .EXT) and declare a sublocal structure without using any indirection symbol.

## 10

Address range violation
-------------------------

This message indicates one of the following conditions:

- A declaration specifies addresses beyond the allowable range; for example:  
`INT i = 'G' + 300;`  
Declare at most 256 words of directly addressable data relative to 'G', 127 words relative to 'L', and 31 words relative to 'S'.
- A subprocedure parameter or a sublocal variable cannot be accessed because other items have been pushed onto the data stack. Reorder the parameters or sublocal data. For more information on "Sublocal Storage Limitations" and "Sublocal Parameter Storage Limitations", see the *TAL Programmer's Guide*, which describes the limitations of sublocal storage.
- The total of primary and secondary global variables exceeds 32K words. Reduce the number or size of your global variables or move some of them to extended memory.
- The zeroth element of a global direct array is outside G-plus addressing. Declare the array so that its zeroth element falls within G-plus addressing. If the array is located at G[0], its lower bound must be a zero or negative value.
- The upper bound of the last sublocal array is less than zero. Specify an upper bound that is equal to or larger than zero.
- The size of an indirect array (of structures) exceeds 32K words. Reduce the size of the array.
- The size of a local variable exceeds 128K bytes. Reduce the size of the local variable.
- A procedure that you compile with the SQL directive has more than 122 words of primary local variables. Declare the excess words as indirect variables.

## 11

Illegal reference [variable-name] [parameter-number]
---

Conditions that cause this error include:

- A variable appears where a constant is expected, or a constant appears where a variable is expected. Replace *variable-name* with a constant or variable as required.

- A CALL statement passes a value parameter to a procedure that expects a reference parameter. Replace the parameter indicated by *parameter-number* with a reference parameter.
- A CODE statement includes indirect branches such as the following. Remove any indirect branches from the CODE statement.

```
CODE (BANZ .test1);  
!Some code here  
test1: CODE (CON @test2);
```

## 12

Nested routine declaration(s)

The following conditions can cause this error:

- One or more procedure declarations appear within another procedure declaration. Either replace the nested procedures with subprocedures or move the nested procedures outside the encompassing procedure.
- The identifier of a procedure declared as a parameter does not appear in the parameter list. Add the procedure identifier to the parameter list. For example:

```
PROC myproc (a);           !Q is missing from parameter list  
    INT a;  
    PROC q;           !Q is declared as parameter  
    BEGIN  
        !Lots of code  
    END;
```

## 13

Only 16-bit INT value(s) allowed

- A value of a data type other than INT appears where only INT values are permitted. Specify INT values in this context.
- An index of a data type other than INT appears for an equivalenced address. Specify an INT index for the equivalenced address.

**14**

Only initialization with constant value(s) is allowed
---

A global initialization expression includes variables. Initialize global data only with constant expressions, which can include `@identifier` when used with standard functions that return a constant value. `@` accesses the address of `identifier`, which must be a nonpointer variable with a 16-bit address. For example:

```

STRUCT .st[0:1];
BEGIN
    INT a;
    INT b;
    INT c;
END;
INT .p1 := @st.b;                                -- error 14
INT .p2 := @st '+' $OFFSET (st.b) / 2;          -- works fine
INT .q1 := @st[1].c;                            -- error 14
INT .q2 := @st '+' $LEN (st) / 2
'+' $OFFSET (st.c) / 2;                          -- works fine

```

**15**

Initialization is illegal with reference specification
--

An equivalenced variable declaration tries to initialize the previously declared variable. Initialize the previous variable before referring to it in the equivalenced variable declaration. For example:

```

INT b;
INT .a = b := 5;           !Cannot initialize B here; error 15
INT b := 5;                !Can initialize B here
INT .a = b;                !No error

```

**16**

Insufficient disk space on swap volume
--

The swap volume cannot accommodate your program. For example, either of the following values are too large for the memory available on the swap volume:

- The SQL PAGES value
- The combined SQL PAGES and SYMBOLPAGES values

Reduce the value that is too large, or specify another swap volume.

## 17

Formal parameter type specification is missing

A declaration for a formal parameter is missing in the procedure or subprocedure header. Declare the missing formal parameter or remove its identifier from the formal parameter list.

## 18

Illegal array bounds specification

Incorrect bounds appear in an array declaration. To correct this error:

- Specify bounds that are constant expressions.
- Specify a lower bound that is smaller than the upper bound. (This is not a requirement when the array is declared within a structure.)
- Specify no bounds when you declare an equivalenced variable:

INT a[0:5] = b; !Cause error 18

## 19

Global or nested SUBPROC declaration

A subprocedure declaration appears either outside a procedure or within another subprocedure. Declare all subprocedures within a procedure but not within a subprocedure.

## 20

Illegal bit field designation

A bit field construct is incorrect. Correct the construct so that both bit numbers are INT constants and the left bit number is less than or equal to the right bit number; for example:

<0:5>

**21**

Label declared more than once

The same identifier appears more than once as a statement label in the same scope. Specify label identifiers that are unique within a scope. For example, MYLABEL appears twice as statement labels within a procedure :

```
PROC q;  
  BEGIN  
    mylabel:  
      !Some statements here  
    mylabel:  
      !More statements here  
  END;
```

!Duplicate statement label

**22**

Only standard indirect variables are allowed

The extended (32-bit) indirection symbol (.EXT) appears where the compiler expects standard indirection. Use the standard (16-bit) indirection symbol (.) in this context.

**23**

Variable size error

The size field of a data type is incorrect. For example, INT(12) is incorrect. Specify a correct data type, such as INT or INT(32).

**24**

Data declaration(s) must precede PROC declaration(s)

A global data declaration follows a procedure declaration. Declare all global data before the first procedure declaration.

**25**

Item does not have an extended address

The argument to the standard function \$LADR does not have an extended address. When you use \$LADR, specify an argument that has an extended address.

**26**

Routine declared forward more than once

More than one FORWARD declaration for the given procedure or subprocedure is present. Declare a procedure or subprocedure FORWARD only once. Delete all duplicate FORWARD declarations.

**27**

Illegal syntax

A statement or the line preceding it contains one or more syntax errors. For example, the following conditions can cause error 27, followed by error 0, which terminates compilation:

- A misplaced or missing semicolon; for example:

```
PROC myproc          !Place semicolon here
  BEGIN;           !Remove this semicolon
  INT num; sum;    !Replace first semicolon with comma
  num := 2;         !Place semicolon here
  sum := 5;
  END;
```

- A reserved word appears as an identifier in the formal parameter list of a procedure declaration. Replace the identifier with a nonreserved identifier.
- An incorrect conditional expression appears in an IF statement or IF expression. Correct the expression.

**28**

Illegal use of code relative variable

A read-only array appears in an incorrect context, such as:

- On the left side of an assignment operator (:=)
- On the left side of a move operator (':=:' or '=:)'
- On the left side of a group comparison expression

You can use a normal array instead. For the syntax of arrays, see [Section 7, Arrays](#).

**29**

Illegal use of identifier <i>name</i>
---------------------------------------

An identifier appears in the formal parameter specification of a procedure or subprocedure declaration but not in the formal parameter list. Either include the identifier in the formal parameter list or remove the identifier from the formal parameter specification.

**30**

Only label or USE variable allowed
------------------------------------

A DROP statement specifies an incorrect identifier. In the DROP statement, specify the identifier of a label or a USE statement variable.

**31**

Only PROC or SUBPROC identifier allowed
---

A CALL statement specifies an incorrect identifier. In the CALL statement, specify the identifier of a procedure, subprocedure, or entry point.

**32**

Type incompatibility
----------------------

This message indicates one of the following conditions:

- An expression contains operands of different data types. To make the types match, use type-transfer standard functions or specify constants correctly. For example, for an INT(32) constant, use the D suffix.
- A procedure without a return type occurs on the right side of an assignment statement. Specify only a function in this context.
- Either a procedure is declared INT and its FORWARD declaration is declared INT(32), or a procedure is declared INT(32) and its FORWARD declaration is declared INT. Make the data type in both declarations match.
- A RETURN statement has no return value and the correct number of words for a return value is not in the register stack. In an INT procedure, for example, specify an INT value in the RETURN statement.
- A RETURN statement specifies a return value of the wrong data type. Specify the correct data type.

- An undeclared variable prefixed by @ is passed as an actual parameter. The compiler treats the undeclared variable as a label, so if the allocated size of the formal parameter does not match that of the label address, the compiler issues the error. Declare the variable before passing it as a parameter.
- A character string is assigned to a FIXED or REAL(64) variable. Change the value from a character string, or change the data type.

## 33

Illegal global declaration(s)

A declaration occurs for an item (such as a label) that cannot be a global item. At the global level, you can declare LITERALs, DEFINEs, simple variables, arrays, structures, simple pointers, structure pointers, and equivalenced variables.

## 34

Missing Variable

A required variable is missing from the current statement. Specify all variables shown as required in the syntax diagrams in this manual.

## 35

Subprocedures cannot be parameters

A subprocedure is declared as a formal parameter or is passed as an actual parameter. You can declare and pass procedures (but not subprocedures) as parameters.

## 36

Illegal Range

A specified value exceeds the allowable range for a given operation. Correct the value.

## 37

Missing Identifier

A required identifier is missing from the current statement. Provide the missing identifier.

**38**

```
Illegal index register specification
```

You tried to reserve more than three registers for use as index registers. Use a DROP statement to reduce the number of reserved registers.

**39**

```
Open failed on file file-name
```

The compiler could not open the file you specified in a SOURCE directive. If NOABORT is in effect, the compiler prompts you for the name of a source file. You can take any of the following alternative actions:

- Retry the same source file name.
- Ignore that source file and continue compilation.
- Substitute another source file name.
- Terminate the compilation.

If you choose to ignore the source file or to terminate the compilation, error 39 appears. If SUPPRESS is in effect, the compiler prints the SOURCE directive before the error message.

**40**

```
Only allowed with a variable
```

- A bit-deposit construct is applied to a variable other than STRING or INT:

```
INT i;  
UNSIGNED(5) uns5;  
i := uns5.<13:14>; !Error 40 appears
```

- To correct the preceding error, change the variable to STRING or INT:

```
INT i, var;  
i := var.<13:14>;
```

- An operation or construct that is valid only when used with a variable appears in some other context, such as appending a bit-deposit field to an expression:

```
INT a, b;  
(a+b).<2:5> := 0; !Error 40 appears
```

- To correct the preceding error, assign the expression to a STRING or INT variable and then append the bit-deposit field to the variable:

```
INT a, b, var;  
!Code to store values in A and B  
var := a + b;  
var.<2:5> := 0;
```

**41**

Undefined ASSERTION procedure: *proc-name*

An ASSERT statement invokes an undeclared procedure specified in an ASSERTION directive. Either declare the procedure or specify an existing procedure in the ASSERTION directive.

**42**

Table overflow *number*

Your source program fills one of the fixed-size tables of the compiler. No recovery from this condition is possible. Correct the source program as indicated in the following table (*number* identifies the affected table):

<b>Number</b>	<b>Table Name</b>	<b>Condition/Action</b>
0	Constant	Place a DUMPCONS directive before the point of overflow to force the constant table to be dumped. Termination does not occur if a block move of a large constant list caused the overflow.
1	Tree	Simplify the expression.
2	Pseudo-Label	You might have too many nested IF statements. Simplify the IF statements.
3	Parametric DEFINE	The DEFINE macro being expanded has parameters that are too long. Shorten the parameters.
4	Section	SOURCE directives access too many sections at one time. Break the sections into two or more groups.

**43**

Illegal Symbol

The current source line contains an invalid character or a character that is invalid in the current context. Specify the correct character.

**44****Illegal Instruction**

The specified mnemonic does not match those for the NonStop system. As of release C00, the compiler does not generate code for the NonStop 1+ system. Replace the instruction with one described in the *System Description Manual* for your system.

**45****Only INT(32) value(s) allowed**

A value of the wrong data type appears. In this context, specify an INT(32) value.

**46****Illegal indirection specification**

The period symbol (.) is used on a variable that is already indirect. Specify only one level of indirection by removing the period symbol from the current context.

**47****Illegal for 16-bit INT**

An INT value appears where the compiler expected an INT(32) value. For the unsigned divide ('/') and unsigned modulo divide ('\') operations, specify an INT(32) dividend and an INT divisor.

**48****Missing *item-specification***

The source code is missing *item-specification*. Supply the missing item.

**49****Undeclared identifier**

- A reference to an undeclared data item appears in the source file. Declare the item or change the reference.
- The string parameter of a parameter pair (*string:length*) is misspelled. Correct the spelling.

**50**

Cannot drop this Label

50

Cannot drop this Label

A DROP statement refers to an undeclared or unused label. Drop a label only after you declare it and after the compiler reads all references to it. Dropping a label saves symbol table space and allows its reuse (as in a DEFINE macro).

**51**

Index register allocation failed

The compiler is unable to allocate an index register. You might have indexed multiple arrays in a single statement and reserved the limit of index registers using USE statements. Modify your program so that it requires no more than three index registers at a time.

**52**

Missing initialization for code relative array

Initialization is missing from a read-only array declaration. When you declare a read-only array, make sure you initialize the array with values (not including " ").

**53**

Edit file has invalid format or sequence *n*

The compiler detects an unrecoverable error in the source file. In the message, *n* is a negative number that identifies one of the following conditions:

Number	Condition/ Action
-3	Text-file format error. Correct the format.'
-4	Sequence error—the line number of the current source line is less than that of the preceding line. Correct the sequence of source lines.

**54**

Illegal reference parameter

A structure appears as a formal value parameter. Declare all structure formal parameters as reference parameters.

**55**

Illegal SUBPROC attribute
---------------------------

A subprocedure declaration includes an incorrect subprocedure attribute such as EXTERNAL or EXTENSIBLE. Remove the incorrect attribute from the subprocedure declaration. Subprocedures can only have the VARIABLE attribute.

**56**

Illegal use of USE variable
-----------------------------

A USE variable is used incorrectly. Correct the usage in accordance with the descriptions of the USE statement or the optimized FOR statement.

**57**

Symbol table overflow
-----------------------

The compiler has not allocated sufficient space for the symbols in your program. You can either:

- Use the SYMBOLPAGES directive to increase the allocation
- Break the program into smaller modules

**58**

Illegal branch
----------------

Your program branches into a FOR statement that uses a USE register as its counter. Branch to the beginning of the FOR statement, not within it.

**59**

Division by zero
------------------

The compiler detects an attempt to divide by 0. Correct the expression to avoid division by 0.

**60**

Only a data variable may be indexed
-------------------------------------

An index is appended to an invalid identifier such as the identifier of a label or an entry point. Append an index only to the identifier of a variable.

**61**

Actual/formal parameter count mismatch
--

A call to a procedure or subprocedure supplies more (or fewer) parameters than you defined in the procedure or subprocedure declaration. Supply all required parameters, and supply at least commas for all optional parameters.

**62**

Forward/external parameter count mismatch
---

The number of parameters specified in a FORWARD or EXTERNAL declaration differs from that specified in the procedure body declaration. Specify the same number of parameters in the procedure body declaration as there are in the FORWARD or EXTERNAL declaration.

**63**

Illegal drop of USE variable in context of FOR loop
---

Within a FOR loop, a DROP statement attempts to drop a USE register that is the index of the FOR loop. The FOR loop can function correctly only if the register remains reserved. Remove the DROP statement from within the FOR loop.

**64**

Scale point must be a constant
--------------------------------

The current source line contains a scale point that is not a constant. Specify the *fpoint* in a FIXED variable declaration and the *scale* argument to the \$SCALE function as INT constants in the range -19 through +19.

**65**

Illegal parameter or routine not variable
---

The *formal-param* supplied to the \$PARAM function is not in the formal parameter list for the procedure, or the \$PARAM function appears in a procedure that is not VARIABLE or EXTENSIBLE. Use the \$PARAM function only in VARIABLE procedures and subprocedures and in EXTENSIBLE procedures.

**66**

Unable to process remaining text
----------------------------------

This message is usually the result of a poorly structured program, when numerous errors are compounded and concatenated to the point where the compiler is unable to proceed with the analysis of the remaining source lines. Review the code to determine how to correct the errors.

**67**

Source commands nested too deeply
-----------------------------------

The compilation unit nests SOURCE directives to more than seven levels, not counting the original outermost source file. That is, a SOURCE directive reads in source code that contains a SOURCE directive that reads in source code that contains a SOURCE directive that reads in source code, and so on, until the seven-level limit is exceeded. Reduce the number of nested levels.

**68**

This identifier cannot be indexed
-----------------------------------

A directly addressable variable was indexed and used in a memory-referencing instruction in a CODE statement. Modify the code to avoid this usage.

**69**

Invalid template access
-------------------------

A template structure is referenced as an allocated data item, such as in the \$OCCURS function. Refer to a template structure only in the declaration of a referral structure or a structure pointer.

**70**

Only items subordinate to a structure may be qualified
--

A qualified reference appears for a nonstructure item. Use the qualified identifier form of *structure-name.substructure-name.item-name* only for data items within a structure.

**71**

Only INT or STRING STRUCT pointers are allowed

A structure pointer of an incorrect data type (attribute) occurs. Specify only the INT or STRING attribute when you declare structure pointers.

**72**

Indirection must be supplied

An indirection symbol is missing from a pointer declaration. When you declare a pointer, specify an indirection symbol preceding the pointer identifier.

**73**

Only a structure identifier may be used as a referral

An incorrect referral occurs in a declaration. For the referral, specify the identifier of a previously declared structure or structure pointer.

**74**

Word addressable items may not be accessed through a STRING structure pointer

A STRING structure pointer attempts to access word-addressed items. To access word-addressed structure items, use an INT structure pointer, either a standard (16-bit) pointer or an extended (32-bit) pointer.

**75**

Illegal UNSIGNED variable declaration

An indirect UNSIGNED variable declaration occurs. Declare an UNSIGNED variable as directly addressed, regardless of its scope (global, local, or sublocal).

**76**

Illegal STRUCT or SUBSTRUCT reference

An incorrect structure or substructure reference occurs. Refer to a structure or substructure only:

- In a move statement

- In a group comparison expression
- In a SCAN or RSCAN statement
- As an actual reference parameter
- As `@identifier` in an expression

**77**

Unsigned variables may not be subscripted

An indexed (subscripted) reference to an UNSIGNED simple variable occurs. Remove the index from the identifier of the UNSIGNED simple variable.

**78**

Invalid number form

A floating-point constant appears in an incorrect form. Use one of the forms described in “REAL and REAL(64) Numeric Constant” in [Examples of REAL and REAL \(64\) Numeric Constants](#) on page 3-15

**79**

REAL underflow or overflow

Underflow or overflow occurred during input conversion of a REAL or REAL(64) number. Specify floating-point numbers in the following approximate range:

$\pm 8.6361685550944446E-78$  through  $\pm 1.15792089237316189E77$

**80**

OPTIMIZE 2 register allocation conflict

The compiler has run out of registers in OPTIMIZE 2 mode. The compiler emits error 80 and continues the compilation.

**81**

Invoked forward PROC converted to external

An EXTERNAL procedure declaration occurs for a procedure that was previously called as if it were a FORWARD procedure. Correct either the EXTERNAL declaration or the call to the FORWARD procedure.

**82**

CROSSREF does not work with USEGLOBALS

The USEGLOBALS directive appears in a source file submitted to the stand-alone Crossref product. The compiler issues error 82 and stops the Crossref product. Before resubmitting the source file to the Crossref product, remove the USEGLOBALS directive from the source file.

**83**

CPU type must be set initially

A CPU directive appears in the wrong place. Specify this directive preceding any data or procedure declarations.

**84**

There is no SCAN instruction for extended memory

An extended indirect array is the object of a SCAN or RSCAN statement. The hardware does not support scans in extended memory. Move the array temporarily into a location in the user data segment and perform the scan operations there.

**85**

Bounds illegal on .SG or .EXT items

A system global or extended pointer declaration includes bound specifications. Remove the bounds from such declarations.

**86**

Constant expected and not found

A variable appears where the compiler expects a constant. Replace the variable with a constant.

**87**

Illegal constant format

A constant appears in an incorrect form. Specify constants in the forms described in [Section 3, Data Representation](#).

**88**

Expression too complex. Please simplify

The current expression is too complex. The compiler's stack overflowed and the compilation terminated. Simplify the expression.

**89**

Only arrays of simple data types can be declared read-only

A structure declaration contains = 'P', which is restricted to read-only array declarations. Either remove = 'P' from the structure declaration, or replace the structure declaration with a read-only array declaration.

**90**

Invalid object file name - *file-name*

The object file name is incorrect. Specify the name of a disk file.

**91**

Invalid default volume or subvolume

The default volume or subvolume in the startup message is incorrect. Correct the volume or subvolume name.

**92**

Branch to entry point not allowed

An entry point is the target of a GOTO statement. Following the entry point, add a label identifier and then specify the label identifier in the GOTO statement.

**93**

Previous data block not ended

A BLOCK or PROC declaration appears before the end of the previous BLOCK declaration. End each BLOCK declaration with the END BLOCK keywords before starting a new BLOCK declaration or the first PROC declaration. This message occurs only if the compilation begins with a NAME declaration.

**94**

Declaration must be in a data block

An unblocked global data declaration appears after a BLOCK declaration. Either place all unblocked global declarations inside BLOCK declarations or place them before the first BLOCK declaration or SOURCE directive that includes a BLOCK. This message occurs only if the compilation begins with a NAME declaration.

**95**

Cannot purge file *file-name*

A security violation occurs in a SAVEGLOBALS compilation and the compiler cannot purge the existing global data declarations file. Correct the security violation and then recompile.

**96**

Address references between global data blocks not allowed

A variable declared in one global data block is initialized with the address of a variable declared in another global data block. Because global data blocks are relocatable, such an initialization is invalid. Include both declarations in the same global data block (or BLOCK declaration). This message occurs only if the compilation begins with a NAME declaration.

**97**

Equivalences between global data blocks not allowed

An equivalenced declaration in a global data block refers to a variable declared in another global data block. Place both declarations in the same global data block (or BLOCK declaration). This message occurs only if the compilation unit begins with the NAME declaration.

**98**

Extended arrays are not allowed in subprocedures

A declaration for an extended indirect array appears in a subprocedure. Remove the extended indirection symbol (.EXT) from the array declaration. Sublocal variables must be directly addressed.

**99**

Initialization list exceeds space allocated

A constant list contains values that exceed the space allocated by the data declaration. List smaller values in the constant list or declare larger variables.

**100**

Nested parametric-DEFINE definition encountered during expansion

Nesting of DEFINE declarations occurs. Remove the DEFINE declaration that is nested within another DEFINE declaration.

**101**

Illegal conversion to EXTENSIBLE

An attempt to convert an ineligible procedure to EXTENSIBLE occurs. Convert only a VARIABLE procedure that has at least one parameter, at most 16 words of parameters, and all one-word parameters except the last, which can be a word or longer. Also specify the number of parameters the procedure had when it was VARIABLE.

**102**

Illegal operand for ACON

A CODE statement contains an incorrect constant for the ACON option. Specify a constant that represents the absolute run-time code address associated with the label in the next instruction location. An absolute code address is relative to the beginning of the code space in which the encompassing procedure resides.

**103**

Indirection mode specified not allowed for P-relative variable

A read-only array declaration includes an indirection symbol. Remove the indirection symbol from the declaration.

**104**

This procedure has missing label - *label-name*

The procedure refers to either:

- A label identifier that is missing from the procedure
- An undeclared variable prefaced with @ in an actual parameter list

Either use *label-name* in the procedure or declare the variable.

**105**

A secondary entry point is missing - *entry-point-name*

The entry point is not present in the procedure. Declare an entry-point identifier and use it in the procedure.

**106**

A referenced subprocedure declared FORWARD is missing - *subproc-name*

The procedure contains a FORWARD subprocedure declaration and a call to that subprocedure but the subprocedure body is missing. Declare the subprocedure body.

**108**

Case label must be signed, 16-bit integer

An incorrect case label appears in a labeled CASE statement. Specify a signed INT constant or a LITERAL for the case label.

**109**

Case label range must be non-empty - *range*

A case alternative in a labeled CASE statement has no values associated with it. Specify at least one value for each case alternative.

**110**

```
This case label (or range) overlaps a previously used case  
label - n
```

The value *n* appears as a case label more than once in a labeled CASE statement. Specify unique case labels within a CASE statement.

**111**

```
The number of sparse case labels is limited to 63
```

Too many case labels appear in a labeled CASE statement. Specify no more than 63 case labels.

**112**

```
USEGLOBALS file was created with an old version of TAL or  
file code is not 105, file-name
```

The USEGLOBALS directive cannot use the global declarations file named in the message. If the global declarations file does not have file code 105, recompile the source code by using the SAVEGLOBALS directive. Use the same version of the compiler for both the SAVEGLOBALS and USEGLOBALS compilations.

**113**

```
File error number, file-name
```

A file error occurred when the compiler tried to process a file named in a directive such as ERRORFILE or SAVEGLOBALS. The message includes the name of the file and the number of the file error. Provide the correct file name.

**114**

```
@ prefix is not allowed on SCAN, MOVE or GROUP COMPARISON  
variable
```

A variable in a SCAN or move statement, or in a group comparison expression, is prefixed with @, which returns the address of the variable. Remove the @ operator.

**115**

Missing FOR part

The FOR *count* clause is missing from a move statement. Include the FOR *count* clause in the move statement.

**116**

Illegal use of period prefix

An incorrect use of the period (.) prefix occurs. Use this prefix only as follows:

- As an indirection symbol in declarations
- As a separator in qualified identifiers of structure items, as in MYSTRUCT.SUBSTRUCT.ITEMX
- As a dereferencing symbol with INT and STRING identifiers to add a level of indirection

**117**

Bounds are illegal on struct pointers

A structure pointer declaration includes bounds. Remove the bounds from the declaration.

**118**

Width of UNSIGNED array elements must be 1, 2, 4, or 8 bits

An incorrect *width* value appears in an UNSIGNED array declaration. For the *width* of UNSIGNED array elements, specify 1, 2, 4, or 8 only.

**119**

Illegal use of @ prefix together with index expression

In an assignment statement, an indexed pointer identifier appears. If the pointer is declared within a structure or substructure, append the index to the structure or substructure identifier, not to the pointer identifier. Otherwise, remove the index from the assignment statement.

**120**

Only type INT and INT(32) index expressions are allowed

An index expression of the wrong data type appears. Specify an INT or INT(32) index expression.

**121**

Only STRUCT items are allowed here

An incorrect argument to the \$BITOFFSET function appears. Specify only a structure item as the \$BITOFFSET argument.

**122**

Extended MOVE or GROUP COMPARISON needs a 32-bit NEXT ADDRESS variable

The next-address (*next-addr*) variable in a move statement or a group comparison expression is the wrong data type. Specify an INT(32) *next-addr* variable because the compiler emits an extended move sequence when:

- The destination variable or the source variable has extended addressing.
- The destination variable or the source variable has byte addressing and the other has word addressing.

**123**

Not allowed with UNSIGNED variables

An incorrect reference to an UNSIGNED variable appears. Correct the reference so that the UNSIGNED identifier:

- Has no @ or period (.) prefix
- Is not sent as an actual parameter to a reference formal parameter
- Is not the source or destination of a move statement, a SCAN or RSCAN statement, or a group comparison expression

**124**

ERRORFILE exists and its file code is not 106, will not purge

The file specified in an ERRORFILE declarative has the wrong file code. The compiler does not purge the file. Specify a file that has file code 106.

MOVE or GROUP COMPARISON count-unit must be BYTES, WORDS, or ELEMENTS

An incorrect *count-unit* appears in a move statement or in a group comparison expression. Specify the *count-unit* as BYTES, WORDS, or ELEMENTS only.

## 126

Initialization of local extended arrays is not allowed

An initial value appears in a local extended array declaration. Remove the initialization from the declaration and use an assignment statement instead.

## 127

Illegal block relocation specifier, expecting either AT or BELOW

An incorrect relocation clause appears in a BLOCK declaration. Specify only the AT or BELOW clause to relocate the block.

## 128

Illegal block relocation position

An incorrect relocation position appears in a BLOCK declaration. Specify AT (0), BELOW (64), or BELOW (256) only.

## 129

This variable must be subscripted

A reference to an UNSIGNED array appears without an index (subscript). When you refer to an UNSIGNED array, always append an index to the array identifier.

## 130

This variable may not be the target of an equivalence

The new variable in an equivalenced variable declaration is type UNSIGNED. Declare the new variable as any type except UNSIGNED.

Procedure code space exceeds 32K words

A single procedure is larger than 32K words long. Reduce the size of the procedure.

## 132

Compiler internal logic error detected in code generator

The code generator detected an internal logic error. Report this occurrence to HP and include a copy of the complete compilation listing (and source, if possible).

## 133

Compiler label table overflow

The code generator detected a label table overflow. Report this occurrence to HP and include a copy of the complete compilation listing (and source, if possible).

## 134

Value assigned to USE variable within argument list may be lost

An assignment to a USE register appears in an actual parameter list. After a procedure call, the compiler restores the USE register to its original value and the changed value is lost. Before issuing the CALL statement, use an assignment statement to change the value in the USE register.

## 135

Use relational expression

UNSIGNED(17–31) operands appear incorrectly in a conditional expression. To correct the expression, either change the data type or change the operator. With Boolean operators and unsigned relational operators, use only STRING, INT, or UNSIGNED(1–16) operands. With signed relational operators, use operands of any data type, including UNSIGNED(17–31) operands.

**136**

Compiler relative reference table overflow

The compiler's relative reference table overflowed. Report this occurrence to HP and include a copy of the complete compilation listing (and source, if possible).

**137**

Cannot purge error file *file-name* - File system error number

The compiler encountered a file error when it tried to purge an error file whose name was specified in an ERRORFILE directive. The message includes the name of the file and the number of the file system error. Supply the correct file name.

**138**

Not a host variable

A data item appears where an SQL host variable is expected. Declare the data item in an EXEC SQL DECLARE block, as described in the *NonStop SQL Programming Manual for TAL*.

**139**

Invalid declaration for length component of string parameter

An incorrect *length* parameter appears in a parameter pair specification. Declare this parameter as an INT simple variable that specifies the length of the *string* parameter in the parameter pair.

**140**

Too many parameters

Too many formal parameters appear in a procedure or subprocedure declaration. For a procedure, include no more than 32 formal parameters. For a subprocedure, include no more than allowed by the space available in the parameter area.

**141**

Invalid declaration for string component of string parameter

An incorrect *string* parameter appears in a parameter pair specification of the form *string:length*. Declare the *string* parameter as a standard indirect or extended indirect STRING array.

**142**

String parameter pair expected

A CALL statement passes an actual parameter to a procedure that expects a parameter pair. In the actual parameter list, replace the incorrect parameter with a parameter pair in the form: *string: length*

**143**

String parameter pair not expected

A CALL statement passes a parameter pair to a procedure that does not expect it. In the actual parameter list, replace the parameter pair with a single parameter.

**144**

Colon not allowed in the actual parameter list

A colon appears incorrectly in an actual parameter list. If the colon represents an omitted actual parameter pair, replace the colon with a comma. Use a colon only between the *string* and *length* parameters of a parameter pair.

**145**

Only 16-bit integer index expression allowed

An index expression of the wrong data type appears. Change the index expression to an INT expression.

**146**

Identifier for SQLMEM length must be an INT literal

An incorrect MAPPED length value appears in the SQLMEM directive. Specify the length value as an INT LITERAL or constant. The LITERAL identifier is interpreted when an EXEC SQL statement occurs, not when the directive occurs.

**147**

Identifier for SQLMEM address must be an extended string pointer.

An incorrect MAPPED address value appears in the SQLMEM directive. Specify the address value as a constant or an extended indirect identifier of type STRING. The LITERAL identifier is interpreted when an EXEC SQL statement occurs, not when the directive occurs.

**148**

Exceeded allocated space for SQLMEM MAPPED

SQL data structures need more space than is allocated. Specify a larger MAPPED length value in the SQLMEM directive.

**149**

Value out of range

The specified value is outside the permissible range of values. For example, the value 256 is outside the range for a BIT\_FILLER field, which has a range of 0 through 255. Specify a value that falls within the range.

**150**

SQLMEM STACK cannot be used in a SUBPROC

Within a subprocedure, the SQLMEM STACK directive is in effect when an SQL statement occurs. Because of addressability limits in subprocedures, parameters or data might not be accessible if you push data onto the stack. Remove the SQLMEM STACK directive from within the subprocedure.

**151**

Only STRING arrays may have SQL attributes

SQL attributes are applied to arrays that are not STRING arrays. Apply SQL attributes only to STRING arrays.

Type mismatch for SQL attribute

An incorrect SQL data type attribute occurs for a TAL variable. Specify an SQL data type attribute that matches the TAL data type of the variable.

## 153

Length mismatch for SQL attribute

The length of the data provided in the SQL attribute does not match the length of the data for the variable. Specify data having a length that conforms to the data type of the variable.

## 154

Exceeded available memory limits

The compiler has exhausted its internal memory resources. Change the swap volume (by using the PARAM SAMECPU command described in Appendix E in the *TAL Programmer's Guide*). If an excessive value appears in the EXTENDTALHEAP, SYMBOLPAGES, or SQL PAGES directive, specify a smaller value.

## 155

This directive not allowed in this part of program

The directive is not appropriate in its current location.

- If the directive belongs elsewhere, such as in the compiler run command or on the first line of the source file, relocate the directive.
- If the directive should not appear because of a previous occurrence of this or another directive, remove the incorrect directive.

## 156

ASSERTION procedure cannot be VARIABLE or EXTENSIBLE

A VARIABLE or EXTENSIBLE procedure is specified in an ASSERTION directive. Specify a procedure that has no parameters.

**160**

Only one language attribute is allowed

More than one language attribute appears in a procedure declaration. Specify only one of the following language attributes following the LANGUAGE keyword (and only in a D-series EXTERNAL procedure declaration):

C  
COBOL  
FORTRAN  
PASCAL  
UNSPECIFIED

**161**

Language attribute only allowed for external procedures

A language attribute appears in a procedure declaration that is not specified as being EXTERNAL. Specify LANGUAGE followed by C, COBOL, FORTRAN, PASCAL, or UNSPECIFIED only in D-series EXTERNAL procedure declarations.

**162**

Illegal size given in procedure parameter declaration

An incorrect parameter declaration appears. Specify the correct parameter type.

**163**

Public name only allowed for external procedures

A public name appears in a procedure declaration that is not specified as being EXTERNAL. Specify a public name only in D-series EXTERNAL procedure declarations.

**164**

Procedure was previously declared in another language

Multiple EXTERNAL declarations have the same procedure identifier but have different language attributes. Delete the incorrect EXTERNAL declaration.

**165**

Procedure was previously declared with a public name

The previous EXTERNAL procedure declaration includes a public name, and the current procedure declaration is a secondary entry point. Remove the public name from the EXTERNAL procedure heading.

**166**

Illegal public name encountered

An incorrect public name appears. Specify a public name only in a D-series EXTERNAL procedure declaration, using the identifier format of the language in which the external routine is written (C, COBOL85, FORTRAN, Pascal, or TAL).

**168**

Use a DUMPCONS directive before the atomic operation

Constants included in the code make it impossible to specify an atomic operation here. Use the DUMPCONS directive to move the constants out of the way.

**169**

Increase the size of the internal heap of the compiler by recompiling with the EXTENDTALHEAP directive

The internal heap of the compiler is too small. Specify the EXTENDTALHEAP directive in the compilation command in all subsequent compilations. An example is:

```
TAL /in mysrc, OUT mylst/ myobj; EXTENDTALHEAP 120
```

**175**

\$OPTIONAL is only allowed as an actual parameter or parameter pair

```
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), !Parameter pair
          $OPTIONAL (j > 2, j) ); !Parameter
END;
```

## 176

The second argument of \$OPTIONAL must be a string parameter pair

The called procedure declares a parameter pair, but the caller does not specify a parameter pair as the second argument to \$OPTIONAL. Replace the incorrect argument with a parameter pair to match the specification in the called procedure. See the example shown for Error 175.

## 177

The first argument of \$OPTIONAL must be a 16-bit integer expression

A conditional expression that does not evaluate to an INT expression appears as the first argument to \$OPTIONAL. Correct the conditional expression so that it evaluates to an INT expression. See the example shown for Error 175.

## 178

\$OPTIONAL allowed only in calls to VARIABLE or EXTENSIBLE procedures

.i.\$OPTIONAL not allowed (error178)

\$OPTIONAL appears in a call to a procedure that is not VARIABLE or EXTENSIBLE. Remove \$OPTIONAL from the CALL statement or declare the called procedure as VARIABLE or EXTENSIBLE. See the example shown for Error 175.

## 179

Undefined toggle: *toggle-name*

*toggle-name* is used before it is created. Create the named toggle in a DEFINETOG, RESETTOG, or SETTOG directive before using the toggle in an IF, IFNOT, or ENDIF directive.

# Warning Messages

The following messages indicate conditions that might affect program compilation or execution. If you get any warning messages, check your code carefully to determine whether you need to make corrections.

**0**

All index registers are reserved

Three index registers are already reserved by USE statements. Three is the maximum number of index registers that you can reserve. The compiler assigned the last identifier to an already allocated index register.

**1**

Identifier exceeds 31 characters in length

An identifier in the current source line is longer than 31 characters, the maximum allowed for an identifier. The compiler ignores all excess characters. Make sure the shortened identifier is still unique within its scope.

**2**

Illegal option syntax

An incorrect compiler directive option appears. The compiler ignores the option. If omitting the option adversely affects the program, specify the correct option.

**3**

Initialization list exceeds space allocated

An initialization list contains more values or characters than can be contained by the variable being initialized. The compiler ignores the excess items. If omitting the excess values adversely affects the program, declare a variable large enough to hold the values.

**4**

P-relative array passed as reference parameter

A procedure call passed the address of a read-only array to a procedure. Make sure the procedure takes explicit action to use the address properly.

**5**

PEP size estimate was too small

Your PEP estimate (in the PEP directive) is not large enough to contain all the entries required. BINSERV has allocated appropriate additional space.

**6**

Invalid ABSLIST addresses may have been generated

When the file reaches the 64K-word limit, the compiler disables ABSLIST, starts printing offsets relative to the procedure base instead of to the code area base, and emits this warning. Also, because the compiler is a one-pass compiler, some addresses are incorrect when:

- The file contains more than 32K words of code
- RESIDENT procedures follow nonresident procedures
- The PEP directive does not supply enough PEP table space
- All procedures are not FORWARD (and no PEP directive appears)

If the file has more than 32K words of code space or if you use the stand-alone Binder, do not use ABSLIST.

**7**

Multiple defined SECTION *name*

The same section name appears more than once in the same SOURCE directive. The compiler ignores all occurrences but the first. If omitting the code denoted by duplicate section names adversely affects the program, replace the duplicate section names with unique section names.

**8**

SECTION *name* not found

A section name listed on a SOURCE directive is not in the specified file. If omitting such code adversely affects the program, specify the correct file name.

**9**

RP or S register mismatch
---------------------------

An operation contains conflicting instructions for the RP (register pointer) or the S register that the compiler cannot resolve. An example of RP conflict is:

```
IF alpha THEN STACK 1 ELSE STACK 1D;
```

An example of S conflict is the following statement. The setting of the S register depends on the value of ALPHA, which the compiler cannot determine at compile time.

```
IF alpha THEN CODE (ADDS 1) ELSE CODE (ADDS 2);
```

The compiler might detect an RP or S conflict in source code that compiled without problem in releases before C00. If the object code does not execute as intended, recode the source code to eliminate the warning. You might only need to insert an RP or DECS directive.

**10**

RP register overflow or underflow
-----------------------------------

A calculation produced an index register number that is greater than 7 or less than 0. If this overflow or underflow adversely affects your program, correct the calculation.

**11**

Parameter type conflict possible
----------------------------------

.i.Parameter type conflict (warning 11)

An actual parameter does not have the parameter type specified by the formal parameter declaration. For example, a procedure passed the address of a byte-aligned (STRING) extended item as an actual parameter to a procedure that expects the address of a word-aligned item. If the address is not on a word boundary, the system ignores the odd-byte number and accesses the entire word. Make sure the size and alignment of the actual parameter matches the requirements of the called procedure.

**12**

Undefined option
------------------

Conditions that cause this warning include:

- An incorrect option in a directive
- Stray characters (such as semicolons) at the end of a directive line

In either case, the compiler ignores the directive. Enter the correct option or remove the stray characters.

## 13

Value out of range

A value exceeds the permissible range for its context (for example, a shift count is greater than the number of existing bits). If the value is important to your program, use a value that falls within the permissible range.

## 14

Index was truncated

This warning appears, for example, when you:

- Equivalence a STRING or INT item to an indexed odd-byte address
- Equivalence a direct variable equivalent to an indexed indirect variable

The compiler truncates the index; for example:

```
STRING .s[0:4]; INT s1 = s[1];      !Result is INT s1 = s[0]
```

## 15

Right shift emitted

A procedure or subprocedure call passed a byte address as a parameter to a procedure that expects a word address. The compiler converts the byte address to a word address by right-shifting the address. If the STRING item begins on an odd-byte boundary, the word-aligned item also includes the even-byte part of the word. If this is a problem, pass only a word address.

## 16

Value passed as reference parameter

A parameter is passed by value to a procedure or subprocedure that expects a reference parameter. If this is your intent, and if the value can be interpreted as a 16-bit address, no error is involved.

**17****Initialization value too complex**

An initialization expression is too complicated to evaluate in the current context. If your program is adversely affected, simplify the expression.

**19****PROC not declared FORWARD with ABSLIST option on**

A PEP directive or a FORWARD declaration is missing. When you use the ABSLIST directive, the compiler must know the size of the PEP table before the procedure occurs in the source program. Enter either a PEP directive at the beginning of the program or a FORWARD declaration for the procedure. The compiler also emits warning 6 at the end of the compilation.

**20****Source line truncated**

A source line extends beyond 132 characters. The compiler ignores the excess characters. If your program is adversely affected, break the source line into lines of less than 132 characters.

**21****Attribute mismatch**

The attributes in a FORWARD declaration do not match those in the procedure body declaration. If your program is adversely affected, correct the set of attributes.

**22****Illegal command list format**

The format of options in a directive is incorrect. The compiler ignores the directive. If your program is adversely affected, correct the format of the compiler options.

**23****The list length has been used for the compare count**

A FOR count clause and a constant list both appear in a group comparison expression, which are mutually exclusive. The compiler obtains the count of items from the length

of the constant list. If your program is adversely affected, correct the group comparison expression as described in [Section 4, Expressions](#).

**24**

A USE register has been overwritten

The evaluation of an expression caused the value in a USE register to be overwritten. Multiplication of two FIXED values, for example, can cause this to occur. If this affects your program adversely, use a local variable in place of the USE register.

**25**

FIXED point scaling mismatch

The *fpoint* of an actual FIXED reference parameter does not match that of the formal parameter. The system applies the *fpoint* of the formal parameter to the actual parameter.

**26**

Arithmetic overflow

A numeric constant represents a value that is too large for its data type, or an overflow occurs while the compiler scales a quadrupleword constant up or down.

**27**

ABS (FPOINT) > (19)

The *fpoint* in a FIXED declaration or the *scale* parameter of the \$SCALE function is less than -19 or greater than 19. The compiler sets the *fpoint* to the maximum limit, either -19 or 19.

**28**

More than one MAIN specified. MAIN is still *name*

Although the source code can contain more than one MAIN procedure, in the object code only the first MAIN procedure the compiler sees retains the MAIN attribute.

If the program contains any COBOL or COBOL85 program units, the MAIN procedure must be written in COBOL or COBOL85, respectively. For more information, see the *COBOL85 Reference Manual*.

**29**

One or more illegal attributes
--------------------------------

Incorrect attributes appear in a subprocedure declaration, which can have only the VARIABLE attribute. The compiler ignores all attributes but VARIABLE.

**32**

RETURN not encountered in typed PROC or SUBPROC
---

A RETURN statement is missing from a function or the RP counter of the compiler is 7 (empty register stack). To return a value from the function, include at least one RETURN statement with an expression; this action automatically places the value on the register stack. You can alternatively use a CODE or STACK statement to place the value on the register stack; however, this practice might be compatible only with TNS systems.

**33**

Redefinition size conflict
----------------------------

In a structure declaration, a variable redefinition appears in which the new item is larger than the previously declared item. If your program is adversely affected, correct the variable redefinition so that the new item is the same size or shorter than the previous item.

**34**

Redefinition offset conflict
------------------------------

In a structure declaration, a variable redefinition appears in which the new item requires word alignment, while the previously declared item has an odd-byte alignment. If your program is adversely affected, correct the variable redefinition so that the new item and the previous item are both word aligned or both byte aligned.

**35**

Segment number information lost
---------------------------------

If you pass an extended indirect actual parameter to a procedure that expects a standard address, the compiler converts the address and the segment number is lost. The resulting address is correct only within the user data segment. Pass an appropriate address.

**36**

Expression passed as reference parameter
--

A calling sequence passes a parameter in the form @identifier to a procedure or subprocedure that expects an extended pointer as a parameter. If the intent is to pass the address of the pointer rather than the address stored in the pointer, no error is involved.

**37**

Array access changed from indirect to direct
--

An indirect array is declared inside a subprocedure or is declared as a read-only array. The compiler changes the indirect array to a direct array because:

- The sublocal area has no secondary storage for indirect data.
- Code-relative addresses are always direct.

**38**

S register underflow
----------------------

The compiler attempted to generate code that decrements the S register below its initial value in a procedure or subprocedure. If you decide that your source program is correct, you must insert a DECS directive at that point to recalibrate the compiler's internal S register. This is essential in subprocedures, because sublocal variables have S-relative addresses.

**39**

This directive cannot be pushed or popped
---

A PUSH or POP prefix appears on a directive that has no directive stack. Specify the directive identifier without the prefix.

**40**

A procedure declared FORWARD is missing - <i>proc-name</i>
--

A FORWARD declaration occurs, but the procedure body declaration is missing. The compiler converts all references to this procedure into EXTERNAL references to the same identifier. If this is not your intent, declare the procedure body.

**41**

File system DEFINES are not enabled

A TACL DEFINE name appears in place of a file name, but the system is not configured to use TACL DEFINEs. Error 39 (Open failed on *file-name*) follows warning 41. Issue the TACL commands that enable the TACL DEFINE.

**42**

Specified bit extract/deposit may be invalid for strings

An incorrect bit specification occurs. To access or deposit bits in a STRING item, specify bit numbers 8 through 15 only. Specifying bit numbers 0 through 7 of a STRING item has no effect, because the system stores STRING items in the right byte of a word and places a zero in the left byte.

**43**

A default OCCURS count of 1 is returned

\$OCCURS appears with an argument that is a simple variable, pointer, or procedure parameter, so OCCURS returns a 1. Use \$OCCURS only with an array (declared in or out of a structure), a structure, or a substructure (but not a template structure or a template substructure).

**44**

A subprocedure declared FORWARD is missing - *subproc-name*

The named subprocedure is declared FORWARD but is not referenced, and its body is not declared. If the absence the body of *subproc-name* adversely affects your program, declare the subprocedure body.

**45**

Variable attribute ignored - no parameters

The VARIABLE attribute appears for a procedure or subprocedure that has no parameters. The compiler ignores the VARIABLE attribute.

**46**

Non-relocatable global reference

A declaration that refers to a G-relative location appears when the RELOCATE directive is in effect and all primary global data is relocatable. This reference might be incorrect if BINSERV relocates the data blocks when it builds the object file. Either change the declaration of the identifier or, if NAME (and BLOCK) statements do not appear, delete the RELOCATE directive.

**47**

Invalid file or subvolume specification

An incorrect file or subvolume appears in a directive. Correct the directive.

**48**

This directive not allowed in this part of program

A directive occurs in an inappropriate place. The compiler ignores the directive. If this adversely affects your program, move the directive to an appropriate location, usually to the first line of the program or to the compilation command.

**49**

Address of entry point used in an expression

The value of the construct `@entry-point-name` for a subprocedure is the address of the first word of code executed after a call to the entry point. If code written for releases before compiler version E01 contains the expression `@ep-1` to calculate the entry-point location, change it to `@ep` for correct execution.

**50**

Literal initialized with address reference

An incorrect global initialization occurs, in which a LITERAL represents the address of a global variable. Because global data is now relocatable, avoid initializing them with addresses. If your program is adversely affected, initialize the global variable properly.

**52**

The compiler no longer generates code for NonStop 1+ systems

As of release C00, the compiler no longer generates code for the NonStop 1+ system. Instructions that apply only to NonStop 1+ systems are CBDA, CBDB, CBUA, CBUB, DS, RMD, SCAL, SLRU, LWBG, MAPP, MNDS, MNSG, MNSS, UMPP, XINC, XSMS, ZZZZ. A CODE statement that specifies such an instruction causes error 44 (Illegal instruction).

**53**

Illegal order of directives on directive line

A directive that must be first on a directive line is not first, or a directive that must be last on a directive line is not last. Place the following directives as indicated below.

Directive	Place in Directive Line	Directive	Place in Directive Line
ASSERTION	Last	IF	Last
COLUMNS	First	SECTION	Alone
ENDIF	Alone	SOURCE	Last

If the COLUMNS directive appears in the compilation command, the compiler does not enforce the above ordering.

**54**

The structure item rather than the define will be referenced

A DEFINE declaration renames a structure item, but the qualified identifier of the DEFINE is the same as that of another structure item. A reference to the DEFINE identifier accesses the other structure item. To ensure proper references, use unique identifiers for all declarations.

**55**

The length of this structure exceeds 32767 bytes at item \*\*  
*item-name*

A structure occurrence exceeds 32,767 bytes in length. The message identifies the item that caused the structure to exceed the legal length; the next item is the one the compiler cannot access. Reduce the length of the structure.

**57**

SAVEGLOBALS and USEGLOBALS cannot appear in the same compilation

Two mutually exclusive directives appear in the same compilation unit. To save global declarations in a file for use by a later compilation, retain SAVEGLOBALS and delete USEGLOBALS. To make use of the saved global declarations, retain USEGLOBALS and delete SAVEGLOBALS.

**58**

Code space exceeds 64k, ABSLIST has been disabled

TAL compiler versions B00 and later support up to 16 \* 64K words of source code. When the code exceeds 64K words, the compiler disables ABSLIST for the remainder of the listing.

**59**

Number of global data blocks exceeds maximum, SAVEGLOBALS disabled

The program attempts to define a global data block beyond the compiler's storage limit of 100 such blocks. Reduce the number of global data blocks.

**60**

Previous errors and warnings will not be included in ERRORFILE

Errors detected and reported before the ERRORFILE directive occurs are not reported to the file specified in the directive.

**61**

This directive can appear only once in a compilation

The directive cited in the warning message is not the first of its kind in the compilation. Remove all duplicate occurrences of this directive.

**62**

Extended address of STRING p-rel. array is incorrect if reference is more than 32K words from array

The STRING read-only array is declared in a procedure or subprocedure, and the compiler may generate an incorrect address when converting the address of a global STRING read-only array to an extended address for use in a move statement, a group comparison, or a procedure call.

**63**

A file system DEFINE name is not permitted in a LIBRARY directive.

A TACL ASSIGN name or a TACL DEFINE name appears in a LIBRARY directive. Specify a disk file name, fully or partially qualified, in the LIBRARY directive.

**64**

No file system DEFINE exists for this logical file name

A TACL DEFINE name (such as =ABLE) occurs, but no such TACL DEFINE has been added in the operating system environment.

**65**

RELEASE1 and RELEASE2 options are mutually exclusive. The most recently specified option is in effect.

SQL directives specifying both the RELEASE1 and RELEASE2 options appear in the source code. The compiler applies the most recently specified option. If this is not your intent, specify only one of the two options.

**66**

WHENEVERLIST and NOWHENEVERLIST options are mutually exclusive. The most recently specified option is in effect.

SQL directives specifying the WHENEVERLIST and NOWHENEVERLIST options appear in the source code. The compiler applies the most recently specified option. If this is not your intent, specify only one of the two options.

**67**

SQLMAP and NOSQLMAP options are mutually exclusive. The most recently specified option is in effect

SQL directives specifying the SQLMAP and NOSQLMAP options appear in the source code. The compiler applies the most recently specified option. If this is not your intent, specify only one of the two options.

**68**

Cannot access SSV

An incorrect ASSIGN SSV (Search Subvolume) number occurs. Specify the correct SSV number.

**69**

Label declaration belongs inside a procedure or subprocedure

A label declaration appears outside of a procedure or subprocedure. Move the label declaration into the appropriate procedure or subprocedure.

**70**

Conflicting TARGET directive ignored

A TARGET directive conflicts with a previous TARGET directive. The compiler accepts only the first TARGET directive that it encounters.

**73**

An ASSIGN SSV number is too large

A TACL ASSIGN command specifies an SSV value larger than 49. Use SSV values in the range 0 through 49; for example:

```
ASSIGN SSV48, \node1.$vol2.subvol3
```

**74**

The language attribute for this procedure conflicts with a prior declaration

An EXTERNAL procedure declaration specifies language attributes that do not match those in the declaration that describes the procedure body.

**75**

There are too many ASSIGN commands

More than 75 ASSIGN commands appear in the compilation. Reduce the number of ASSIGN commands to 75 or fewer.

**76**

Cannot use \$OFFSET or \$LEN until base structure is complete

\$LEN or \$OFFSET is applied to an unfinished structure or to a substructure that is declared in an unfinished structure. The compiler does not calculate these values until the encompassing structure is complete. Complete the encompassing structure.

**77**

SYMSERV died while processing the cross-reference listing

The compiler has passed inconsistent symbol information to the Crossref process. Use the SYMBOLS directive for all your source code.

**78**

TAL cannot set RP value for forward branch

A CODE statement causes a conditional jump out of an optimized FOR loop. The compiler can generate correct code for the nonjump condition of the CODE statement, but not for the jump condition. To set the RP value for the forward branch, use a CODE (STRP ...) statement or an RP directive.

**79**

Invalid directive option. Remainder of line skipped.

An incorrect option appears in a directive. The compiler ignores the remainder of the directive line. Replace the incorrect option with a correct option.

**80**

This is a reserved toggle name

A reference to a reserved toggle name appears in the source code. Choose a different toggle name for your code.

**81**

The PAGES option of the SQL directive must be on the command line

An SQL directive specifying the PAGES option appears in a directive line. The compiler ignores this directive. Specify this directive only in the compilation command.

**83**

Too many user-defined named toggles

You have exceeded the maximum number of named toggles. Use fewer named toggles.

**84**

Invalid parameter list

An incorrect parameter list appears in a procedure or subprocedure declaration. If your program is adversely affected, correct the parameter list.

**86**

Return condition code must be 16-bit integer expression

A procedure returns a condition code that does not evaluate to an INT expression. Correct the condition code so that it evaluates to an INT expression.

**87**

The register stack should be empty, but is not

The register stack should be empty after each statement (except CODE, STACK, and STORE). If you have not left items on the register stack deliberately, change your code.

**88**

Address size mismatch

This warning appears, for example, when:

- You pass the content of a standard (16-bit) pointer by reference to a procedure that expects the content of an extended (32-bit) pointer.

- You pass the content of an extended (32-bit) pointer by reference to a procedure that expects the content of a standard (16-bit) pointer.

**89**

CROSSREF does not work with USEGLOBALS

The CROSSREF and USEGLOBALS directives both appear in a compilation unit. The compiler issues warning 89 and turns off the CROSSREF directive. If you need to collect cross-reference information, remove the USEGLOBALS directive from the compilation unit.

**90**

Initialization of UNSIGNED arrays is not supported

An array of type UNSIGNED is initialized. Remove the initialization from the declaration of the UNSIGNED array.

**91**

MAIN procedure cannot return a value

A RETURN value appears in a RETURN statement in the MAIN procedure. When the MAIN procedure terminates, it calls a system procedure, so the return value is meaningless.

**93**

Do not use an SQL statement in a parametrized DEFINE

An EXEC SQL statement appears in the text of a DEFINE that has parameters. The compiler evaluates the parameters and the SQL statement in the same buffer space.

**94**

A template structure is not addressable

The address of a template structure is specified. Specify only the address of a definition structure of a referral structure.

# SYMSERV Messages

The following message might appear during compilation:

```
SYMSERV FATAL ERROR
```

This error appears only when the compiler detects a logic error within its operation. If you are using versions of the TAL compiler and SYMSERV from the same release, and if no other error message appears that would explain this behavior, please report this occurrence to HP. Include a copy of the complete compilation listing and the source, if possible.

# BINSERV Messages

For BINSERV diagnostic messages, see the *Binder Manual*.

# Common Run-Time Environment Messages

For Common Run-Time Environment (CRE) diagnostic messages, see the *CRE Programmer's Guide*.



# B

## TAL Syntax Summary (Railroad Diagrams)

This appendix provides a syntax summary of railroad diagrams for specifying:

- Constants
- Expressions
- Declarations
- Statements
- Standard Functions
- Compiler Directives

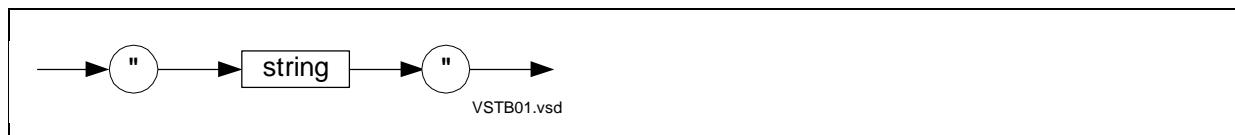
### Constants

The following syntax diagrams describe:

- Character string constants (all data types)
- STRING numeric constants
- INT numeric constants
- INT(32) numeric constants
- FIXED numeric constants
- REAL and REAL(64) numeric constants
- Constant lists

#### Character String Constants

A character string constant consists of one or more ASCII characters stored in a contiguous group of bytes.



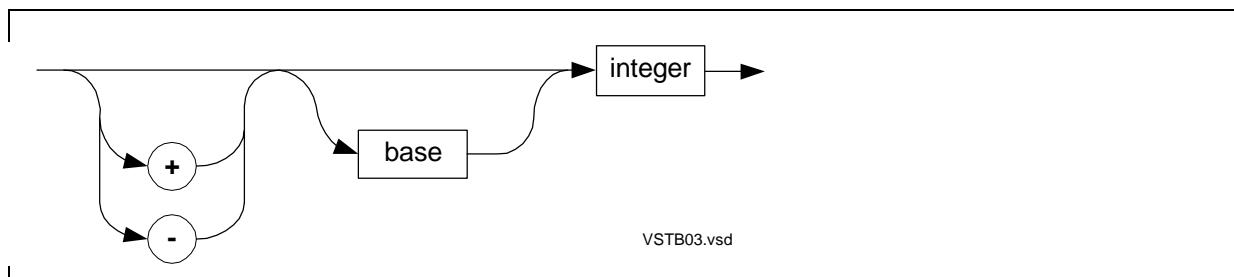
## STRING Numeric Constants

A STRING numeric constant consists of an unsigned 8-bit integer.



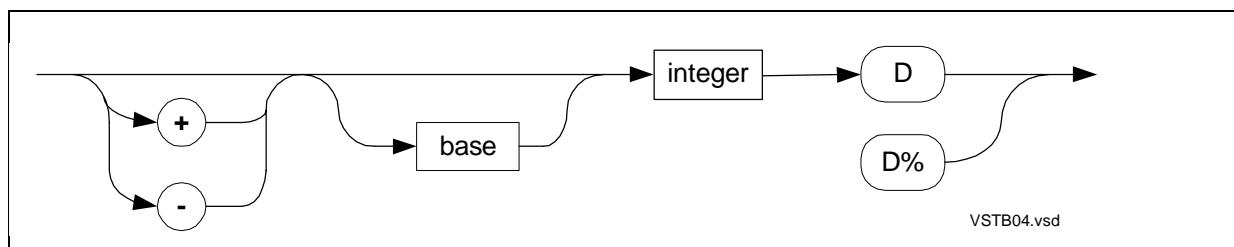
## INT Numeric Constants

An INT numeric constant is a signed or unsigned 16-bit integer.



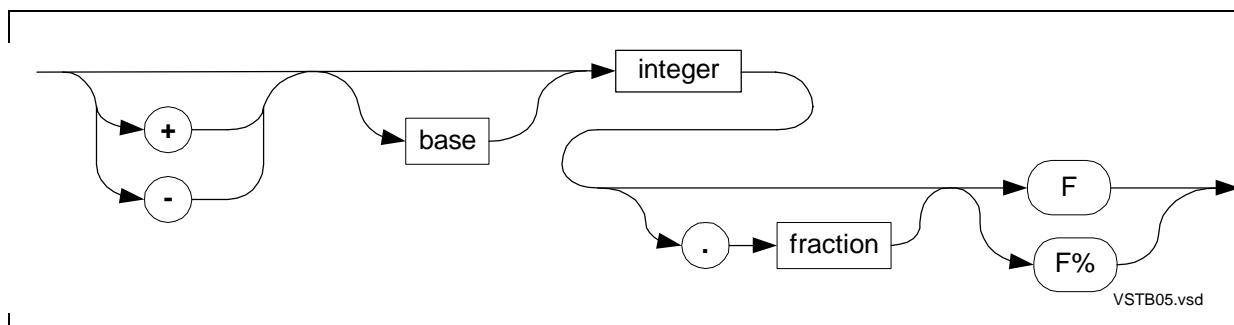
## INT(32) Numeric Constants

An INT(32) numeric constant is a signed or unsigned 32-bit integer.



## FIXED Numeric Constants

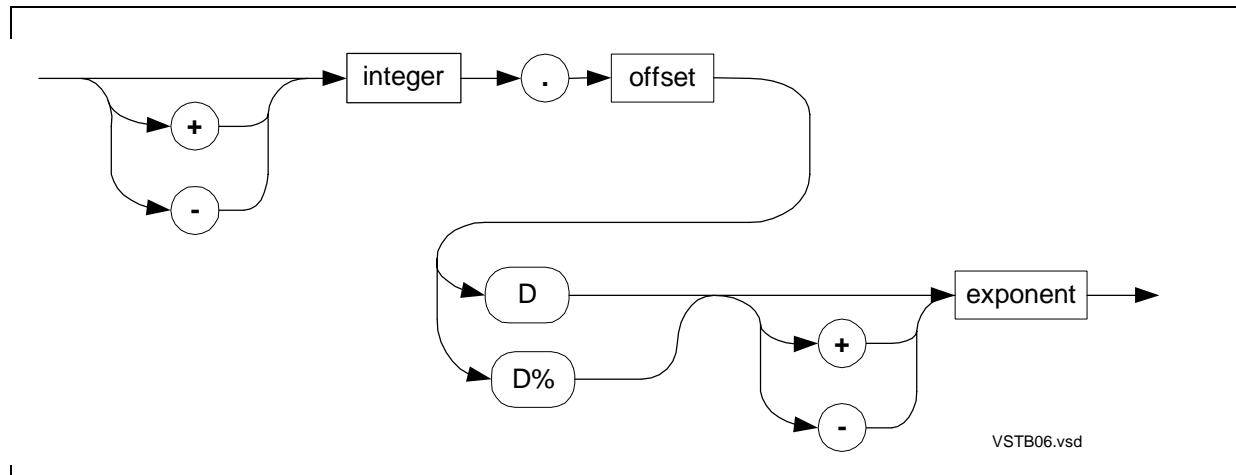
A FIXED numeric constant is a signed 64-bit fixed-point integer.



## REAL and REAL(64) Numeric Constants

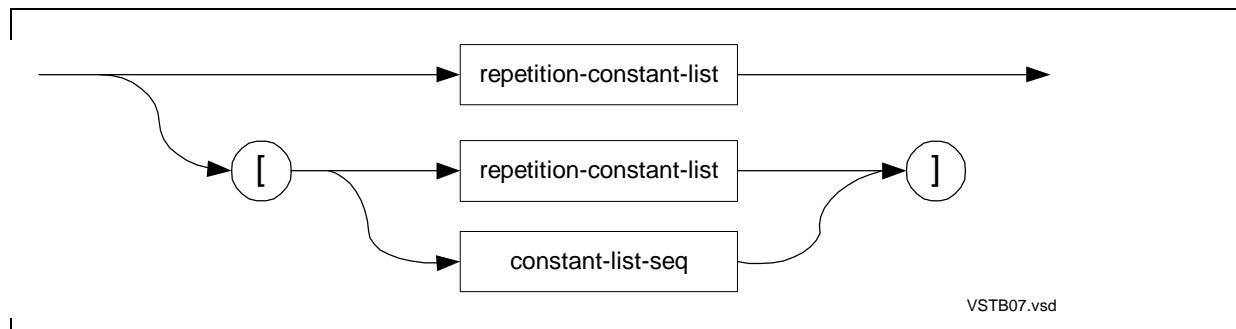
A REAL numeric constant is a signed 32-bit floating-point number that is precise to approximately 7 significant digits.

A REAL(64) numeric constant is a signed 64-bit floating-point number that is precise to approximately 17 significant digits.

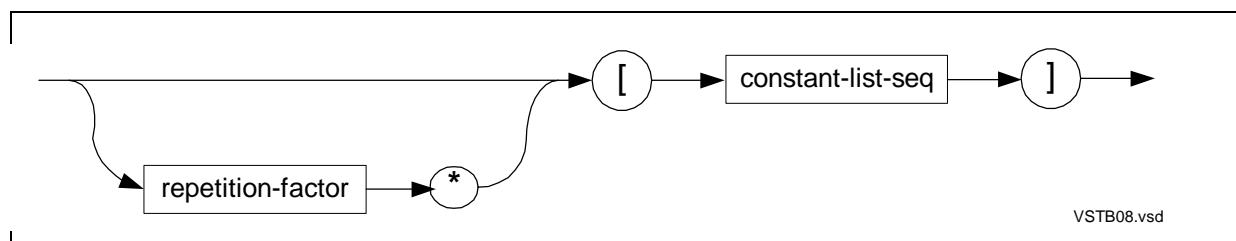


## Constant Lists

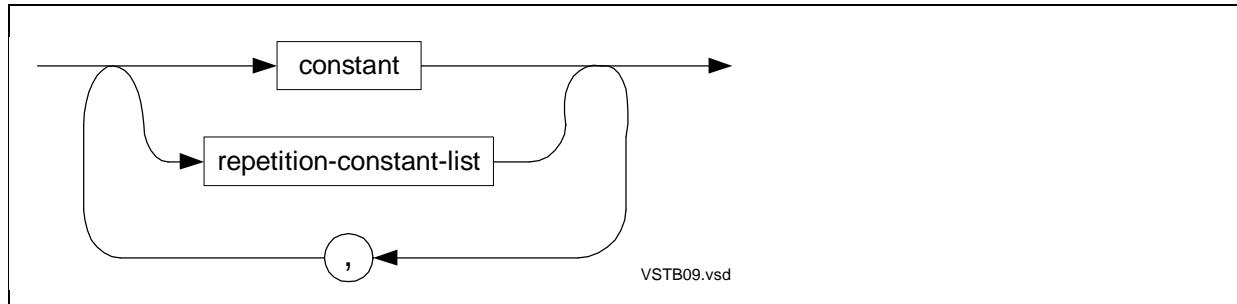
A constant list is a list of one or more constants.



`repetition-constant-list`



constant-list-seq



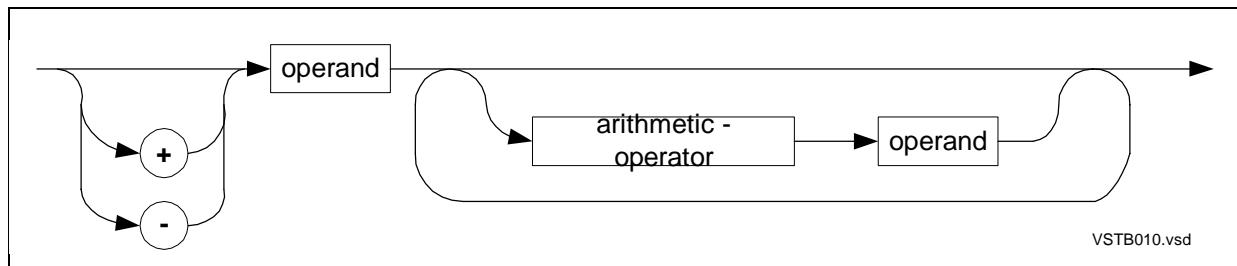
## Expressions

The following syntax diagrams describe:

- Arithmetic expressions
- Conditional expressions
- Assignment expressions
- CASE expressions
- IF expressions
- Group comparison expressions
- Bit extractions
- Bit shifts

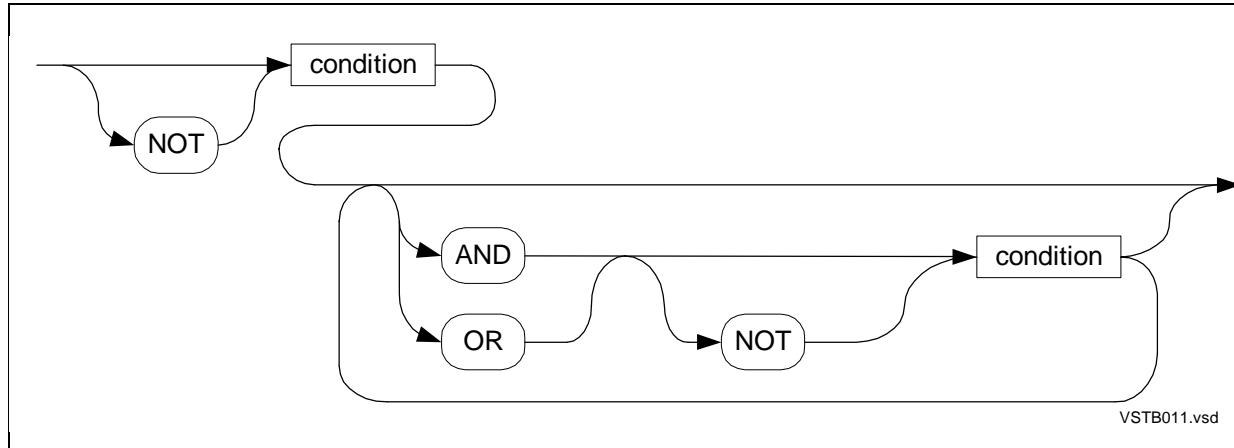
## Arithmetic Expressions

An arithmetic expression is a sequence of operands and arithmetic operators that computes a single numeric value of a specific data type.



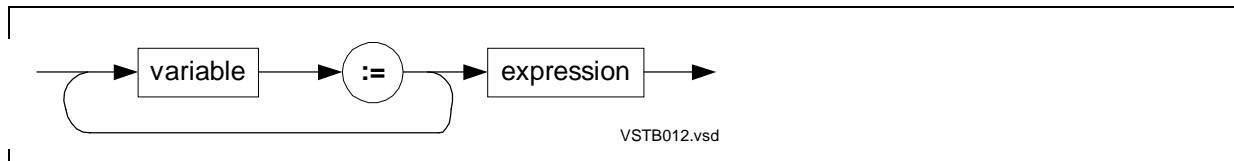
## Conditional Expressions

A conditional expression is a sequence of conditions and Boolean or relational operators that establishes the relationship between values.



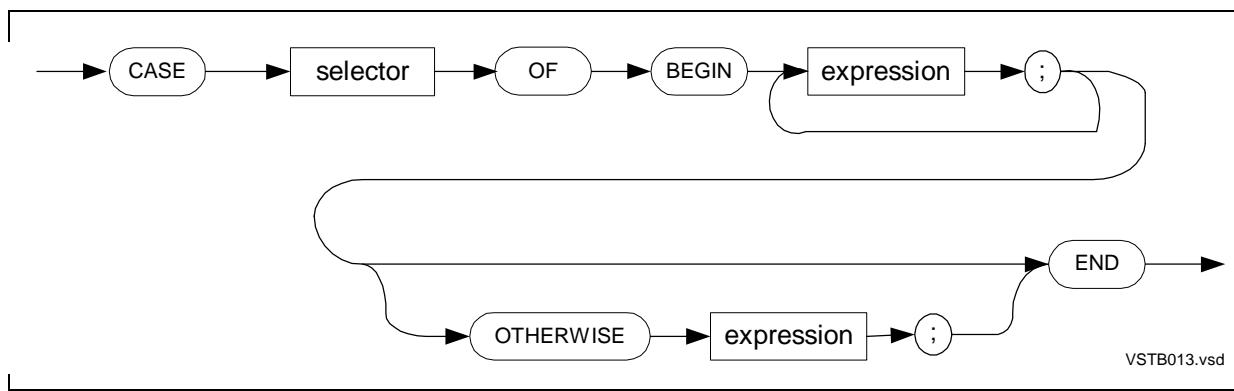
## Assignment Expressions

The assignment expression assigns the value of an expression to a variable.



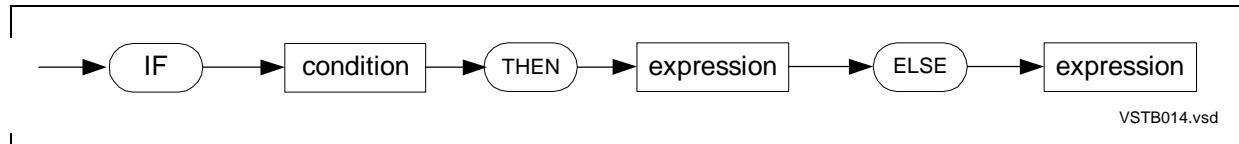
## CASE Expressions

The CASE expression selects one of several expressions.



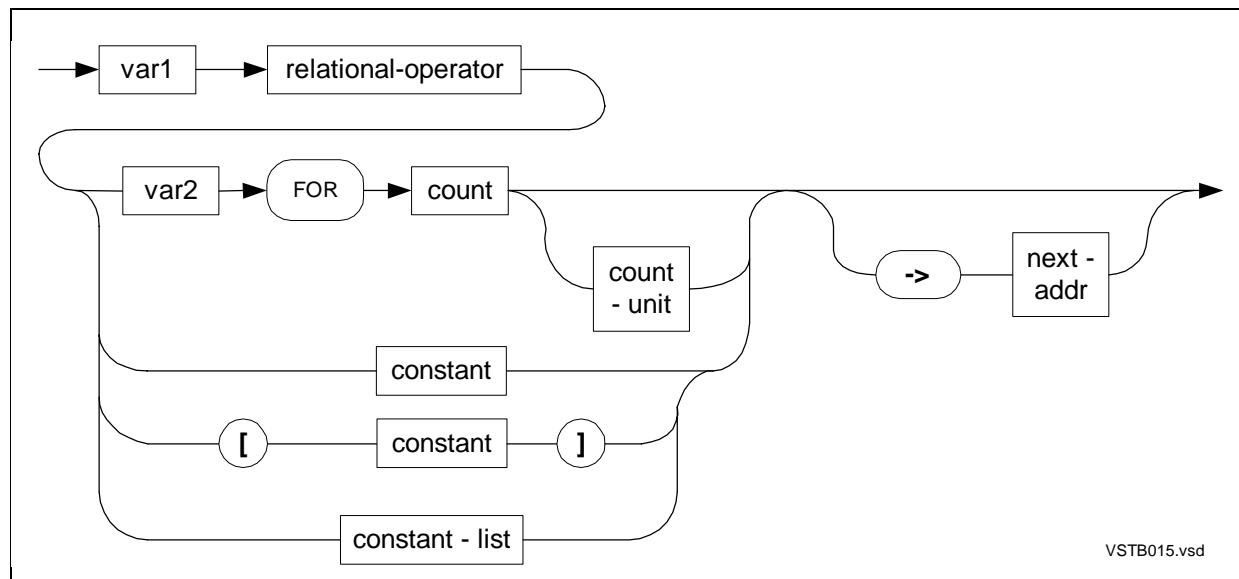
## IF Expressions

The IF expression conditionally selects one of two expressions, usually for assignment to a variable.



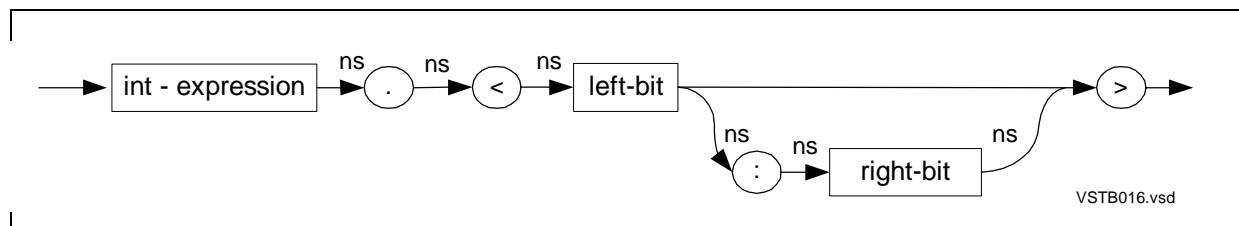
## Group Comparison Expressions

The group comparison expression compares a variable with a variable or a constant.



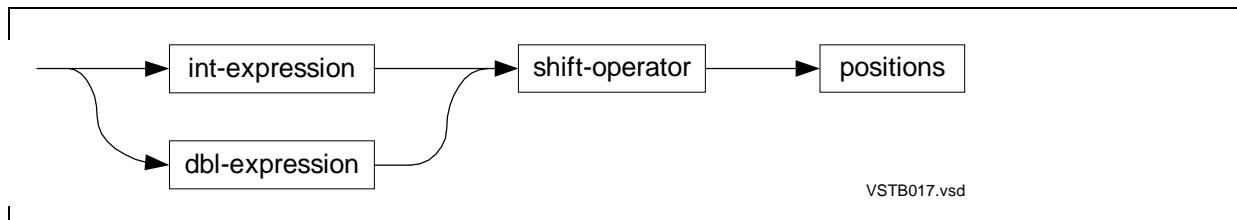
## Bit Extractions

A bit extraction accesses a bit field in an INT expression without altering the expression.



## Bit Shifts

A bit shift operation shifts a bit field a specified number of positions to the left or to the right within a variable without altering the variable.



## Declarations

Declaration syntax diagrams describe:

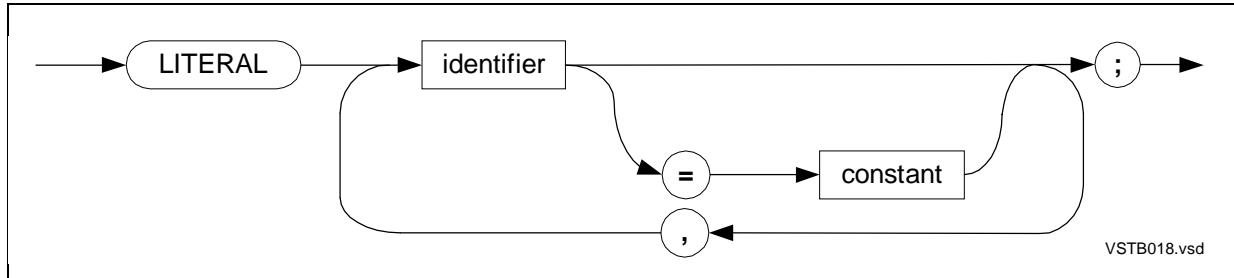
- LITERALs and DEFINEs
- Simple variables
- Arrays and read-only arrays
- Structures—definition structures, template structures, referral structures
- Structure items—simple variables, arrays, substructures, filler bytes, filler bits,
- simple pointers, structure pointers, and redefinitions
- Simple pointers and structure pointers
- Equivalenced variables
- NAMEs and BLOCKs
- Procedures and subprocedures

## LITERAL and DEFINE Declarations

The following syntax diagrams describe LITERAL and DEFINE declarations.

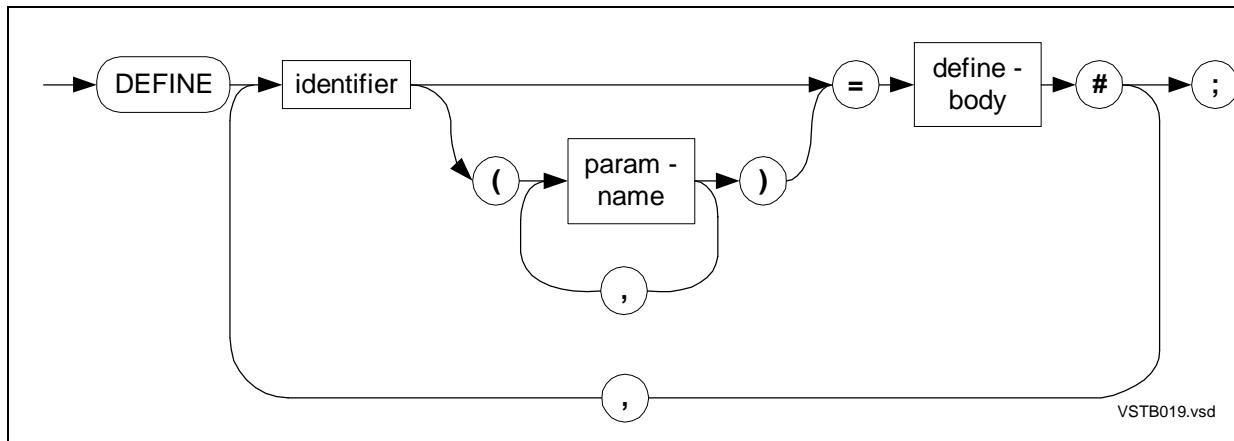
### LITERALs

A LITERAL associates an identifier with a constant.



## DEFINEs

A DEFINE associates an identifier (and parameters if any) with text.

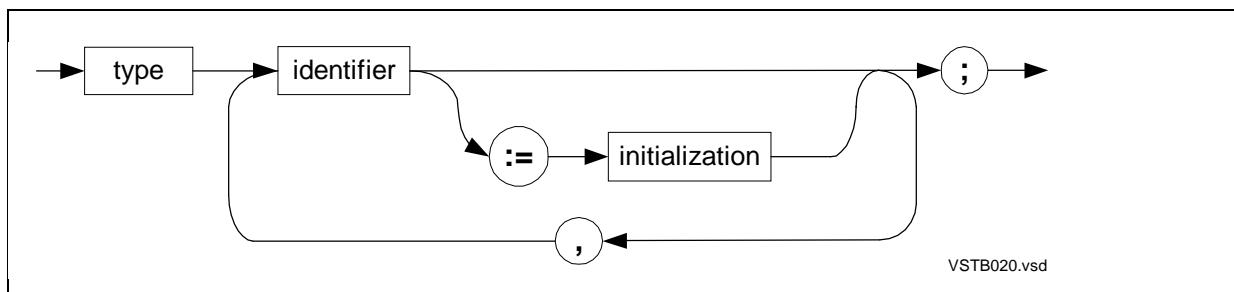


## Simple Variable Declarations

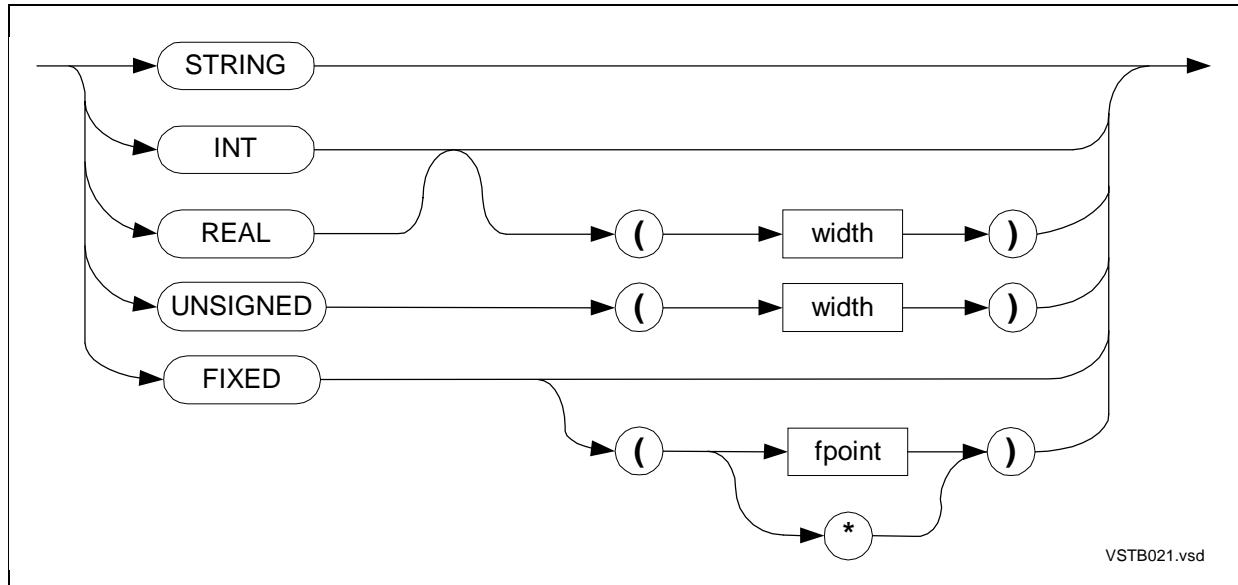
The following syntax diagram describes simple variable declarations:

### Simple Variables

A simple variable associates an identifier with a single-element data item of a specified data type.



type

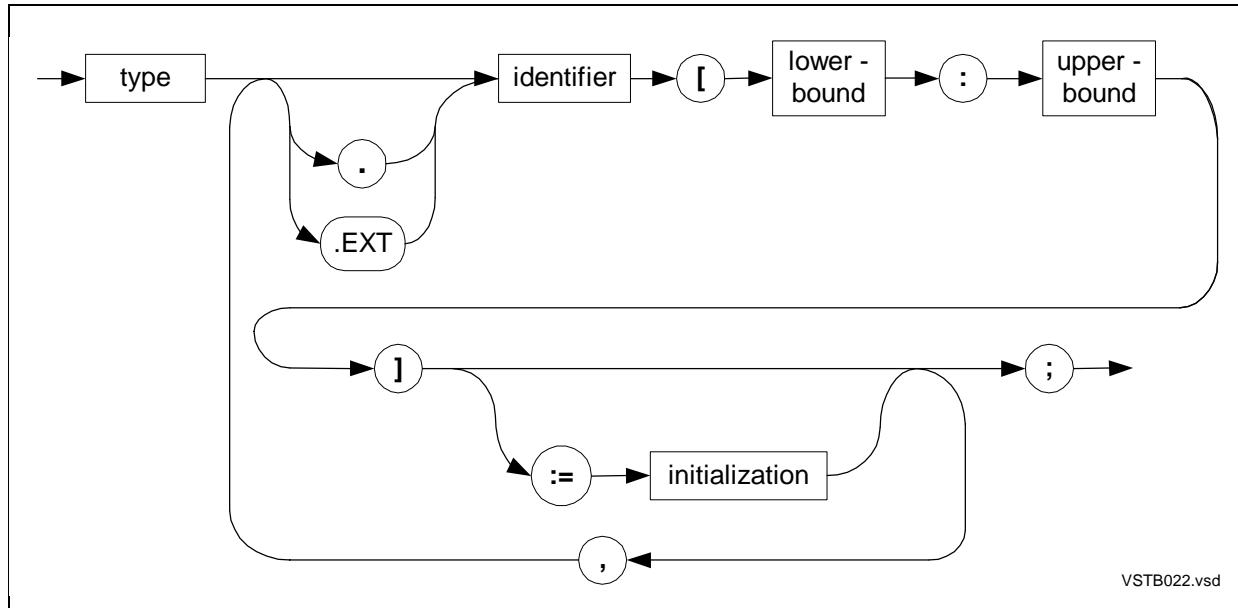


## Array Declarations

The following syntax diagrams describe array and read-only array declarations:

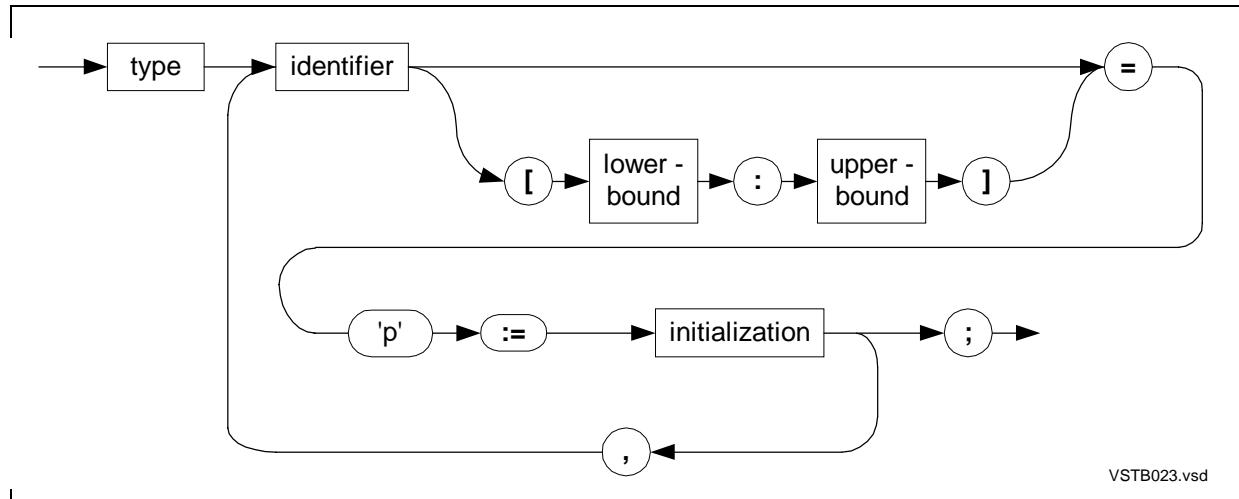
### Arrays

An array associates an identifier with a one-dimensional set of elements of the same data type.



## Read-Only Arrays

A read-only array associates an identifier with a one-dimensional and nonmodifiable set of elements of the same data type. A read-only array is located in a user code segment.



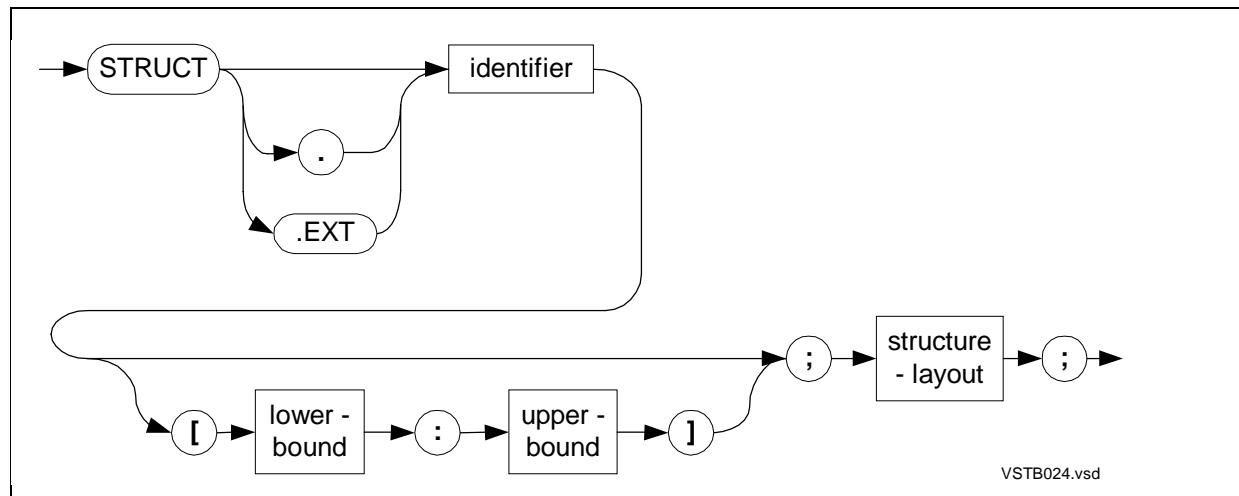
## Structure Declarations

The following syntax diagrams describe:

- Structures—definition structures, template structures, referral structures
- Structure items—simple variables, arrays, substructures, filler bytes, filler bits, simple pointers, structure pointers, and redefinitions

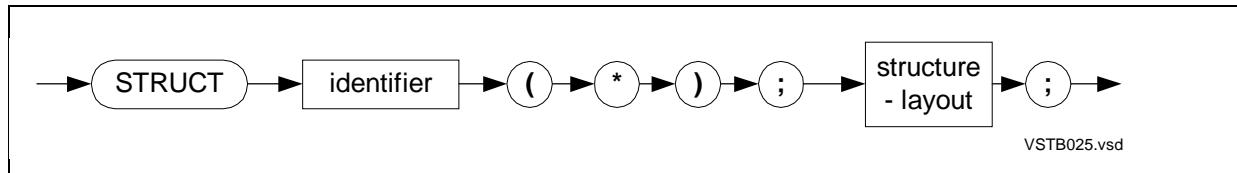
## Definition Structures

A definition structure associates an identifier with a structure layout and allocates storage for it.



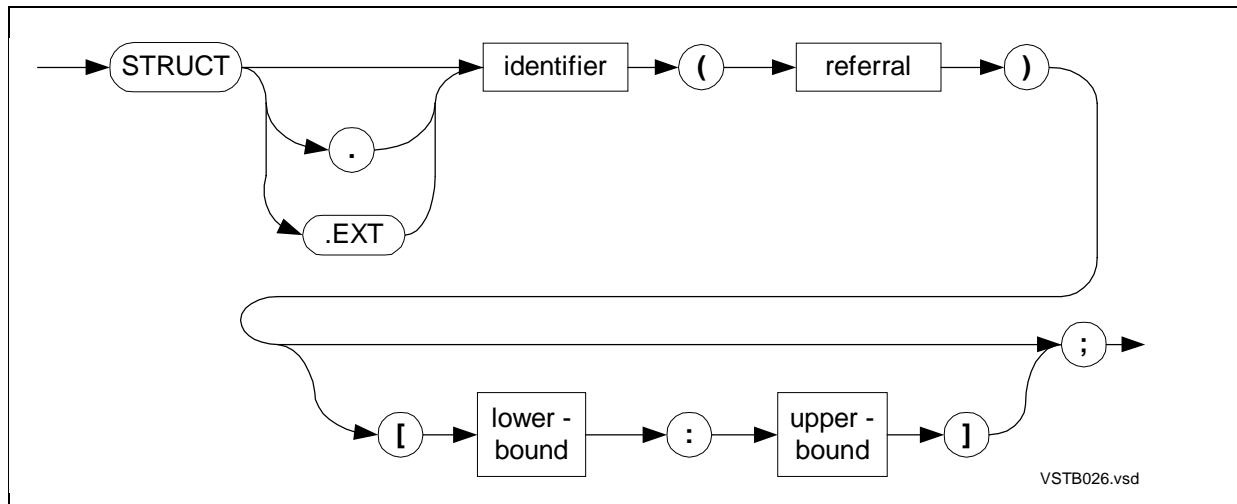
## Template Structures

A template structure associates an identifier with a structure layout but allocates no storage for it.



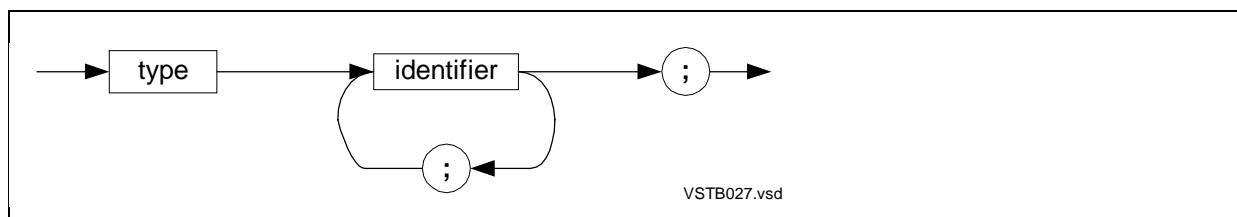
## Referral Structures

A referral structure associates an identifier with a structure whose layout is the same as a previously declared structure and allocates storage for it.



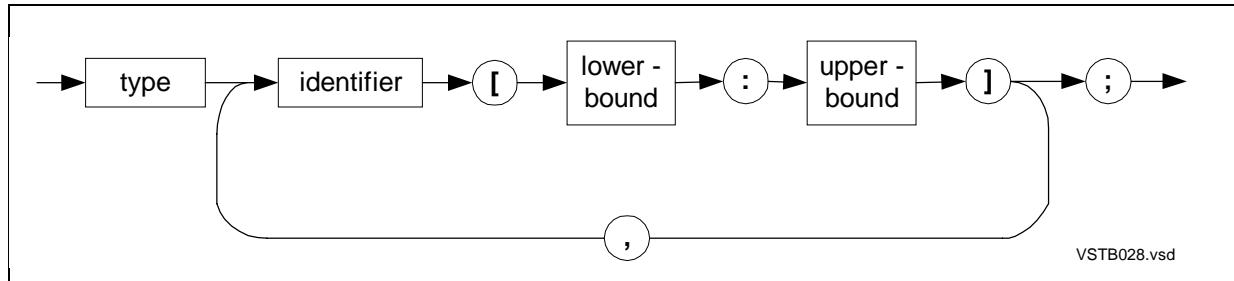
## Simple Variables Declared in Structures

A simple variable can be declared inside a structure.



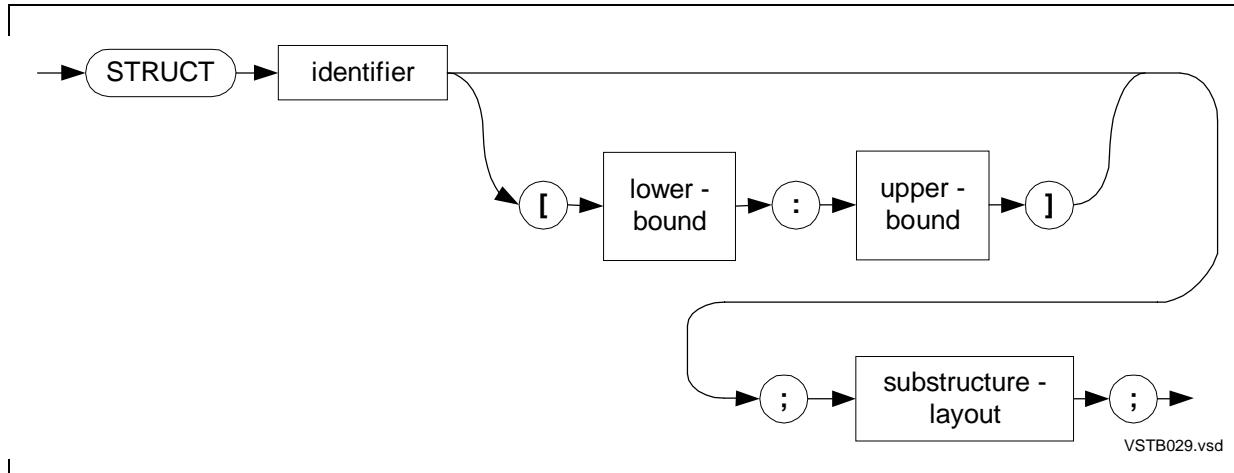
## Arrays Declared in Structures

An array can be declared inside a structure.



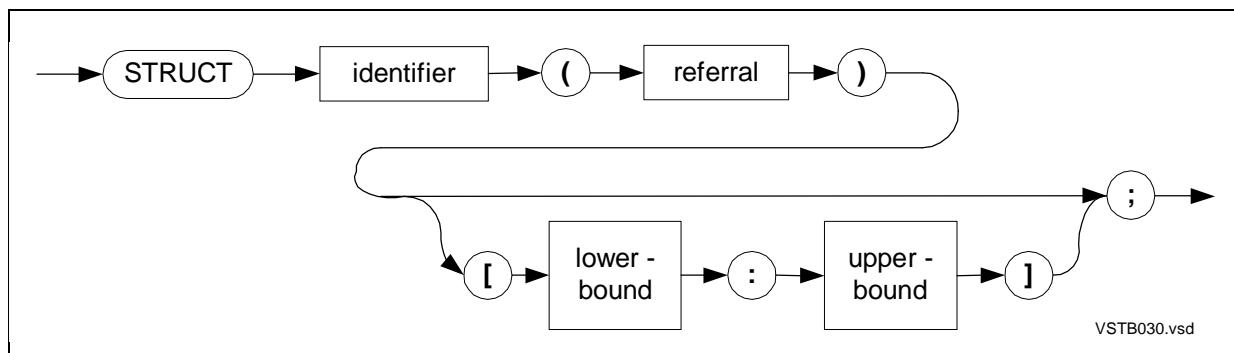
## Definition Substructures

A definition substructure can be declared inside a structure.



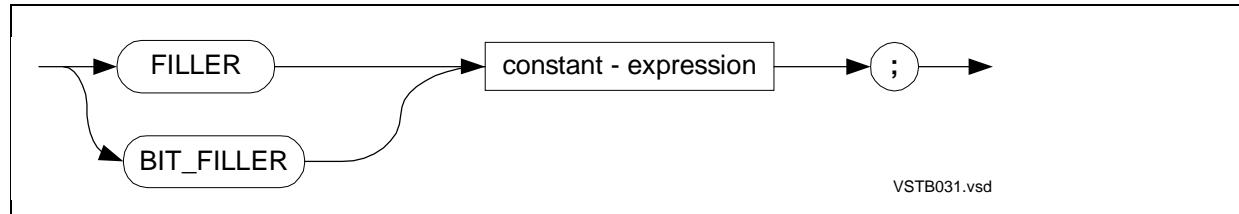
## Referral Substructures

A referral substructure can be declared inside a structure.



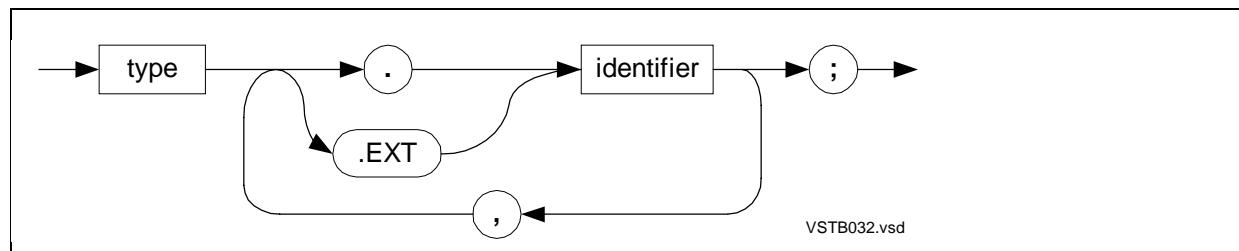
## Fillers in Structures

A filler is a byte or bit place holder in a structure.



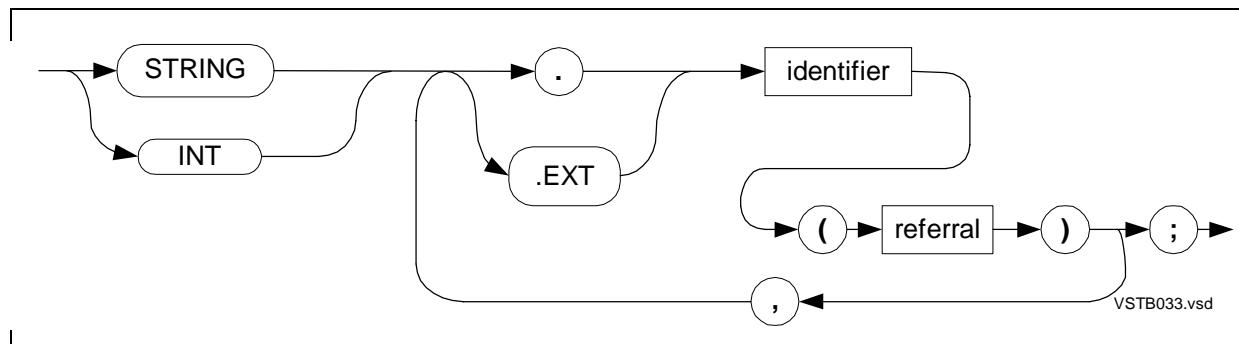
## Simple Pointers Declared in Structures

A simple pointer can be declared inside a structure.



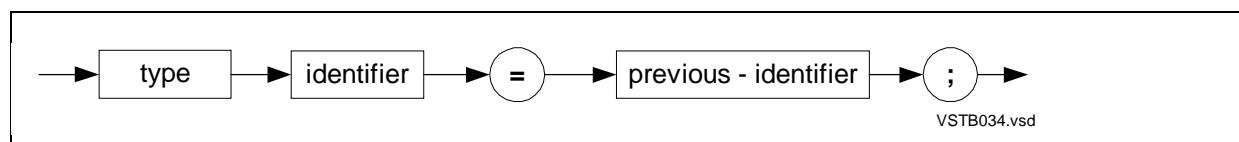
## Structure Pointers Declared in Structures

A structure pointer can be declared inside a structure.



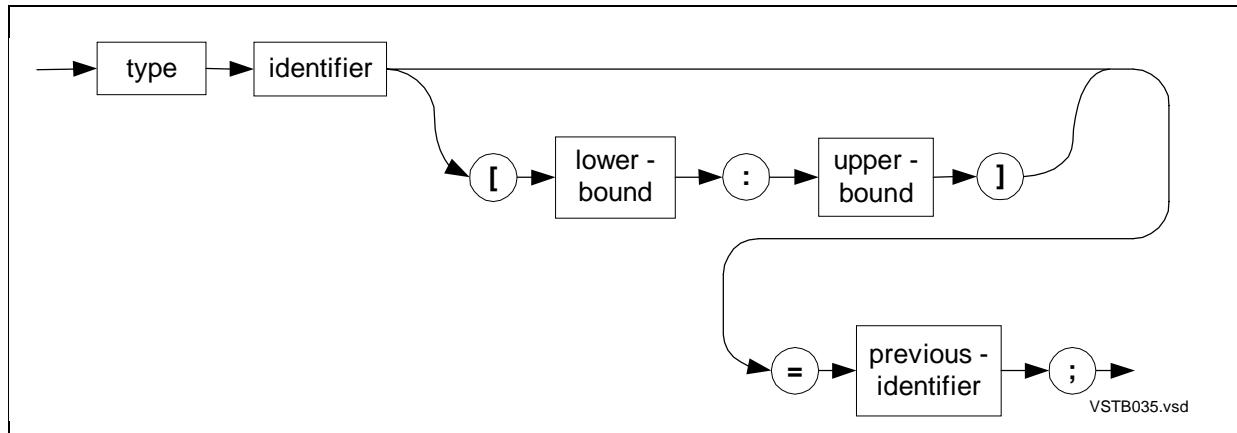
## Simple Variable Redefinitions

A simple variable redefinition associates a new simple variable with a previous item at the same BEGIN-END level of a structure.



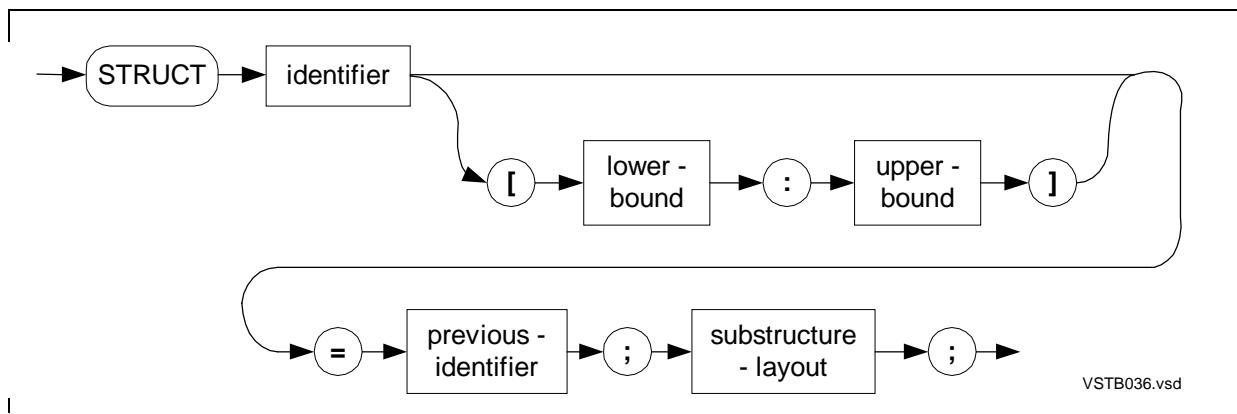
## Array Redefinitions

An array redefinition associates a new array with a previous item at the same BEGIN-END level of a structure.



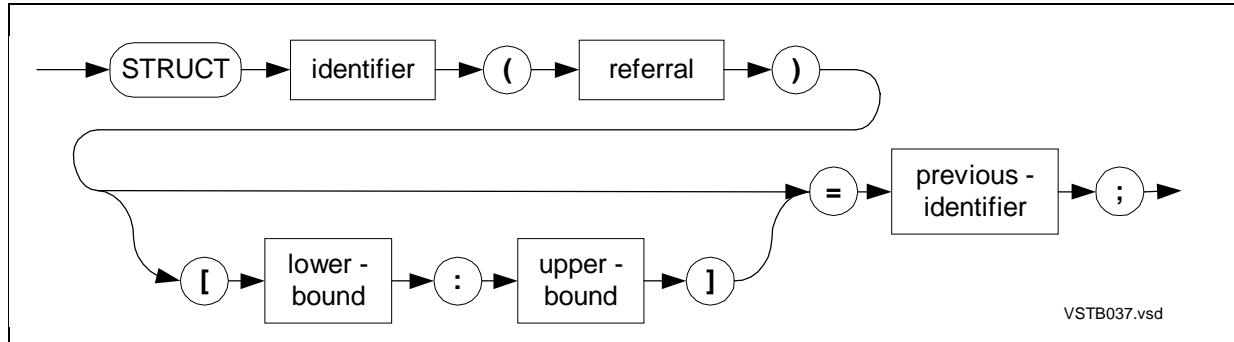
## Definition Substructure Redefinitions

A definition substructure redefinition associates a definition substructure with a previous item at the same BEGIN-END level of a structure.



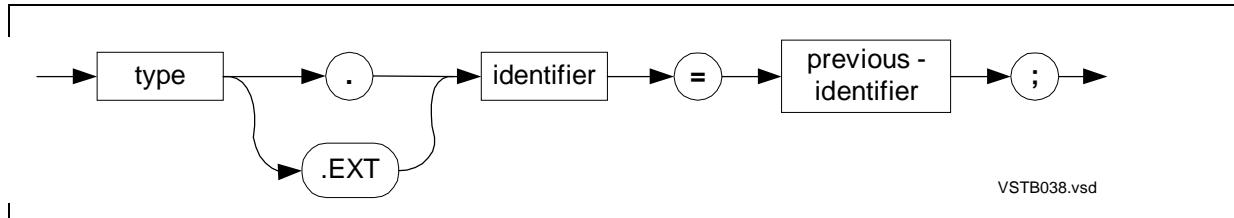
## Referral Substructure Redefinitions

A referral substructure redefinition associates a referral substructure with a previous item at the same BEGIN-END level of a structure.



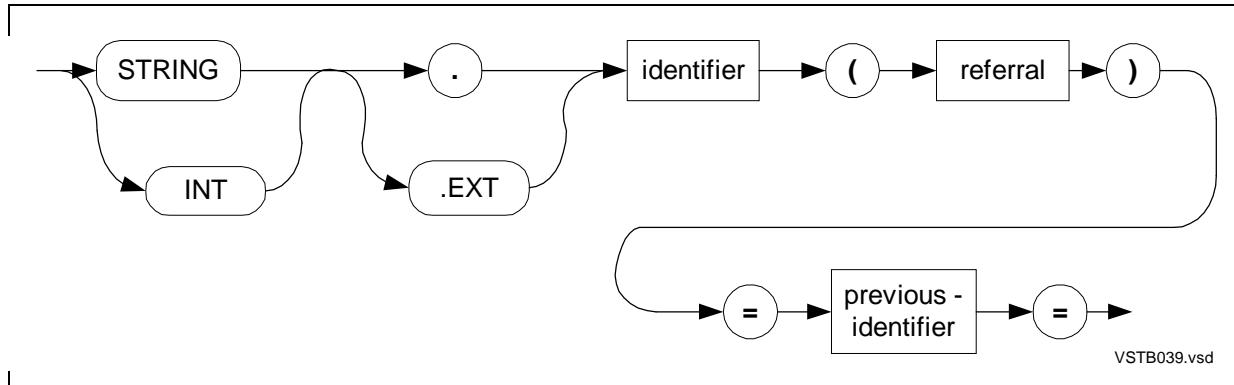
## Simple Pointer Redefinitions

A simple pointer redefinition associates a new simple pointer with a previous item at the same BEGIN-END level of a structure.



## Structure Pointer Redefinitions

A structure pointer redefinition associates a structure pointer with a previous item at the same BEGIN-END level of a structure.

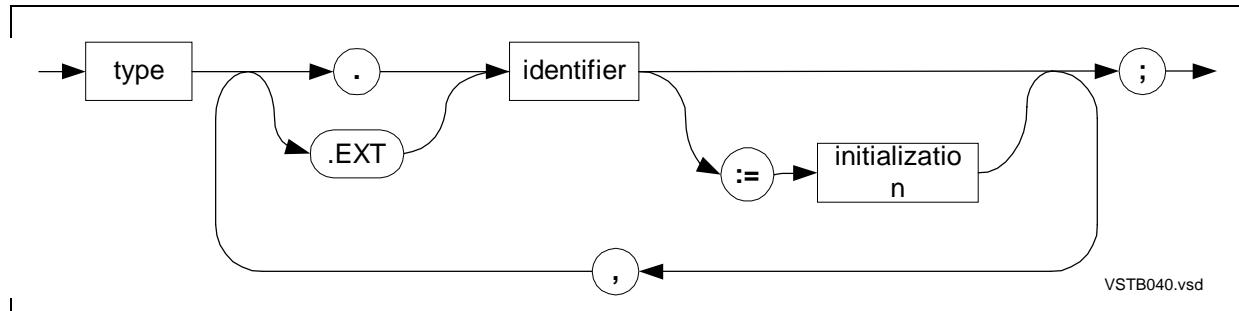


## Pointer Declarations

The following syntax diagrams describe simple pointer and structure pointer declarations.

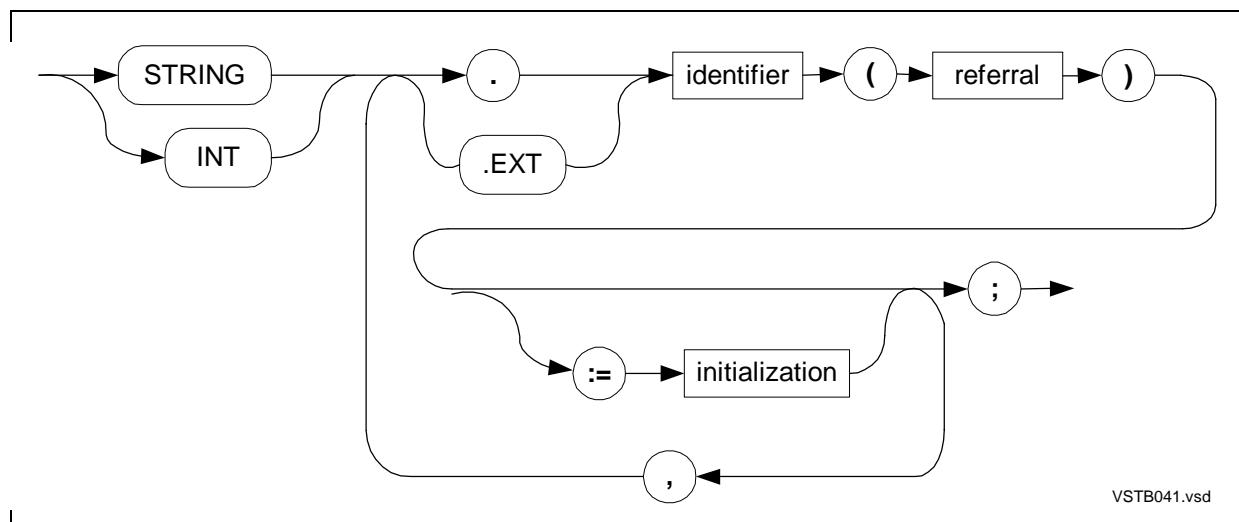
## Simple Pointers

A simple pointer is a variable you load with a memory address, usually of a simple variable or array, which you access with this simple pointer.



## Structure Pointers

A structure pointer is a variable you load with a memory address of a structure, which you access with this structure pointer.

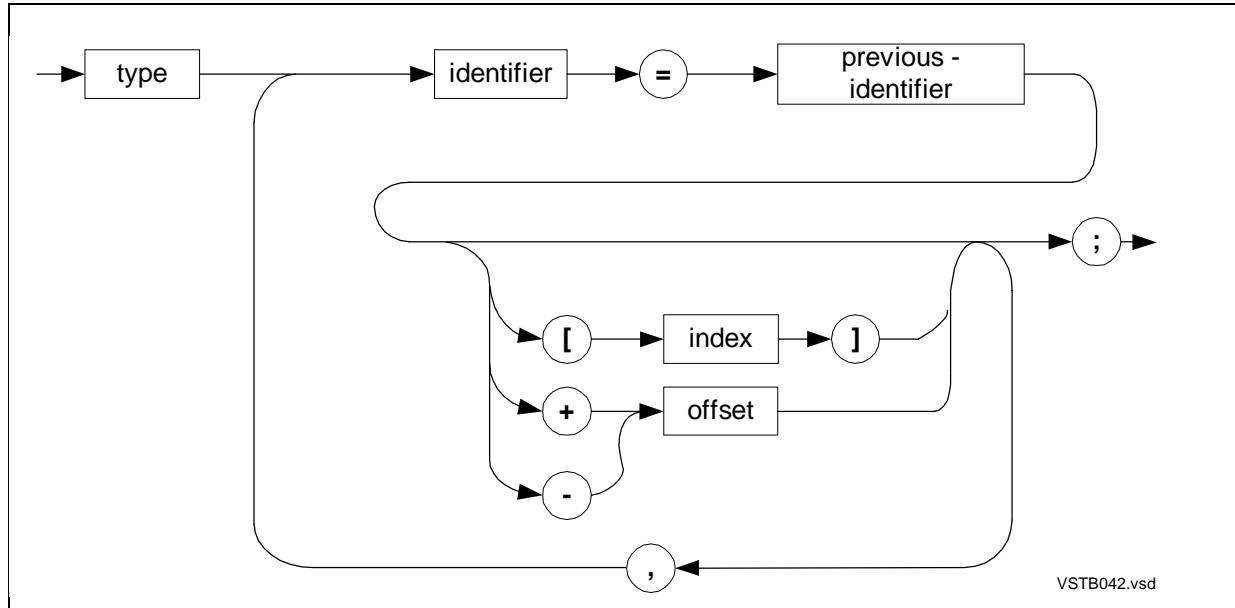


## Equivalenced Variable Declarations

The following syntax diagrams describe equivalenced variable declarations for simple variables, simple pointers, structures, and structure pointers.

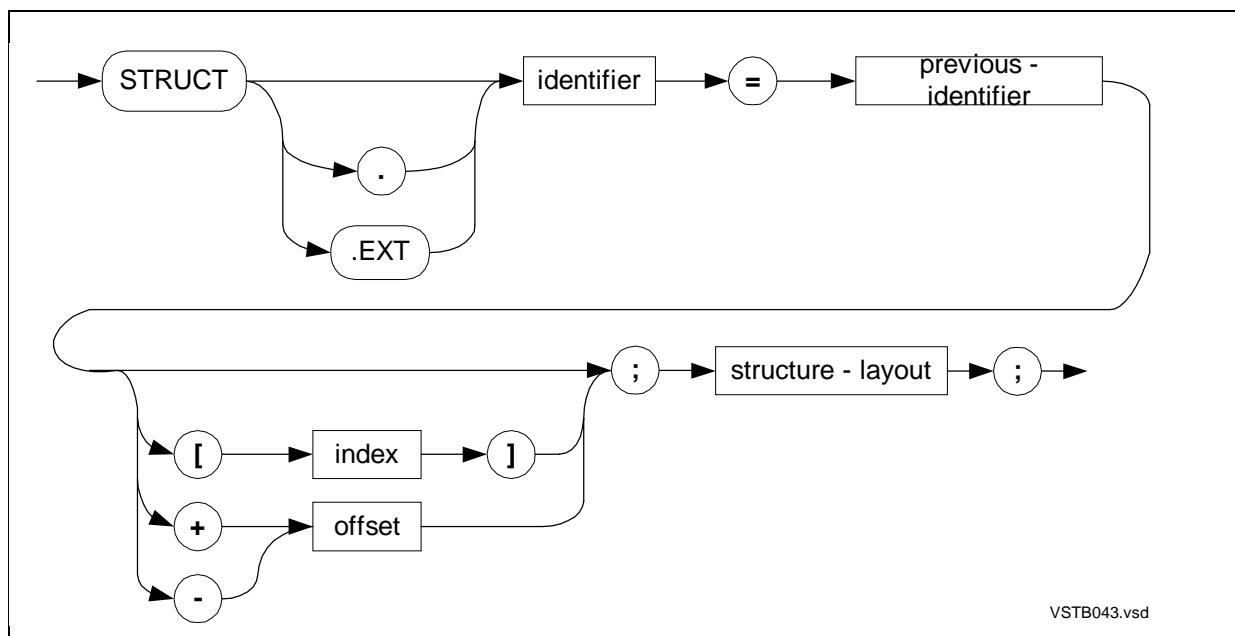
## Equivalenced Simple Variables

An equivalenced simple variable associates a new simple variable with a previously declared variable.



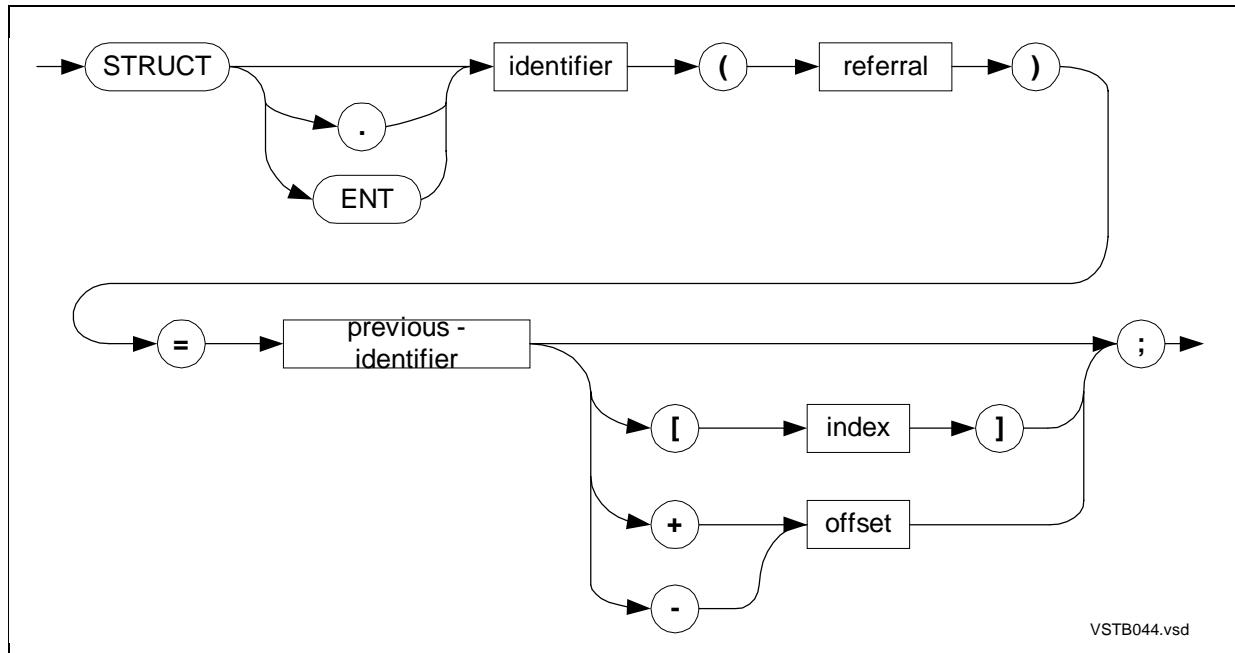
## Equivalenced Definition Structures

An equivalenced definition structure associates a new definition structure with a previously declared variable.



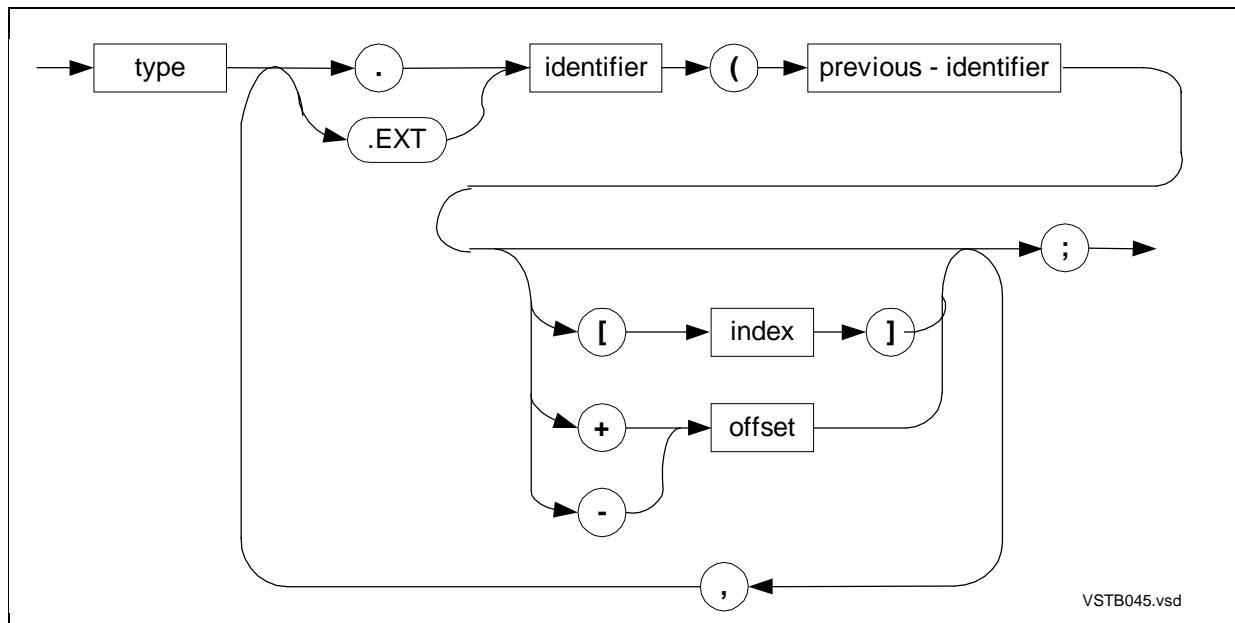
## Equivalenced Referral Structures

An equivalenced referral structure associates a new referral structure with a previously declared variable.



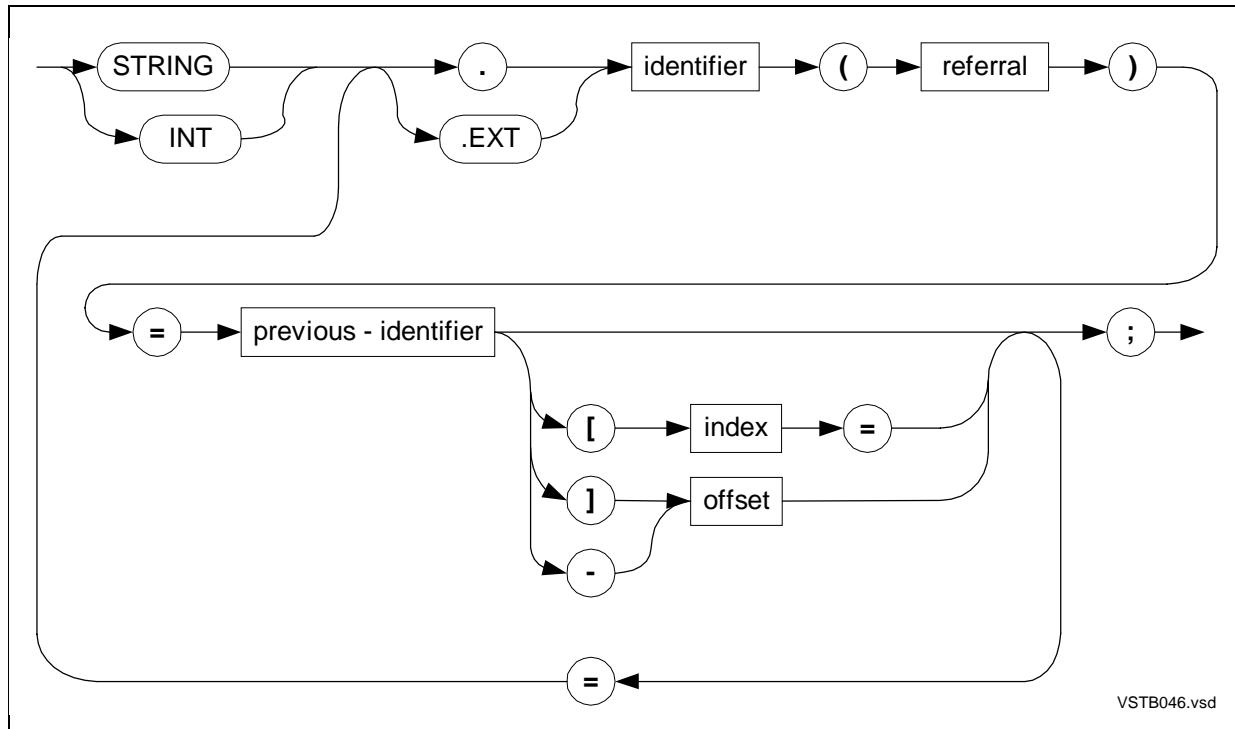
## Equivalenced Simple Pointers

An equivalenced simple pointer associates a new simple pointer with a previously declared variable.



## Equivalenced Structure Pointers

An equivalenced structure pointer associates a new structure pointer with a previously declared variable.

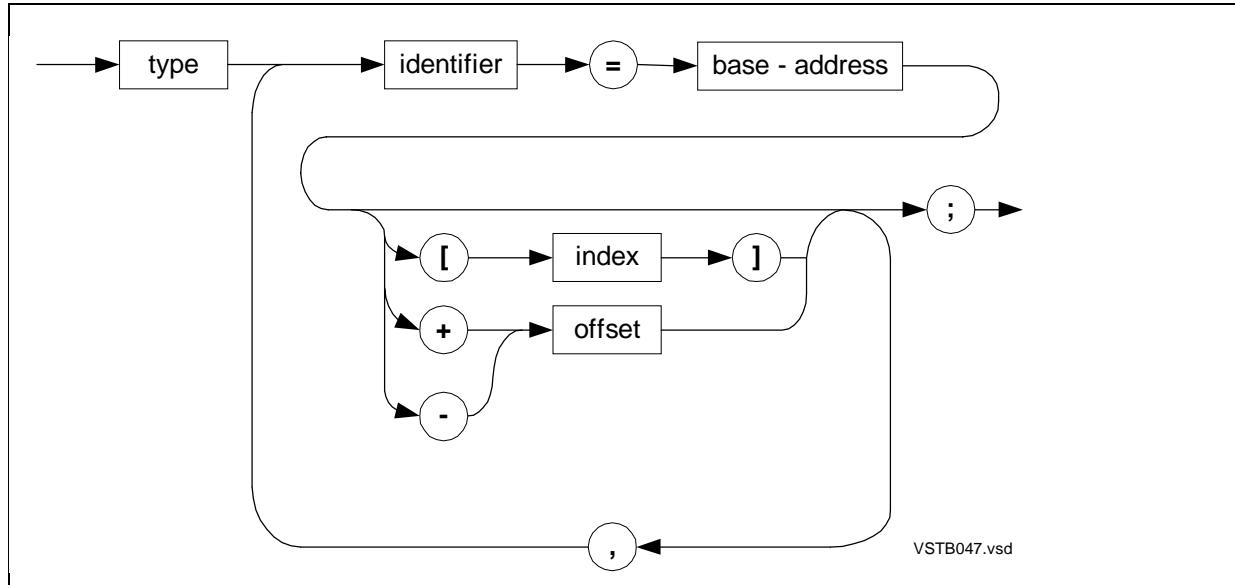


## Base-Address Equivalenced Variable Declarations

The following syntax diagrams describe base-addressed equivalenced variable declarations for simple variables, simple pointers, structures, and structure pointers.

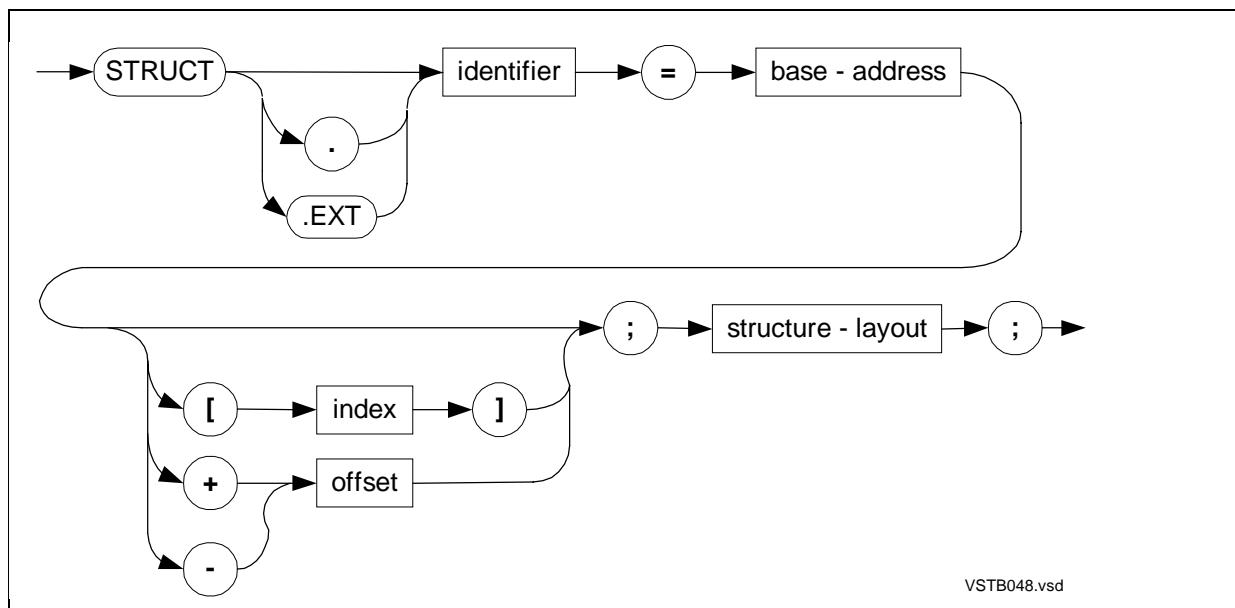
### Base-Address Equivalenced Simple Variables

A base-addressed equivalenced simple variable associates a simple variable with a global, local, or top-of-stack base address.



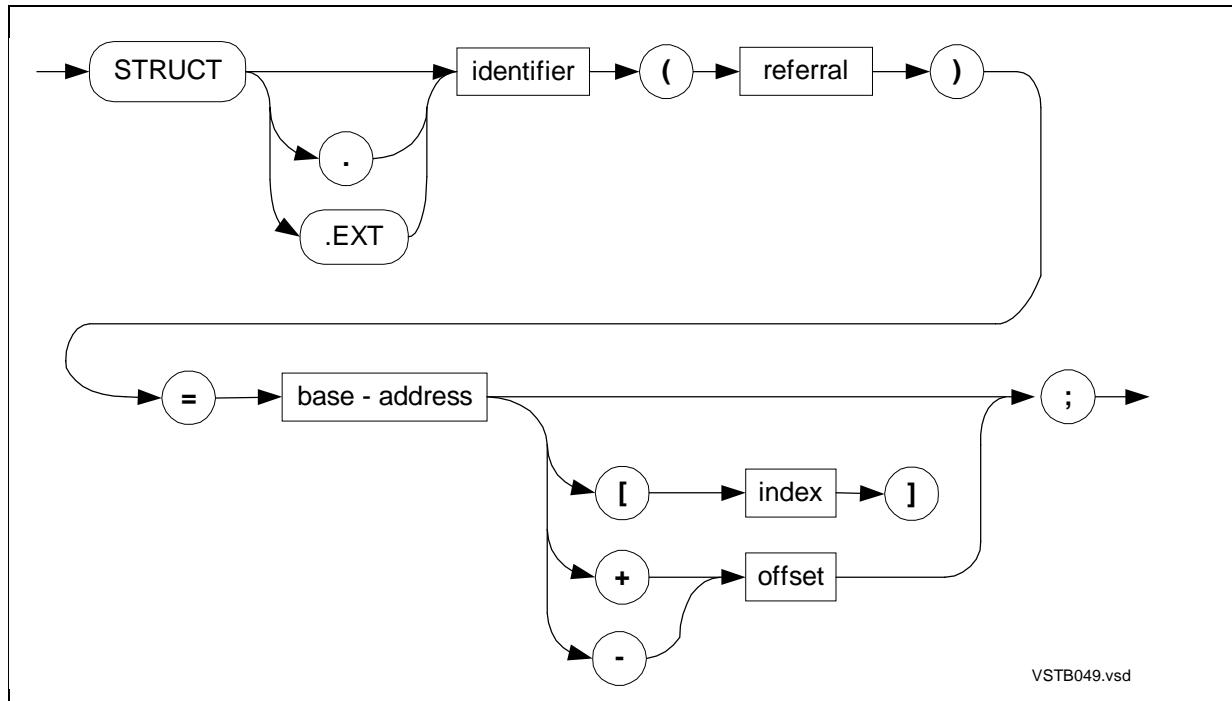
## Base-Address Equivalenced Definition Structures

A base-addressed equivalenced definition structure associates a definition structure with a global, local, or top-of-stack base address.



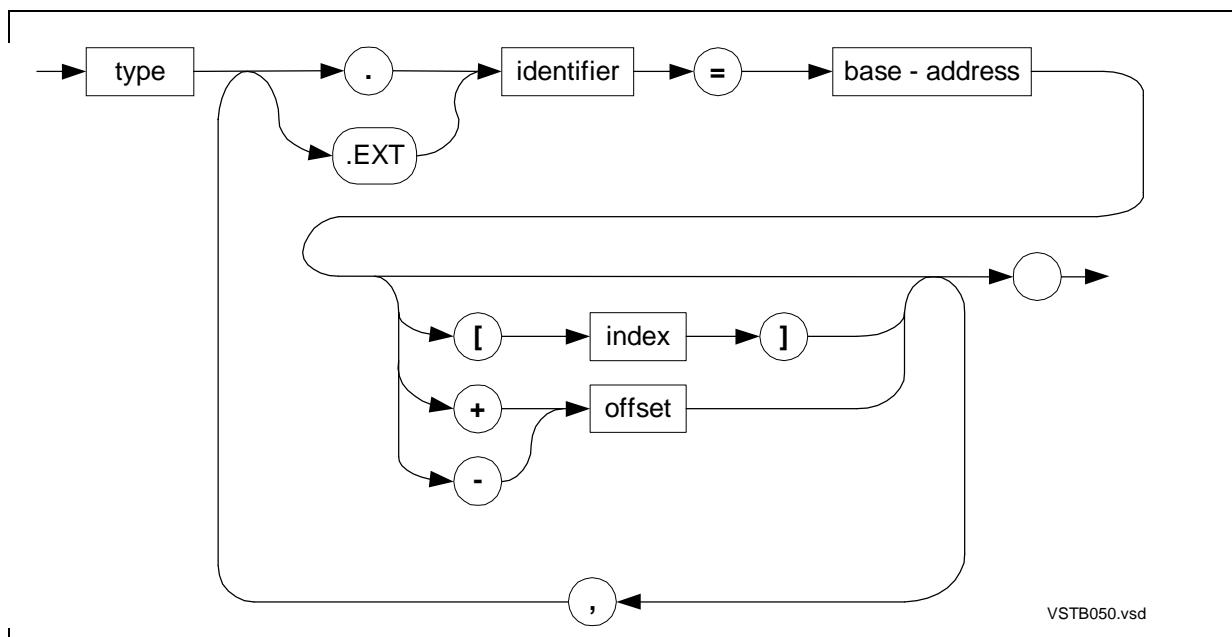
## Base-Address Equivalenced Referral Structures

A base-addressed equivalenced referral structure associates a referral structure with a global, local, or top-of-stack base address.



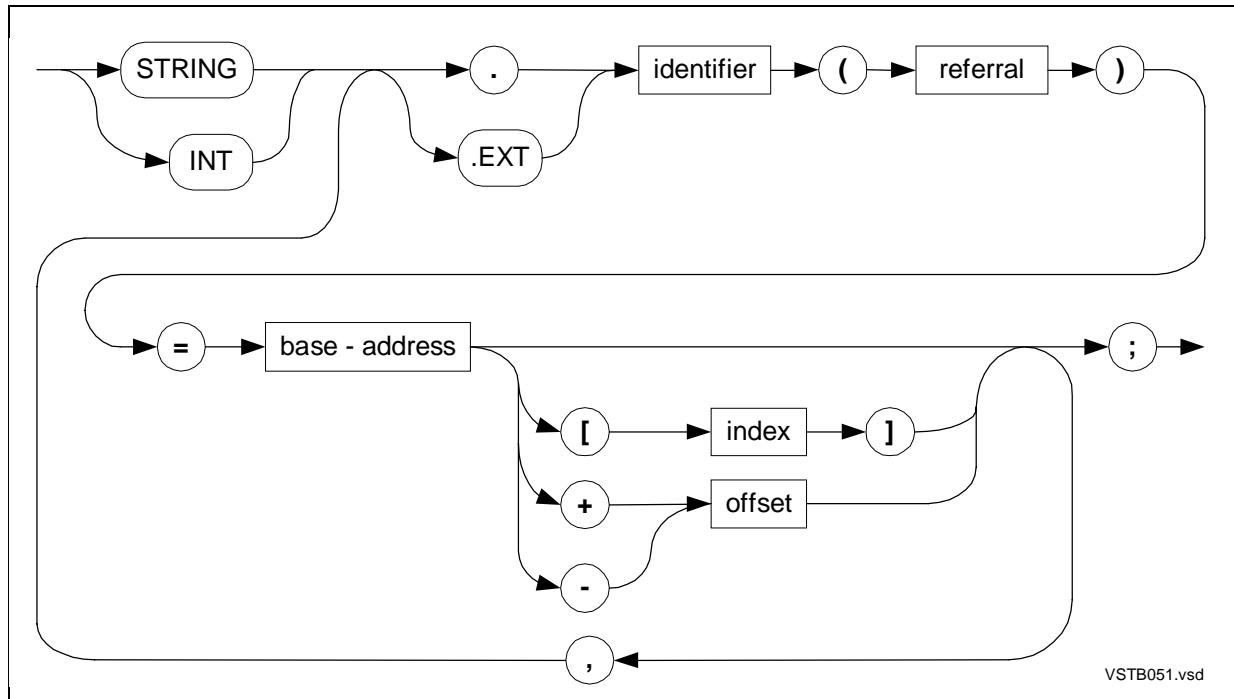
## Base-Address Equivalenced Simple Pointers

A base-addressed equivalenced simple pointer associates a simple pointer with a global, local, or top-of-stack base address.



## Base-Address Equivalenced Structure Pointers

A base-addressed equivalenced structure pointer associates a structure pointer with a global, local, or top-of-stack base address.

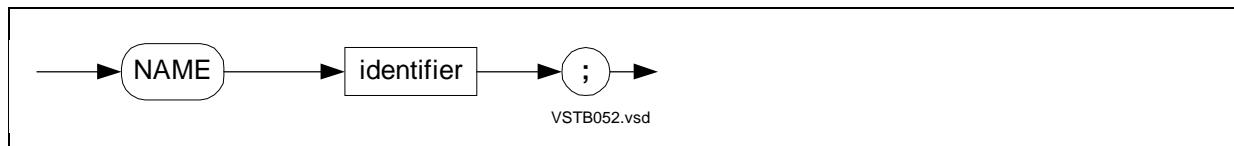


## NAME and BLOCK Declarations

The following syntax diagrams describe NAME and BLOCK declarations.

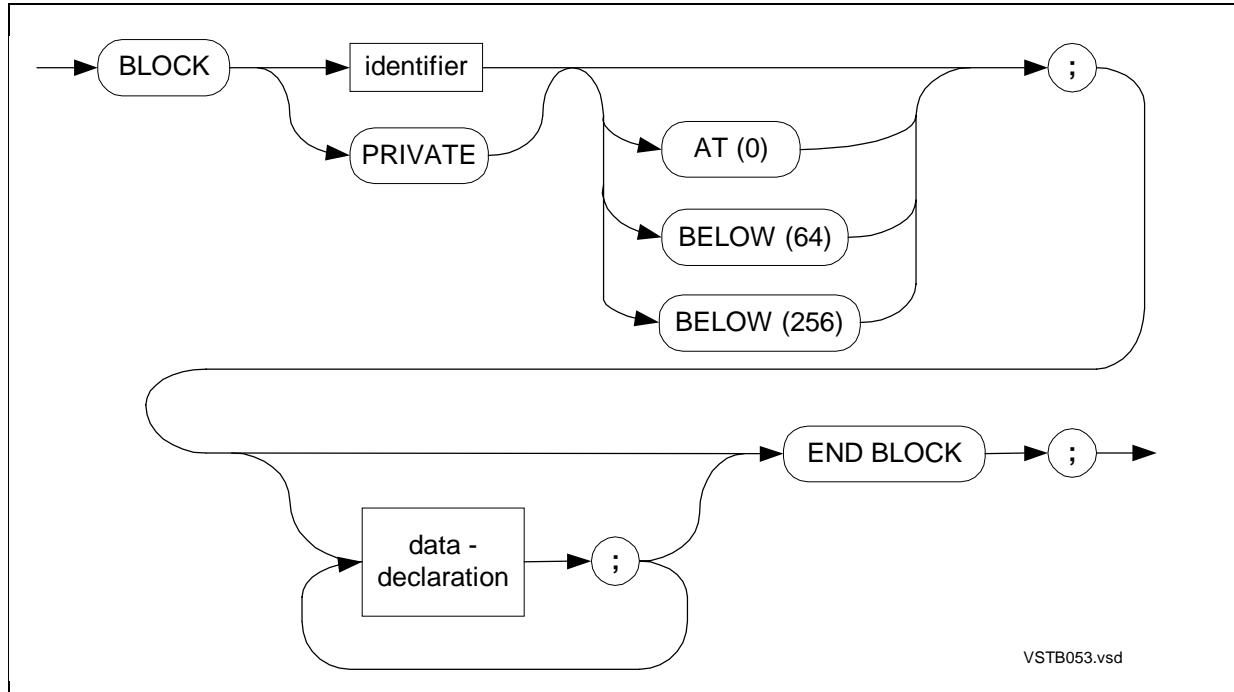
### NAMEs

The NAME declaration assigns an identifier to a compilation unit and to its private global data block if it has one.



### BLOCKs

The BLOCK declaration groups global data declarations into a named or private relocatable global data block.

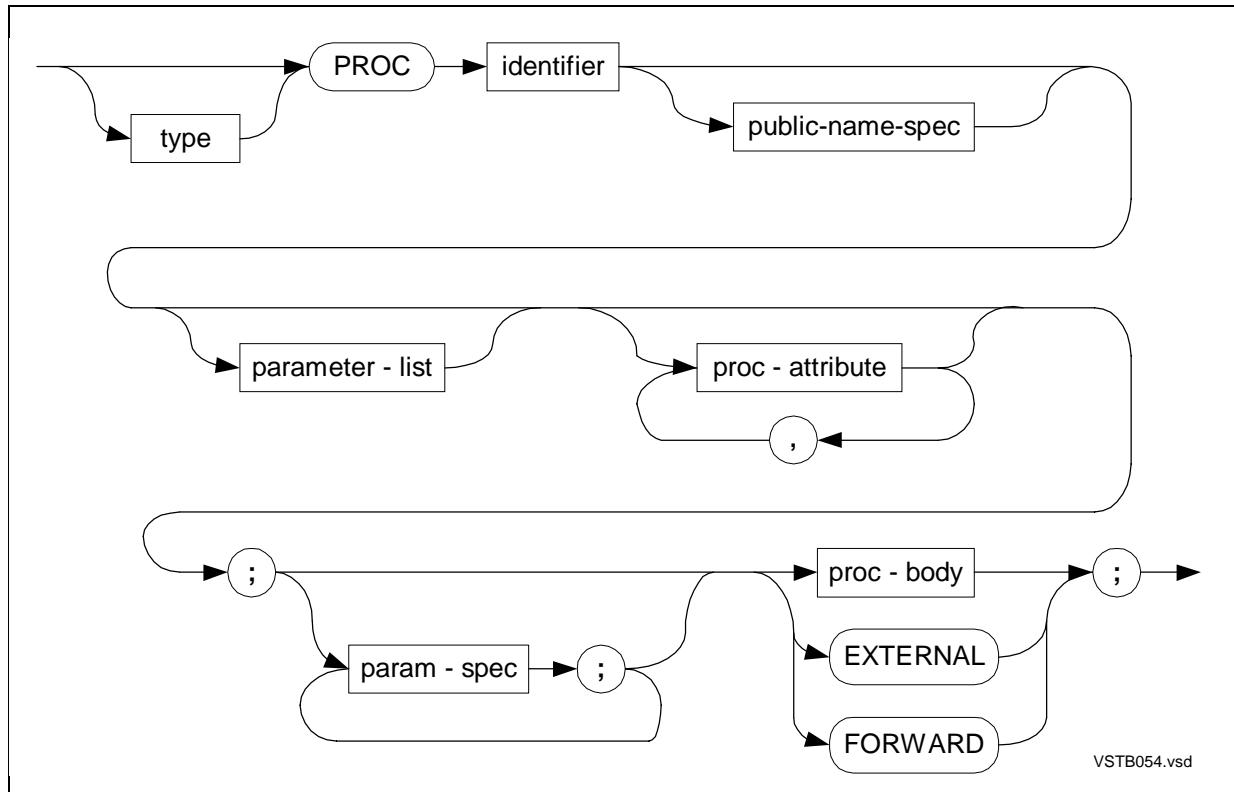


## Procedure and Subprocedure Declarations

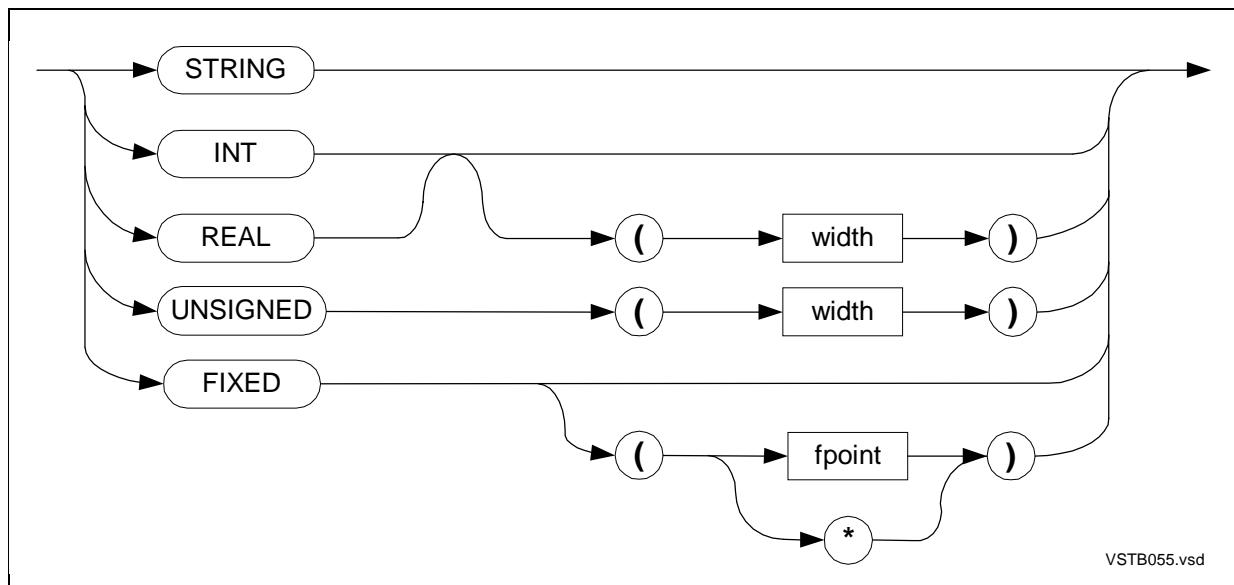
The following syntax diagrams describe procedure, subprocedure, entry-point, and label declarations.

### Procedures

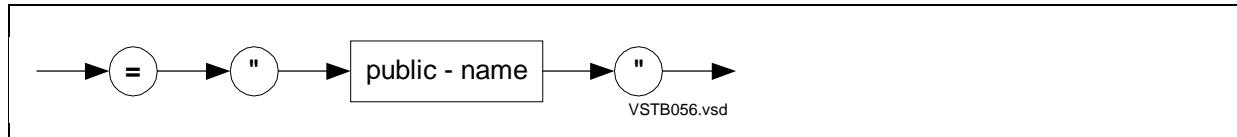
A procedure is a program unit that is callable from anywhere in the program.



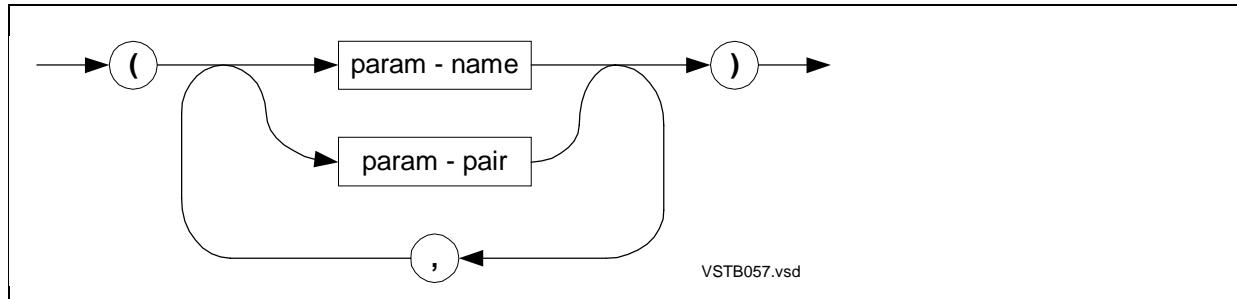
type



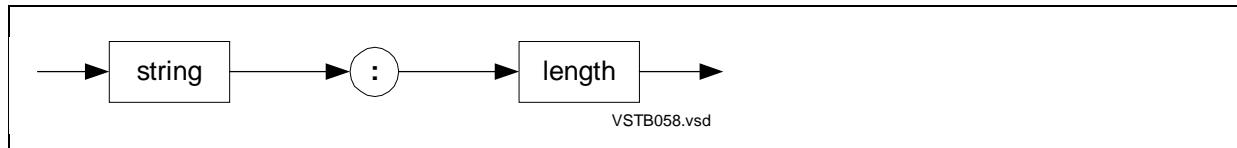
public-name-spec



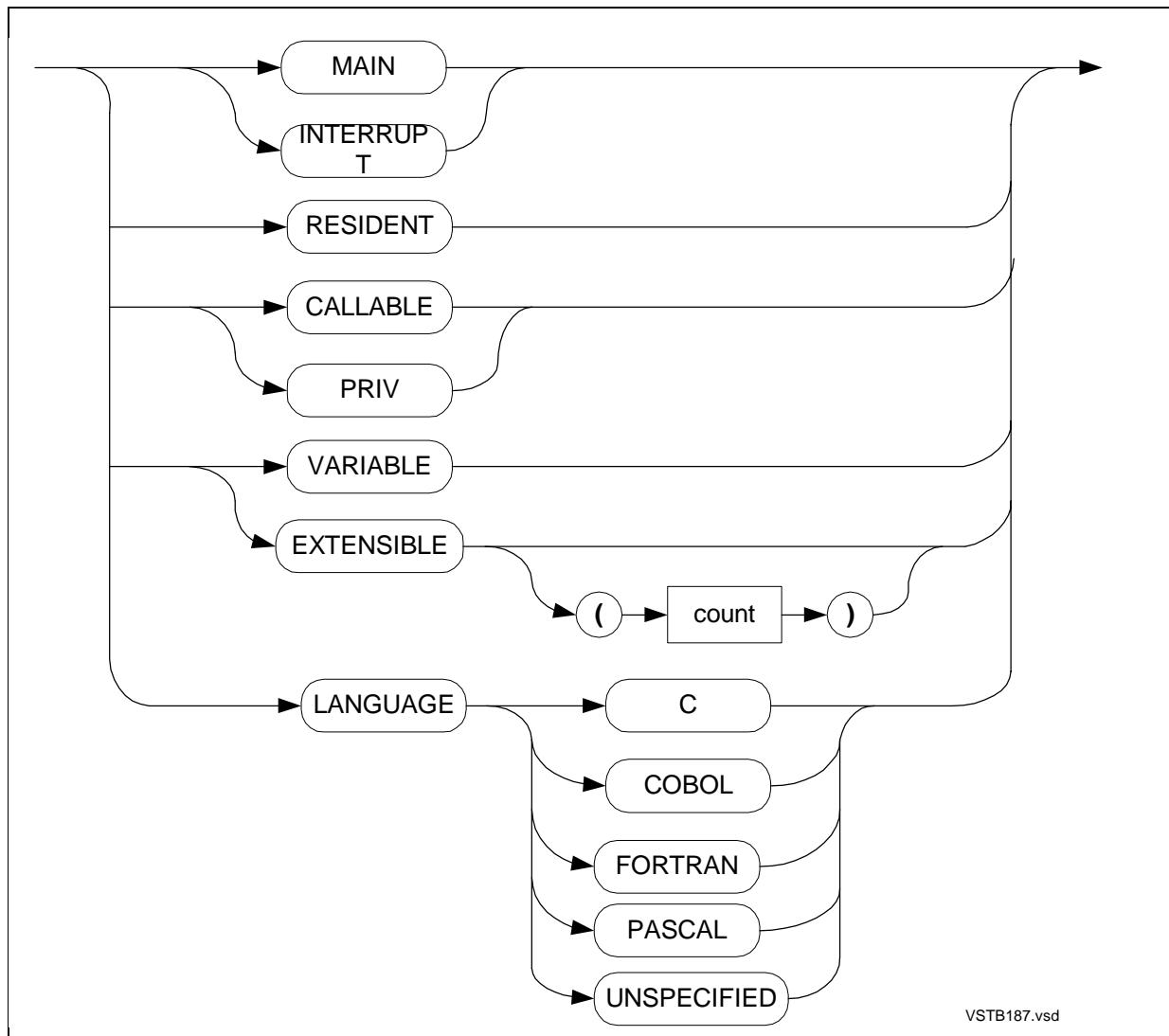
parameter-list



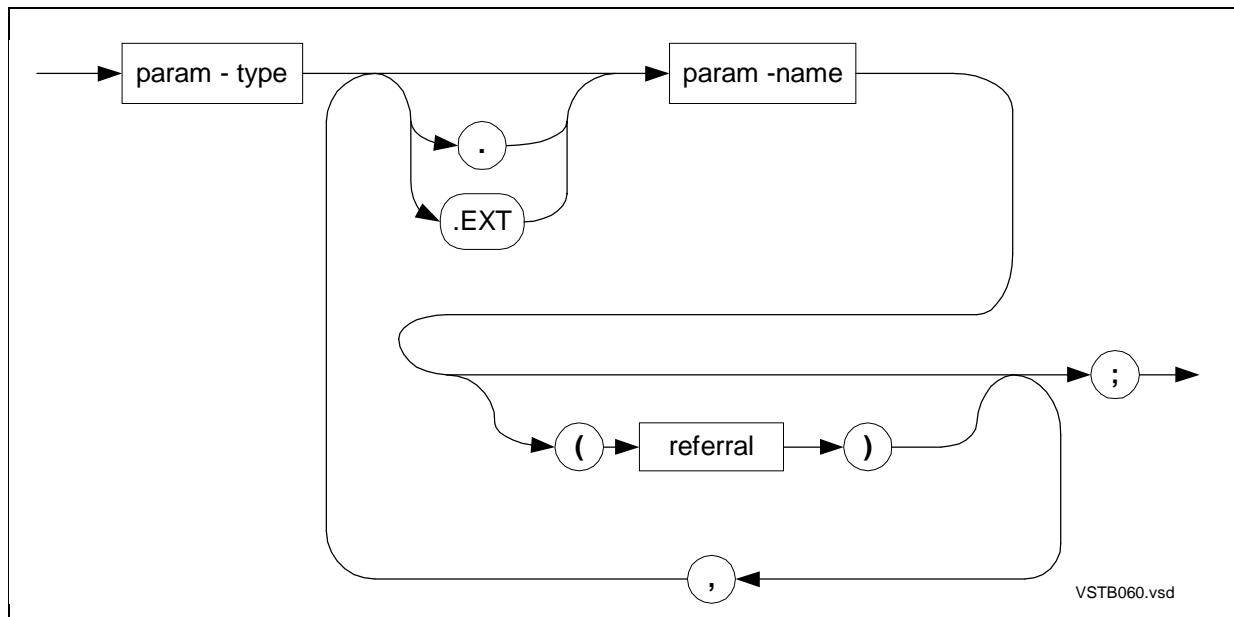
param-pair



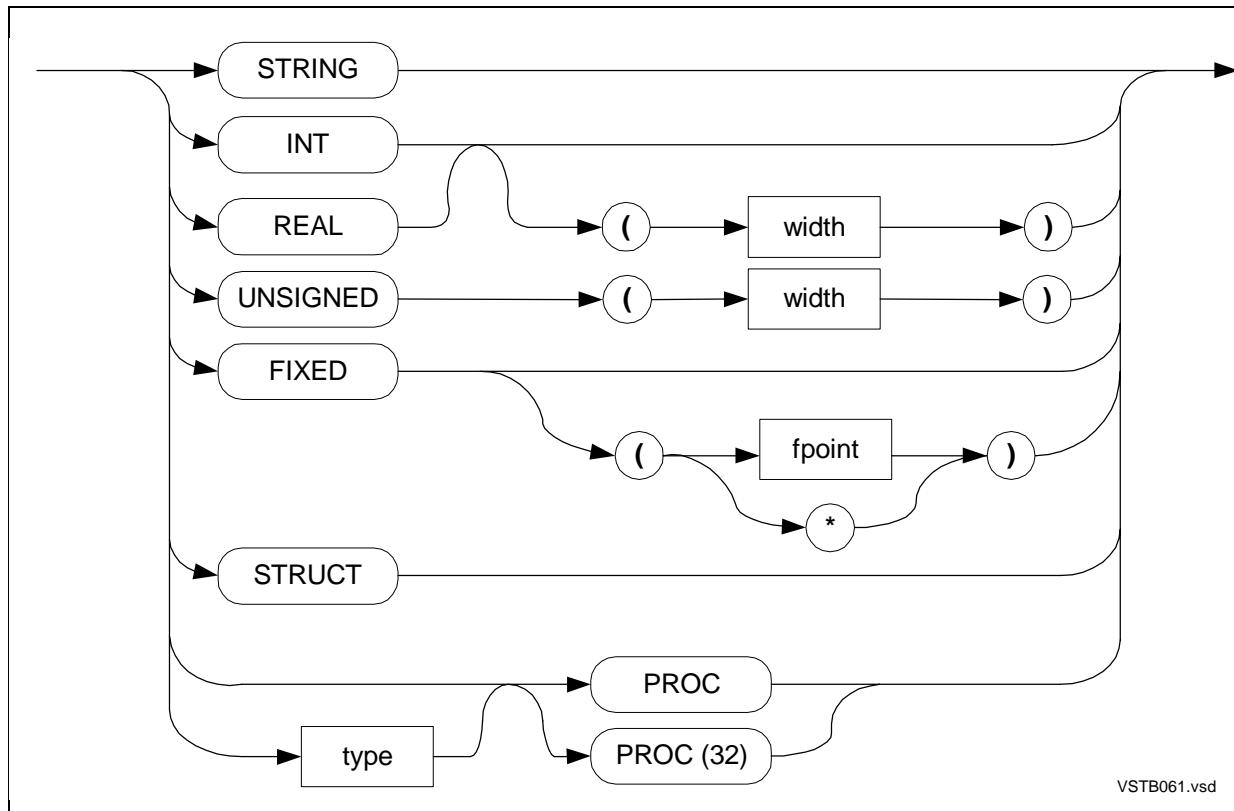
proc-attribute



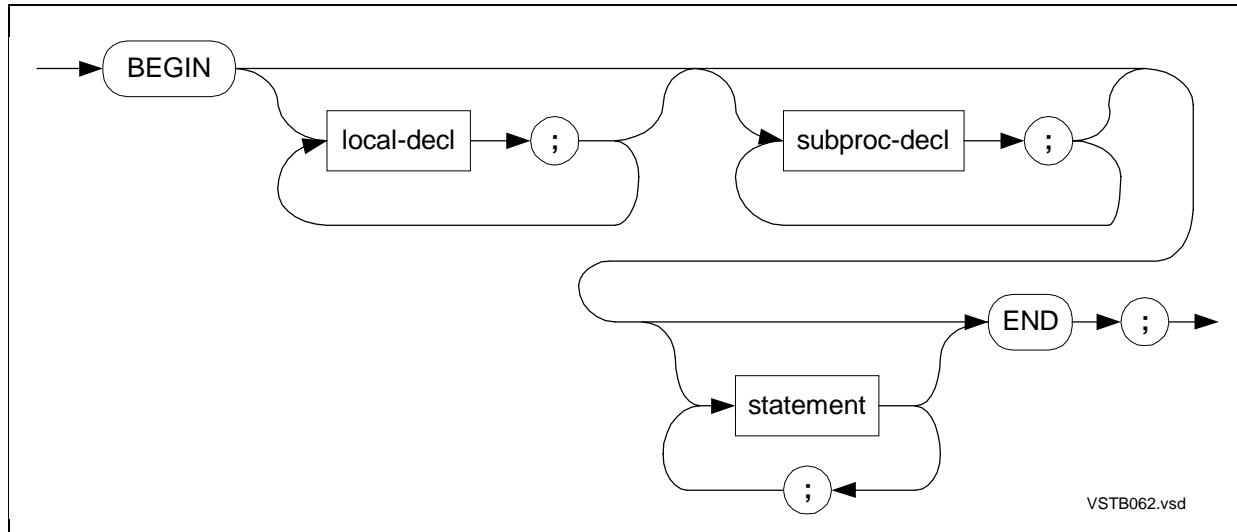
param - spec



param - type

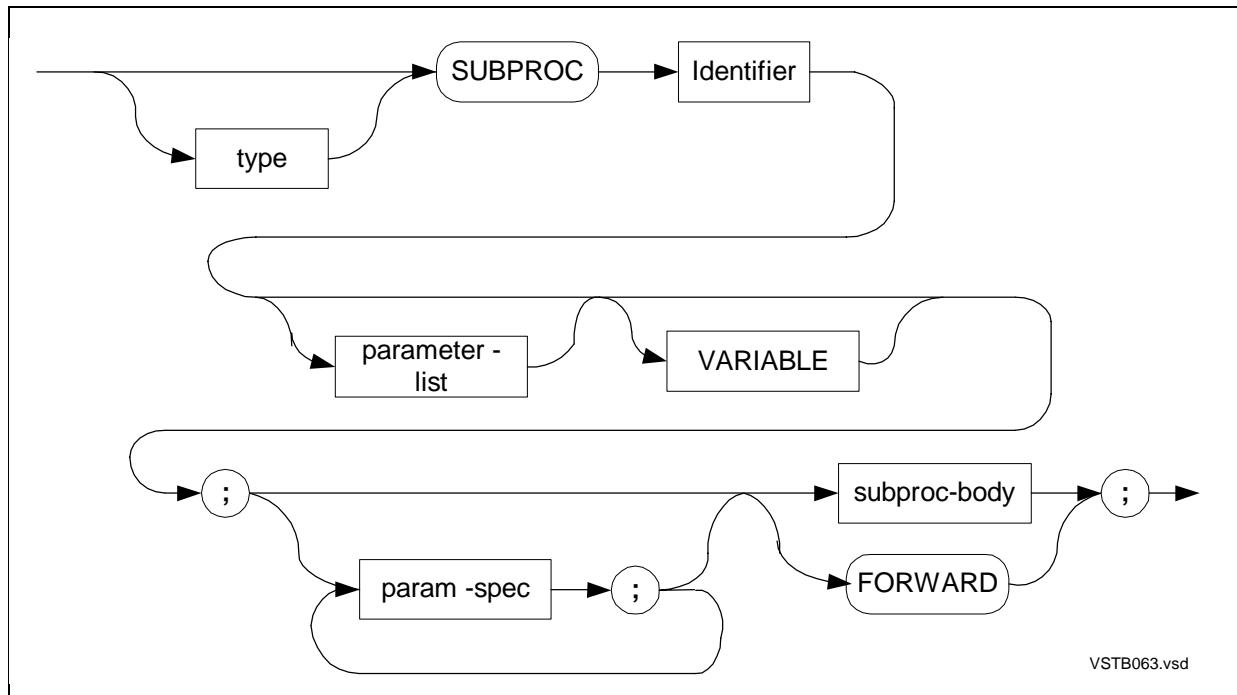


proc - body

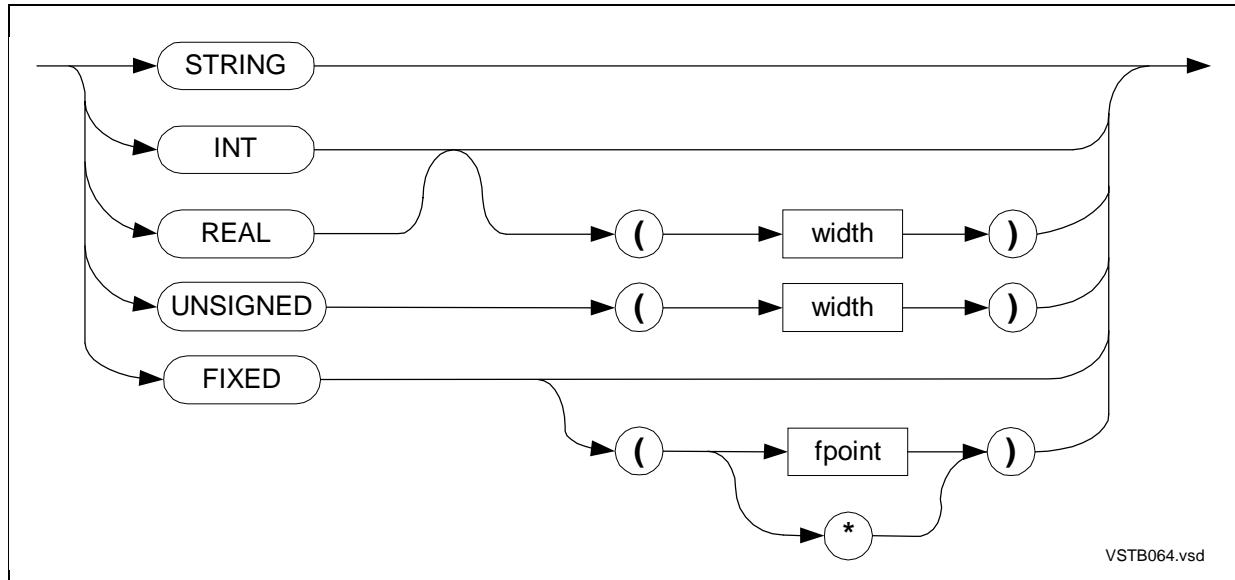


## Subprocedures

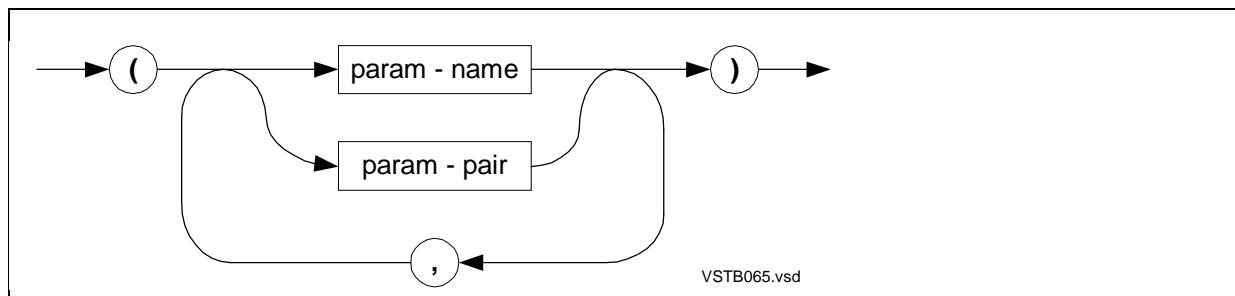
A subprocedure is a program unit that is callable from anywhere in the procedure.



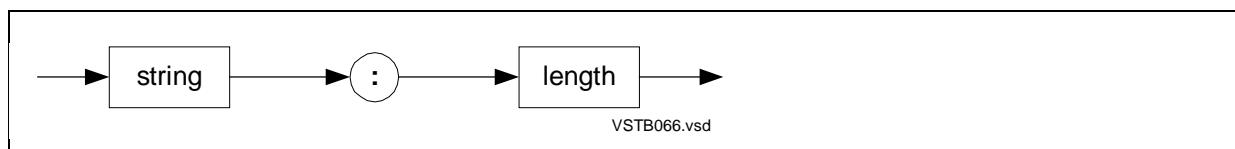
type



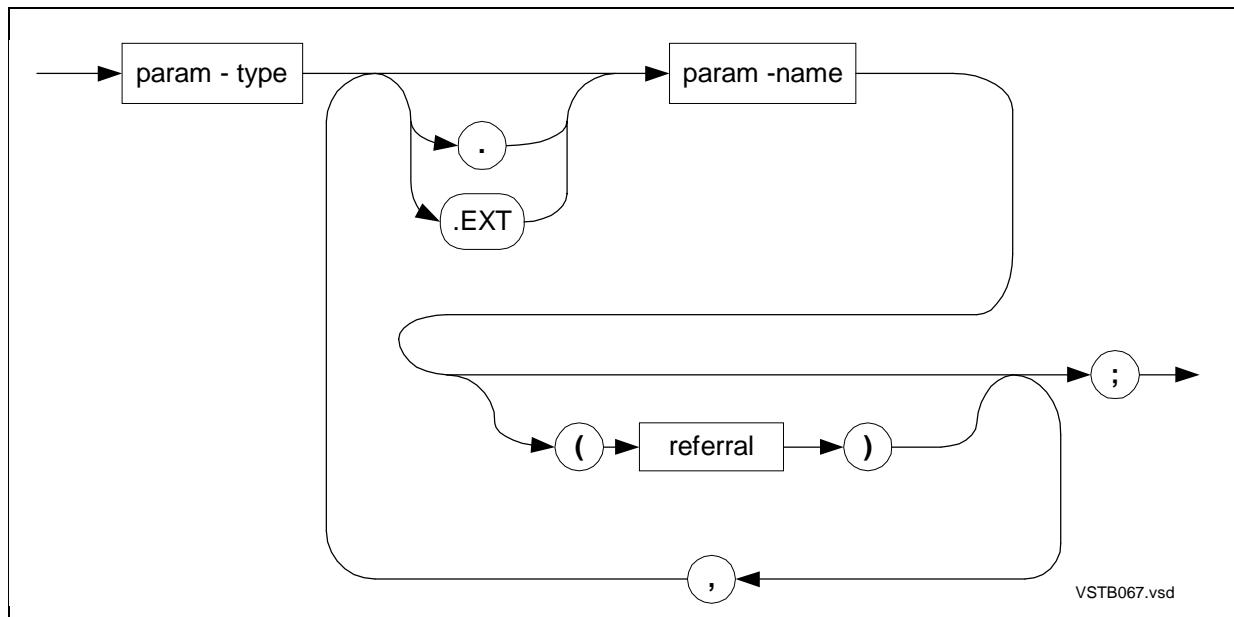
parameter - list



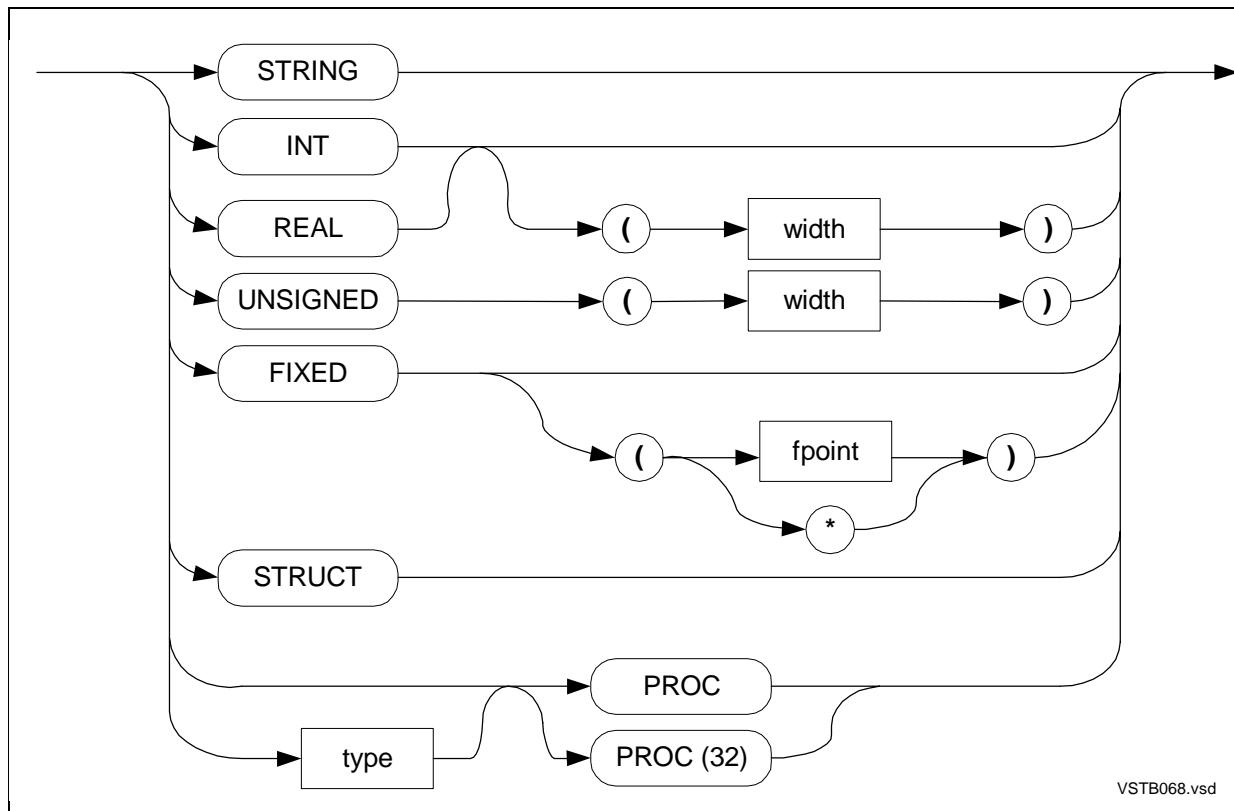
param - pair



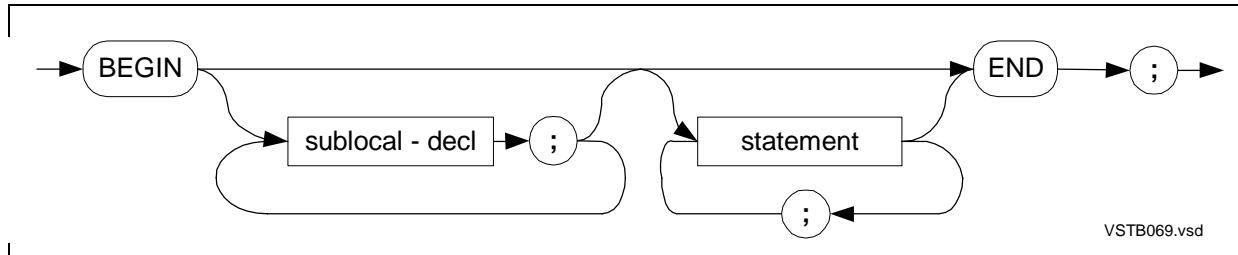
param - spec



param - type

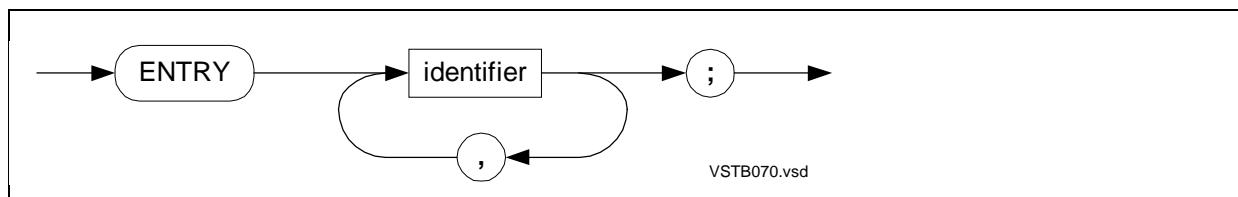


subproc - body



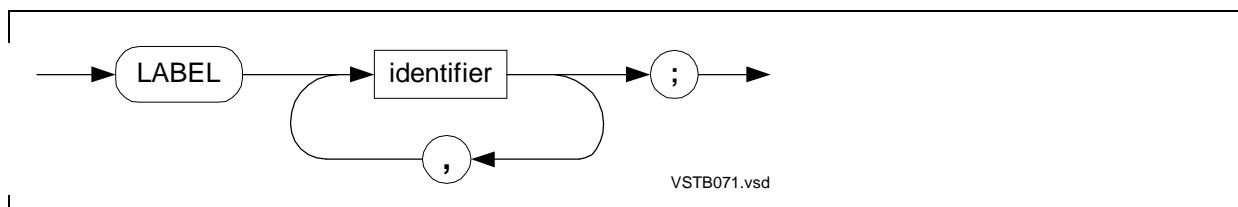
## Entry Points

The entry-point declaration associates an identifier with a secondary location from which execution can start in a procedure or subprocedure.



## Labels

The LABEL declaration reserves an identifier for later use as a label within the encompassing procedure or subprocedure.

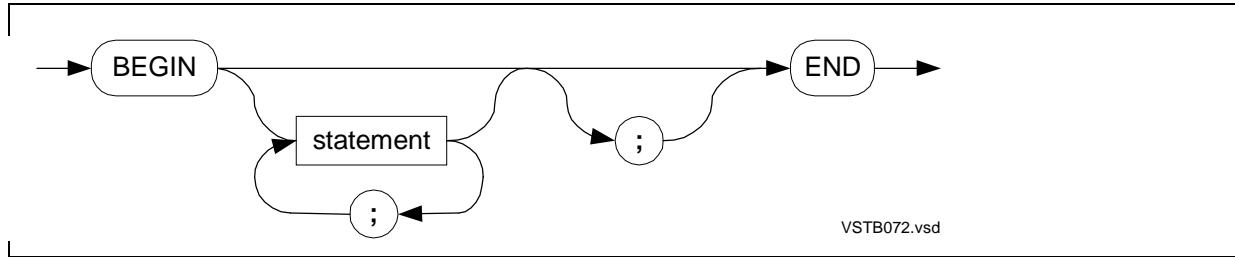


## Statements

The following syntax diagrams describe statements in alphabetic order.

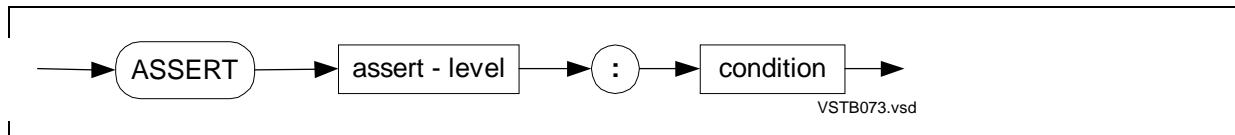
### Compound Statements

A compound statement is a BEGIN-END construct that groups statements to form a single logical statement.



## ASSERT Statement

The ASSERT statement conditionally invokes the procedure specified in an ASSERTION directive.



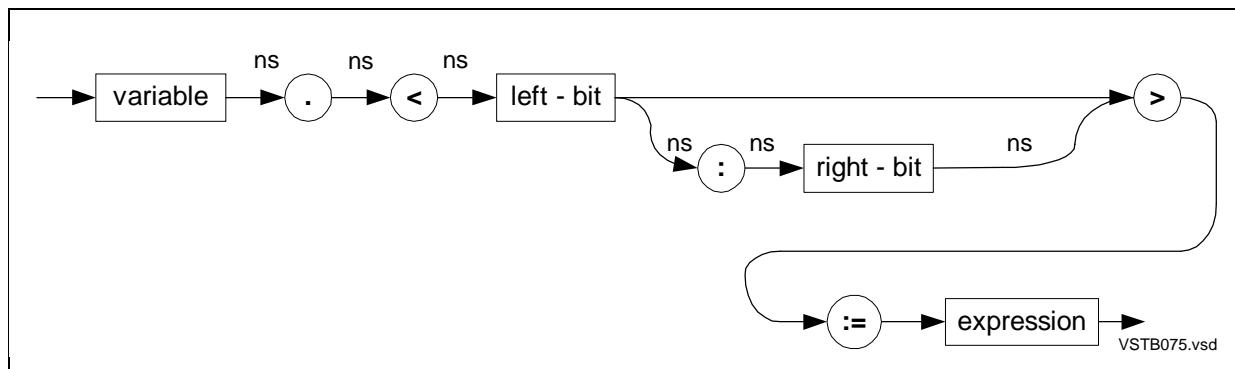
## Assignment Statement

The assignment statement assigns a value to a previously declared variable.



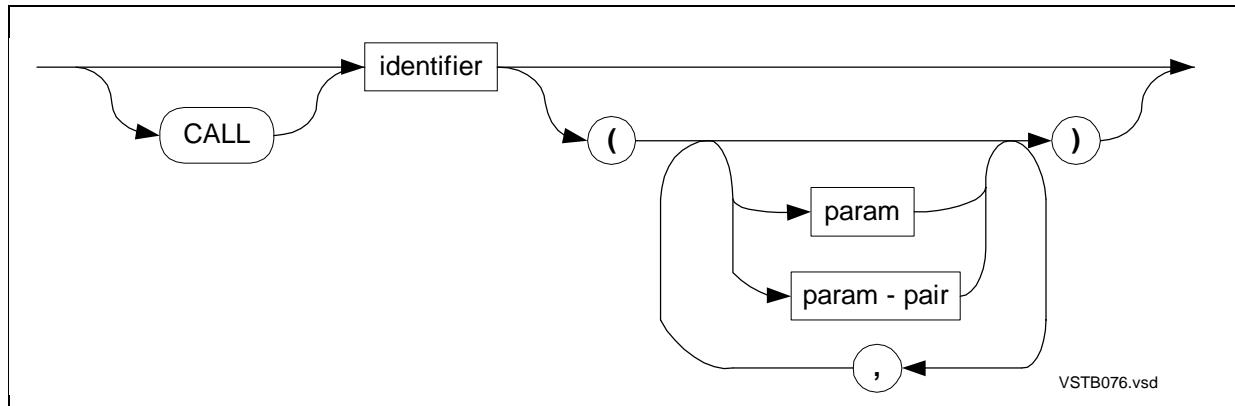
## Bit Deposit Assignment Statement

The bit-deposit assignment statement assigns a value to a bit field in a variable.

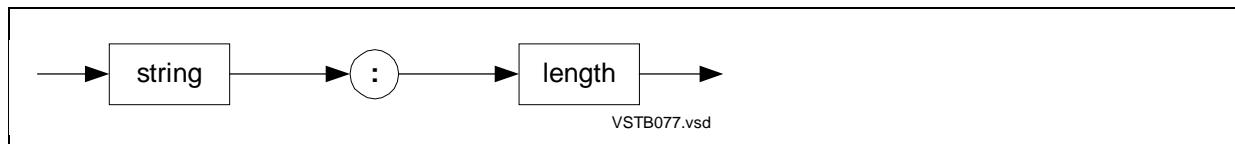


## CALL Statement

The CALL statement invokes a procedure, subprocedure, or entry point, and optionally passes parameters to it.

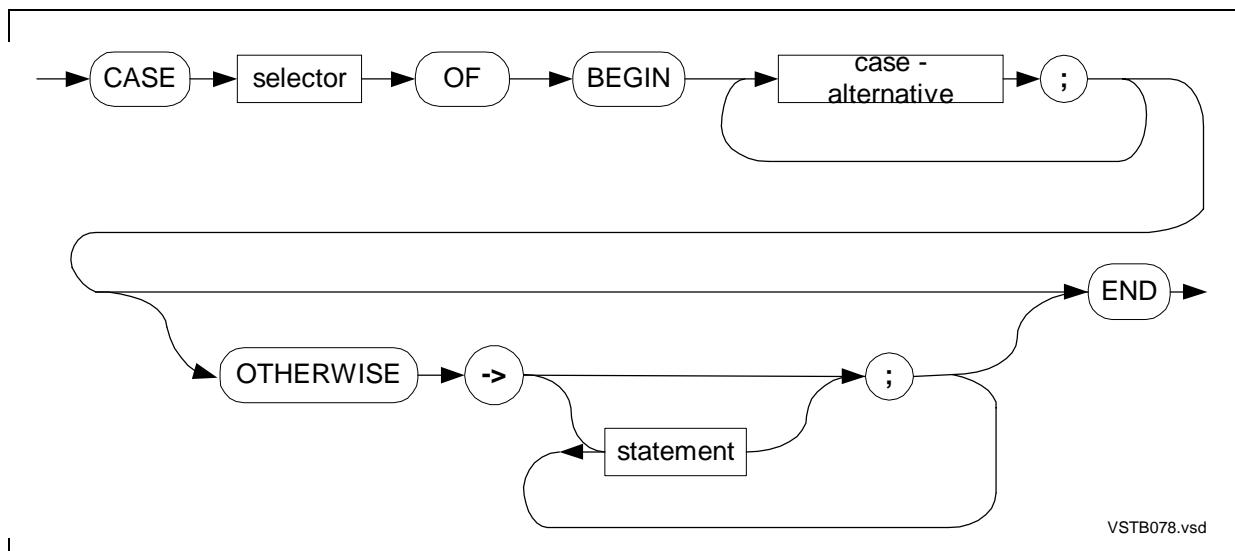


param - pair

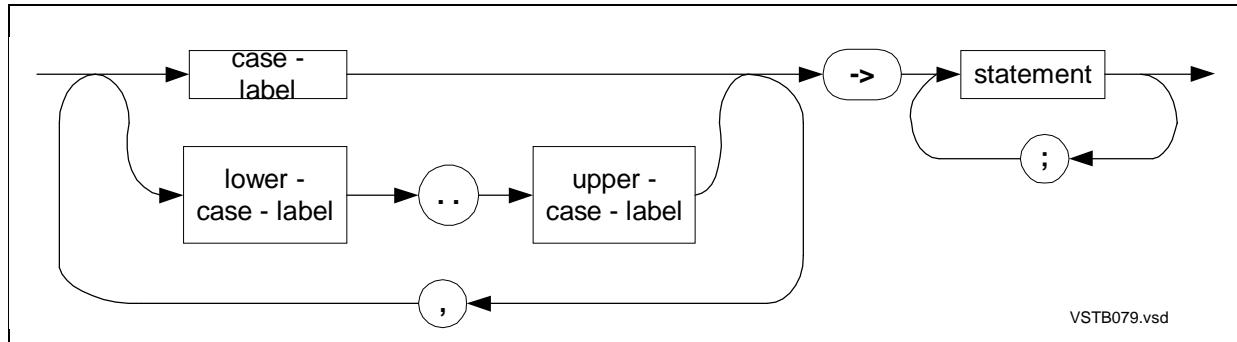


## Labeled CASE Statement

The labeled CASE statement executes a choice of statements the selector value matches a case label associated with those statements.



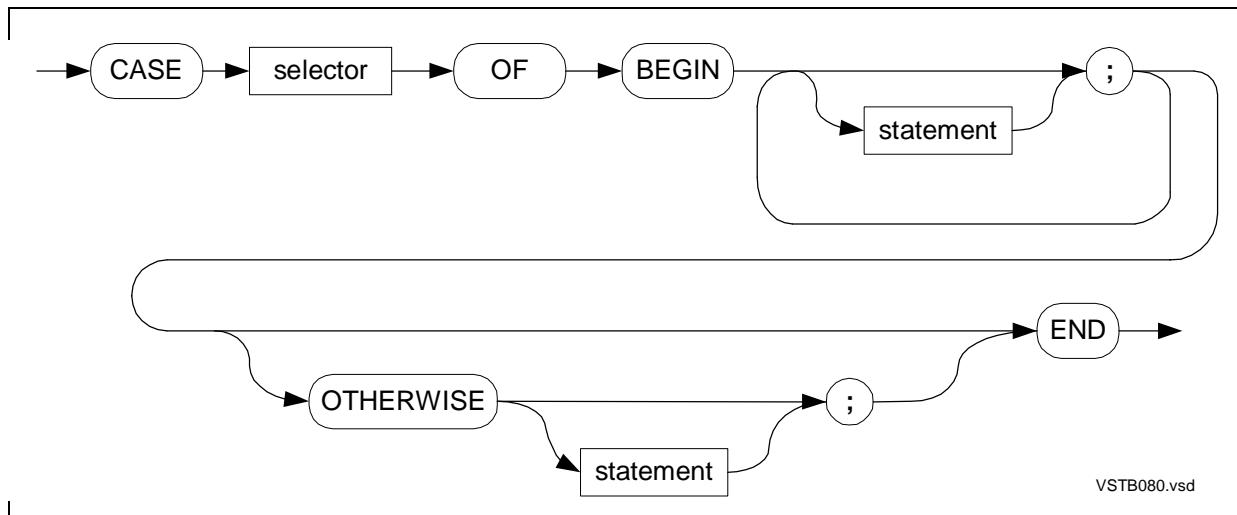
case - alternative



VSTB079.vsd

## Unlabeled CASE Statement

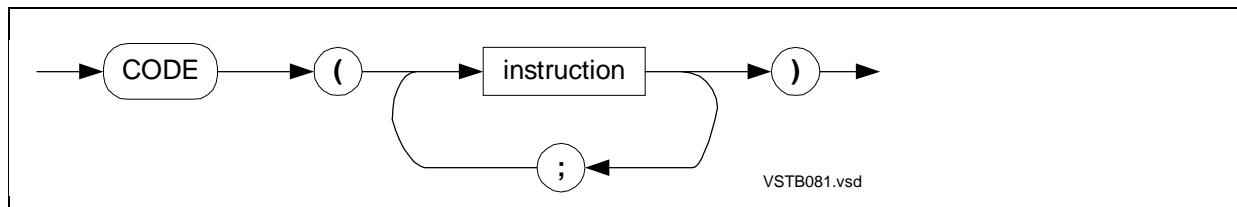
The unlabeled CASE statement executes a choice of statements based on an inclusive range of implicit selector values, from 0 through  $n$ , with one statement for each value.



VSTB080.vsd

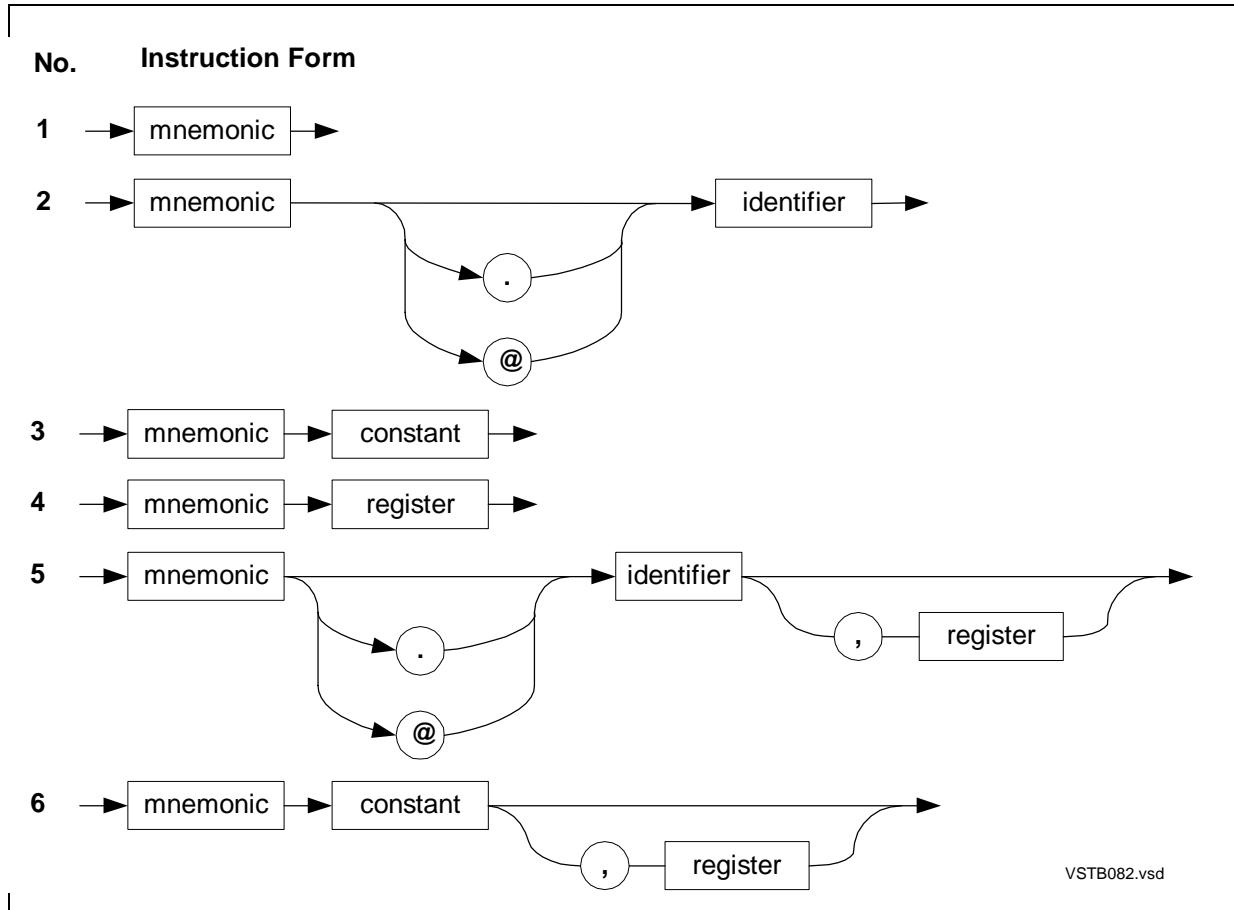
## CODE Statement

The CODE statement specifies machine-level instructions and pseudocodes to compile into the object file.



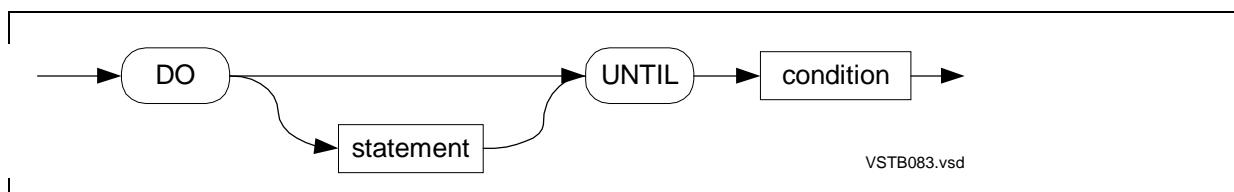
VSTB081.vsd

instruction



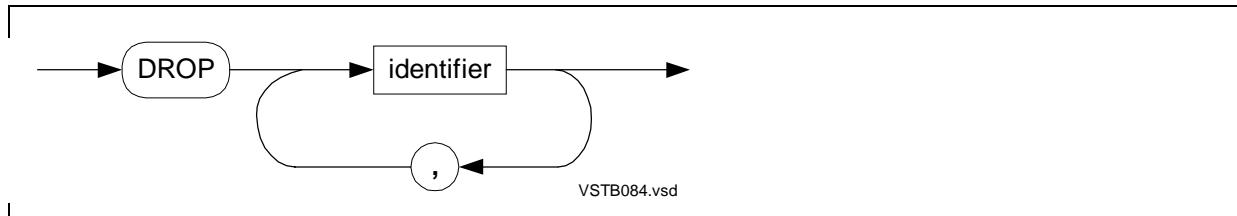
## DO Statement

The DO statement is a posttest loop that repeatedly executes a statement until a specified condition becomes true.



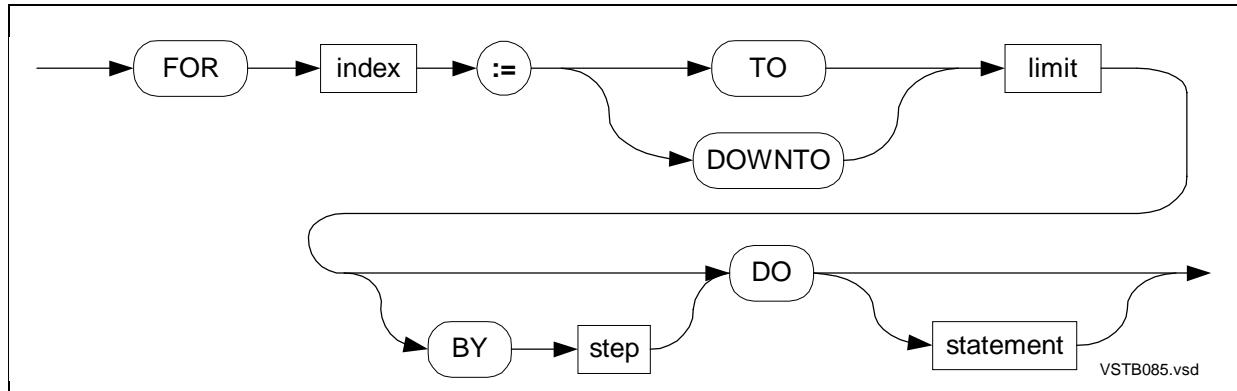
## DROP Statement

The DROP statement disassociates an identifier from an index register reserved by a USE statement or from a label.



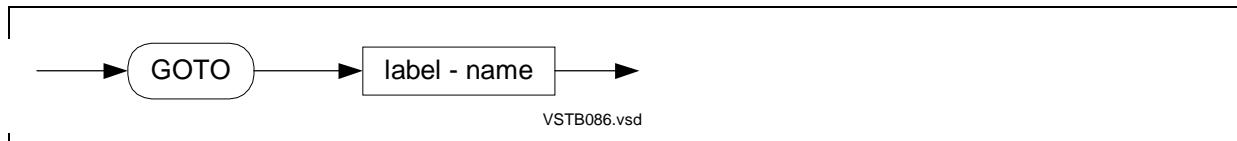
## FOR Statement

The FOR statement is a pretest loop that repeatedly executes a statement while incrementing or decrementing an index.



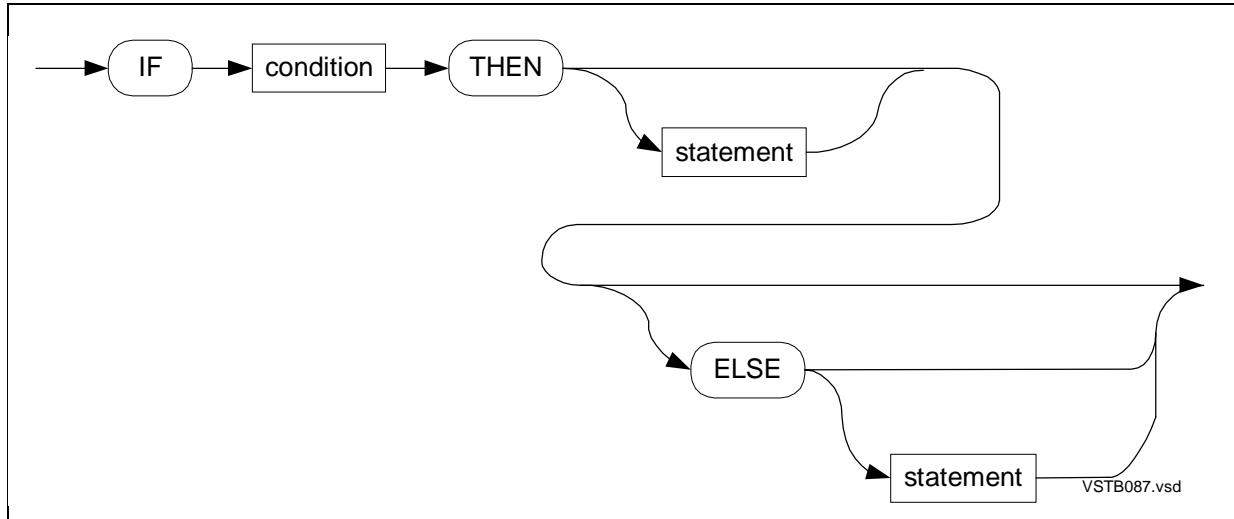
## GOTO Statement

The GOTO statement unconditionally transfers program control to a labeled statement.



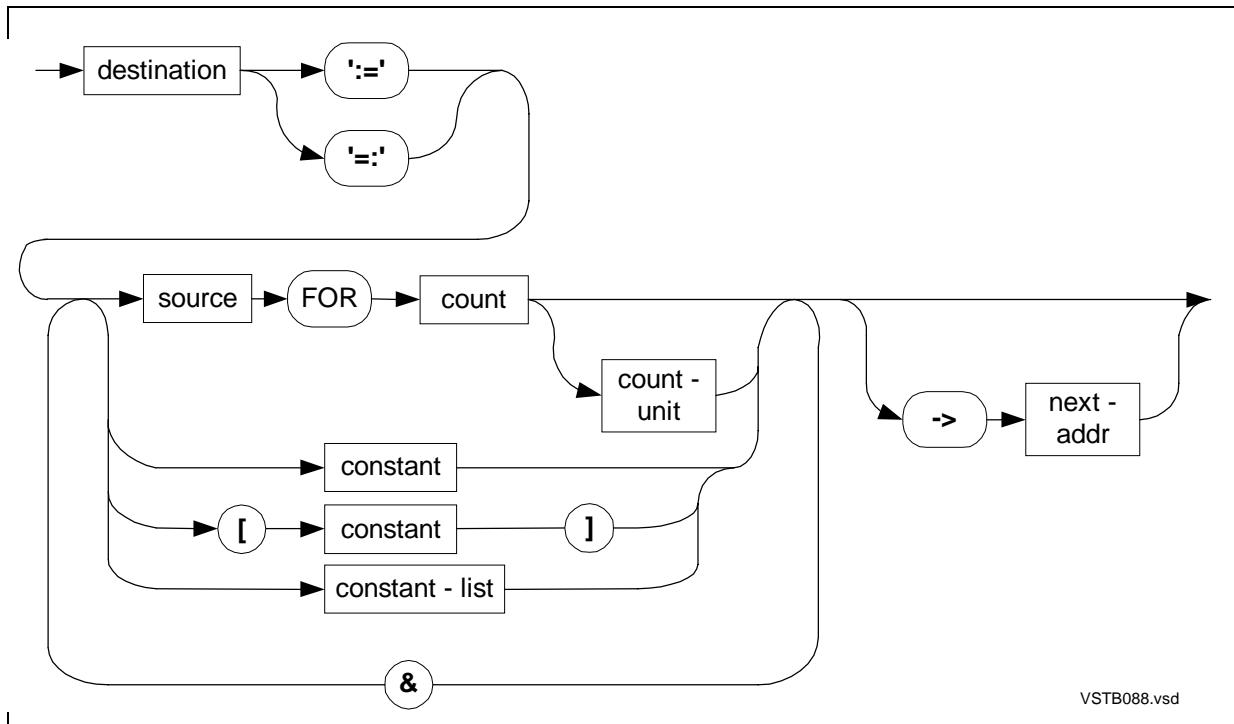
## IF Statement

The IF statement conditionally selects one of two statements.



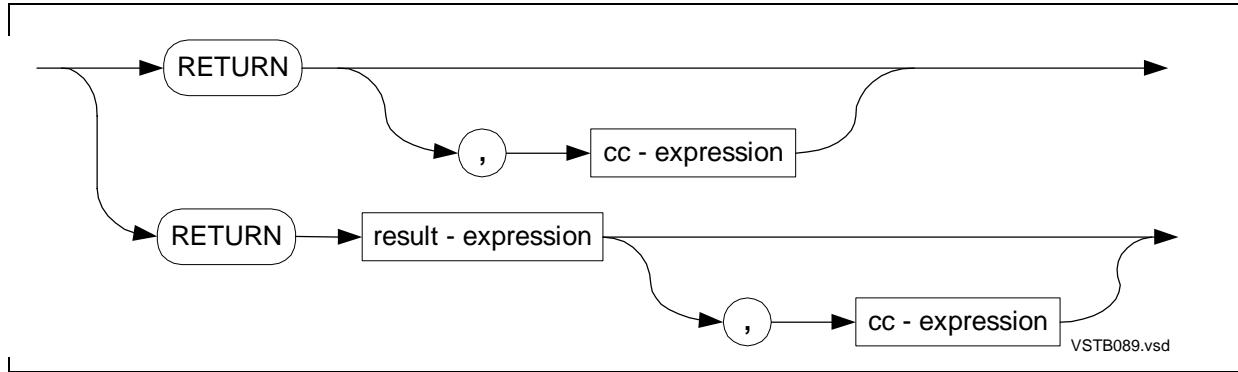
## Move Statement

The move statement copies contiguous bytes, words, or elements to a new location.



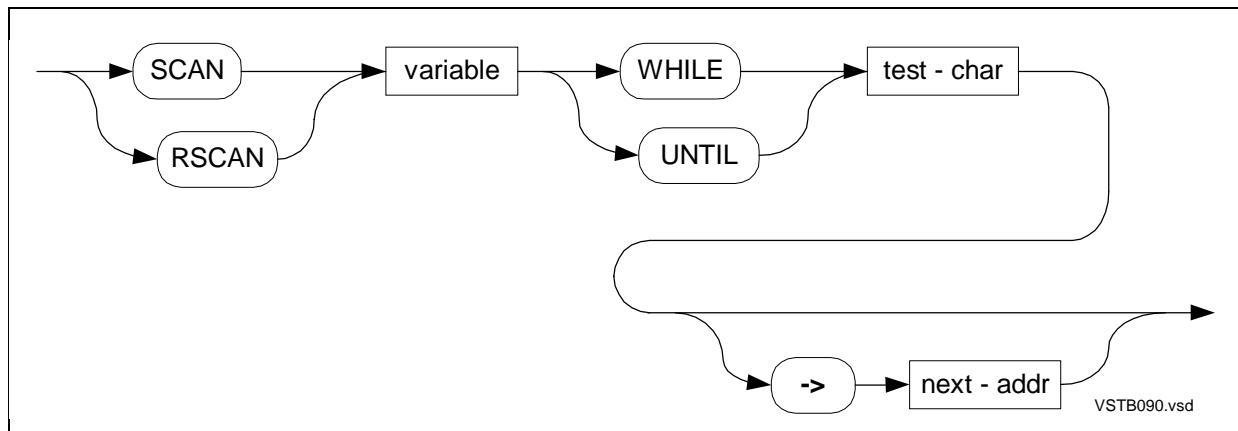
## RETURN Statement

The RETURN statement returns control to the caller. For a function, RETURN must return a result expression. The RETURN directive can also return a condition-code value.



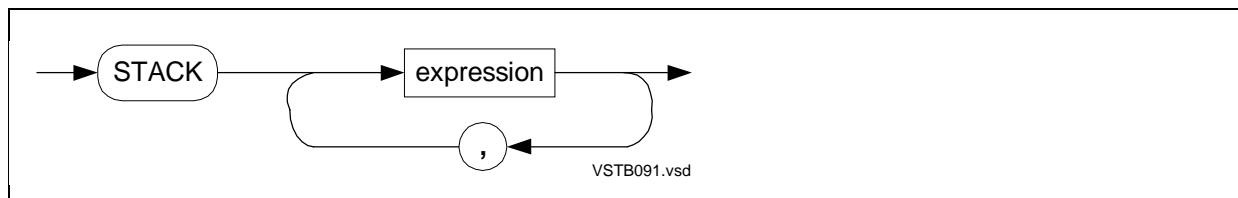
## Scan Statement

The SCAN or RSCAN statement scans sequential bytes for a test character from left to right or from right to left, respectively.



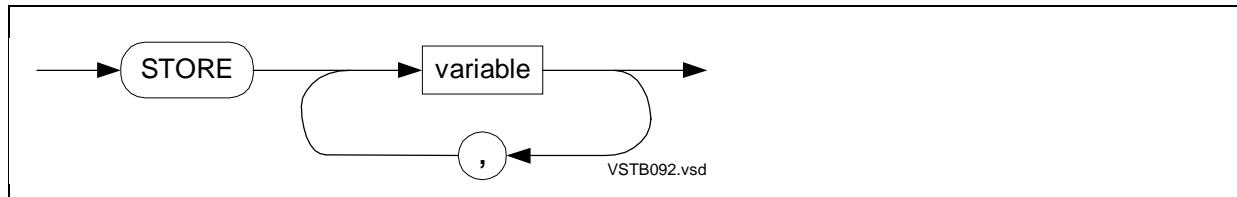
## STACK Statement

The STACK statement loads values onto the register stack.



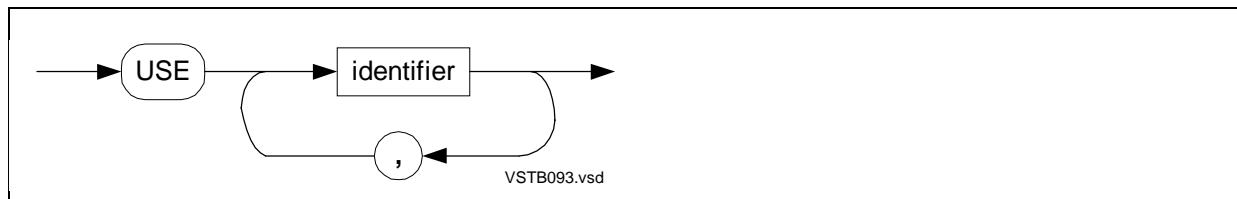
## STORE Statement

The STORE statement removes values from the register stack and stores them into variables.



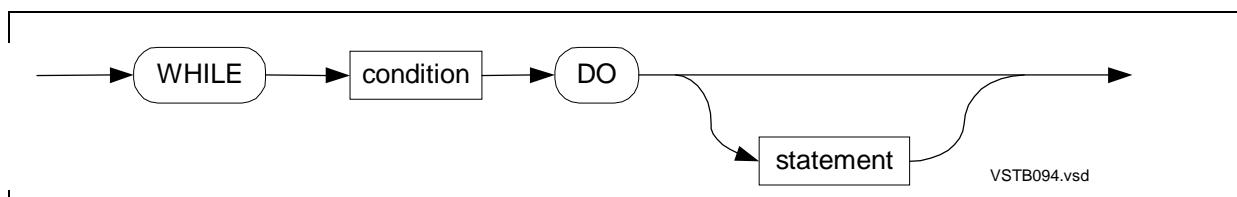
## USE Statement

The USE statement reserves an index register for your use.



## WHILE Statement

The WHILE statement is a pretest loop that repeatedly executes a statement while a condition is true.

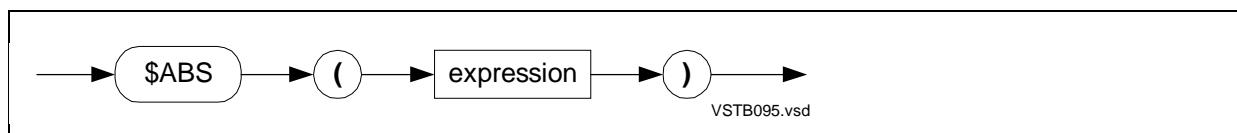


## Standard Functions

The following syntax diagrams describe standard functions in alphabetic order.

### \$ABS Function

The \$ABS function returns the absolute value of an expression. The returned value has the same data type as the expression.



## \$ALPHA Function

The \$ALPHA function tests the right byte of an INT value for the presence of an alphabetic character.

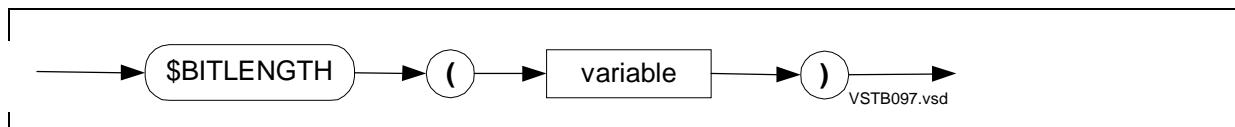


## \$AXADR Function

See [Section 15, Privileged Procedures](#)

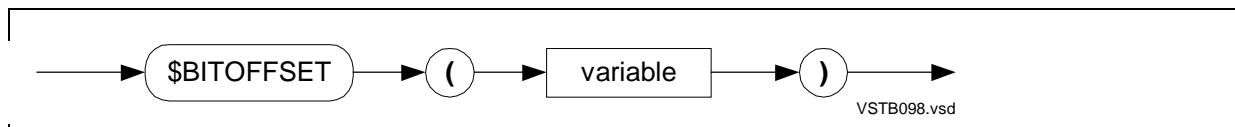
## \$BITLENGTH Function

The \$BITLENGTH function returns the length, in bits, of a variable.



## \$BITOFFSET Function

The \$BITOFFSET function returns the number of bits from the address of the zeroth structure occurrence to a structure data item.

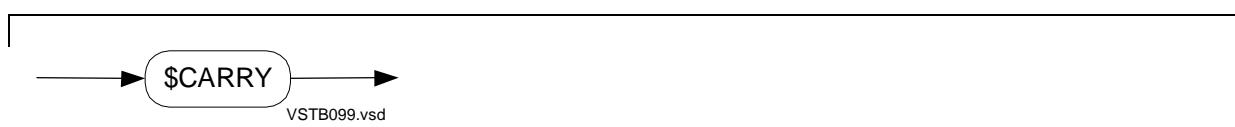


## \$BOUNDS Function

See [Section 15, Privileged Procedures](#)

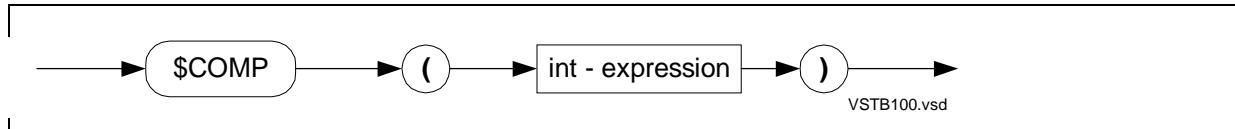
## \$CARRY Function

The \$CARRY function checks the state of the carry bit in the environment register and indicates whether a carry out of the high-order bit position occurred.



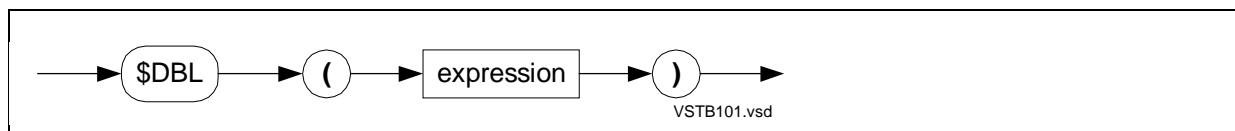
## \$COMP Function

The \$COMP function obtains the one's complement of an INT expression.



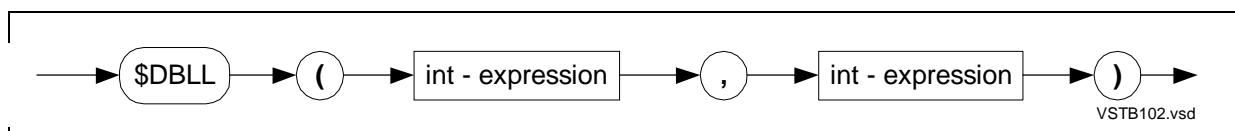
## \$DBL Function

The \$DBL function returns an INT(32) value from an INT, FIXED(0), REAL, or REAL(64) expression.



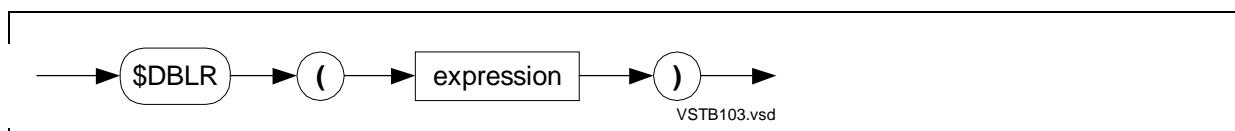
## \$DBLL Function

The \$DBLL function returns an INT(32) value from two INT values.



## \$DBLR Function

The \$DBLR function returns an INT(32) value from an INT, FIXED(0), REAL, or REAL(64) expression and applies rounding to the result.



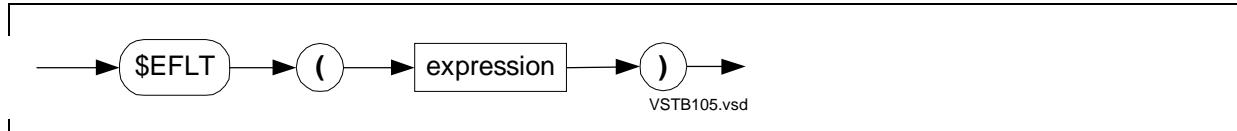
## \$DFIX Function

The \$DFIX function returns a FIXED (*fpoint*) expression from an INT(32) expression.



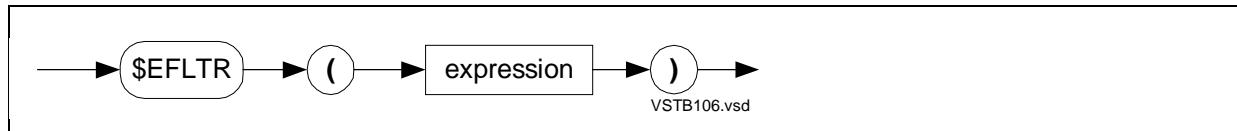
## \$EFLT Function

The \$EFLT function returns a REAL(64) value from an INT, INT(32), FIXED (*fpoint*), or REAL expression.



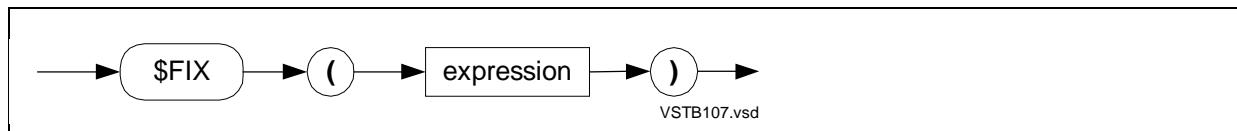
## \$EFLTR Function

The \$EFLTR function returns a REAL(64) value from an INT, INT(32), FIXED (*fpoint*), or REAL expression and applies rounding to the result.



## \$FIX Function

The \$FIX function returns a FIXED(0) value from an INT, INT(32), REAL, or REAL(64) expression.



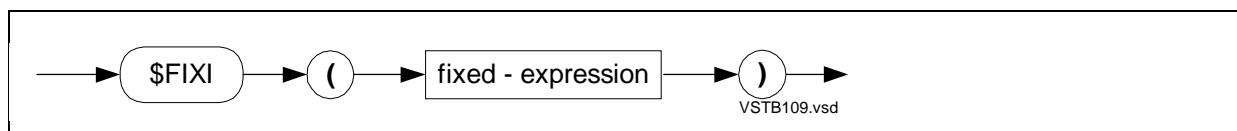
## \$FIXD Function

The \$FIXD function returns an INT(32) value from a FIXED(0) expression.



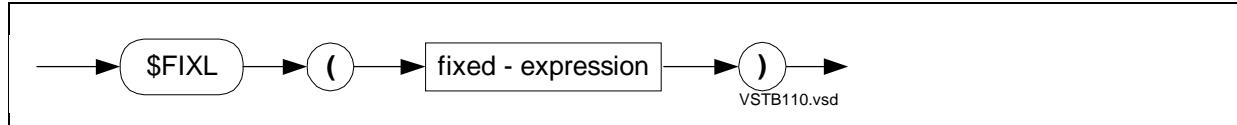
## \$FIXI Function

The \$FIXI function returns the signed INT equivalent of a FIXED(0) expression.



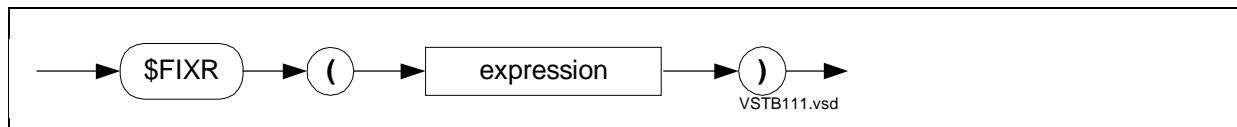
## \$FIXL Function

The \$FIXL function returns the unsigned INT equivalent of a FIXED(0) expression.



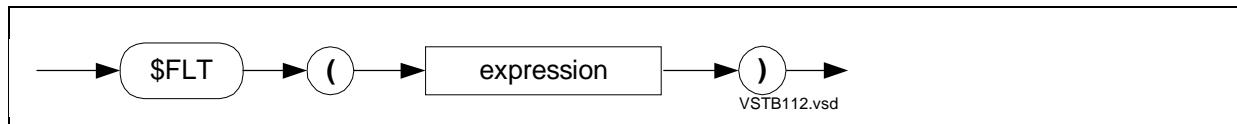
## \$FIXR Function

The \$FIXR function returns a FIXED(0) value from an INT, INT(32), FIXED, REAL, or REAL(64) expression and applies rounding to the result.



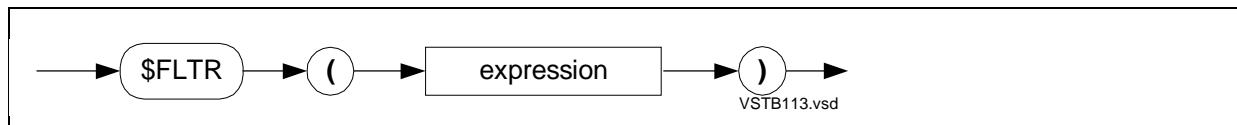
## \$FLT Function

The \$FLT function returns a REAL value from an INT, INT(32), FIXED (*fpoint*), or REAL(64) expression.



## \$FLTR Function

The \$FLTR function returns a REAL value from an INT, INT(32), FIXED (*fpoint*), or REAL(64) expression and applies rounding to the result.



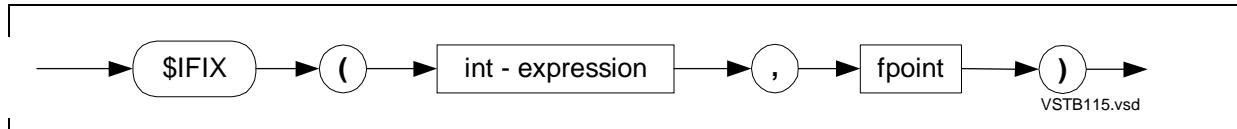
## \$HIGH Function

The \$HIGH function returns an INT value that is the high-order 16 bits of an INT(32) expression.



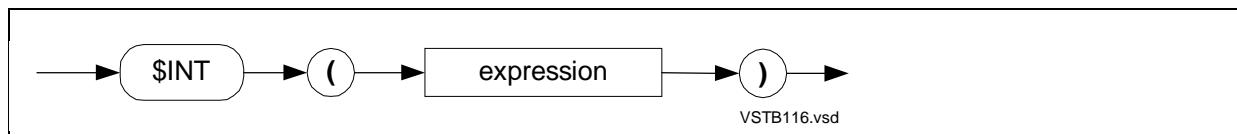
## \$IFIX Function

The \$IFIX function returns a FIXED (*fpoint*) value from a signed INT expression.



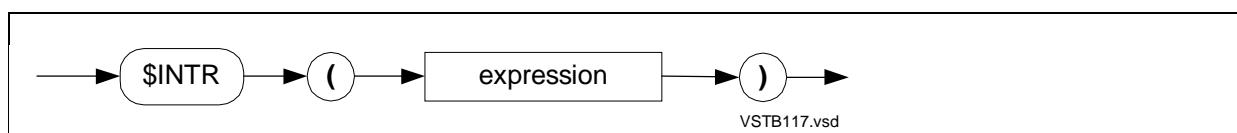
## \$INT Function

The \$INT function returns an INT value from the low-order 16 bits of an INT(32 or FIXED(0) expression. \$INT returns a fully converted INT expression from a REAL or REAL(64) expression.



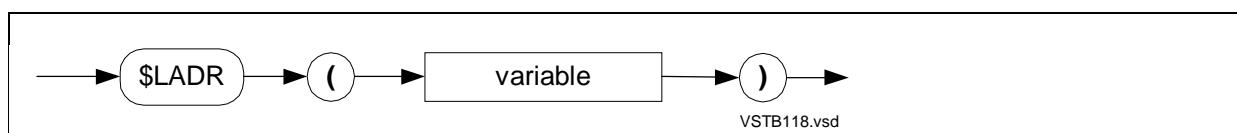
## \$INTR Function

The \$INTR function returns an INT value from the low-order 16 bits of an INT(32) or FIXED(0) expression. \$INTR returns a fully converted and rounded INT expression from a REAL or REAL(64) expression.



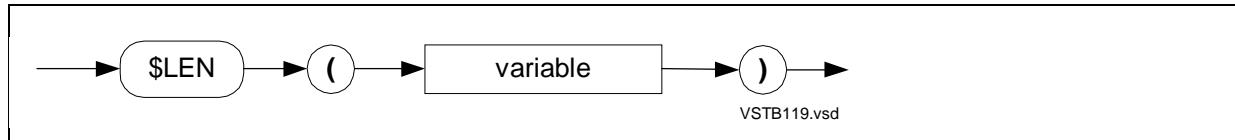
## \$LADR Function

The \$LADR function returns the standard (16-bit) address of a variable that is accessed through an extended (32-bit) pointer.



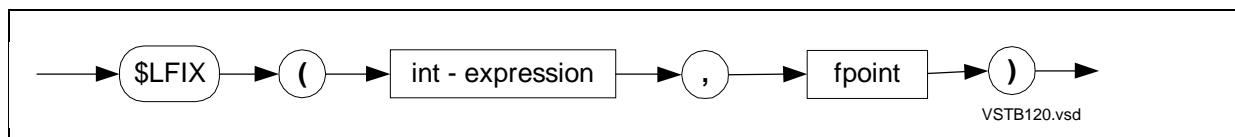
## \$LEN Function

The \$LEN function returns the length, in bytes, of one occurrence of a variable.



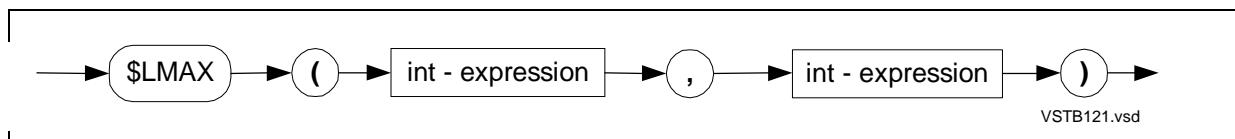
## \$LFIX Function

The \$LFIX function returns a FIXED (*fpoint*) expression from an unsigned INT expression.



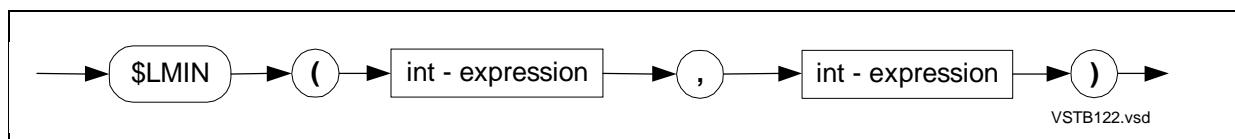
## \$LMAX Function

The \$LMAX function returns the maximum of two unsigned INT expressions.



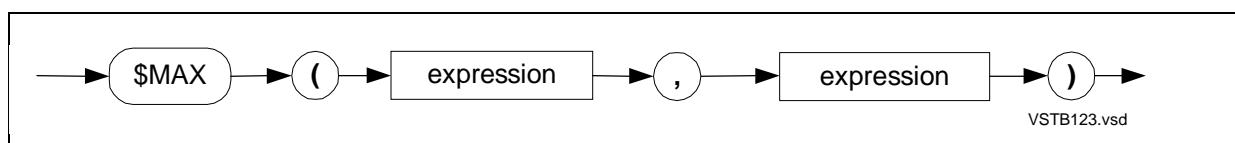
## \$LMIN Function

The \$LMIN function returns the minimum of two unsigned INT expressions.



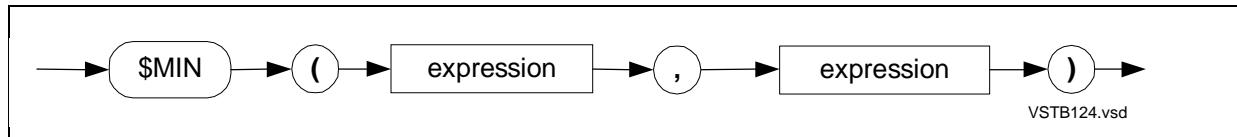
## \$MAX Function

The \$MAX function returns the maximum of two signed INT, INT(32), FIXED (*fpoint*), REAL, or REAL(64) expressions.



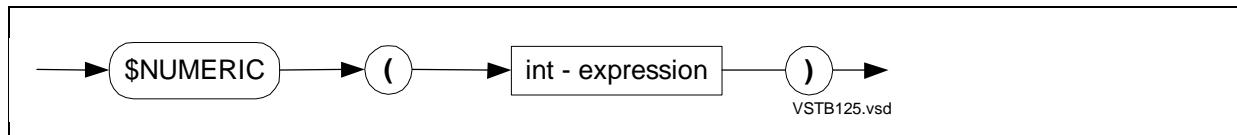
## \$MIN Function

The \$MIN function returns the minimum of two INT, INT(32), FIXED (*fpoint*), REAL, or REAL(64) expressions.



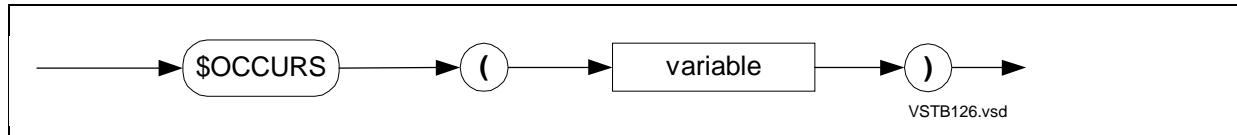
## \$NUMERIC Function

The \$NUMERIC function tests the right half of an INT value for the presence of an ASCII numeric character.



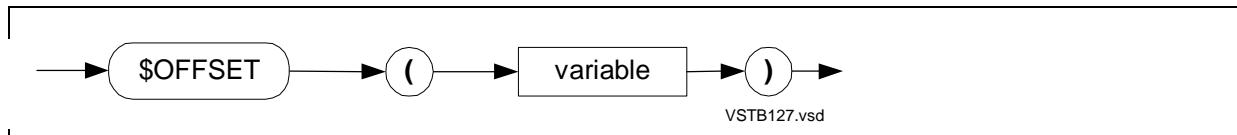
## \$OCCURS Function

The \$OCCURS function returns the number of occurrences of a variable.



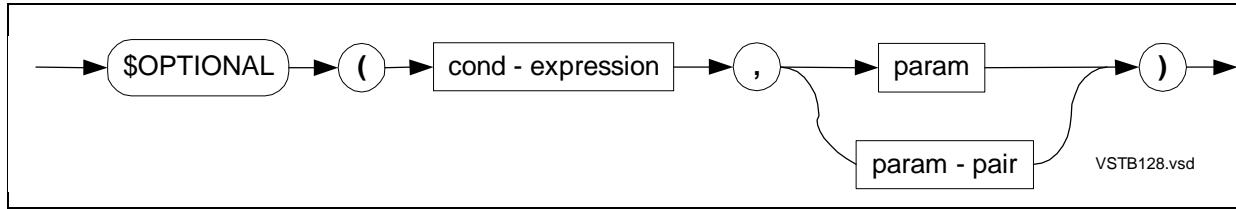
## \$OFFSET Function

The \$OFFSET function returns the number of bytes from the address of the zeroth structure occurrence to a structure data item.



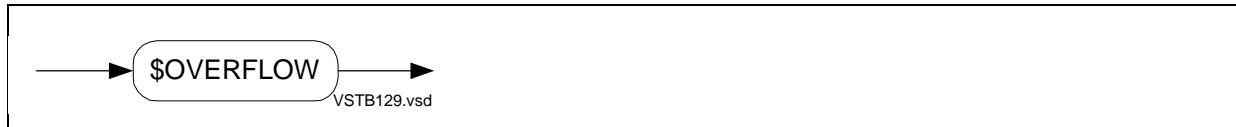
## \$OPTIONAL Function

The \$OPTIONAL function controls whether a given parameter or parameter pair is passed to a VARIABLE or EXTENSIBLE procedure. OPTIONAL is a D20 or later feature.



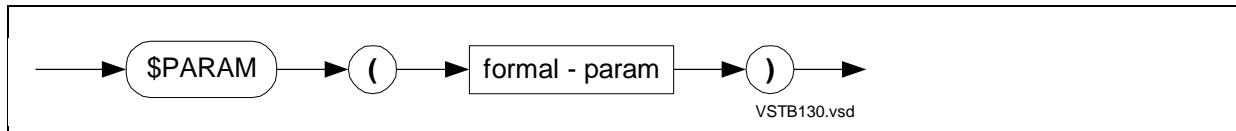
## \$OVERFLOW Function

The \$OVERFLOW function checks the state of the overflow indicator and indicates whether an overflow occurred during an arithmetic operation.



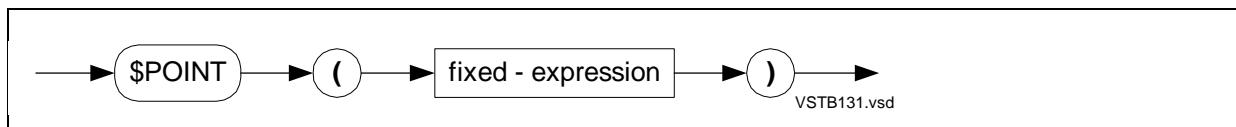
## \$PARAM Function

The \$PARAM function checks for the presence or absence of an actual parameter in the call that invoked the current procedure or subprocedure.



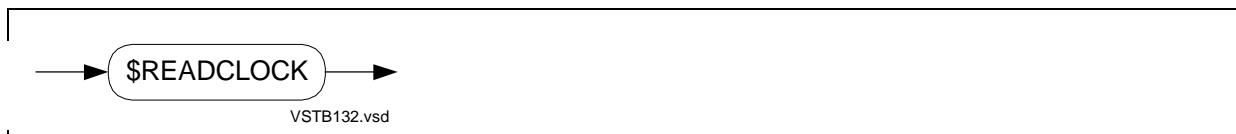
## \$POINT Function

The \$POINT function returns the *fpoint* value, in integer form, associated with a FIXED expression.



## \$READCLOCK Function

The \$READCLOCK function returns the current setting of the system clock.



## \$RP Function

The \$RP function returns the current setting of the compiler's internal RP counter. (RP is the register stack pointer.)



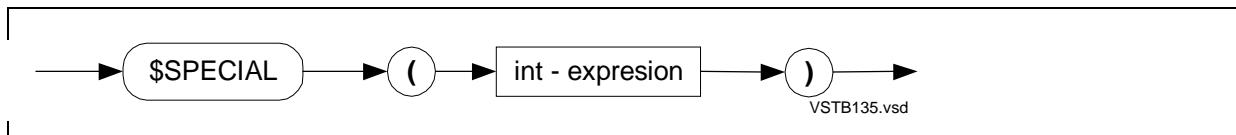
## \$SCALE Function

The \$SCALE function moves the position of the implied decimal point by adjusting the internal representation of a FIXED (*fpoint*) expression.



## \$SPECIAL Function

The \$SPECIAL function tests the right half of an INT value for the presence of an ASCII special (non-alphanumeric) character.

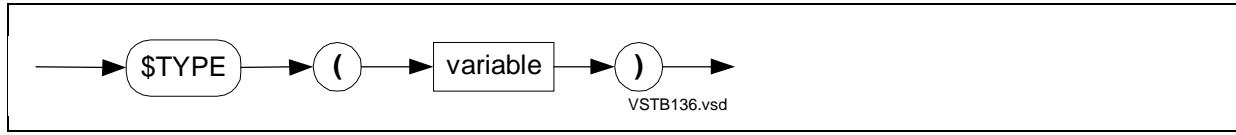


## \$SWITCHES Function

See [Section 15, Privileged Procedures](#)

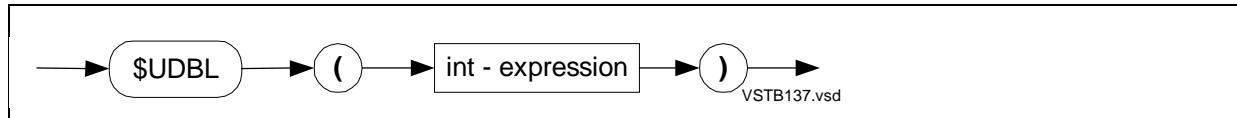
## \$TYPE Function

The \$TYPE function returns a value that indicates the data type of a variable.



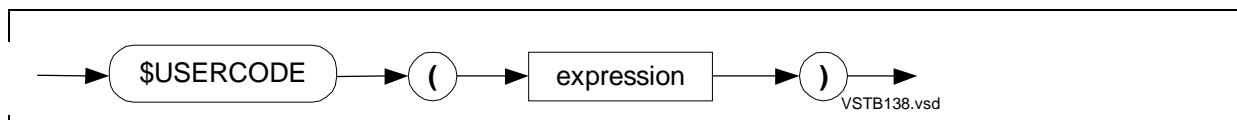
## \$UDBL Function

The `$UDBL` function returns an INT(32) value from an unsigned INT expression.



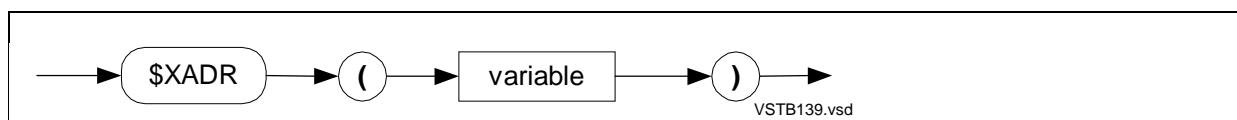
# \$USERCODE Function

The \$USERCODE function returns the content of the word at the specified location in the current user code segment.



## \$XADR Function

The \$XADR function converts a standard address to an extended address.

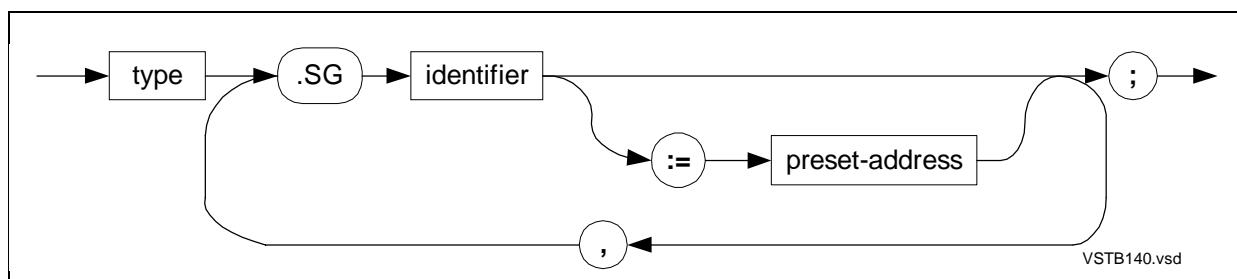


# Privileged Procedures

The following syntax diagrams describe declarations for privileged procedures.

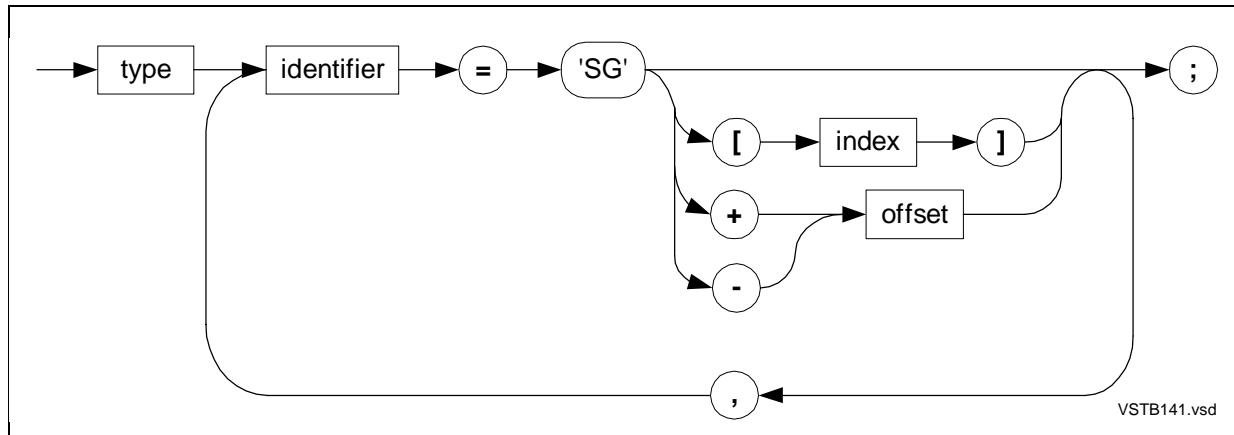
# System Global Pointers

The system global pointer declaration associates an identifier with a memory location that you load with the address of a variable located in the system global data area.



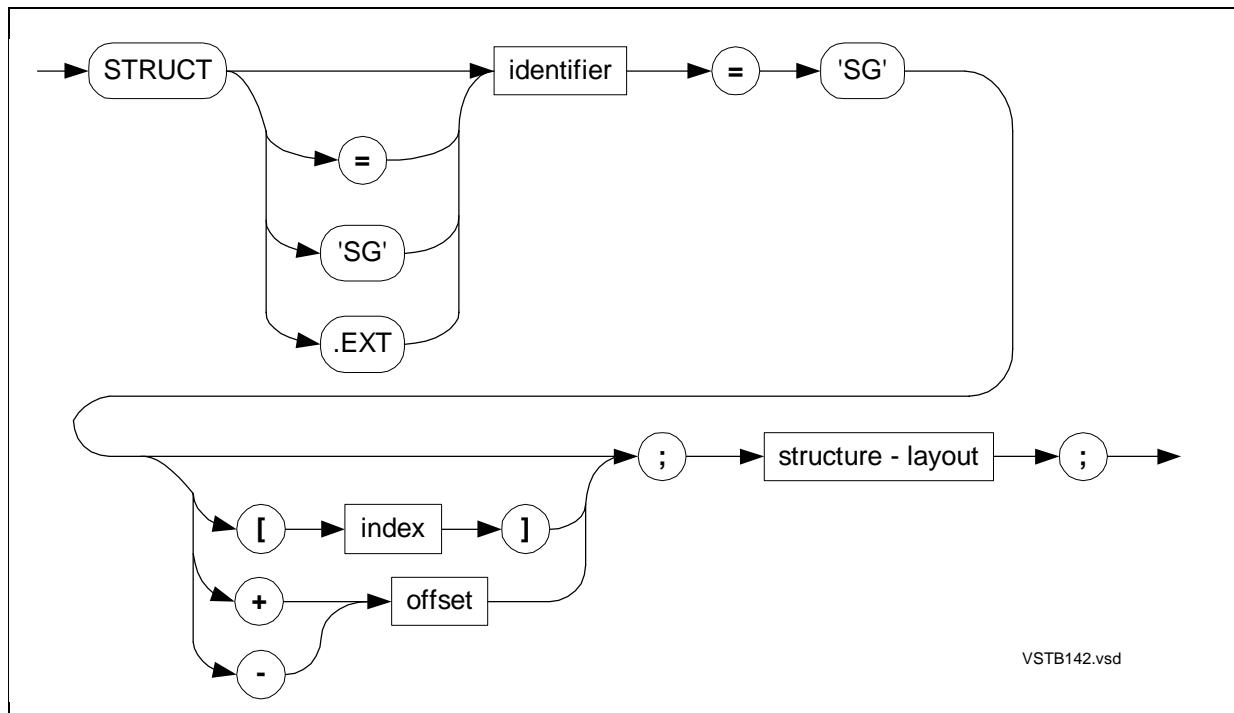
## 'SG'-Equivalenced Simple Variables

The 'SG'-equivalenced simple variable declaration associates a simple variable with a location that is relative to the base address of the system global data area.



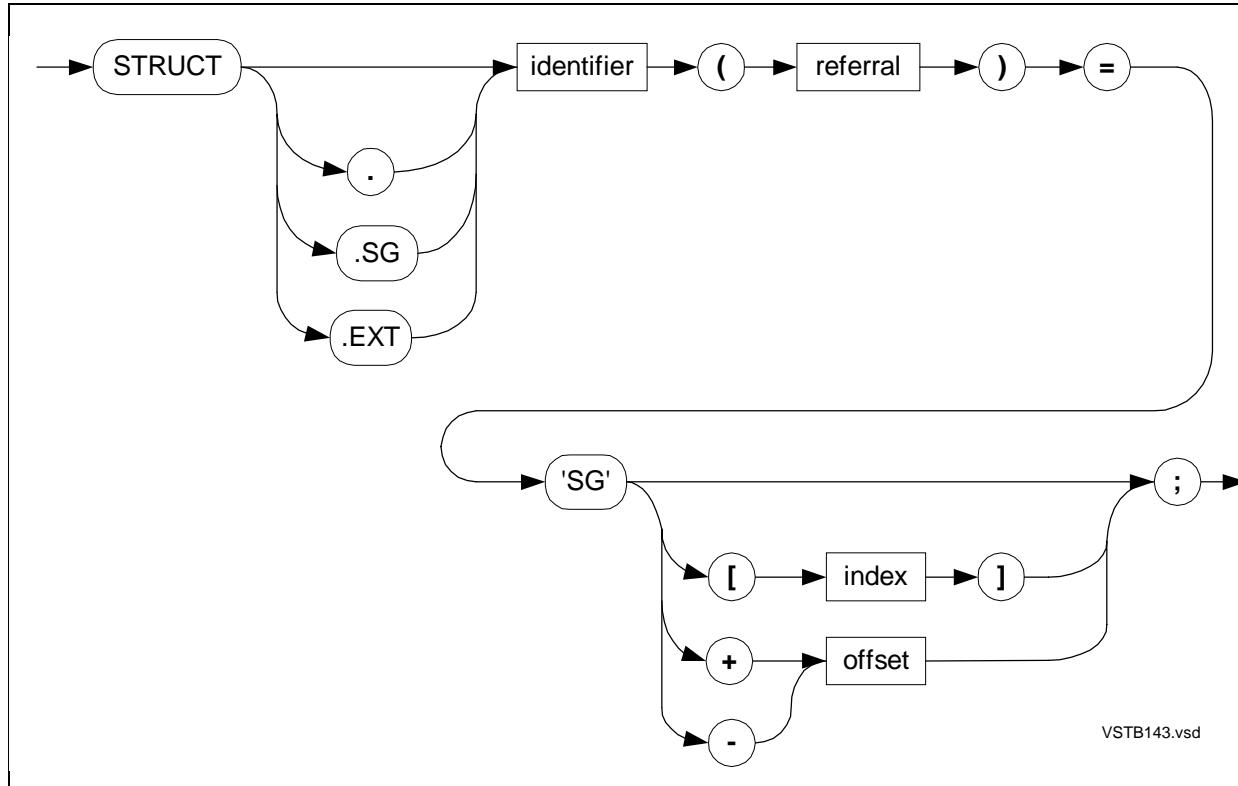
## 'SG'-Equivalenced Definition Structures

The 'SG'-equivalenced definition structure declaration associates a definition structure with a location relative to the base address of the system global data area.



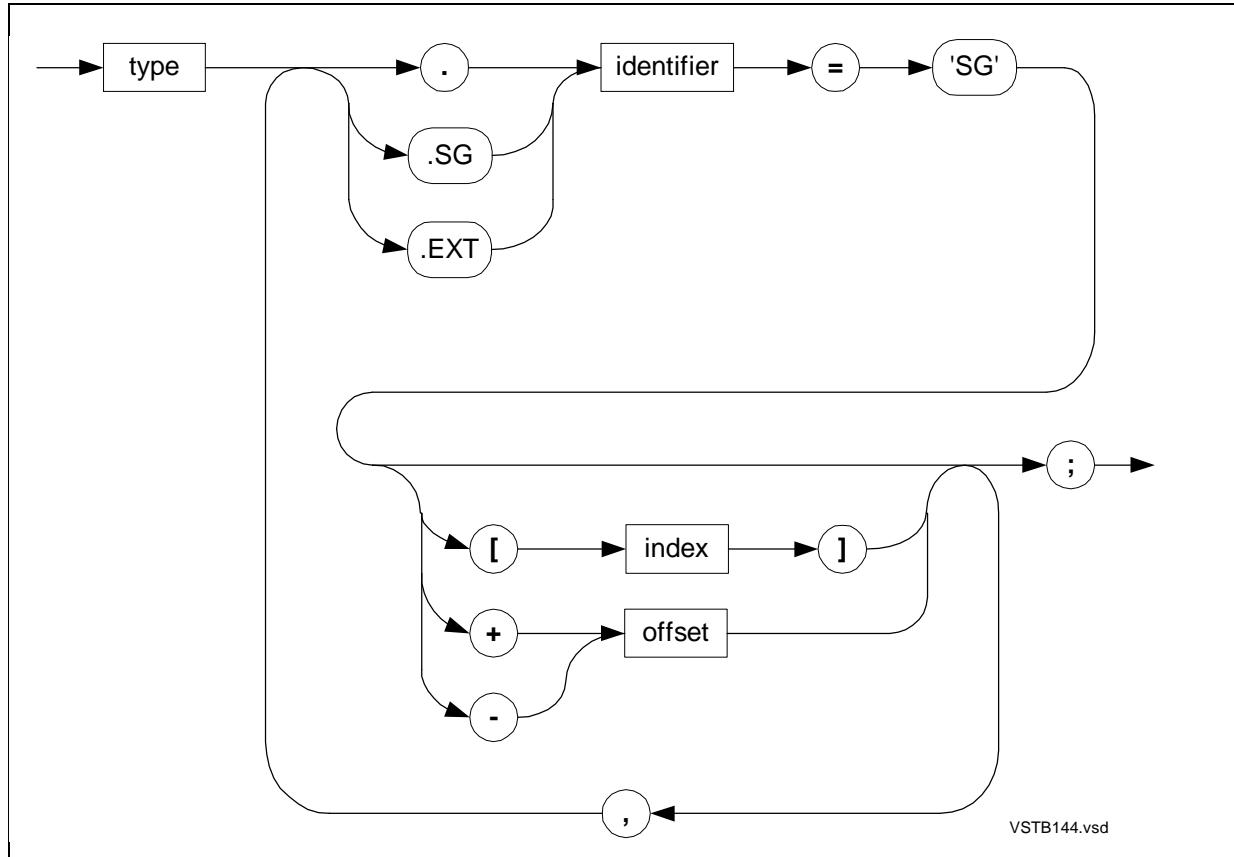
## 'SG'-Equivalenced Referral Structures

The 'SG'-equivalenced referral structure declaration associates a referral structure with a location relative to the base address of the system global data area.



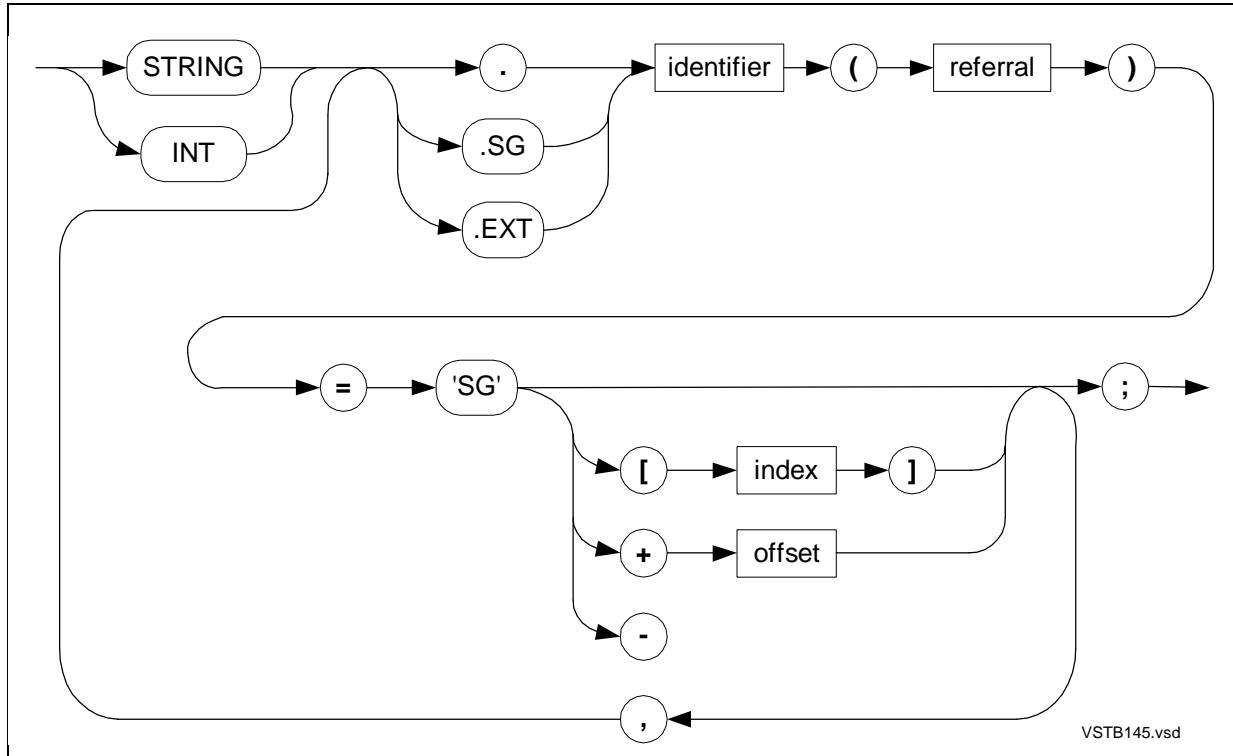
## 'SG'-Equivalenced Simple Pointers

The 'SG'-equivalenced simple pointer declaration associates a simple pointer with a location relative to the base address of the system global data area.



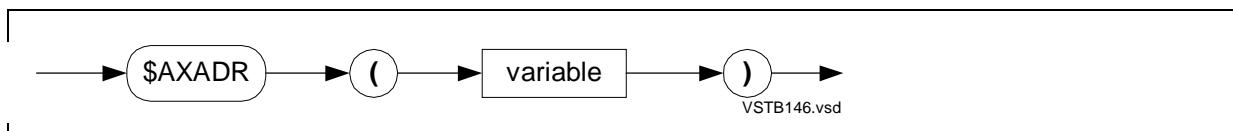
## 'SG'-Equivalenced Structure Pointers

The 'SG'-equivalenced structure pointer declaration associates a structure pointer with a location relative to the base address of the system global data area.



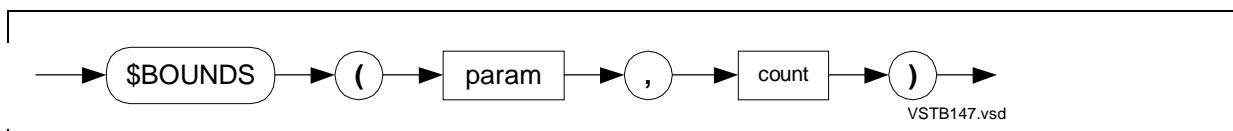
## **\$AXADR Function**

The \$AXADR function returns an absolute extended address.



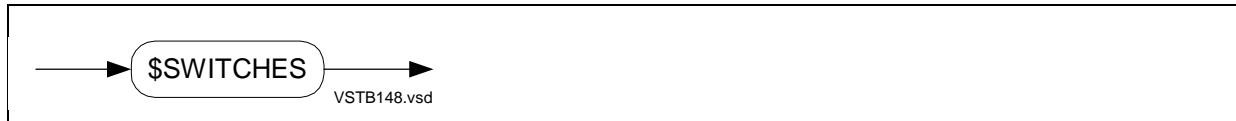
## \$BOUNDS Function

The \$BOUNDS function checks the location of a parameter passed to a system procedure to prevent a pointer that contains an incorrect address from overlaying the stack (S) register with data.



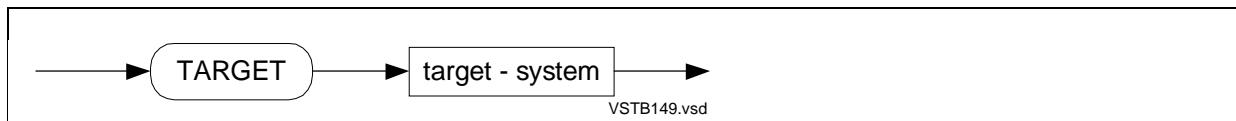
## \$SWITCHES Function

The \$SWITCHES function returns the current content of the switch register.



## TARGET Directive

The TARGET directive specifies the target system for which you have written conditional code. TARGET works in conjunction with the IF and ENDIF directives in D20 or later object files.

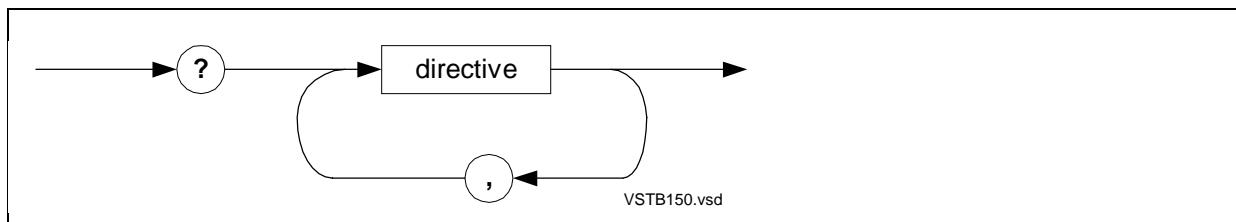


## Compiler Directives

The following syntax diagrams describe directive lines, followed by compiler directives in alphabetic order.

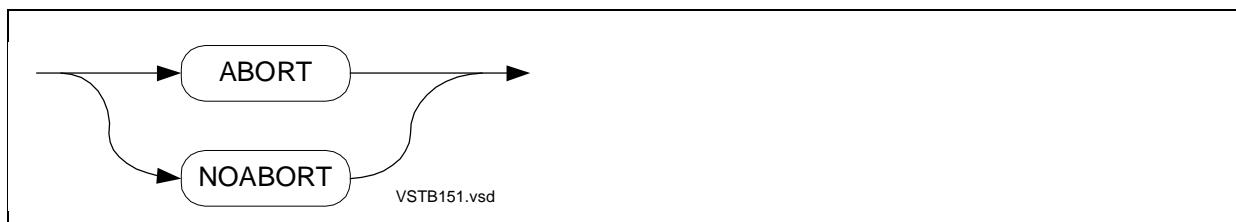
### Directive Lines

A directive line in your source code contains one or more compiler directives.



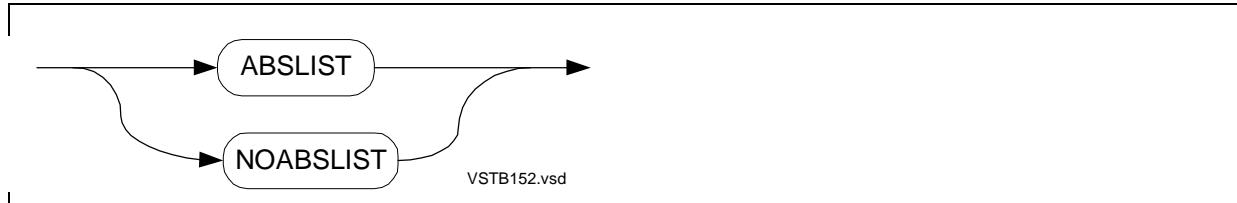
## ABORT Directive

The ABORT directive terminates compilation if the compiler cannot open a file specified in a SOURCE directive. The default is ABORT.



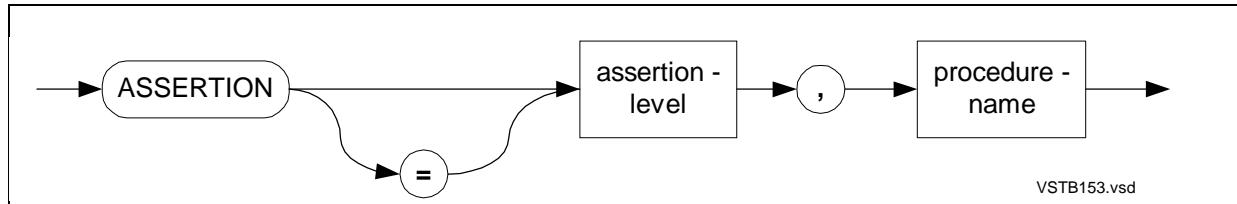
## ABSLIST Directive

ABSLIST lists code addresses relative to the code area base. The default is NOABSLIST.



## ASSERTION Directive

ASSERTION invokes a procedure when a condition defined in an ASSERT statement is true.



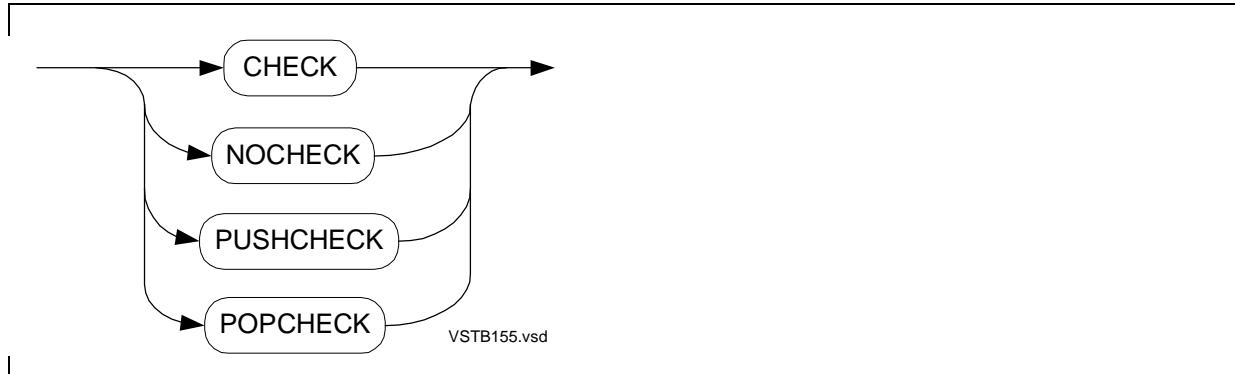
## BEGINCOMPIRATION Directive

BEGINCOMPIRATION marks the point in the source file where compilation is to begin if the USEGLOBALS directive is in effect.



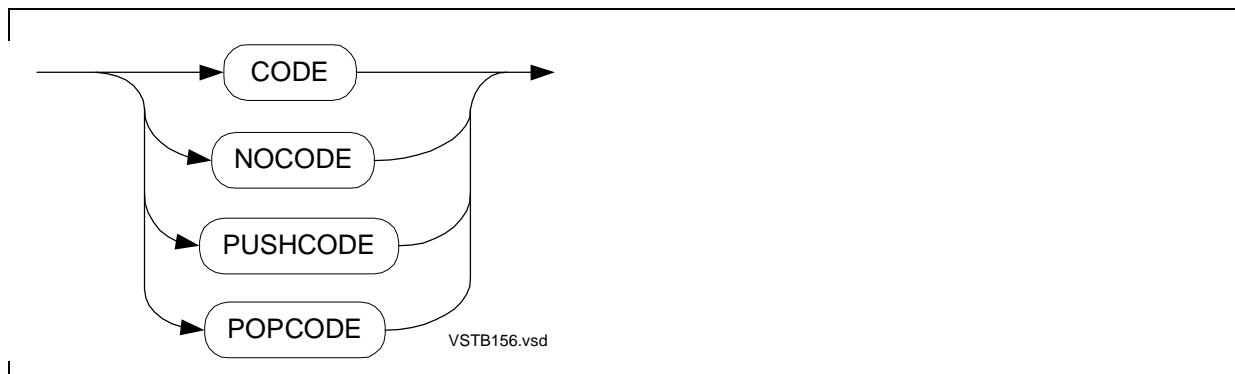
## CHECK Directive

CHECK generates range-checking code for certain features. The default is NOCHECK.



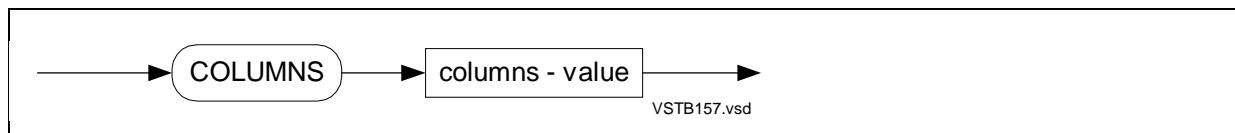
## CODE Directive

CODE lists instruction codes and constants in octal format after each procedure. The default is CODE.



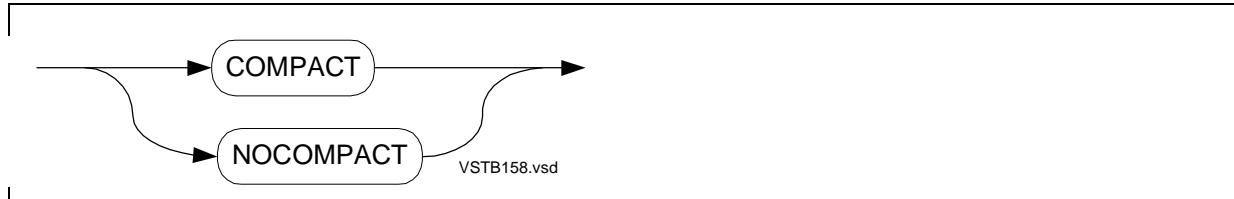
## COLUMNS Directive

COLUMNS directs the compiler to treat any text beyond the specified column as comments.



## COMPACT Directive

COMPACT moves procedures into gaps below the 32K-word boundary of the code area if they fit. The default is COMPACT.



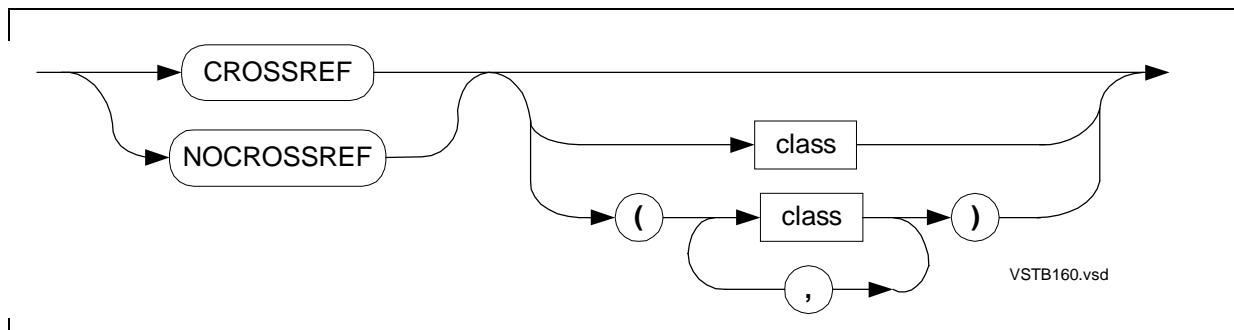
## CPU Directive

CPU specifies that the object file runs on a TNS system. (The need for this directive no longer exists. This directive has no effect on the object file and is retained only for compatibility with programs that still specify it.)



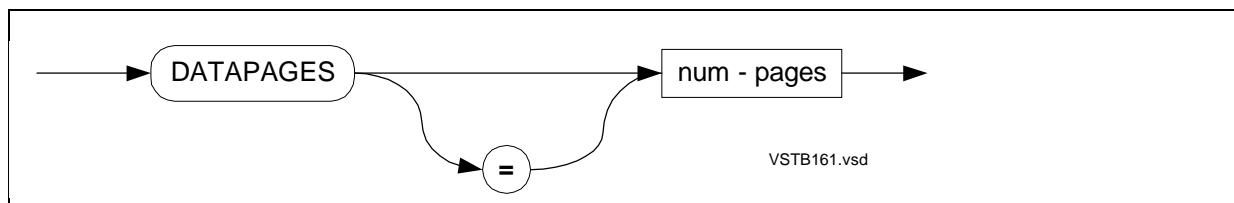
## CROSSREF Directive

CROSSREF collects source-level declarations and cross-reference information or specifies CROSSREF classes. The default is NOCROSSREF.



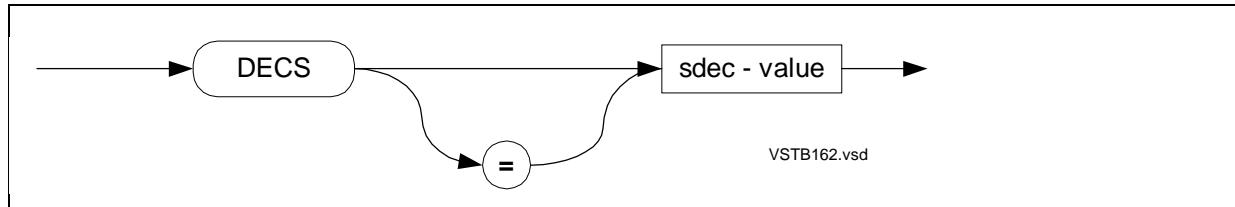
## DATAPAGES Directive

DATAPAGES sets the size of the data area in the user data segment.



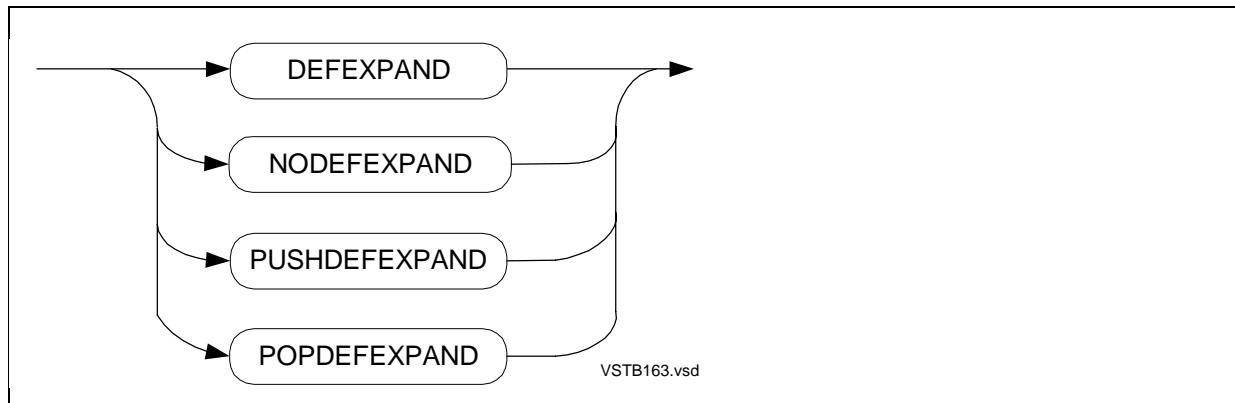
## DECS Directive

DECS decrements the compiler's internal S-register counter.



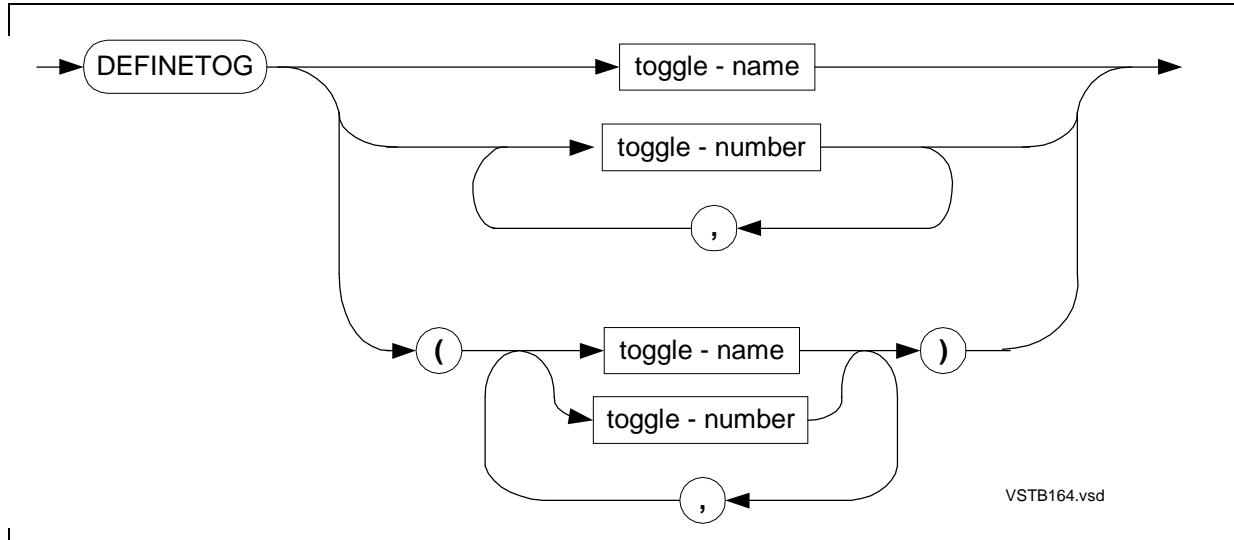
## DEFEXPAND Directive

Directive DEFEXPAND lists expanded DEFINEs and SQL-TAL code in the compiler listing. The default is NODEFEXPAND.



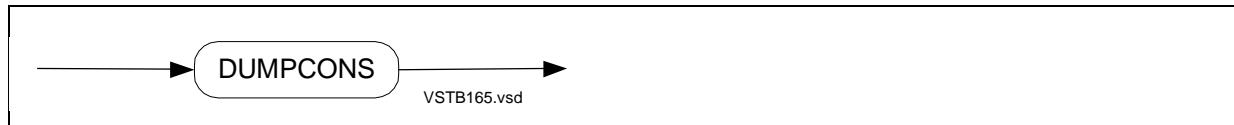
## DEFINETOG Directive

DEFINETOG specifies named or numeric toggles, without changing any prior settings, for use in conditional compilation. DEFINETOGL is a D20 or later feature.



## DUMPCONS Directive

DUMPCONS inserts the contents of the compiler's constant table into the object code.

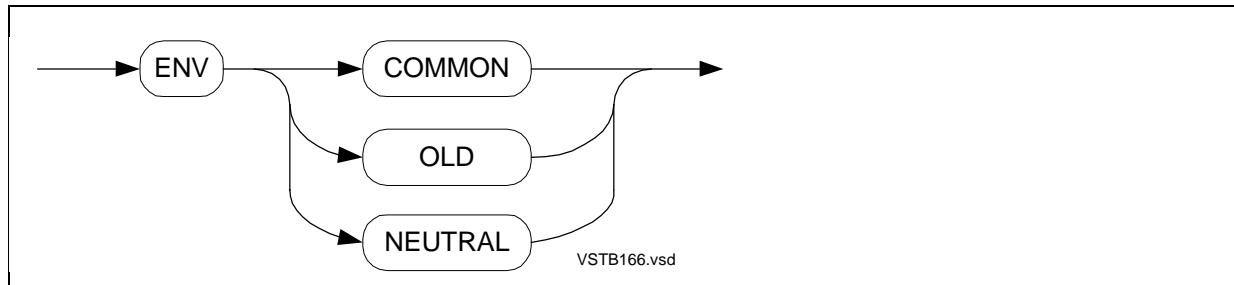


## ENDIF Directive

See [IF and ENDIF Directives](#) on page 16-47.

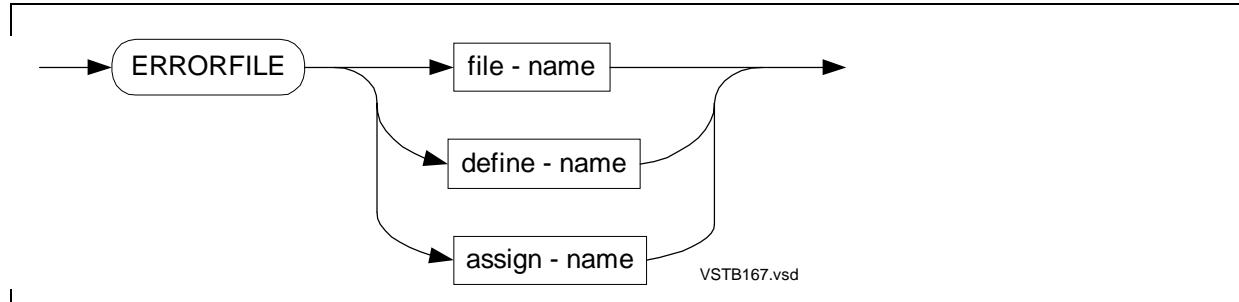
## ENV Directive

ENV specifies the intended run-time environment of a D-series object file. The default is ENV NEUTRAL.



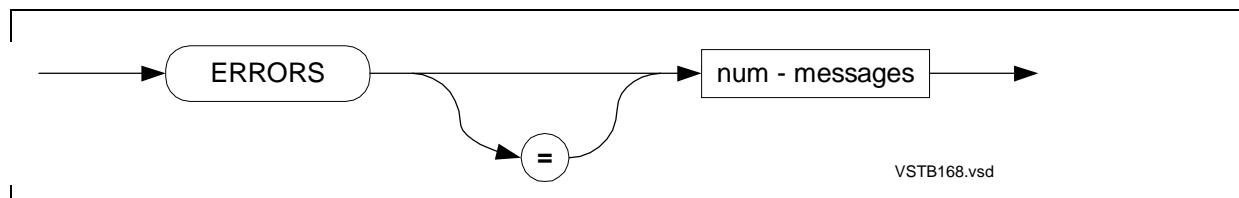
## ERRORFILE Directive

ERRORFILE logs compilation errors and warnings to an error file so you can use the TACL FIXERRS macro to view the diagnostic messages in one PS Text Edit window and correct the source file in another window.



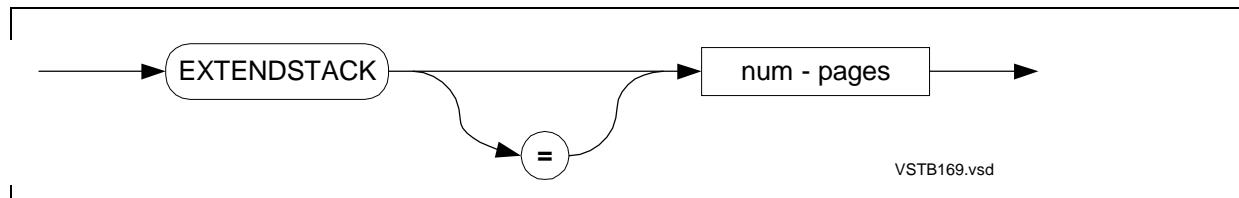
## ERRORS Directive

ERRORS sets the maximum number of error messages to allow before the compiler terminates the compilation.



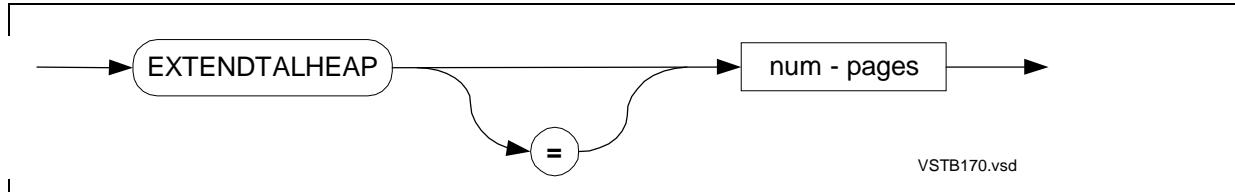
## EXTENDSTACK Directive

EXTENDSTACK increases the size of the data stack in the user data segment.



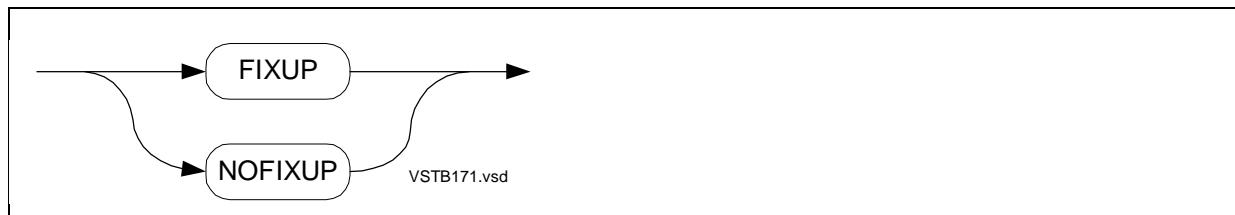
## EXTENDTALHEAP Directive

EXTENDTALHEAP increases the size of the compiler's internal heap for a D-series compilation unit.



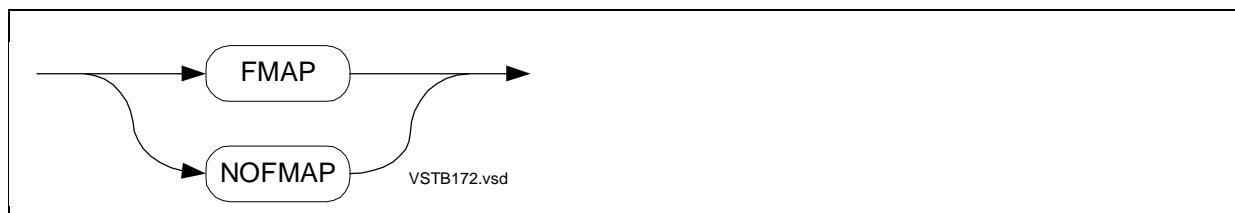
## FIXUP Directive

FIXUP directs BINSERV to perform its fixup step. The default is FIXUP.



## FMAP Directive

FMAP lists the file map. The default is NOFMAP.



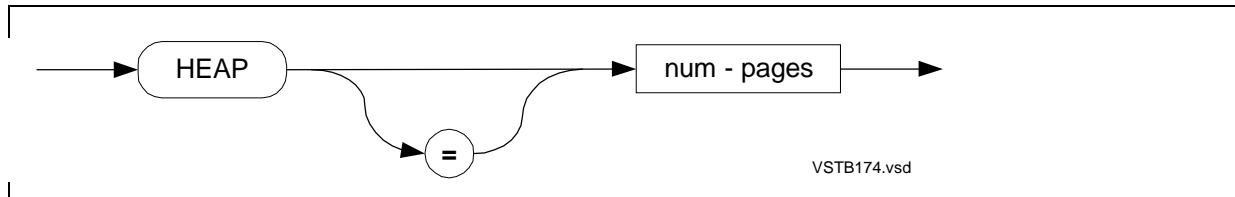
## GMAP Directive

GMAP lists the global map. The default is GMAP.



## HEAP Directive

HEAP sets the size of the CRE user heap for a D-series compilation unit if the ENV COMMON directive is in effect.



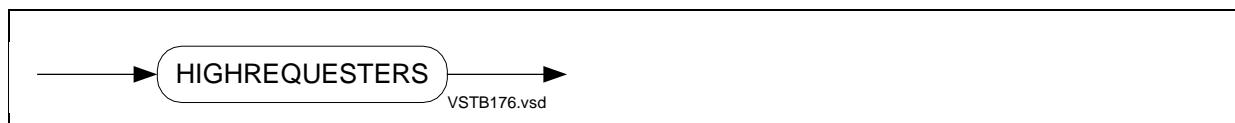
## HIGHPIN Directive

HIGHPIN sets the HIGHPIN attribute in a D-series object file.



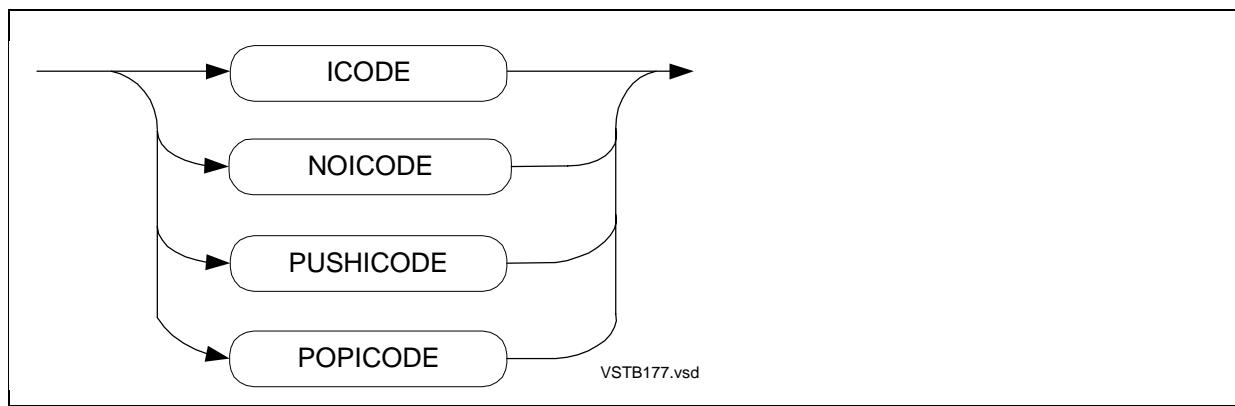
## HIGHREQUESTERS Directive

HIGHREQUESTERS sets the HIGHREQUESTERS attribute in a D-series object file.



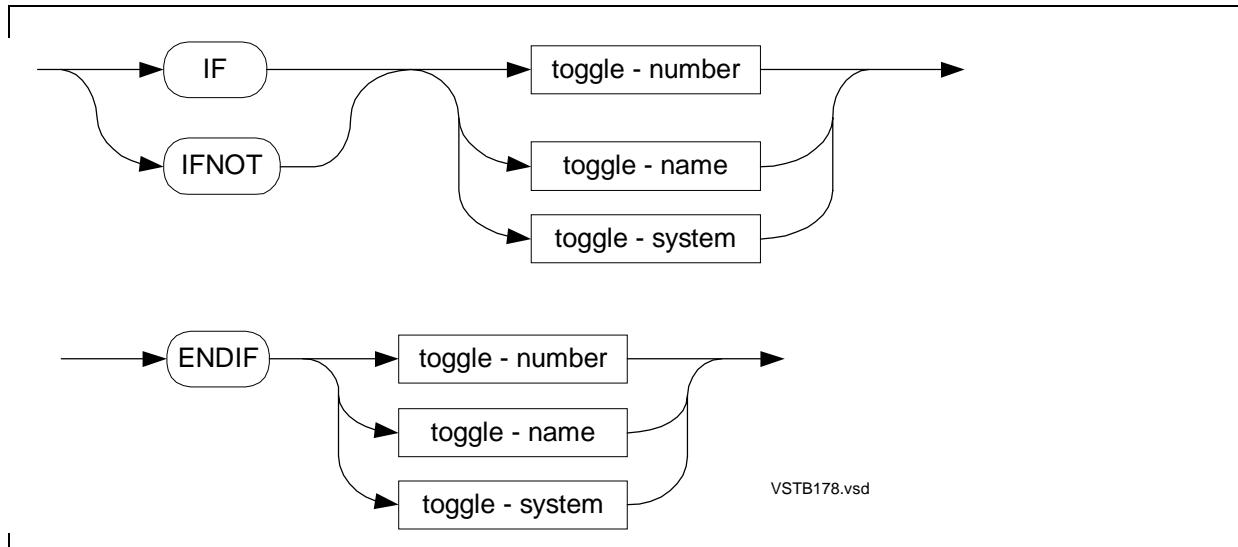
## ICODE Directive

ICODE lists the instruction-code (icode) mnemonics for subsequent procedures. The default is NOICODE.



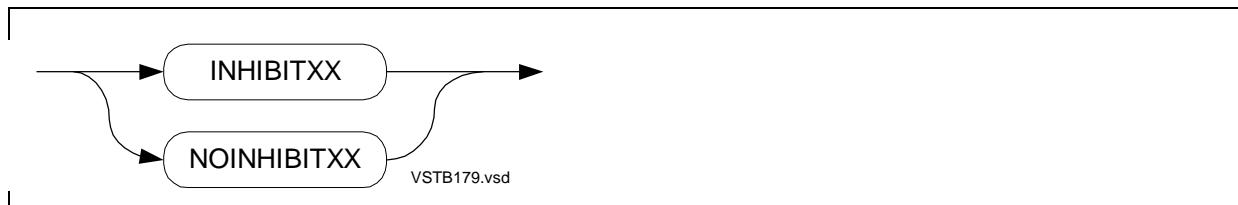
## IF and ENDIF Directives

IF and IFNOT control conditional compilation based on a condition. The ENDIF directive terminates the range of the matching IF or IFNOT directive. The D20 or later RVU supports named toggles and target-system toggles as well as numeric toggles.



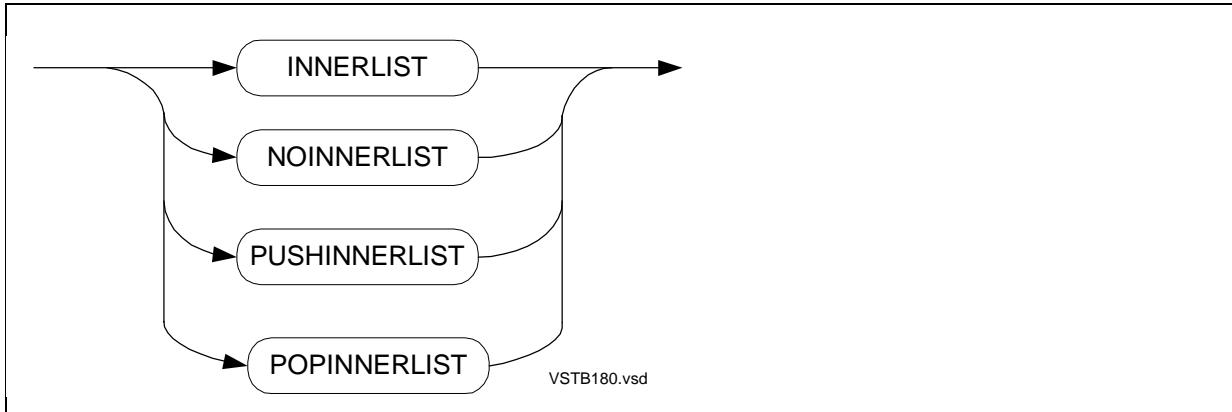
## INHIBITXX Directive

INHIBITXX generates inefficient but correct code for extended global declarations in relocatable blocks that Binder might locate after the first 64 words of the primary global area of the user data segment. The default is NOINHIBITXX.



## INNERLIST Directive

INNERLIST lists the instruction code mnemonics (and the compiler's RP setting) for each statement. The default is NOINNERLIST.



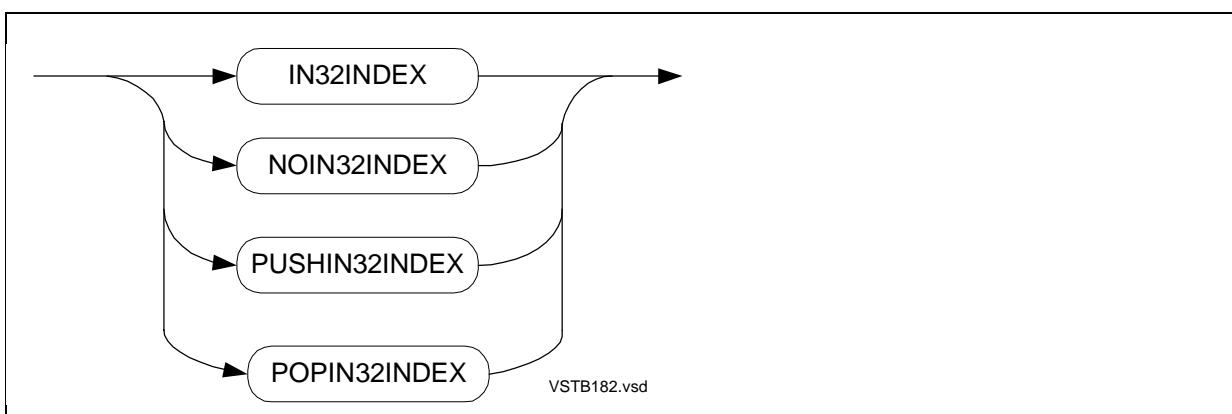
## INSPECT Directive

INSPECT sets the Inspect product as the default debugger for the object file. The default is NOINSPECT.



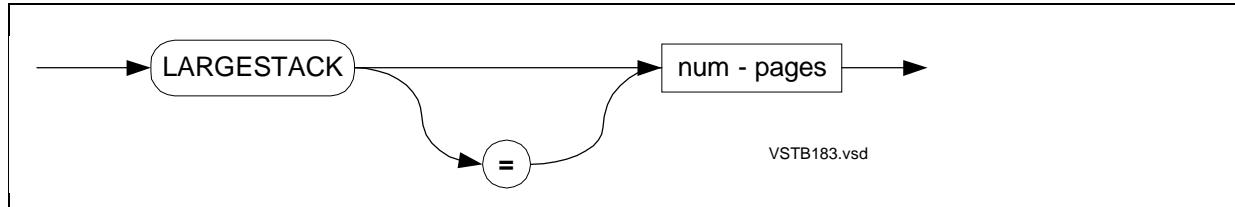
## INT32INDEX Directive

INT32INDEX generates INT(32) indexes from INT indexes for accessing items in an extended indirect structure in a D-series program. The default is NOINT32INDEX.



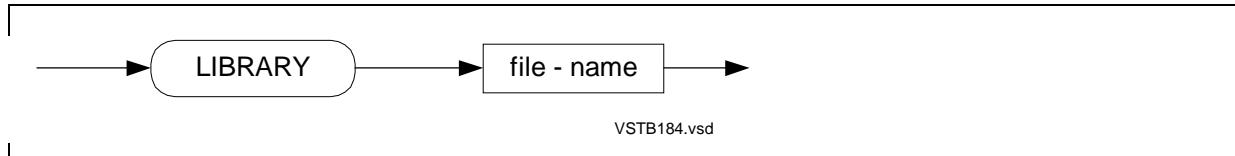
## LARGESTACK Directive

LARGESTACK sets the size of the extended stack in the automatic extended data segment.



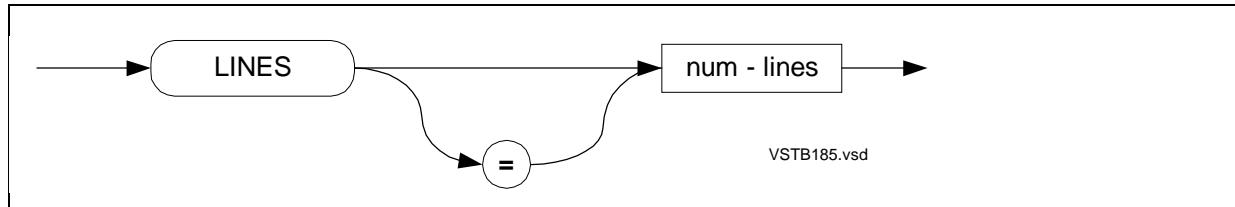
## LIBRARY Directive

LIBRARY specifies the name of the TNS software user run-time library to be associated with the object file at run time.



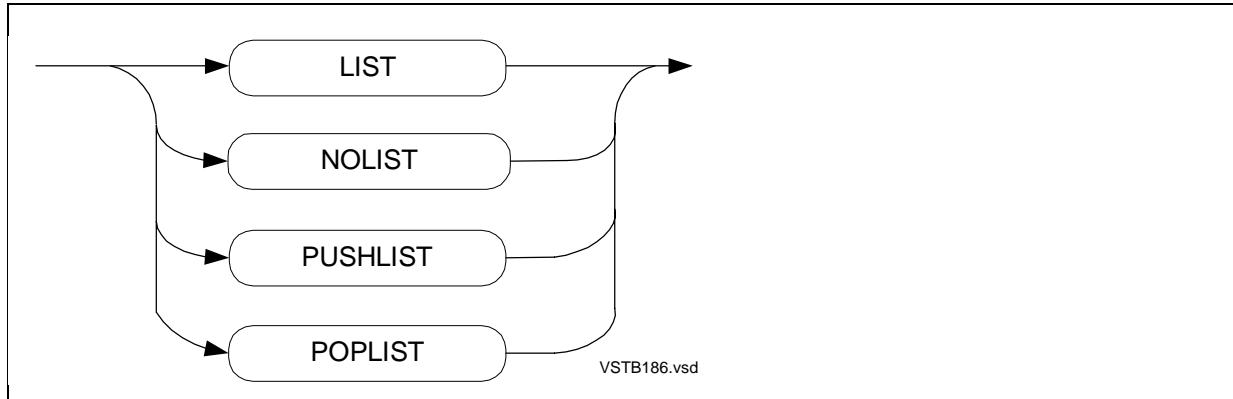
## INES Directive

INES sets the maximum number of output lines per page.



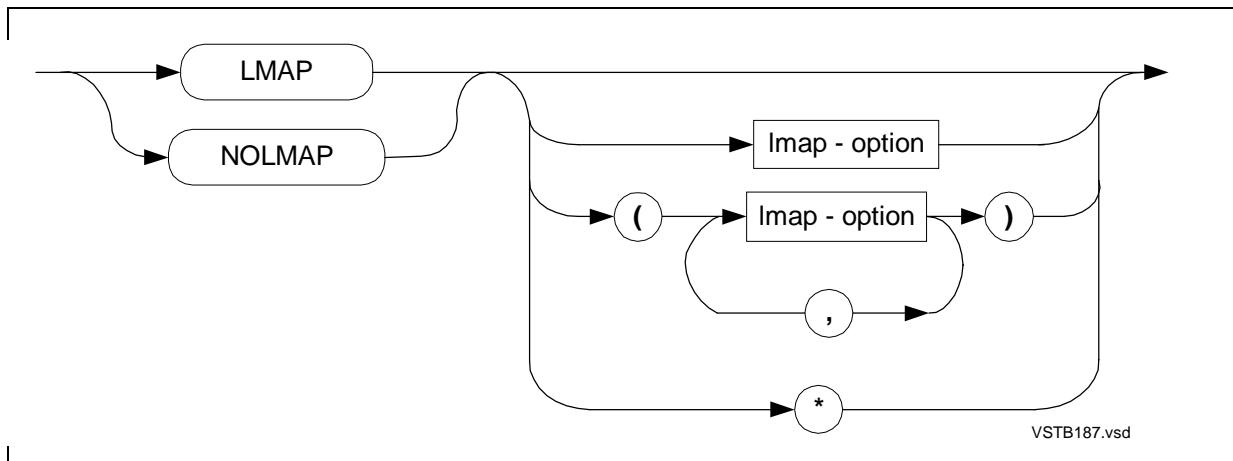
## LIST Directive

LIST lists the source text for subsequent source code if NOSUPPRESS is in effect. The default is LIST.



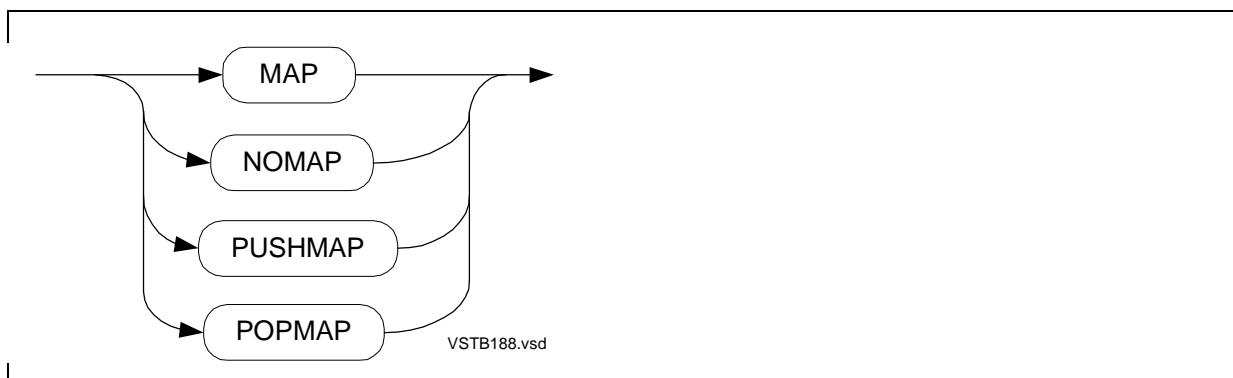
## LMAP Directive

LMAP lists load-map and cross-reference information. The default is LMAP ALPHA.



## MAP Directive

MAP lists the identifier maps. The default is MAP.



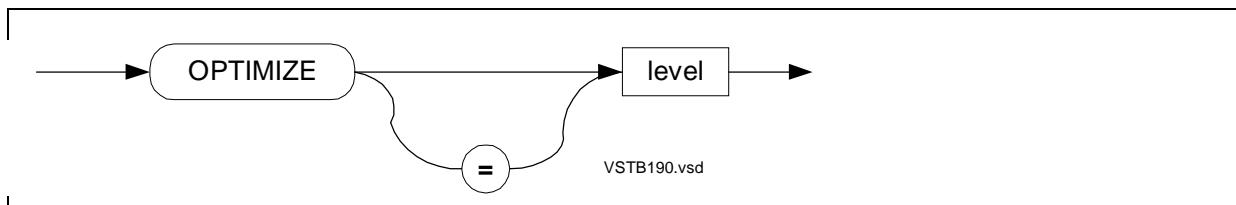
## OLDFLTSTDFUNC Directive

OLDFLTSTDFUNC treats arguments to the \$FLT, \$FLTR, \$EFLT, and \$EFLTR standard functions as if they were FIXED(0) values.



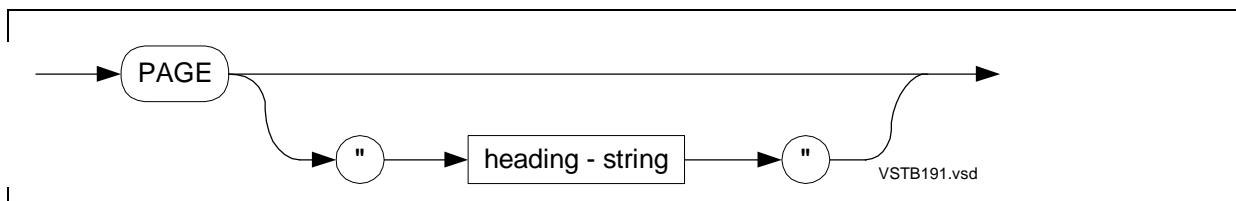
## OPTIMIZE Directive

OPTIMIZE specifies the level at which the compiler optimizes the object code.



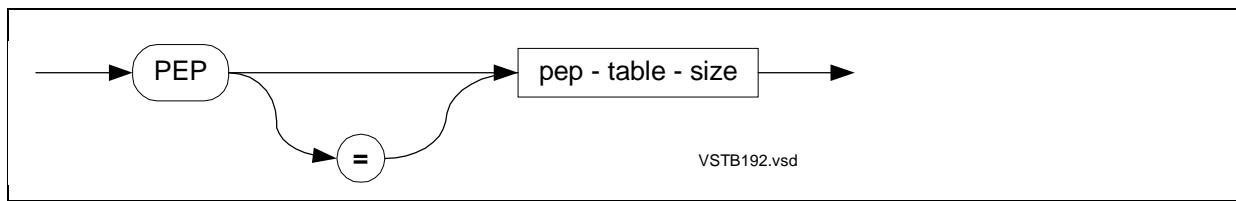
## PAGE Directive

PAGE optionally prints a heading and causes a page eject.



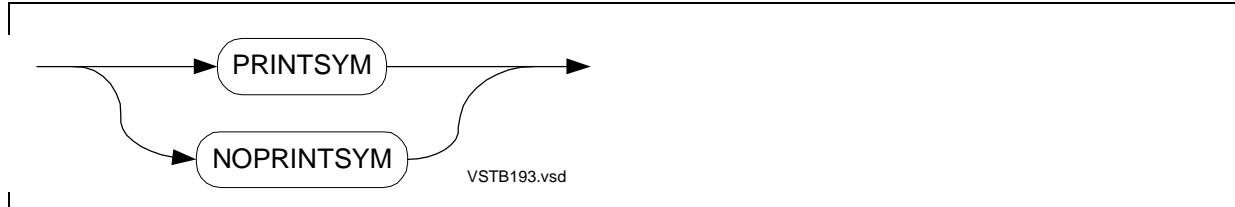
## PEP Directive

PEP specifies the size of the procedure entry-point (PEP) table.



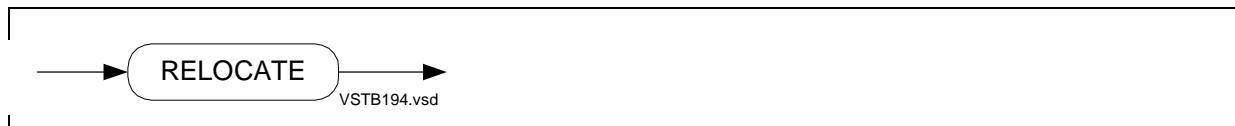
## PRINTSYM Directive

PRINTSYM lists symbols. The default is PRINTSYM.



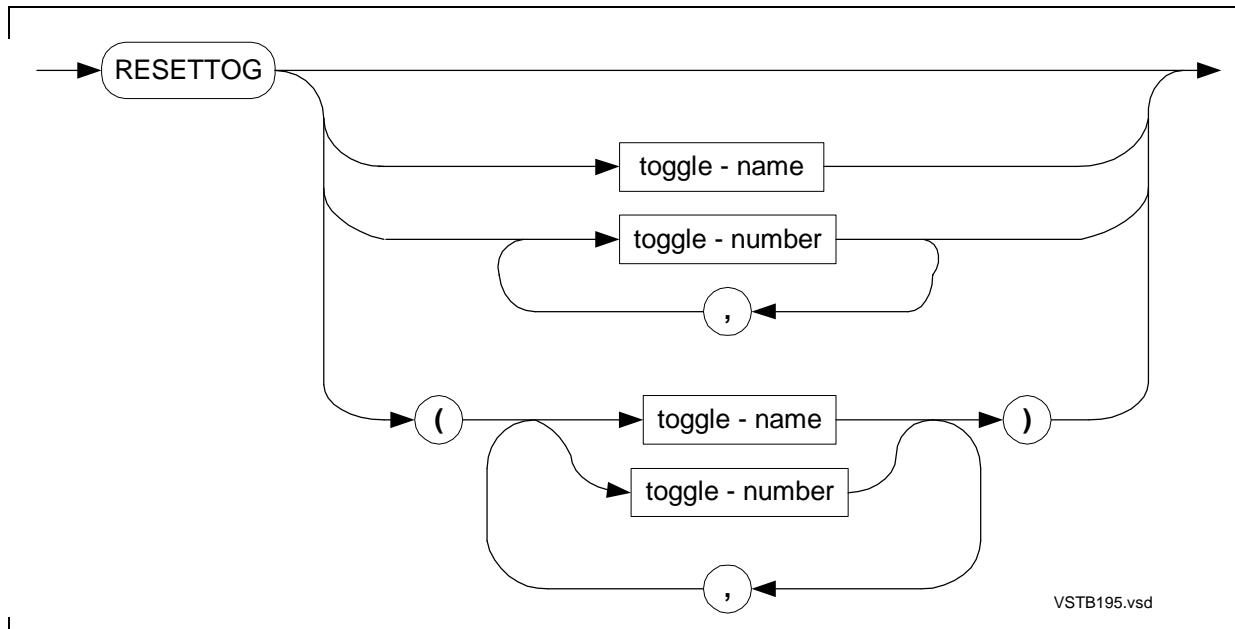
## RELOCATE Directive

RELOCATE lists BINSERV warnings for declarations that depend on absolute addresses in the primary global data area of the user data segment.



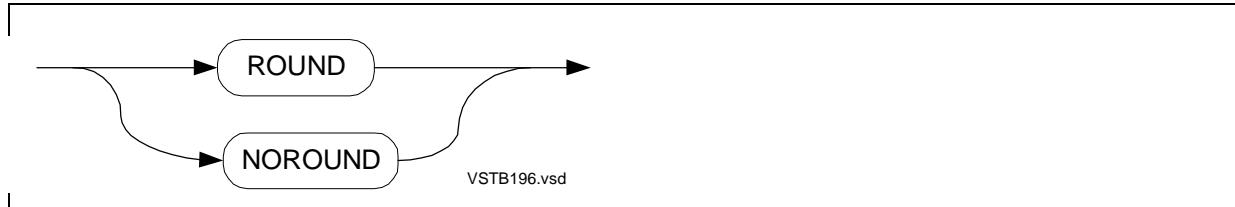
## RESETTOG Directive

RESETTOG creates new toggles in the off state and turns off toggles created by SETTOG. The RESETTOG directive supports named toggles as well as numeric toggles.



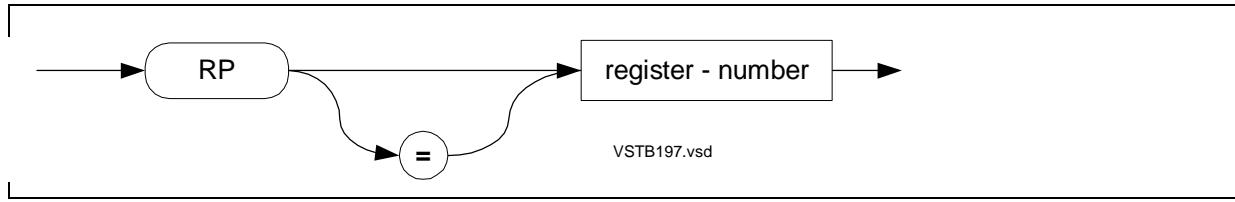
## ROUND Directive

ROUND rounds FIXED values assigned to FIXED variables that have smaller fpoint values than the values you are assigning. The default is NOROUND.



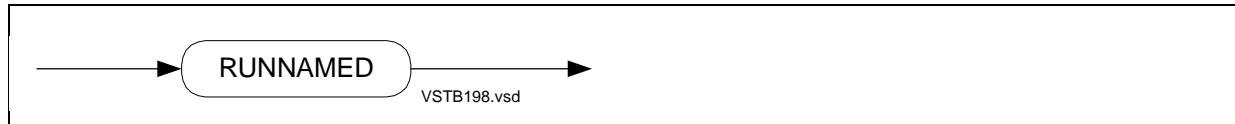
## RP Directive

RP sets the compiler's internal register pointer (RP) count. RP tells the compiler how many registers are currently in use on the register stack.



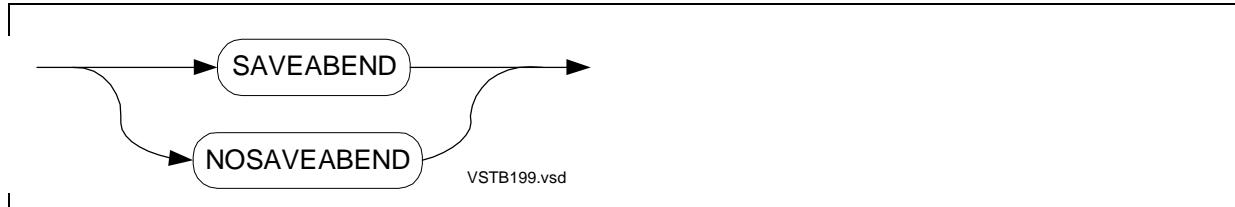
## RUNNAMED Directive

RUNNAMED causes a D-series object file to run on a D-series system as a named process even if you do not provide a name for it.



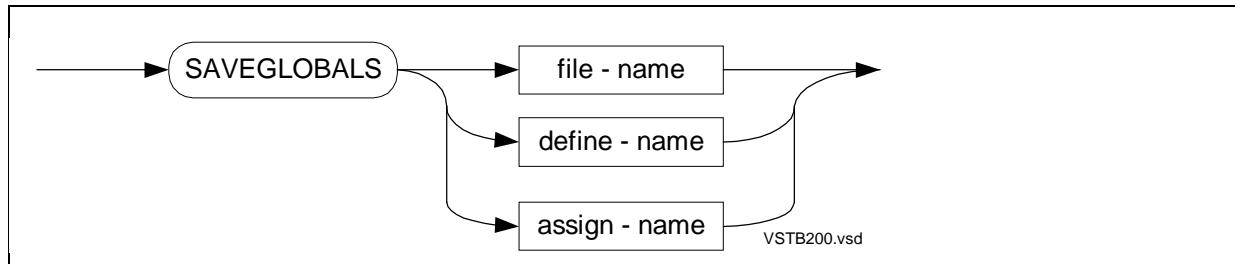
## SAVEABEND Directive

SAVEABEND directs the Inspect product to generate a save file if your process terminates abnormally during execution. The default is NOSAVEABEND.



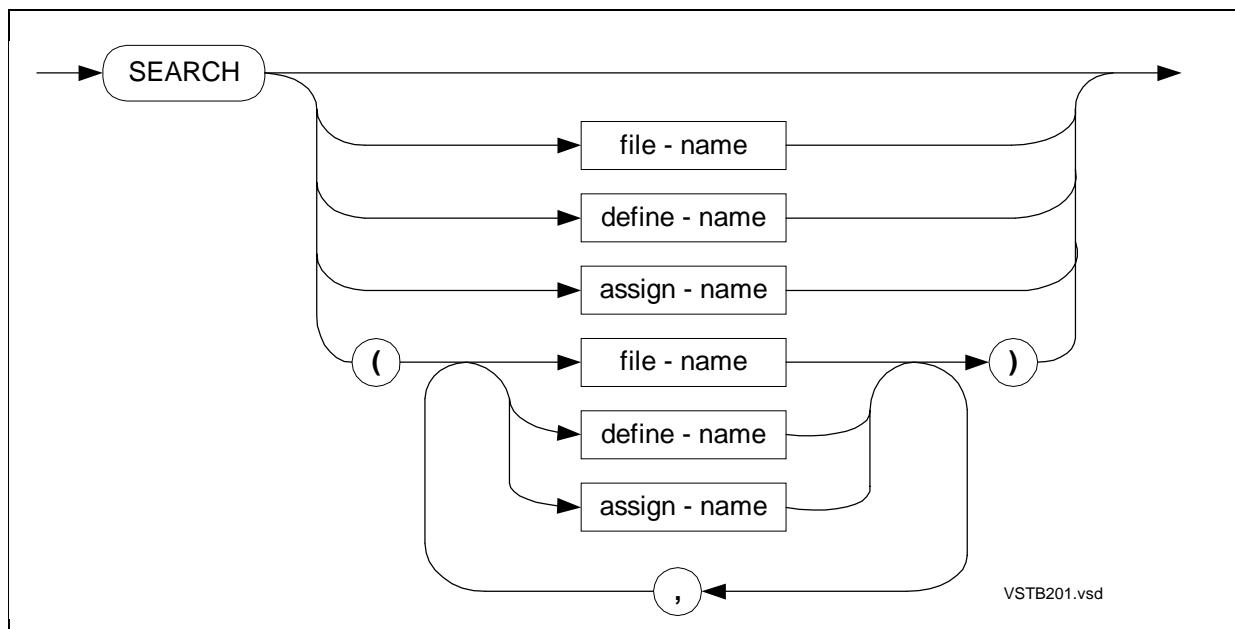
## SAVEGLOBALS Directive

SAVEGLOBALS saves all global data declarations in a file for use in subsequent compilations that specify the USEGLOBALS directive.



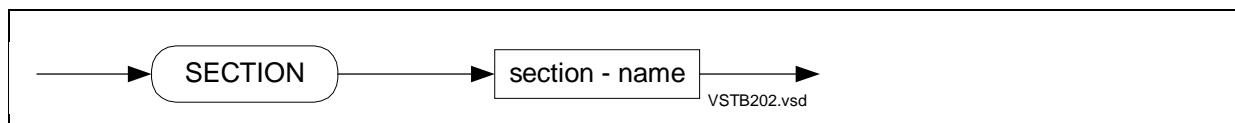
## SEARCH Directive

SEARCH specifies object files from which BINSERV can resolve unsatisfied external references and validate parameter lists at the end of compilation. By default, BINSERV does not attempt to resolve unsatisfied external references.



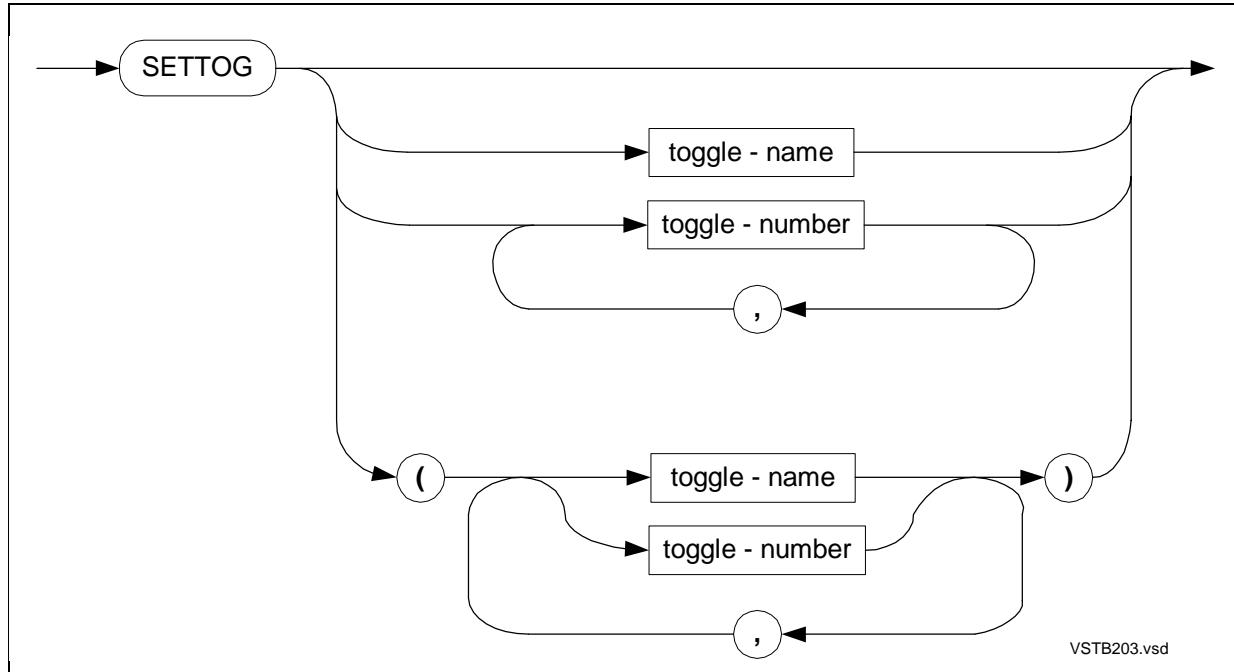
## SECTION Directive

SECTION gives a name to a section of a source file for use in a SOURCE directive.



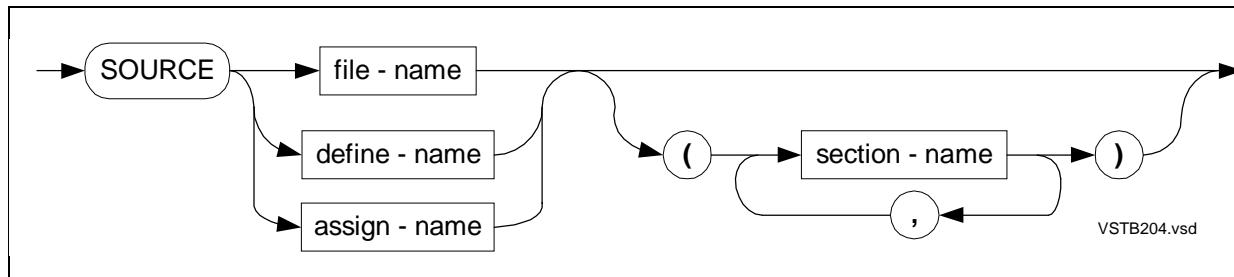
## SETTOG Directive

SETTOG turns the specified toggles on for use in conditional compilations. The SETTOG directive supports named toggles as well as numeric toggles.



## SOURCE Directive

SOURCE specifies source code to include from another source file.



## SQL Directive

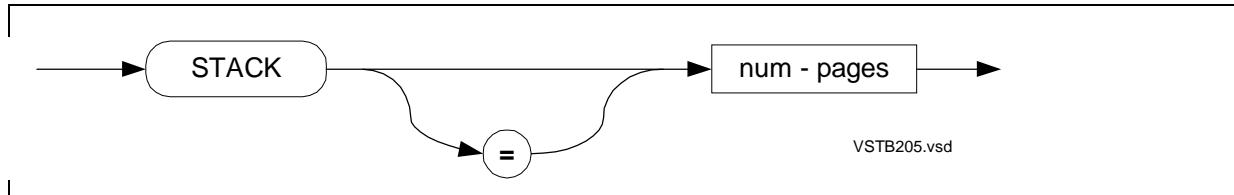
See the *NonStop SQL Programming Manual for TAL*.

## SQLMEM Directive

See the *NonStop SQL Programming Manual for TAL*.

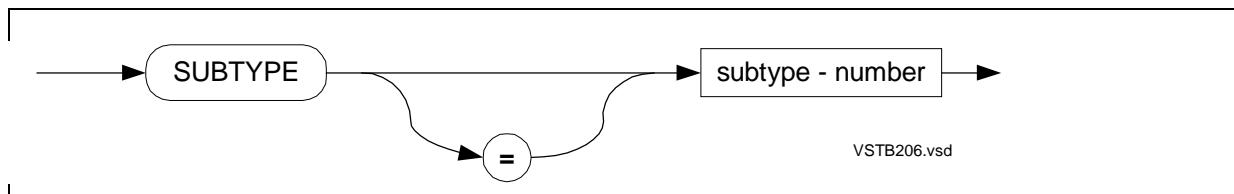
## STACK Directive

STACK sets the size of the data stack in the user data segment.



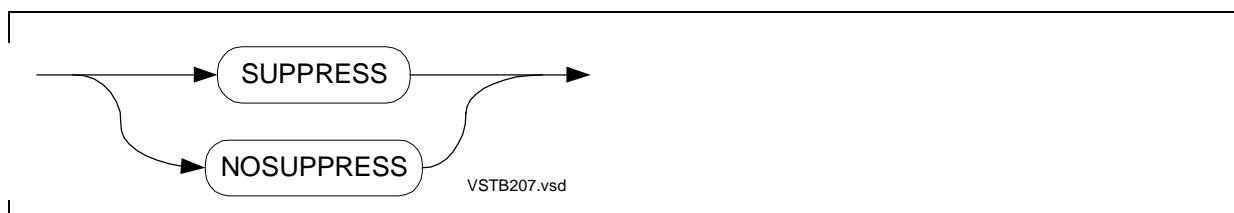
## SUBTYPE Directive

SUBTYPE specifies that the object file is to execute as a process of a specified subtype.



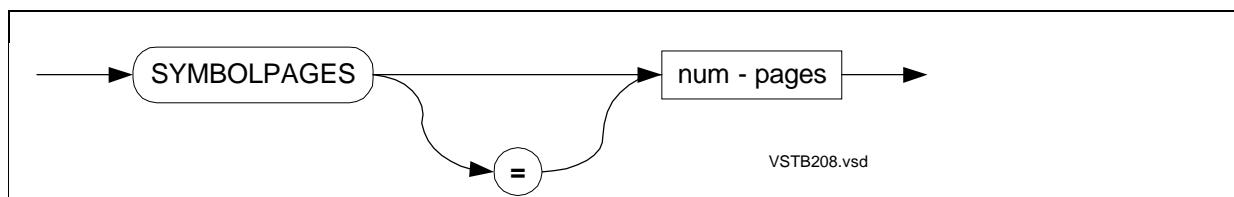
## SUPPRESS Directive

SUPPRESS overrides all the listing directives. The default is NOSUPPRESS.



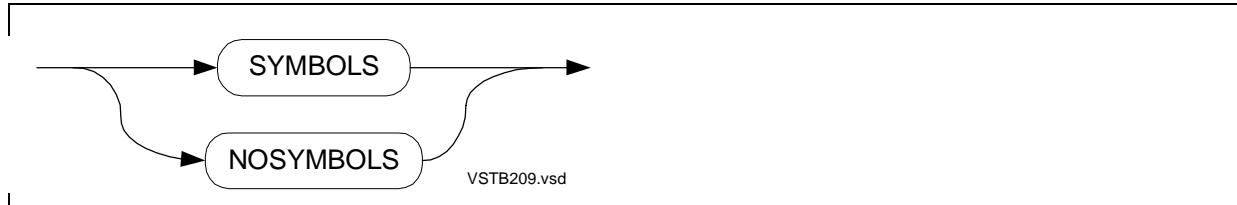
## SYMBOLPAGES Directive

SYMBOLPAGES sets the size of the internal symbol table the compiler uses as a temporary storage area for processing variables and SQL statements.



## SYMBOLS Directive

SYMBOLS saves symbols in a symbol table (for Inspect symbolic debugging) in the object file. The default is NOSYMBOLS.



## SYNTAX Directive

SYNTAX checks the syntax of the source text without producing an object file.

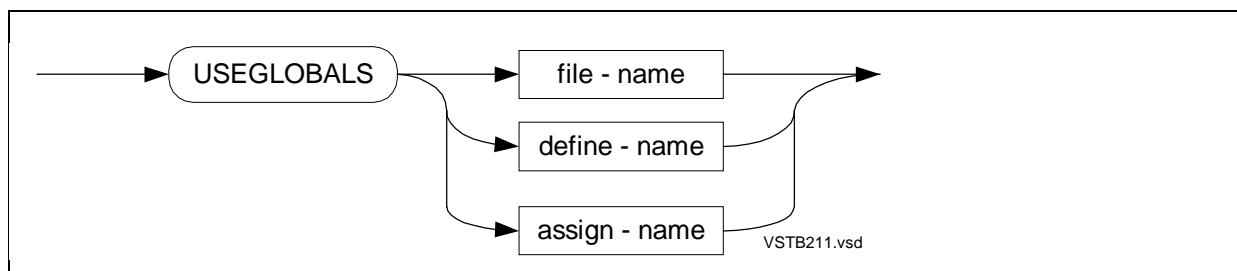


## TARGET Directive

See [Section 15, Privileged Procedures](#).

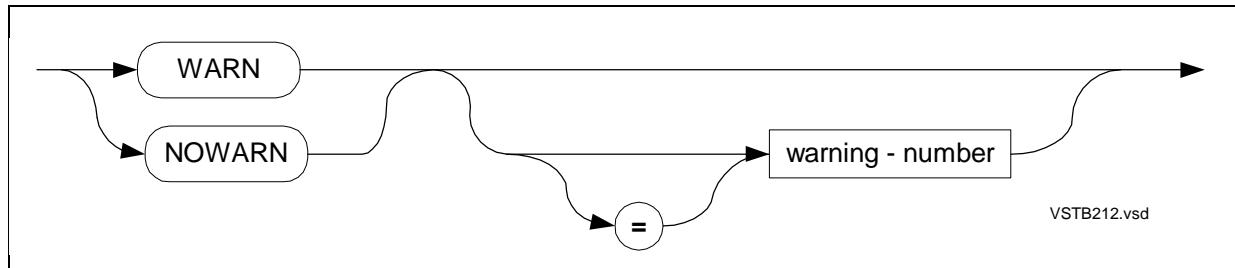
## USEGLOBALS Directive

Directive USEGLOBALS retrieves the global data declarations saved in a file by SAVEGLOBALS during a previous compilation.



## WARN Directive

WARN instructs the compiler to print a selected warning or all warnings. The default is WARN.



# C

## TAL Syntax Summary (Bracket-and-Brace Diagrams)

This appendix provides a syntax summary of bracket-and-brace diagrams for specifying:

- Constants
- Expressions
- Declarations
- Statements
- Standard Functions
- Compiler Directives

### General Syntax Notation

In this appendix, the following syntax notation conventions are used in the format of bracket-and-brace diagrams.

#### UPPERCASE LETTERS

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

SOURCE

#### lowercase italic letters

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

*file-name*

#### Brackets [ ]

Brackets enclose optional syntax items. For example:

[ base ] integer

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

[ + ] *operand*  
 [ - ]  
 [ + | - ] *operand*

## Braces { }

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

{ E } *exponent*  
 { L }  
 { E | L } *exponent*

## Vertical Line |

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

[ + | - ] *operand*

## Ellipsis ...

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

[ { AND } [ NOT ] *condition* ] ... ]  
 { OR }

## Punctuation

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

STRUCT *identifier* (\*) ;

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

" [ " *repetition-constant-list* " ] "

## Item Spacing

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

STRUCT *identifier* (\*) ;

If there is no space between two items, spaces are not permitted. In the following examples, there are no spaces permitted between the items:

```
[ NO ] SUPPRESS
int-expression.< left-bit[: right-bit]>
```

## Line Spacing

Continuation lines are indented and separated from the preceding line by a blank line. This spacing distinguishes items in a continuation line from items in a vertical list of selections. For example:

```
STRUCT [ . . ] identifier ( referral )
[ .EXT]
[ " [ " lower-bound : upper-bound " ] " ] ;
```

## Constants

The following syntax diagrams describe:

- Character string constants (all data types)
- STRING numeric constants
- INT numeric constants
- INT(32) numeric constants
- FIXED numeric constants
- REAL and REAL(64) numeric constants
- Constant lists

### Character String Constants

A character string constant consists of one or more ASCII characters stored in a contiguous group of bytes.

“*string*”

### STRING Numeric Constants

A STRING numeric constant is an unsigned 8-bit integer.

[ *base* ] *integer*

## INT Numeric Constants

An INT numeric constant is a signed or unsigned 16-bit integer.

[ + ] [ base ] <i>integer</i>
[ - ]

## INT(32) Numeric Constants

An INT(32) numeric constant is a signed or unsigned 32-bit integer.

[ + ] [ base ] <i>integer</i> { D }
[ - ] { %D }

## FIXED Numeric Constants

A FIXED numeric constant is a signed 64-bit fixed-point integer.

[ + ] [ base ] <i>integer</i> [ . <i>fraction</i> ] { F }
[ - ] { %F }

## REAL and REAL(64) Numeric Constants

A REAL numeric constant is a signed 32-bit floating-point number that is precise to approximately 7 significant digits.

A REAL(64) numeric constant is a signed 64-bit floating-point number that is precise to approximately 17 significant digits.

[ + ] <i>integer. fraction</i> { E } [ + ] <i>exponent</i>
[ - ] { L } [ - ]

## Constant Lists

A constant list is a list of one or more constants.

{ <i>repetition-constant-list</i> }
{ "[" <i>repetition-constant-list</i> "]" }
{ "[" <i>constant-list-seq</i> "]" }

*repetition-constant-list* has the form:

[ *repetition-factor*\* ] "[" *constant-list-seq* "]"

*constant-list-seq* has the form:

```
{ constant }
{ repetition-constant-list }

[ , { constant } ] ...
{ repetition-constant-list }
```

## Expressions

The following syntax diagrams describe:

- Arithmetic expressions
- Conditional expressions
- Assignment expressions
- CASE expressions
- IF expressions
- Group comparison expressions
- Bit extractions
- Bit shifts

### Arithmetic Expressions

An arithmetic expression is a sequence of operands and arithmetic operators that computes a single numeric value of a specific data type.

[ + ] operand [ [ arithmetic-operator operand ] ... ]
[ - ]

### Conditional Expressions

A conditional expression is a sequence of conditions and Boolean or relational operators that establishes the relationship between values.

[ NOT ] condition [ [ { AND } [ NOT ] condition ] ... ]
{ OR }

### Assignment Expressions

The assignment expression assigns the value of an expression to a variable.

variable := [ variable := ] ... expression
--

## CASE Expressions

The CASE expression selects one of several expressions.

```
CASE selector OF BEGIN expression ; [ expression ; ] ...
[ OTHERWISE expression ; ] END
```

## IF Expressions

The IF expression conditionally selects one of two expressions, usually for assignment to a variable.

```
IF condition THEN expression ELSE expression
```

## Group Comparison Expressions

The group comparison expression compares a variable with a variable or a constant.

```
var1 relational-operator
{ var2 FOR count [ count-unit ] } [ -> next-addr ]
{ constant }
{ "[" constant "]" }
{ constant-list }
```

## Bit Extractions

A bit extraction accesses a bit field in an INT expression without altering the expression.

```
int-expression < left-bit [: right-bit]>
```

## Bit Shifts

A bit shift operation shifts a bit field a specified number of positions to the left or to the right within a variable without altering the variable.

```
{ int-expression } shift-operator positions
{ dbl-expression }
```

## Declarations

Declaration syntax diagrams describe:

- LITERALs and DEFINEs

- Simple variables
- Arrays and read-only arrays
- Structures—definition structures, template structures, referral structures
- Structure items—simple variables, arrays, substructures, filler bytes, filler bits, simple pointers, structure pointers, and redefinitions
- Simple pointers and structure pointers
- Equivalenced variables
- NAMEs and BLOCKs
- Procedures and subprocedures

## LITERAL and DEFINE Declarations

The following syntax diagrams describe LITERAL and DEFINE declarations.

### LITERALS

A LITERAL associates an identifier with a constant.

```
LITERAL identifier [ = constant ]
[ , identifier [ = constant ] ] . . . ;
```

### DEFINES

A DEFINE associates an identifier (and parameters if any) with text.

```
DEFINE identifier [ ( param-name [ , param-name ] . . . ) ]
= define-body #
[ , identifier [ ( param-name [ , param-name ] . . . ) ]
= define-body # ] . . . ;
```

## Simple Variable Declarations

The following syntax diagram describes simple variable declarations:

### Simple Variables

A simple variable associates an identifier with a single-element data item of a specified data type.

```
type identifier [ := initialization ]
[ , identifier [ := initialization ] ] . . . ;
```

*type* is one of:

```
{ STRING }
{ { INT | REAL } [ ( width ) ] }
{ UNSIGNED ( width ) }
{ FIXED [ ( fpoint | * ) ] }
```

## Array Declarations

The following syntax diagrams describe array and read-only array declarations:

### Arrays

An array associates an identifier with a one-dimensional set of elements of the same data type.

```
type [ . . . ] identifier " [ " lower-bound : upper-bound " ] "
[ .EXT ]
[ := initialization ]
[ , [ . . . ] identifier " [ " lower-bound : upper-bound " ] "
[ .EXT ]
[ := initialization ] ] . . . ;
```

### Read-Only Arrays

A read-only array associates an identifier with a one-dimensional and nonmodifiable set of elements of the same data type. A read-only array is located in a user code segment.

```
type identifier [ "[" lower-bound : upper-bound "]" ]
= 'P' := initialization
[ , identifier [ "[" lower-bound : upper-bound "]" ]
= 'P' := initialization ] . . . ;
```

*type* is one of:

```
{ STRING
{ { INT | REAL } [ ( width ) ] }
{ UNSIGNED ( width ) }
{ FIXED [ ( fpoint | * ) ] }
```

# Structure Declarations

The following syntax diagrams describe:

- Structures—definition structures, template structures, referral structures
- Structure items—simple variables, arrays, substructures, filler bytes, filler bits, simple pointers, structure pointers, and redefinitions

## Definition Structures

A definition structure associates an identifier with a structure layout and allocates storage for it.

```
STRUCT [ . ] identifier
      [ .EXT ]
      [ "[" lower-bound : upper-bound "]" ];
      structure-layout;
```

## Template Structures

A template structure associates an identifier with a structure layout but allocates no storage for it.

```
STRUCT identifier (*) ;
      structure-layout;
```

## Referral Structures

A referral structure associates an identifier with a structure whose layout is the same as a previously declared structure and allocates storage for it.

```
STRUCT [ . ] identifier ( referral )
      [ . EXT ]
      [ "[" lower-bound : upper-bound "]" ];
```

## Simple Variables Declared in Structures

A simple variable can be declared inside a structure.

```
type identifier [ , identifier ] ... ;
```

## Arrays Declared in Structures

An array can be declared inside a structure.

```
type identifier" [ " lower-bound : upper-bound " ] "
[., identifier" [ " lower-bound : upper-bound " ] " . . . ;
```

## Definition Substructures

A definition substructure can be declared inside a structure.

```
STRUCT identifier
[ " [ " lower-bound : upper-bound " ] " ] ; substructure-layout;
```

## Referral Substructures

A referral substructure can be declared inside a structure.

```
STRUCT identifier( referral )
[ " [ " lower-bound : upper-bound " ] " ] ;
```

## Fillers in Structures

A filler is a byte or bit place holder in a structure.

```
{ FILLER } constant-expression ;
{ BIT_FILLER }
```

## Simple Pointers Declared in Structures

A simple pointer can be declared inside a structure.

```
type { . . . } identifier
{ .EXT }
[ , { . . . } identifier ] . . . ;
{ .EXT }
```

## Structure Pointers Declared in Structures

A structure pointer can be declared inside a structure.

```
{ STRING } { . . . } identifier( referral )
{ INT } { .EXT }
[ , { . . . } identifier( referral ) ] . . . ;
{ .EXT }
```

## Simple Variable Redefinitions

A simple variable redefinition associates a new simple variable with a previous item at the same BEGIN-END level of a structure.

```
type identifier = previous-identifier ;
```

## Array Redefinitions

An array redefinition associates a new array with a previous item at the same BEGIN-END level of a structure.

```
type identifier [ " [ " lower-bound : upper-bound " ] " ]  
= previous-identifier ;
```

## Definition Substructure Redefinitions

A definition substructure redefinition associates a new definition substructure with a previous item at the same BEGIN-END level of a structure.

```
STRUCT identifier [ " [ " lower-bound : upper-bound " ] " ]  
= previous-identifier ; substructure-layout ;
```

## Referral Substructure Redefinitions

A referral substructure redefinition associates a new referral substructure with a previous item at the same BEGIN-END level of a structure.

```
STRUCT identifier ( referral )  
[ " [ " lower-bound : upper-bound " ] " ] = previous-identifier ;
```

## Simple Pointer Redefinitions

A simple pointer redefinition associates a new simple pointer with a previous item at the same BEGIN-END level of a structure.

```
type { . . . } identifier = previous-identifier ;  
{ .EXT }
```

## Structure Pointer Redefinitions

A structure pointer redefinition associates a new structure pointer with a previous item at the same BEGIN-END level of a structure.

```
{ STRING } { . . . } identifier ( referral )
{ INT     } { .EXT}
            = previous-identifier ;
```

## Pointer Declarations

The following syntax diagrams describe simple pointer and structure pointer declarations.

### Simple Pointers

A simple pointer is a variable you load with a memory address, usually of a simple variable or array, which you access with this simple pointer.

```
type { . . . } identifier [ := initialization ]
      { .EXT }
[ , { . . . } identifier [ := initialization ] ] ... ;
      { .EXT }
```

### Structure Pointers

A structure pointer is a variable you load with a memory address of a structure, which you access with this structure pointer.

```
{ STRING } { . . . } identifier ( referral )
{ INT     } { .EXT }
            [ := initialization ]
[ , { . . . } identifier ( referral )
      { .EXT }
            [ := initialization ] ] . . . ;
```

## Equivalenced Variable Declarations

The following syntax diagrams describe equivalenced variable declarations for simple variables, simple pointers, structures, and structure pointers.

### Equivalenced Simple Variables

An equivalenced simple variable associates a new simple variable with a previously declared variable.

```

type identifier = previous-identifier [ " [ " index " ] " ]
[ { + } offset ]
{ - }

[ , identifier = previous-identifier [ " [ " index " ] " ] ... ;
[ { + } offset ]
{ - }

```

## Equivalenced Definition Structures

An equivalenced definition structure associates a new definition structure with a previously declared variable.

```

STRUCT [ . . . ] identifier = previous-identifier
[ .EXT ]
[ " [ " index " ] " ] ; structure-layout ;
[ { + } offset ]
{ - }

```

## Equivalenced Referral Structures

An equivalenced referral structure associates a new referral structure with a previously declared variable.

```

STRUCT [ . . . ] identifier ( referral ) = previous-identifier
[ .EXT ]
[ " [ " index " ] " ] ;
[ { + } offset ]
{ - }

```

## Equivalenced Simple Pointers

An equivalenced simple pointer associates a new simple pointer with a previously declared variable.

```

type { . . . } identifier = previous-identifier
{ .EXT }

[ " [ " index " ] " ]
[ { + } offset ]
{ - }

[ , { . . . } identifier = previous-identifier
{ .EXT }

[ " [ " index " ] " ] ... ;
[ { + } offset ]
{ - }

```

## Equivalenced Structure Pointers

An equivalenced structure pointer associates a new structure pointer with a previously declared variable.

```
{ STRING } { .      } identifier ( referral )
{ INT     } { .EXT } 
    = previous-identifier [ " [ " index " ] " ]
        [ { + } offset ]
        { - }

[ , { .      } identifier ( referral )
  { .EXT }
    = previous-identifier [ " [ " index " ] " ] ... ;
        [ { + } offset ]
        { - }
```

## Base-Address Equivalenced Variable Declarations

The following syntax diagrams describe base-addressed equivalenced variable declarations for simple variables, simple pointers, structures, and structure pointers.

### Base-Address Equivalenced Simple Variables

A base-addressed equivalenced simple variable associates a simple variable with a global, local, or top-of-stack base address.

```
type identifier = base-address [ " [ " index " ] " ]
                                [ { + } offset ]
                                { - }

[ , identifier = base-address [ " [ " index " ] " ] ... ;
                                [ { + } offset ]
                                { - }
```

### Base-Address Equivalenced Definition Structures

A base-addressed equivalenced definition structure associates a definition structure with a global, local, or top-of-stack base address.

```
STRUCT [ .      ] identifier = base-address
      [ .EXT ]
      [ " [ " index " ] " ] ; structure-layout ;
          [ { + } offset ]
          { - }
```

## Base-Address Equivalenced Referral Structures

A base-addressed equivalenced referral structure associates a referral structure with a global, local, or top-of-stack base address.

```
STRUCT [ . . . ] identifier ( referral ) = base-address
      [ .EXT ]
      [ " [ " index " ] " ];
      [ { + } offset ]
      { - }
```

## Base-Address Equivalenced Simple Pointers

A base-addressed equivalenced simple pointer associates a simple pointer with a global, local, or top-of-stack base address.

```
type { . . . } identifier = base-address [ " [ " index " ] " ]
      { .EXT }
      [ { + } offset ]
      { - }

[ , { . . . } identifier = base-address
  { .EXT }
  [ " [ " index " ] " ] . . . ;
  [ { + } offset ]
  { - }
```

## Base-Address Equivalenced Structure Pointers

A base-addressed equivalenced structure pointer associates a structure pointer with a global, local, or top-of-stack base address.

```
{ STRING } { . . . } identifier ( referral )
{ INT } { .EXT }
      = base-address [ " [ " index " ] " ]
      [ { + } offset ]
      { - }

[ , { . . . } identifier ( referral )
  { .EXT }
  = base-address [ " [ " index " ] " ] . . .
  [ { + } offset ]
  { - }
```

## NAME and BLOCK Declarations

The following syntax diagrams describe NAME and BLOCK declarations.

### NAMES

The NAME declaration assigns an identifier to a compilation unit and to its private global data block if it has one.

```
NAME identifier ;
```

### BLOCKS

The BLOCK declaration groups global data declarations into a named or private relocatable global data block.

```
BLOCK { identifier } [ AT (0) ] ;
      { PRIVATE } [ BELOW (64) ]
                  [ BELOW (256) ]
[ data-declaration ; ] ...
END BLOCK ;
```

## Procedure and Subprocedure Declarations

The following syntax diagrams describe procedure, subprocedure, entry-point, and label declarations.

### Procedures

A procedure is a program unit that is callable from anywhere in the program.

```
[ type ] PROC identifier [ public-name-spec ]
      [ parameter-list ]
      [ proc-attribute [ , proc-attribute ] ... ] ;
      [ param-spec ; ] ...
      { proc-body } ;
      { EXTERNAL }
      { FORWARD }
```

*type* is one of:

```
{ STRING }
{ { INT | REAL } [ ( width ) ] }
{ UNSIGNED ( width ) }
{ FIXED [ ( fpoint | * ) ] }
```

*public-name-spec* has the form:

= " *public-name* "

*parameter-list* has the form:

( { *param-name* } [ , { *param-name* } ] ... )  
   { *param-pair* }   { *param-pair* }

*param-pair* has the form:

string : length

*proc-attrib* is one of:

{ MAIN   INTERRUPT	}
{ RESIDENT	}
{ CALLABLE   PRIV	}
{ VARIABLE   EXTENSIBLE [ ( count ) ]	}
{ LANGUAGE { C }	}
{ COBOL }	
{ FORTRAN }	
{ PASCAL }	
{ UNSPECIFIED }	

*param-spec* has the form:

*param-type* [ . . . ] *param-name* [ ( *referral* ) ]  
   [ .EXT ]  
   [ , . . . ] *param-name* [ ( *referral* ) ] ... ;  
   [ .EXT ]

*param-type* is one of:

{ STRING	}
{ { INT   REAL } [ ( <i>width</i> ) ]	}
{ UNSIGNED ( <i>width</i> )	}
{ FIXED [ ( <i>fpoint</i>   * ) ]	}
{ STRUCT	}
{ [ <i>type</i> ] { PROC   PROC(32) }	}

*proc-body* has the form:

BEGIN [ *local-decl* ; ] ...  
   [ *subproc-decl* ; ] ...  
   [ *statement* [ ; *statement* ] ... ]  
 END ;

## Subprocedures

A subprocedure is a program unit that is callable from anywhere in the procedure.

```
[ type ] SUBPROC identifier [ parameter-list ] [ VARIABLE ] ;
  [ param-spec ; ] ...
  { subproc-body } ;
  { FORWARD }
```

*type* is one of:

```
{ STRING }
{ { INT | REAL } [ ( width ) ] }
{ UNSIGNED ( width ) }
{ FIXED [ ( fpoint | * ) ] }
```

*parameter-list* has the form:

```
( { param-name } [ , { param-name } ] . . .
  { param-pair } { param-pair } )
```

*param-pair* has the form:

*string* : *length*

*param-spec* has the form:

```
param-type [ . . . ] param-name [ ( referral ) ]
  [ .EXT ]
  [ , . . . ] param-name [ ( referral ) ] . . .
  [ .EXT ]
```

*param-type* is one of:

```
{ STRING }
{ { INT | REAL } [ ( width ) ] }
{ UNSIGNED ( width ) }
{ FIXED [ ( fpoint | * ) ] }
{ STRUCT }
{ [ type ] { PROC | PROC(32) } }
```

*subproc-body* has the form:

```
BEGIN [ sublocal-decl ; ] . . .
  [ statement [ ; statement ] . . . ]
END ;
```

## Entry Points

An entry-point declaration associates an identifier with a secondary location from which execution can start in a procedure or subprocedure.

```
ENTRY identifier [ , identifier] ... ;
```

## Labels

The LABEL declaration reserves an identifier for later use as a label within the encompassing procedure or subprocedure.

```
LABEL identifier [ , identifier] ... ;
```

# Statements

The following syntax diagrams describe statements in alphabetic order.

## Compound Statements

A compound statement is a BEGIN-END construct that groups statements to form a single logical statement.

```
BEGIN  
  [ statement ] [ ; statement ] . . . [ ; ]  
END
```

## ASSERT Statement

The ASSERT statement conditionally invokes the procedure specified in an ASSERTION directive.

```
ASSERT assert-level : condition
```

## Assignment Statement

The assignment statement assigns a value to a previously declared variable.

```
variable := [ variable := ] ... expression
```

## Bit-D eposit Assignment Statement

The bit-deposit assignment statement assigns a value to a bit field in a variable.

```
variable.< left-bit [: right-bit] > := expression
```

## CALL Statement

The CALL statement invokes a procedure, subprocedure, or entry point, and optionally passes parameters to it.

```
[ CALL ] identifier
  [ ( [ param      ][ , [ param      ] ] ... ) ]
    [ param-pair ] [ param-pair ]
```

## Labeled CASE Statement

The labeled CASE statement executes a choice of statements the selector value matches a case label associated with those statements.

```
CASE selector OF
  BEGIN
    case-alternative ; [ case-alternative ; ] ...
    [ OTHERWISE -> [ statement [ ; statement ] . . . ] ; ]
  END
```

*case-alternative* has the form:

```
{ case-label }
{ lower-case-label .. upper-case-label }

[ , { case-label } ] ...
  { lower-case-label .. upper-case-label }

-> statement [ ; statement ] ...
```

## Unlabeled CASE Statement

The unlabeled CASE statement executes a choice of statements based on an inclusive range of implicit selector values, from 0 through *n*, with one statement for each value.

```
CASE selector OF
  BEGIN [ statement [ ; statement ] ... ] ;
    [ OTHERWISE [ statement ] ; ]
  END
```

## CODE Statement

The CODE statement specifies machine-level instructions and pseudocodes to compile into the object file.

`CODE ( instruction [ ; instruction ] ... )`

*instruction* has one of the following forms:

No.	Instruction Form
1.	{ mnemonic }
2.	{ mnemonic [ .   @ ] identifier }
3.	{ mnemonic constant }
4.	{ mnemonic register }
5.	{ mnemonic [ .   @ ] identifier [ , register ] }
6.	{ mnemonic constant [ , register ] }

- |    |  |   |
|----|--|---|
| 1. | { mnemonic }                                     | } |
| 2. | { mnemonic [ .   @ ] identifier }                | } |
| 3. | { mnemonic constant }                            | } |
| 4. | { mnemonic register }                            | } |
| 5. | { mnemonic [ .   @ ] identifier [ , register ] } | } |
| 6. | { mnemonic constant [ , register ] }             | } |

## DO Statement

The DO statement is a posttest loop that repeatedly executes a statement until a specified condition becomes true.

`DO [ statement ] UNTIL condition`

## DROP Statement

The DROP statement disassociates an identifier from an index register reserved by a USE statement or from a label.

`DROP identifier [ , identifier ] ...`

## FOR Statement

The FOR statement is a pretest loop that repeatedly executes a statement while incrementing or decrementing an index.

`FOR index := initial-value { TO } limit [ BY step ]  
  { DOWNTO }  
  DO [ statement ]`

## GOTO Statement

The GOTO statement unconditionally transfers program control to a labeled statement.

GOTO <i>label-name</i>
------------------------

## IF Statement

The IF statement conditionally selects one of two statements.

IF <i>condition</i> THEN [ <i>statement</i> ] [ ELSE [ <i>statement</i> ] ]
---

## Move Statement

The move statement copies contiguous bytes, words, or elements to a new location.

<pre> <i>destination</i> { ':=' }               { '=' }  { <i>source</i> FOR <i>count</i> [ <i>count-unit</i> ] } { <i>constant</i> } { " [ " <i>constant</i> " ] " { <i>constant-list</i> }  [ &amp; { <i>source</i> FOR <i>count</i> [ <i>count-unit</i> ] } ] ... { <i>constant</i> } { " [ " <i>constant</i> " ] " { <i>constant-list</i> }  [ -&gt; <i>next-addr</i> ] </pre>
--

## RETURN Statement

The RETURN statement returns control to the caller. For a function, RETURN must return a result expression. The RETURN statement can also return a condition-code value.

<pre> { RETURN [ , <i>cc-expression</i> ] } { RETURN <i>result-expression</i> [ , <i>cc-expression</i> ] } </pre>
---

## SCAN Statement

The SCAN or RSCAN statement scans sequential bytes for a test character from left to right or from right to left, respectively.

<pre> { SCAN } <i>variable</i> { WHILE } <i>test-char</i> [ -&gt; <i>next-addr</i> ] { RSCAN }      { UNTIL } </pre>
--

## STACK Statement

The STACK statement loads values onto the register stack.

STACK *expression* [ , *expression* ] ...

## STORE Statement

The STORE statement removes values from the register stack and stores them into variables.

STORE *variable* [ , *variable* ] ...

## USE Statement

The USE statement reserves an index register for your use.

USE *identifier* [ , *identifier* ] ...

## WHILE Statement

The WHILE statement is a pretest loop that repeatedly executes a statement while a condition is true.

WHILE *condition* DO [ *statement* ]

# Standard Functions

The following syntax diagrams describe standard functions in alphabetic order.

## \$ABS Function

The \$ABS function returns the absolute value of an expression. The returned value has the same data type as the expression.

\$ABS ( *expression* )

## \$ALPHA Function

The \$ALPHA function tests the right byte of an INT value for the presence of an alphabetic character.

\$ALPHA ( *int-expression* )

## \$AXADR Function

See [Section 15, Privileged Procedures](#)

## \$BITLENGTH Function

The \$BITLENGTH function returns the length, in bits, of a variable.

```
$BITLENGTH ( variable )
```

## \$BITOFFSET Function

The \$BITOFFSET function returns the number of bits from the address of the zeroth structure occurrence to a structure data item.

```
$BITOFFSET ( variable )
```

## \$BOUNDS Function

See [Section 15, Privileged Procedures](#)

## \$CARRY Function

The \$CARRY function checks the state of the carry bit in the environment register and indicates whether a carry out of the high-order bit position occurred.

```
$CARRY
```

## \$COMP Function

The \$COMP function obtains the one's complement of an INT expression.

```
$COMP ( int-expression )
```

## \$DBL Function

The \$DBL function returns an INT(32) value from an INT, FIXED(0), REAL, or REAL(64) expression.

```
$DBL ( expression )
```

## \$DBLL Function

The \$DBLL function returns an INT(32) value from two INT values.

\$DBLL ( <i>int-expression</i> , <i>int-expression</i> )
--

## \$DBLR Function

The \$DBLR function returns an INT(32) value from an INT, FIXED(0), REAL, or REAL(64) expression and applies rounding to the result.

\$DBLR ( <i>expression</i> )
------------------------------

## \$DFIX Function

The \$DFIX function returns a FIXED (*fpoint*) expression from an INT(32) expression.

\$DFIX ( <i>dbl-expression</i> , <i>fpoint</i> )
--

## \$EFLT Function

The \$EFLT function returns a REAL(64) value from an INT, INT(32), FIXED (*fpoint*), or REAL expression.

\$EFLT ( <i>expression</i> )
------------------------------

## \$EFLTR Function

The \$EFLTR function returns a REAL(64) value from an INT, INT(32), FIXED (*fpoint*), or REAL expression and applies rounding to the result.

\$EFLTR ( <i>expression</i> )
-------------------------------

## \$FIX Function

The \$FIX function returns a FIXED(0) value from an INT, INT(32), REAL, or REAL(64) expression.

\$FIX ( <i>expression</i> )
-----------------------------

## \$FIXD Function

The \$FIXD function returns an INT(32) value from a FIXED(0) expression.

\$FIXD ( <i>fixed-expression</i> )
------------------------------------

## \$FIXI Function

The \$FIXI function returns the signed INT equivalent of a FIXED(0) expression.

\$FIXI ( *fixed-expression* )

## \$FIXL Function

The \$FIXL function returns the unsigned INT equivalent of a FIXED(0) expression.

\$FIXL ( *fixed-expression* )

## \$FIXR Function

The \$FIXR function returns a FIXED(0) value from an INT, INT(32), FIXED, REAL, or REAL(64) expression and applies rounding to the result.

\$FIXR ( *expression* )

## \$FLT Function

The \$FLT function returns a REAL value from an INT, INT(32), FIXED (*fpoint*), or REAL(64) expression.

\$FLT ( *expression* )

## \$FLTR Function

The \$FLTR function returns a REAL value from an INT, INT(32), FIXED (*fpoint*), or REAL(64) expression and applies rounding to the result.

\$FLTR ( *expression* )

## \$HIGH Function

The \$HIGH function returns an INT value that is the high-order 16 bits of an INT(32) expression.

\$HIGH ( *dbl-expression* )

## \$IFIX Function

The \$IFIX function returns a FIXED (*fpoint*) value from a signed INT expression.

\$IFIX ( <i>int-expression</i> , <i>fpoint</i> )
--

## \$INT Function

The \$INT function returns an INT value from the low-order 16 bits of an INT(32 or FIXED(0) expression. \$INT returns a fully converted INT expression from a REAL or REAL(64) expression.

\$INT ( <i>expression</i> )
-----------------------------

## \$INTR Function

The \$INTR function returns an INT value from the low-order 16 bits of an INT(32) or FIXED(0) expression. \$INTR returns a fully converted and rounded INT expression from a REAL or REAL(64) expression.

\$INTR ( <i>expression</i> )
------------------------------

## \$LADR Function

The \$LADR function returns the standard (16-bit) address of a variable that is accessed through an extended (32-bit) pointer.

\$LADR ( <i>variable</i> )
----------------------------

## \$LEN Function

The \$LEN function returns the length, in bytes, of one occurrence of a variable.

\$LEN ( <i>variable</i> )
---------------------------

## \$LFIX Function

The \$LFIX function returns a FIXED (fpoint) expression from an unsigned INT expression.

\$LFIX ( <i>int-expression</i> , <i>fpoint</i> )
--

## \$LMAX Function

The \$LMAX function returns the maximum of two unsigned INT expressions.

\$LMAX ( <i>int-expression</i> , <i>int-expression</i> )
--

## \$LMIN Function

The \$LMIN function returns the minimum of two unsigned INT expressions.

```
$LMIN ( int-expression , int-expression )
```

## \$MAX Function

The \$MAX function returns the maximum of two signed INT, INT(32), FIXED (*fpoint*), REAL, or REAL(64) expressions.

```
$MAX ( expression , expression )
```

## \$MIN Function

The \$MIN function returns the minimum of two INT, INT(32), FIXED (*fpoint*), REAL, or REAL(64) expressions.

```
$MIN ( expression , expression )
```

## \$NUMERIC Function

The \$NUMERIC function tests the right half of an INT value for the presence of an ASCII numeric character.

```
$NUMERIC ( int-expression )
```

## \$OCCURS Function

The \$OCCURS function returns the number of occurrences of a variable.

```
$OCCURS ( variable )
```

## \$OFFSET Function

The \$OFFSET function returns the number of bytes from the address of the zeroth structure occurrence to a structure data item.

```
$OFFSET ( variable )
```

## \$OPTIONAL Function

The \$OPTIONAL function controls whether a given parameter or parameter pair is passed to a VARIABLE or EXTENSIBLE procedure. \$OPTIONAL is a D20 or later feature.

```
$OPTIONAL ( cond-expression , { param } )  
          { param-pair }
```

## \$OVERFLOW Function

The \$OVERFLOW function checks the state of the overflow indicator and indicates whether an overflow occurred during an arithmetic operation.

```
$OVERFLOW
```

## \$PARAM Function

The \$PARAM function checks for the presence or absence of an actual parameter in the call that invoked the current procedure or subprocedure.

```
$PARAM ( formal-param )
```

## \$POINT Function

The \$POINT function returns the fpoint value, in integer form, associated with a FIXED expression.

```
$POINT ( fixed-expression )
```

## \$READCLOCK Function

The \$READCLOCK function returns the current setting of the system clock.

```
$READCLOCK
```

## \$RP Function

The \$RP function returns the current setting of the compiler's internal RP counter. (RP is the register stack pointer.)

```
$RP
```

## \$SCALE Function

The \$SCALE function moves the position of the implied decimal point by adjusting the internal representation of a FIXED (*fpoint*) expression.

```
$SCALE ( fixed-expression , scale )
```

## \$SPECIAL Function

The \$SPECIAL function tests the right half of an INT value for the presence of an ASCII special (non-alphanumeric) character.

`$SPECIAL ( int-expression )`

## \$SWITCHES Function

See [Section 15, Privileged Procedures](#)

## \$TYPE Function

The \$TYPE function returns a value that indicates the data type of a variable.

`$TYPE ( variable )`

## \$UDBL Function

The \$UDBL function returns an INT(32) value from an unsigned INT expression.

`$UDBL ( int-expression )`

## \$USERCODE Function

The \$USERCODE function returns the content of the word at the specified location in the current user code segment.

`$USERCODE ( expression )`

## \$XADR Function

The \$XADR function converts a standard address to an extended address.

`$XADR ( variable )`

# Privileged Procedures

The following syntax diagrams describe declarations for privileged procedures.

## System Global Pointers

The system global pointer declaration associates an identifier with a memory location that you load with the address of a variable located in the system global data area.

```
type .SG identifier [ := preset-address ]
[ , .SG identifier [ := preset-address ] ] ... ;
```

## 'SG'-Equivalenced Simple Variables

The 'SG'-equivalenced simple variable declaration associates a simple variable with a location that is relative to the base address of the system global data area.

```
type identifier = 'SG' [ " [ " index " ] "
[ { + } offset ]
{ - }

[ , identifier = 'SG' [ " [ " index " ] " ] ] . . .
[ { + } offset ]
{ - }
```

## 'SG'-Equivalenced Definition Structures

The 'SG'-equivalenced definition structure declaration associates a definition structure with a location relative to the base address of the system global data area.

```
STRUCT [ . . . ] identifier = 'SG'
[ .SG ]
[ .EXT]
[ " [ " index " ] " ] ; structure-layout ;
[ { + } offset ]
{ - }
```

## 'SG'-Equivalenced Referral Structures

The 'SG'-equivalenced referral structure declaration associates a referral structure with a location relative to the base address of the system global data area.

```
STRUCT [ . . . ] identifier ( referral ) = 'SG'
[ .SG ]
[ .EXT]
[ " [ " index " ] " ;
[ { + } offset ]
{ - }
```

## 'SG'-Equivalenced Simple Pointers

The 'SG'-equivalenced simple pointer declaration associates a simple pointer with a location relative to the base address of the system global data area.

```

type { . . . } identifier = 'SG' [ " [ " index " ] " ]
    { .SG }           [ { + } offset ]
    { .EXT}           { - }

[ , { . . . } identifier = 'SG'
  { .SG }
  { .EXT}

  [ " [ " index " ] " ] ] ... ;
    [ { + } offset ]
    { - }

```

## 'SG'-Equivalenced Structure Pointers

The 'SG'-equivalenced structure pointer declaration associates a structure pointer with a location relative to the base address of the system global data area.

```

{ STRING } { . . . } identifier ( referral )
{ INT }     { .SG }
            { .EXT }

            = 'SG' [ " [ " index " ] " ]
                  [ { + } offset ]
                  { - }

[ , { . . . } identifier ( referral )
  { .SG }
  { .EXT }

  = 'SG' [ " [ " index " ] " ] ] ... ;
    [ { + } offset ]
    { - }

```

## \$AXADR Function

The \$AXADR function returns an absolute extended address.

```
$AXADR ( variable )
```

## \$BOUNDS Function

The \$BOUNDS function checks the location of a parameter passed to a system procedure to prevent a pointer that contains an incorrect address from overlaying the stack (S) register with data.

```
$BOUNDS ( param , count )
```

## \$SWITCHES Function

The \$SWITCHES function returns the current content of the switch register.

\$SWITCHES
------------

## TARGET Directive

The TARGET directive specifies the target system for conditional code. TARGET works in conjunction with the IF and ENDIF directives. TARGET is a D20 or later feature.

TARGET <i>target-system</i>
-----------------------------

# Compiler Directives

The following syntax diagrams describe directive lines, followed by compiler directives in alphabetic order.

## Directive Lines

A directive line in your source code contains one or more compiler directives.

? <i>directive</i> [ , <i>directive</i> ] ...
---

## ABORT Directive

The ABORT directive terminates compilation if the compiler cannot open a file specified in a SOURCE directive. The default is ABORT.

[NO] ABORT
------------

## ABSLIST Directive

ABSLIST lists code addresses relative to the code area base. The default is NOABSLIST.

[NO] ABSLIST
--------------

## ASSERTION Directive

ASSERTION invokes a procedure when a condition defined in an ASSERT statement is true.

ASSERTION [ = ] <i>assertion-level</i> , <i>procedure-name</i>
--

## BEGINCOMPILATION Directive

BEGINCOMPILATION marks the point in the source file where compilation is to begin if the USEGLOBALS directive is in effect.

```
BEGINCOMPILATION
```

## CHECK Directive

CHECK generates range-checking code for certain features. The default is NOCHECK.

```
[ NO]
[PUSH] CHECK
[ POP]
```

## CODE Directive

CODE lists instruction codes and constants in octal format after each procedure. The default is CODE.

```
[ NO]
[PUSH] CODE
[ POP]
```

## COLUMNS Directive

COLUMNS directs the compiler to treat any text beyond the specified column as comments.

```
COLUMNS columns-value
```

## COMPACT Directive

COMPACT moves procedures into gaps below the 32K-word boundary of the code area if they fit. The default is COMPACT.

```
[NO] COMPACT
```

## CPU Directive

CPU specifies that the object file runs on a TNS system. (The need for this directive no longer exists. This directive has no effect on the object file and is retained only for compatibility with programs that still specify it.)

CPU *cpu-type*

## CROSSREF Directive

CROSSREF collects source-level declarations and cross-reference information or specifies CROSSREF classes. The default is NOCROSSREF.

[NO] CROSSREF [ *class* ]  
           [ ( *class* [ , *class* ] ... ) ]

## DATAPAGES Directive

DATAPAGES sets the size of the data area in the user data segment.

DATAPAGES [ = ] *num-pages*

## DECS Directive

DECS decrements the compiler's internal S-register counter.

DECS [ = ] *sdec-value*

## DEFEXPAND Directive

DEFEXPAND lists expanded DEFINEs and SQL-TAL code in the compiler listing. The default is NODEFEXPAND.

[ NO]  
 [PUSH] DEFEXPAND  
 [ POP]

## DEFINETOG Directive

DEFINETOG specifies named or numeric toggles, without changing any prior settings, for use in conditional compilation. DEFINETOOG is a D20 or later feature.

DEFINETOG { *toggle-name* }  
           { *toggle-number* [ , *toggle-number* ] ... }  
           { ( { *toggle-name* } [ , { *toggle-name* } ] ... ) }  
           { *toggle-number* } { *toggle-number* }

## DUMPCONS Directive

DUMPCONS inserts the contents of the compiler's constant table into the object code.

DUMPCONS
----------

## ENDIF Directive

See [IF and ENDIF Directives](#) on page 16-47.

## ENV Directive

ENV specifies the intended run-time environment of a D-series object file. The default is ENV NEUTRAL.

ENV { COMMON } { OLD      } { NEUTRAL }
---

## ERRORFILE Directive

ERRORFILE logs compilation errors and warnings to an error file so you can use the TACL FIXERRS macro to view the diagnostic messages in one PS Text Edit window and correct the source file in another window.

ERRORFILE { file-name    } { define-name } { assign-name }
--

## ERRORS Directive

ERRORS sets the maximum number of error messages to allow before the compiler terminates the compilation.

ERRORS [ = ] <i>num-messages</i>
----------------------------------

## EXTENDSTACK Directive

EXTENDSTACK increases the size of the data stack in the user data segment.

EXTENDSTACK [ = ] <i>num-pages</i>
------------------------------------

## EXTENDTALHEAP Directive

EXTENDTALHEAP increases the size of the compiler's internal heap for a D-series compilation unit.

EXTENDTALHEAP [ = ] <i>num-pages</i>
--------------------------------------

## FIXUP Directive

FIXUP directs BINSERV to perform its fixup step. The default is FIXUP.

[NO] FIXUP
------------

## FMAP Directive

FMAP lists the file map. The default is NOFMAP.

[NO] FMAP
-----------

## GMAP Directive

GMAP lists the global map. The default is GMAP.

[NO] GMAP
-----------

## HEAP Directive

HEAP sets the size of the CRE user heap for a D-series compilation unit if the ENV COMMON directive is in effect.

HEAP [ = ] <i>num-pages</i>
-----------------------------

## HIGHPIN Directive

HIGHPIN sets the HIGHPIN attribute in a D-series object file.

HIGHPIN
---------

## HIGHREQUESTERS Directive

HIGHREQUESTERS sets the HIGHREQUESTERS attribute in a D-series object file.

HIGHREQUESTERS
----------------

## ICODE Directive

ICODE lists the instruction-code (icode) mnemonics for subsequent procedures. The default is NOICODE.

[ NO]
[PUSH] ICODE
[ POP]

## IF and ENDIF Directive

IF and IFNOT control conditional compilation based on a condition. The ENDIF directive terminates the range of the matching IF or IFNOT directive. The D20 or later RVU supports named toggles and target-system toggles as well as numeric toggles.

IF[NOT] { <i>toggle-name</i> }
{ <i>toggle-number</i> }
{ <i>target-system</i> }
ENDIF { <i>toggle-name</i> }
{ <i>toggle-number</i> }
{ <i>target-system</i> }

## INHIBITXX Directive

INHIBITXX generates inefficient but correct code for extended global declarations in relocatable blocks that Binder might locate after the first 64 words of the primary global area of the user data segment. The default is NOINHIBITXX.

[NO] INHIBITXX
----------------

## INNERLIST Directive

INNERLIST lists the instruction code mnemonics (and the compiler's RP setting) for each statement. The default is NOINNERLIST.

[ NO]
[PUSH] INNERLIST
[ POP]

## INSPECT Directive

INSPECT sets the Inspect product as the default debugger for the object file. The default is NOINSPECT.

[NO] INSPECT
--------------

## INT32INDEX Directive

INT32INDEX generates INT(32) indexes from INT indexes for accessing items in an extended indirect structure in a D-series program. The default is NOINT32INDEX.

```
[ NO]  
[PUSH] INT32INDEX  
[ POP]
```

## LARGESTACK Directive

LARGESTACK sets the size of the extended stack in the automatic extended data segment.

```
LARGESTACK [ = ] num-pages
```

## LIBRARY Directive

LIBRARY specifies the name of the TNS software user run-time library to be associated with the object file at run time.

```
LIBRARY file-name
```

## LINES Directive

LINES sets the maximum number of output lines per page.

```
LINES [ = ] num-lines
```

## LIST Directive

LIST lists the source text for subsequent source code if NOSUPPRESS is in effect. The default is LIST.

```
[ NO]  
[PUSH] LIST  
[ POP]
```

## LMAP Directive

LMAP lists load-map and cross-reference information. The default is LMAP ALPHA.

[NO] LMAP [ <i>lmap-option</i> ]
[ ( <i>lmap-option</i> [ , <i>lmap-option</i> ] ... ) ]
[ * ]

## MAP Directive

MAP lists the identifier maps. The default is MAP.

[ NO]
[PUSH] MAP
[ POP]

## OLDFLTSTDFUNC Directive

OLDFLTSTDFUNC treats arguments to the \$FLT, \$FLTR, \$EFLT, and \$EFLTR standard functions as if they were FIXED(0) values.

OLDFLTSTDFUNC
---------------

## OPTIMIZE Directive

OPTIMIZE specifies the level at which the compiler optimizes the object code.

OPTIMIZE [ = ] <i>level</i>
-----------------------------

## PAGE Directive

PAGE optionally prints a heading and causes a page eject.

PAGE [ " <i>heading-string</i> " ]
------------------------------------

## PEP Directive

PEP specifies the size of the procedure entry-point (PEP) table.

PEP [ = ] <i>pep-table-size</i>
---------------------------------

## PRINTSYM Directive

PRINTSYM lists symbols. The default is PRINTSYM.

[NO]PRINTSYM
--------------

## RELOCATE Directive

RELOCATE lists BINSERV warnings for declarations that depend on absolute addresses in the primary global data area of the user data segment.

```
RELOCATE
```

## RESETTOG Directive

RESETTOG creates new toggles in the off state and turns off toggles created by SETTOG. The RESETTOG directive supports named toggles as well as numeric toggles.

```
RESETTOG [ toggle-name ]  
        [ toggle-number [ , toggle-number ] ... ]  
        [ ( { toggle-name } [ , { toggle-name } ] ) ]  
        { toggle-number } { toggle-number }
```

## ROUND Directive

ROUND rounds FIXED values assigned to FIXED variables that have smaller float values than the values you are assigning. The default is NOROUND.

```
[NO] ROUND
```

## RP Directive

RP sets the compiler's internal register pointer (RP) count. RP tells the compiler how many registers are currently in use on the register stack.

```
RP [ = ] register-number
```

## RUNNAMED Directive

RUNNAMED causes a D-series object file to run on a D-series system as a named process even if you do not provide a name for it.

```
RUNNAMED
```

## SAVEABEND Directive

SAVEABEND directs the Inspect product to generate a save file if your process terminates abnormally during execution. The default is NOSAVEABEND.

```
[NO] SAVEABEND
```

## SAVEGLOBALS Directive

SAVEGLOBALS saves all global data declarations in a file for use in subsequent compilations that specify the USEGLOBALS directive.

```
SAVEGLOBALS { file-name }
             { define-name }
             { assign-name }
```

## SEARCH Directive

SEARCH specifies object files from which BINSERV can resolve unsatisfied external references and validate parameter lists at the end of compilation. By default, BINSERV does not attempt to resolve unsatisfied external references.

```
SEARCH [ file-name ]
       [ define-name ]
       [ assign-name ]
       [ ( { file-name } [ , { file-name } ... ] )
           { define-name } { define-name }
           { assign-name } { assign-name } ]
```

## SECTION Directive

SECTION gives a name to a section of a source file for use in a SOURCE directive.

```
SECTION [ section-name ]
```

## SETTOG Directive

SETTOG turns the specified toggles on for use in conditional compilations. The SETTOG directive supports named toggles as well as numeric toggles.

```
SETTOG [ toggle-name ]
       [ toggle-number [ , toggle-number ] ... ]
       [ ( { toggle-name } [ , { toggle-name } ] ) ]
       { toggle-number } { toggle-number }
```

## SOURCE Directive

SOURCE specifies source code to include from another source file.

```
SOURCE { file-name }
       { define-name }
       { assign-name }
       [ ( section-name [ , section-name ] ... ) ]
```

## SQL Directive

For more information, see the *NonStop SQL Programming Manual for TAL*.

## SQLMEM Directive

For more information, see the *NonStop SQL Programming Manual for TAL*.

## STACK Directive

STACK sets the size of the data stack in the user data segment.

```
STACK [=] num-pages
```

## SUBTYPE Directive

SUBTYPE specifies that the object file is to execute as a process of a specified subtype.

```
SUBTYPE [=] subtype-number
```

## SUPPRESS Directive

SUPPRESS overrides all the listing directives. The default is NOSUPPRESS.

```
[NO] SUPPRESS
```

## SYMBOLPAGES Directive

SYMBOLPAGES sets the size of the internal symbol table the compiler uses as a temporary storage area for processing variables and SQL statements.

```
SYMBOLPAGES [=] num-pages
```

## SYMBOLS Directive

SYMBOLS saves symbols in a symbol table (for Inspect symbolic debugging) in the object file. The default is NOSYMBOLS.

```
[NO] SYMBOLS
```

## SYNTAX Directive

SYNTAX checks the syntax of the source text without producing an object file.

**SYNTAX****USEGLOBALS Directive**

USEGLOBALS retrieves the global data declarations saved in a file by SAVEGLOBALS during a previous compilation.

```
USEGLOBALS { file-name }
            { define-name }
            { assign-name }
```

**WARN Directive**

WARN instructs the compiler to print a selected warning or all warnings. The default is WARN.

```
WARN [ [ = ] warning-number ]
```

# 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 an HP file name. The HP 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  $<n>$  or  $<n:n>$ .

**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  $<n>$  or  $<n:n>$ .

**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  $<n>$  or  $<n:n>$ , 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 NonStop 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 RVU 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 RVU 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.

**HP NonStop Series system.** See [TNS](#)

**HP NonStop Series/RISC system** . See [TNS/R](#)

**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 HP 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. The RETURN statement 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.

**template structure.** A declaration that describes a structure layout but allocates no storage for the structure. Contrast with [definition structure](#) and [referral structure](#)

**TNS.** HP computers that support the NonStop Kernel operating system and that are based on complex instruction-set computing (CISC) technology. TNS processors implement the TNS instruction set. Contrast with [TNS/R](#)

**TNS/R.** HP computers that support the NonStop Kernel operating system and that are based on reduced instruction-set computing (RISC) technology. TNS/R processors implement the RISC instruction set and are upwardly compatible with the TNS system-level architecture. Systems with these processors include most of the NonStop servers. Contrast with [TNS](#)

**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

## Numbers

16-bit (standard) pointers [9-1](#)  
32-bit (extended) pointers [9-1](#)

## A

ABORT Directive [16-12](#)  
ABS fpont > 19 (warning 27) [A-45](#)  
ABSLIST Directive  
    description [16-13](#)  
ABSLIST directive  
    with PEP directive [16-13](#)  
Absolute value, obtaining with \$ABS [14-6](#)  
ACON pseudocode, CODE statement [12-17](#)  
Actual/formal parameter count (error 61) [A-19](#)  
Addition operator  
    signed [4-5, 4-6](#)  
    unsigned [4-5, 4-9](#)  
Address base symbol [2-1](#)  
Address conversions  
    bit-shift operations [4-30](#)  
    simple pointers  
        standard-to-extended [9-1](#)  
    \$AXADR (relative-to-absolute) [15-11](#)  
    \$LADR (extended-to-standard) [14-23](#)  
    \$XADR (standard-to-extended) [14-43](#)  
Address of entry point used (warning 49) [A-49](#)  
Address range violation [A-6](#)  
Address references between blocks (error96) [A-25](#)  
Address size mismatch [A-55](#)  
Addresses  
    in simple pointers [9-3](#)  
    in simple pointers in structures [8-14](#)  
    in structure pointers [9-7](#)  
    in structure pointers in structures [8-16](#)  
Addresses (continued)  
    nonrelocatable, warnings of [16-68](#)  
Addressing  
    arrays [7-3](#)  
    definition structures [8-4](#)  
    read-only arrays [7-1, 7-5](#)  
    referral structures [8-1](#)  
Aliases, data types [3-3, 3-4](#)  
All index registers are reserved (warning 0) [A-40](#)  
ALPHA option, LMAP directive [16-61](#)  
AND operator [4-14](#)  
ANY  
    TARGET directive option [15-13](#)  
Arithmetic expressions  
    description [4-5](#)  
    in conditional expressions [4-12](#)  
Arithmetic operators  
    signed [4-6](#)  
    unsigned [4-9](#)  
Arithmetic overflow (warning 26) [A-45](#)  
Arithmetic overflow, testing with \$OVERFLOW [14-35](#)  
Array access changed (warning 37) [A-47](#)  
Arrays  
    as structure items [8-1, 8-2](#)  
    declaring [7-1](#)  
    length, obtaining  
        in bits [14-29](#)  
        in bytes [14-29](#)  
    multidimensional [8-1](#)  
    of arrays [8-1](#)  
    of structures [8-1](#)  
    read-only (P-relative) arrays [7-5](#)  
    redefinitions [8-17](#)  
    syntax summary  
        bracket-and-brace diagrams [C-1](#)  
        railroad diagrams [B-1](#)

ASCII character set [2-1](#)  
 ASCII character, finding  
   \$ALPHA [14-7](#)  
   \$NUMERIC [14-28](#)  
   \$SPECIAL [14-40](#)  
**ASSERT** statement  
   description [12-3](#)  
   with ASSERTION directive [16-14](#)  
**ASSERTION** directive  
   description [16-14](#)  
   with ASSERT statement [12-3](#)  
**ASSERTION** procedure cannot  
 (error156) [A-36](#)  
**ASSIGN SSV** commands [16-4](#)  
**ASSIGN SSV** too large (warning 75) [A-53](#)  
**Assignment expression** [4-19](#)  
**Assignment statement** [12-4](#)  
**AT keyword**, BLOCK declaration [11-3](#)  
**Attribute mismatch** (warning 21) [A-44](#)

## B

**Base address symbol** [2-7](#)  
**Base-address equivalenced variables**  
   definition structures [10-14](#)  
   description [10-11](#)  
   referral structures [10-16](#)  
   simple pointers [10-13](#)  
   simple variables [10-12](#)  
   syntax summary  
     bracket-and-brace diagrams [C-1](#)  
     railroad diagrams [B-1](#)

**BEGIN keyword**  
   compound statement [12-2](#)  
**BEGINCOMPILEMENT** directive  
   description [16-16](#)  
   with SAVEGLOBALS directive [16-16](#)  
   with SOURCE directive [16-77](#)  
   with USEGLOBALS directive [16-94](#)  
**BELOW keyword**, BLOCK declaration [11-3](#)  
**Bit extractions** [4-28](#)

**Bit fields**  
   deposit assignments [12-7](#)  
   UNSIGNED data type [3-5](#)  
**Bit operations**  
   extractions [4-28](#)  
   shifts [4-29](#)  
**Bit shifts** [4-29](#)  
**Bitwise logical operators** [4-11](#)  
**Bit-deposit assignment statement** [12-7](#)  
**BIT\_FILLER** declaration [8-12](#)  
**BLOCKS**  
   declaring [11-2](#)  
   syntax summary  
     bracket-and-brace diagrams [C-16](#)  
     railroad diagrams [B-22](#)  
**Boolean operators** [4-12](#)  
**Bounds illegal** (error 117) [A-23](#)  
**Bounds illegal** (error 85) [A-29](#)  
**Bounds, lower-bound**  
   arrays [7-2](#)  
**Bounds, upper and lower**  
   definition structures [8-1](#)  
   read-only arrays [7-5](#)  
**Bounds, upper-bound**  
   arrays [7-2](#)  
**Branch to entry point** (error 92) [A-24](#)  
**Built-in functions** [14-1](#)  
**Byte** [3-5](#)  
**BYTES keyword**  
   group comparison expression [4-24](#)  
   move statement [12-28](#)

## C

**CALL statement** [12-9](#)  
**CALLABLE attribute of procedures** [13-6](#)  
**Cannot access SSV** (warning 68) [A-53](#)  
**Cannot drop label** (error 50) [A-17](#)  
**Cannot purge error file** (error137) [A-33](#)  
**Cannot purge file** (error 95) [A-25](#)

- Cannot use \$OFFSET or \$LEN (warning 76) [A-54](#)
- Carry and overflow functions
  - summary of [14-3](#)
  - \$CARRY [14-10](#)
  - \$OVERFLOW [14-35](#)
- Carry indicator, testing [4-17](#), [14-11](#)
- CASE expression [4-20](#)
- Case label must be signed (error 108) [A-27](#)
- Case label or range overlaps (error 110) [A-28](#)
- Case label range is empty (error 109) [A-27](#)
- CASE statement
  - labeled [12-11](#)
  - unlabeled [12-13](#)
  - with CHECK directive [16-17](#)
- CCE (condition code equal to) [4-13](#)
- CCG (condition code greater than) [4-13](#)
- CCL (condition code less than) [4-13](#)
- Character set [2-1](#)
- Character strings [3-7](#)
- Character-test functions
  - summary of [14-1](#)
  - \$ALPHA [14-3](#)
  - \$NUMERIC [14-3](#), [14-28](#)
  - \$SPECIAL [14-3](#), [14-40](#)
- CHECK directive
  - description [16-17](#)
  - with unlabeled CASE statement [16-18](#)
- COBOL
  - run-time environment, specifying with ENV directive [16-33](#)
  - TAL procedure language attribute [13-8](#)
- Code addresses, listing with ABSLIST directive [16-13](#)
- CODE directive [16-18](#)
- Code space exceeds 64K (warning 58) [A-51](#)
- Code space items, in arithmetic expressions [4-6](#)
- CODE statement
  - description [12-15](#)
- CODE statement (continued)
  - in DEFINEs [5-4](#)
  - with DECS directive [16-27](#)
  - with DUMPCONS directive [16-32](#)
  - with FOR statement [12-24](#)
  - with RP directive [16-72](#)
- Colon not allowed (error 144) [A-34](#)
- COLUMNS directive [16-19](#)
- Comments and COLUMNS directive [16-19](#)
- COMMON attribute, ENV directive [16-33](#)
- COMPACT directive [16-21](#)
- Compilation command [16-1](#)
- Compilation units [11-1](#)
  - BLOCK declarations [11-1](#), [11-3](#)
  - global data blocks [11-1](#)
  - naming [11-1](#)
- Compiler data structure directives
  - DECS [16-26](#)
  - EXTENDTALHEAP [16-38](#)
  - RP [16-71](#)
  - summary of [16-10](#)
  - SYMBOLPAGES [16-89](#)
- Compiler error (error 0) [A-2](#)
- Compiler heap, increasing with EXTENDTALHEAP directive [16-38](#)
- Compiler input directives
  - ABORT [16-12](#)
  - BEGINCOMPILE [16-16](#)
  - COLUMNS [16-19](#)
  - SAVEGLOBALS [16-75](#)
  - SECTION [16-81](#)
  - SOURCE [16-84](#)
  - summary of [16-6](#)
  - USEGLOBALS [16-93](#)
- Compiler internal error (error 132) [A-32](#)
- Compiler label overflow (error 133) [A-32](#)
- Compiler listing directives
  - ABSLIST [16-13](#)
  - CODE [16-19](#)
  - CROSSREF [16-22](#)

Compiler listing directives (continued)

- DEFEXPAND [16-27](#)
- FMAP [16-40](#)
- GMAP [16-41](#)
- ICODE [16-46](#)
- INNERLIST [16-52](#)
- LINES [16-59](#)
- LIST [16-59](#)
- LMAP [16-61](#)
- MAP [16-62](#)
- PAGE [16-65](#)
- PRINTSYM [16-67](#)
- SUPPRESS [16-88](#)

Compiler messages [A-1](#)

Compiler no longer generates (warning 52) [A-50](#)

Compiler relative reference (error 136) [A-33](#)

Compound statements [12-2](#)

CON pseudocode, CODE statement [12-17](#)

Condition code indicator, testing [4-16](#)

Conditional compilation directives

- DEFINETOG [16-29](#)
- ENDIF [16-32](#)
- IF [16-47](#)
- IFNOT [16-47](#)
- RESETTOG [16-68](#)
- SETTOG [16-82](#)
- summary [16-10](#)

Conditional expressions [4-2](#), [4-12](#)

Conflicting TARGET directive (warning 70) [A-53](#)

Constant

- group comparison expression [4-24](#)

Constant expected (error 86) [A-23](#)

Constant expressions

- as parameters [13-12](#)
- description [4-2](#)

Constant lists

- format [3-16](#)
- group comparison expression [4-25](#)

Constant lists (continued)

- MOVE statement [12-27](#)

Constants

- character strings [3-8](#)
- emitting with DUMPCONS directive [16-31](#)
- LITERALs [5-2](#)
- numeric
- FIXED format [3-13](#)
- INT format [3-10](#)
- INT (32) format [3-11](#)
- REAL format [3-14](#)
- REAL (64) format [3-14](#)
- String format [3-9](#)

syntax summary

- bracket-and-brace diagrams [C-3](#)
- railroad diagrams [B-1](#)

Continuation directive lines [16-2](#)

Copy operation (move statement) [12-27](#)

CPU directive [16-22](#)

CPU type must be set (error 83) [A-23](#)

CRE, directives for

- ENV [16-33](#)
- HEAP [16-42](#)

CROSSREF directive

- description [16-22](#)
- with USEGLOBALS directive [16-23](#)

CROSSREFT directive

- with SYNTAX directive [16-92](#)

Cross-references

- collecting with CROSSREF directive [16-22](#)
- listing with LMAP directive [16-61](#)

**D**

## D

(suffix for INT(32) nonhexadecimal numbers) [3-12](#)

Data blocks, relocatable [11-1](#)

- Data declarations must precede (error 24) [A-10](#)
- Data operations  
  scan statements [12-34](#)
- Data Representation [3-1](#)
- Data sets  
  as a TAL feature [1-2](#)
- Data stack  
  increasing with EXTENDSTACK directive [16-37](#)  
  setting size with STACK directive [16-87](#)
- Data transfer  
  assignment statement [12-4](#)  
  bit-deposit assignments [12-4](#)  
  move statement [12-27](#)  
  STACK statement [12-36](#)  
  STORE statement [12-37](#)
- Data types  
  descriptions [3-1](#)  
  obtaining with \$TYPE [14-41](#)  
  of expressions [4-2](#)  
    arithmetic, Boolean [4-12](#)  
    arithmetic, logical [4-3](#)  
    arithmetic, signed [4-3](#)  
    arithmetic, unsigned [4-3](#)  
    relational, signed [4-14](#)  
    relational, unsigned [4-15](#)  
  of standard function arguments [14-5](#)  
  operations for [3-6](#)  
  standard functions for [3-6](#)  
  storage units [3-5](#)
- DATAPAGES directive  
  description [16-25](#)  
  with EXTENDSTACK directive [16-38](#)  
  with STACK directive [16-87](#)
- Debugger, selecting with INSPECT directive [16-54](#)
- Declaration must be in block (error 94) [A-25](#)
- base-address equivalenced variables [10-1](#)
- Declaration must be in block (error 94) (continued)  
  BLOCKs [11-1](#)  
  compilation-unit names [11-1](#)  
  DEFINEs [5-4](#)  
  entry points [13-19](#)  
  equivalenced variables [10-1](#)  
  global data blocks [11-5](#)  
  labels [13-22](#)  
  LITERALs [5-1](#)  
  NAME [11-1](#)  
  procedures [13-2](#)  
  read-only arrays [7-5](#)  
  simple variables [6-1](#)  
  structure pointers [9-7](#)  
  subprocedures [13-15](#)  
  syntax summary  
    bracket-and-brace diagrams [C-1](#)  
    railroad diagrams [B-1](#)  
    system global pointers [15-3](#)  
    'SG'-equivalenced variables [15-4](#)
- Declarations  
  structures [8-1](#)
- Declarations arrays [7-1](#)
- DECS directive  
  description [16-27](#)  
  in DEFINEs [5-4](#)
- DECS pseudocode, CODE statement [12-17](#)
- Default OCCURS count (warning 43) [A-48](#)
- DEFEXPAND directive  
  DEFINE expansion [5-3, 5-5](#)  
  description [16-27](#)
- DEFINE commands, TACL [16-36](#)
- DEFINEs  
  allocation [5-7](#)  
  as parameters [5-7](#)  
  declaring [5-7](#)  
  expansion of [5-6](#)  
  invoking [5-6](#)

DEFINES (continued)  
 passing parameters [5-7](#)  
 specifying parameters [5-3](#)  
 syntax summary  
   bracket-and-brace diagrams [C-1](#)  
   railroad diagrams [B-1](#)

DEFINETO<sub>G</sub> directive  
 description [16-29](#)  
 with IF directive [16-29](#)

Definition structures  
 base-address equivalenced [10-14](#)  
 declaring [8-3](#)

Definition substructures [8-10](#)  
 declaring [8-10](#)  
 redefinitions [8-20](#)

Delimiters [2-7](#)

Dereferencing operator  
 description [2-9](#)

Diagnostic directives  
 ERRORFILE [16-34](#)  
 ERRORS [16-37](#)  
 RELOCATE [16-67](#)  
 summary of [16-8](#)  
 WARN [16-95](#)

Diagnostic messages [A-1](#)

CROSSREF [A-56](#)

Directive can appear once (warning 61) [A-51](#)

Directive cannot be pushed (warning 39) [A-47](#)

Directive not allowed (error 155) [A-36](#), [A-49](#)

Directive stacks [16-3](#)

Directives  
 in DEFINES [5-4](#)  
 syntax summary  
   bracket-and-brace diagrams [C-1](#)  
   railroad diagrams [B-1](#)

Disk file names [16-4](#)

Division by powers of 2 [4-30](#)

Division by zero (error 59) [A-18](#)

Division operator  
 unsigned [4-10](#)  
 unsigned modulo [4-10](#)

DO keyword  
 DO statement [12-19](#)  
 FOR statement [12-22](#)  
 WHILE statement [12-40](#)

Do not use SQL statement (warning 93) [A-56](#)

DO statement [12-19](#)  
 does not work (error 82) [A-23](#)  
 does not work (warning 89) [A-56](#)

Doubleword [3-5](#)

DOWNTO keyword, FOR statement [12-22](#)

DROP statement  
 description [12-20](#)  
 with FOR statement [12-23](#)  
 with USE statement [12-39](#)

DUMPCONS directive  
 description [16-31](#)  
 with CODE statement [16-32](#)

## E

E (suffix for REAL numbers) [3-15](#)

Edit file invalid format (error 53) [A-17](#)

ELEMENTS keyword  
 group comparison expression [4-24](#)  
 move statement [12-28](#)

ELSE keyword  
 IF expression [4-22](#)  
 IF statement [12-26](#)

END keyword  
 compound statement [12-1](#)

ENDIF  
 description [16-32](#)

ENDIF directive  
 with RESETTO<sub>G</sub> directive [16-69](#)  
 with SETTO<sub>G</sub> directive [16-83](#)

Entry-point declarations [13-18](#)

- ENV directive  
     description [16-32](#)  
     with EXTENDSTACK directive [16-38](#)  
     with STACK directive [16-87](#)
- Equivalenced variables [10-1](#)  
     definition structures [10-5](#)  
     referral structures [10-8](#)  
     simple pointers [10-4](#)  
     simple variables [10-3](#)  
     structure pointers [10-11](#)  
     syntax summary  
         bracket-and-brace diagrams [C-1](#)  
         railroad diagrams [B-1](#)
- Equivalences between blocks (error 97) [A-25](#)
- Error file exists and (error 124) [A-30](#)
- Error handling, file-system errors [4-16](#)
- Error Messages  
     descriptions [A-2](#)
- Error messages  
     maximum allowed, specifying [16-37](#)
- ERRORFILE directive [16-34](#)
- ERRORS directive [16-37](#)
- Exceeded allocated space SQLMEM (error 148) [A-35](#)
- Exceeded available memory (error 154) [A-36](#)
- Exponents, REAL or REAL (64) format [3-15](#)
- Expression passed by reference (warning 36) [A-47](#)
- Expression too complex (error 88) [A-24](#)
- Expressions  
     arithmetic [4-5](#)  
     assignment [4-19](#)  
     CASE [4-19](#)  
     conditional [4-12](#)  
     constant [4-2](#)  
     data types of [4-2](#)  
     description of [4-1](#)  
     effect on hardware indicators [4-9](#)
- Expressions (continued)  
     fixed-point, scaling of [4-7](#)  
     group comparison [4-23](#)  
     IF [4-21](#)  
     logical operations [4-11](#)  
     precedence of operators [4-3](#)  
     relational operations [4-13](#)  
     syntax summary  
         bracket-and-brace diagrams [C-1](#)  
         railroad diagrams [B-1](#)
- Extended address of STRING P-rel, (warning 62) [A-52](#)
- Extended arrays in subprocedures (error 98) [A-25](#)
- Extended move or group comparison (error 122) [A-30](#)
- Extended stack  
     setting size with LARGESTACK directive [16-57](#)  
     stack overflow trap with CHECK directive [16-17](#)
- Extended (32-bit) pointers [9-1, 9-6](#)
- EXTENDSTACK directive  
     description [16-37](#)  
     with DATAPAGES directive [16-26](#)
- EXTENDTALHEAP directive [16-38](#)
- EXTENSIBLE attribute of procedures [13-7](#)
- External Entry Point table [13-4](#)
- EXTERNAL keyword  
     procedure declaration [13-8](#)  
     procedure entry-point declaration [13-19](#)
- External references, resolving with SEARCH directive [16-79](#)

## F

- F (suffix for FIXED nonhexadecimal numbers) [3-13](#)
- File error (error 113) [A-28](#)
- File map, listing with FMAP directive [16-40](#)
- File names [16-4](#)

- File records (structures) [8-1](#)
- File system DEFINE name not permitted (warning 63) [A-52](#)
- File system DEFINEs not enabled (warning 41) [A-48](#)
- File-system errors, testing for [4-16](#)
- Filler declaration [8-12](#)
- First argument of \$OPTIONAL (error 177) [A-39](#)
- FIXED data type
- description [3-3](#)
  - in assignment statements [12-6](#)
  - numeric format [3-14](#)
  - obtaining
    - \$DFIX [14-14](#)
    - \$FIX [14-16](#)
    - \$FIXD [14-16](#)
    - \$FIXR [14-18](#)
    - \$IFIX [14-21](#)
    - \$LFIX [14-25](#)
  - rounding with ROUND directive [16-70](#)
- FIXED parameter type [13-12](#)
- Fixed point
- functions
    - summary of [14-37](#)
    - \$POINT [14-37](#)
    - \$SCALE [14-39](#)
  - numbers
    - ranges by data type [3-2](#)
- FIXED point scaling mismatch (warning 25) [A-45](#)
- FIXED (\*)
- data type [3-2](#)
  - parameter type [13-12](#)
- Fixed-point
- arithmetic [4-7](#)
  - implied setting [3-3](#)
- FIXERRS TACL macro, ERRORFILE directive [16-34](#)
- FIXUP directive [16-39](#)
- Floating-point numbers
- ranges by data type [3-2](#)
  - REAL format [3-14](#)
  - REAL (64) format [3-14](#)
- FMAP directive [16-40](#)
- FOR
- group comparison expression [4-13](#)
- FOR keyword
- FOR statements [12-18](#)
- FOR statement
- description [12-22](#)
  - optimized [12-22, 12-23](#)
  - standard [12-22, 12-23](#)
- For statement
- with USE statement [12-23](#)
- Formal parameter type missing (error 17) [A-9](#)
- FORTRAN
- run-time environment, specifying with ENV directive [16-33](#)
  - TAL procedure language attribute [16-58](#)
- FORWARD keyword
- procedure declaration [13-4](#)
  - procedure entry-point declaration [13-19](#)
  - subprocedure declaration [13-17](#)
  - subprocedure entry-point [13-20](#)
- Forward/external parameter count (error 62) [A-19](#)
- fpoint
- changing with \$SCALE [14-39](#)
  - obtaining with \$POINT [14-37](#)
  - positive or negative [3-3](#)
  - rounding with ROUND directive [16-70](#)
  - specifying with \$LFIX [14-25](#)
- Fractions
- FIXED format [3-13](#)
  - REAL format [3-15](#)
  - REAL (64) format [3-15](#)
- FULL pseudocode, CODE statement [12-17](#)

## Functions

- declaring [13-2](#)
- description [13-1](#)
- procedures [13-2](#)
- subprocedures [13-1](#)
- with RETURN statement [12-32](#)

## G

- Global data declarations
  - in BLOCK declaration [11-1](#)
  - relocatable [11-1](#)
  - retrieving with USEGLOBALS directive [16-77](#)
  - saving with SAVEGLOBALS directive [16-75](#)
  - unblocked [11-5](#)
- Global map, listing with GMAP directive [16-41](#)
- Global or nested subprocedure (error 19) [A-9](#)
- Global primary exceeds 256 words (error 5) [A-4](#)
- GMAP directive [16-41](#)
- GOTO statement
  - description [12-25](#)
  - labels [13-22](#)
- Group comparison expressions
  - description [4-13](#)
  - in conditional expressions [4-13](#)

## H

- Hardware indicators, testing [4-16](#)
- HEAP directive [16-42](#)
- Heap, compiler, increasing with EXTENDTALHEAP directive [16-38](#)
- Heap, user, setting with HEAP directive [16-43](#)
- HIGHPIN directive [16-43](#)
  - description [16-43](#)
  - with RUNNAMED directive [16-73](#)
- HIGHREQUESTERS directive [16-45](#)

## I

- ICODE directive [16-46](#)
- Identifier cannot be indexed (error 68) [A-20](#)
- Identifier declared more than once (error 2) [A-4](#)
- Identifier exceeds (warning 1) [A-40](#)
- Identifier for SQLMEM address (error 147) [A-35](#)
- Identifier for SQLMEM length (error 146) [A-34](#)
- Identifier maps, listing with MAP directive [16-62](#)
- Identifiers
  - classes [2-4](#)
  - format [2-4](#)
  - listing with GMAP directive [16-41](#)
- IF directive
  - description [16-47](#)
  - with DEFINETOG directive [16-69](#)
  - with RESETTOG directive [16-69](#)
  - with SETTOG directive [16-48](#)
  - with TARGET directive [16-49](#)
- IF expression [4-21](#)
- IF statement [12-26](#)
- IFNOT directive
  - description [16-47](#)
  - with DEFINETOG directive [16-48](#)
  - with RESETTOG directive [16-48](#)
  - with SETTOG directive [16-48](#)
  - with TARGET directive [16-49](#)
- Illegal array bounds (error 18) [A-9](#)
- Illegal bit field (error 20) [A-9](#)
- Illegal block relocation position (error 128) [A-31](#)
- Illegal block relocation specifier (error 127) [A-31](#)
- Illegal branch (error 58) [A-18](#)
- Illegal command list (warning 22) [A-44](#)
- Illegal constant format (error 87) [A-23](#)
- Illegal conversion to EXTENSIBLE (error 101) [A-26](#)

Illegal digit (error 6) [A-5](#)  
 Illegal drop of USE variable (error 63) [A-19](#)  
 Illegal for INT (error 47) [A-16](#)  
 Illegal global declarations (error 33) [A-13](#)  
 Illegal index register (error 38) [A-14](#)  
 Illegal indirection specification (error 46) [A-16](#)  
 Illegal instruction (error 44) [A-16](#)  
 Illegal move or group comparison (error 4) [A-4](#)  
 Illegal operand for ACON (error 102) [A-26](#)  
 Illegal option (warning 2) [A-40](#)  
 Illegal order of directives (warning 53) [A-50](#)  
 Illegal parameter or (error 65) [A-19](#)  
 Illegal public name (error 166) [A-38](#)  
 Illegal range (error 36) [A-13](#)  
 Illegal reference parameter (error 54) [A-17](#)  
 Illegal reference (error 11) [A-6](#)  
 Illegal size given (error 162) [A-37](#)  
 Illegal STRUCT or SUBSTRUCT reference (error 76) [A-21](#)  
 Illegal SUBPROC attribute (error 55) [A-18](#)  
 Illegal symbol (error 43) [A-15](#)  
 Illegal syntax (error 27) [A-11](#)  
 Illegal UNSIGNED variable (error 75) [A-21](#)  
 Illegal use of code relative (error 28) [A-11](#)  
 Illegal use of identifier (error 29) [A-12](#)  
 Illegal use of period (error 116) [A-29](#)  
 Illegal use of @ (error 119) [A-29](#)  
 Illegal USE variable (error 56) [A-18](#)  
 Increase size internal heap (error 169) [A-38](#)  
**Index**

- base-address equivalenced variables [10-12](#)
- extended, generating
  - INHIBITXX directive [16-50](#)
  - INT32INDEX directive [16-55](#)

**Index register**

- reserving with USE statement [12-20](#)

**Index register allocation** (error 51) [A-17](#)

**Index registers**

- dropping with DROP statement [12-39](#)

**Index truncated** (warning 14) [A-43](#)  
**Indirection mode specified** (error 103) [A-26](#)  
**Indirection must be supplied** (error 72) [A-21](#)  
**Indirection symbols** [2-7](#)  
**INHIBITXX directive**

- description [16-50](#)
- with USEGLOBALS directive [16-77](#)

**Initialization**

- arrays [7-2](#)
  - read-only arrays [7-5](#)
  - simple pointers [9-2](#)
  - simple variables [6-1](#)
  - structure pointers [9-6](#)

- Initialization exceeds space (error 99) [A-26](#)
- Initialization illegal with reference (error 15) [A-8](#)
- Initialization list exceeds (warning 3) [A-40](#)
- Initialization of local (error 126) [A-31](#)
- Initialization of UNSIGNED arrays (warning 90) [A-56](#)
- Initialization value too complex (warning 17) [A-44](#)

**INNERLIST directive**

- description [16-52](#)
- with OPTIMIZE directive [16-53](#)
- with USE directive [12-39](#)

- Inspect debugger, selecting
  - with INSPECT directive [16-54](#)
  - with SAVEABEND directive [16-54](#)

**INSPECT directive** [16-54](#)  
**Instruction codes**

- listing
  - in octal with CODE directive [16-18](#)
  - procedure mnemonics with ICODE directive [16-46](#)
  - statement mnemonics with INNERLIST directive [16-52](#)
  - specifying with CODE statement [12-16](#)

Instruction trap and CHECK directive [16-17](#)  
 Insufficient disk space on swap volume  
 (error 16) [A-8](#)  
 INT attribute [9-6](#)  
 INT data type  
   description [3-2](#)  
   numeric format [3-10](#)  
   obtaining  
     high-order word with \$HIGH [14-20](#)  
     low-order word with \$INT [14-21](#)  
     rounded low-order word with  
     \$INTR [14-22](#)  
     signed value with \$FIXI [14-17](#)  
     unsigned value with \$FIXL [14-18](#)  
 INT parameter type [13-12](#)  
 INT (16), alias of INT [3-4](#)  
 INT (32) data type  
   description [3-2](#)  
   numeric format [3-11](#)  
 INT (64), alias of FIXED (0) [3-4](#)  
 INT32INDEX directive  
   description [16-55](#)  
   with INHIBITXX directive [16-51](#)  
 Integers  
   FIXED format [3-13](#)  
   INT format [3-10](#)  
   INT (32) format [3-12](#)  
   REAL format [3-15](#)  
   REAL (64) format [3-15](#)  
   STRING format [3-9](#)  
 INTERRUPT attribute of procedures [13-6](#)  
 INT(16), alias of INT [3-3](#)  
 INT(32) data type  
   obtaining  
     \$DBL [14-11](#)  
     \$DBLL [14-12](#)  
     \$DBLR [14-13](#)  
     \$UDBL [14-41](#)  
 INT(32) parameter type [13-12](#)

Invalid ABSLIST addresses (warning  
 6) [A-41](#)  
 Invalid declaration for length (error  
 139) [A-33](#)  
 Invalid declaration for string (error  
 141) [A-34](#)  
 Invalid default volume (error 91) [A-24](#)  
 Invalid directive option (warning 79) [A-54](#)  
 Invalid file or subvolume (warning 47) [A-49](#)  
 Invalid number form (error 78) [A-22](#)  
 Invalid object file (error 90) [A-24](#)  
 Invalid parameter list (warning 84) [A-55](#)  
 Invalid template access (error 69) [A-20](#)  
 Invoked forward PROC (error 81) [A-22](#)  
 Item does not have extended (error  
 25) [A-10, A-53](#)

## L

L (suffix for REAL(64) numbers) [3-15](#)  
 LABEL  
   declaring [13-21](#)  
 Label  
   dropping with DROP statement [12-20](#)  
 Label declared more than once (error  
 21) [A-10](#)  
 Labeled CASE statement [12-11](#)  
 Labels  
   emitting with DUMPCONS  
   directive [16-32](#)  
 LAND operator [4-11](#)  
 Language attribute conflict (warning  
 74) [A-53](#)  
 LANGUAGE attribute of procedures [13-8](#)  
 Language attribute only (error 161) [A-37](#)  
 LARGESTACK directive [16-57](#)  
 Layout  
   definition structures [8-10](#)  
   structures [8-2](#)  
   substructures [8-10](#)  
   template structures [8-5](#)  
 Left shifts, bit [4-30](#)  
 Length mismatch SQL (error 153) [A-36](#)

Length of structure (warning 55) [A-50](#)  
 Length parameters  
   CALL statement [12-9](#)  
   passing conditionally with  
   \$OPTIONAL [14-33](#)  
 LIBRARY directive [16-58](#)  
 LINES directive [16-59](#)  
 LIST directive  
   description [16-59](#)  
   with SOURCE directive [16-86](#)  
 List length used for compare (warning 23) [A-44](#)  
 Literal initialized with address (warning 50) [A-49](#)  
 LITERALS  
   declaring [5-1](#)  
   in arithmetic expressions [4-2](#)  
   syntax summary  
     bracket-and-brace diagrams [C-7](#)  
     railroad diagrams [B-7](#)  
 LMAP directive [16-61](#)  
 Load map, listing with LMAP directive [16-61](#)  
 Local declarations [13-13](#)  
 Logical file names [16-5](#)  
 Logical operators [4-11](#)  
 LOR operator [4-11](#)

## M

Machine instruction statements  
   CODE statement [12-15](#)  
   DROP statement [12-1](#)  
   USE statement [12-1](#)  
 MAIN attribute of procedures [13-5](#)  
 MAIN cannot return value (warning 91) [A-56](#)  
 MAP directive [16-62](#)  
 Memory pages, setting with DATAPAGES directive [16-87](#)  
 Messages  
   compiler [A-1](#)

Messages (continued)  
   error [A-1](#)  
   SYMSERV [A-57](#)  
   warning  
     general description [A-1](#)  
     specific descriptions [A-40](#)  
 Minimum-maximum functions  
   summary of [14-1](#)  
   \$LMAX [14-26](#)  
   \$LMIN [14-26](#)  
   \$MAX [14-27](#)  
   \$MIN [14-27](#)  
 Minus operator  
   unary [4-5](#)  
 Missing FOR part (error 115) [A-29](#)  
 Missing identifier (error 37) [A-13](#)  
 Missing initialization code relative (error 52) [A-17](#)  
 Missing item (error 48) [A-16](#)  
 Missing variable (error 34) [A-13](#)  
 Modulo division operator [4-10](#)  
 Move or group comparison (error 125) [A-31](#)  
 MOVE statement [12-27](#)  
 Multiple defined SECTION (warning 7) [A-41](#)  
 Multiplication by powers of two [4-30](#)  
 Multiplication operator  
   signed [4-3](#)  
   unsigned [4-3](#)

## N

Named toggles  
   specifying with DEFINETOG [16-29](#)  
   turning off with RESETTOG [16-68](#)  
   turning on with SETTOG [16-82](#)  
   using with IF or IFNOT directive [16-29](#)  
 NAMES  
   syntax summary  
     bracket-and-brace diagrams [C-16](#)

- NAMEs (continued)  
     railroad diagrams [B-22](#)
- Nested routine declarations (error 12) [A-7](#)
- Nesting parametric-DEFINE (error 100) [A-26](#)
- NEUTRAL attribute, ENV directive [16-32](#)
- Next address  
     group comparison expression [4-25](#),  
[4-26](#)  
     move statement [12-29](#)  
     RSCAN statement [12-35](#)  
     SCAN statement [12-35](#)
- No file system DEFINE (warning 64) [A-52](#)
- No SCAN for extended (error 84) [A-23](#)
- NOABORT directive [16-12](#)
- NOABSLIST directive [16-7](#)
- NOCHECK directive [16-17](#)
- NOCODE directive [16-19](#)
- NOCOMPACT directive [16-21](#)
- NOCROSSREF directive [16-23](#)
- NODEFEXPAND directive [16-28](#)
- NOFIXUP directive [16-39](#)
- NOFMAP directive [16-40](#)
- NOGMAP directive [16-41](#)
- NOICODE directive [16-46](#)
- NOINHIBITXX directive [16-50](#)
- NOINNERLIST directive [16-53](#)
- NOINSPECT directive [16-54](#)
- NOINT32INDEX directive [16-55](#)
- NOLIST directive [16-60](#)
- NOMAP directive [16-62](#)
- Nonrelocatable global reference (warning 46) [A-49](#)
- NOPRINTSYM directive [16-67](#)
- NOROUND directive [16-71](#)
- NOSAVEABEND directive [16-74](#)
- NOSUPPRESS directive [16-89](#)
- NOSYMBOLS directive [16-90](#)
- Not allowed with UNSIGNED (error 123) [A-30](#)
- Not defined for INT(32), FIXED, or REAL (error 8) [A-5](#)
- Not host variable (error 138) [A-33](#)
- NOT operator [4-12](#)
- NOWARN directive [16-96](#)
- Null statement [12-2](#)
- Number bases  
     FIXED format [3-13](#)  
     INT format [3-10](#)  
     INT (32) format [3-11](#)  
     STRING format [3-9](#)
- Number of global data blocks (warning 59) [A-51](#)
- Number of sparse (error 111) [A-28](#)
- Numbers  
     FIXED format [3-13](#)  
     INT format [3-10](#)  
     INT (32) format [3-11](#)  
     REAL format [3-15](#)  
     REAL (64) format [3-15](#)  
     STRING format [3-9](#)
- Numeric toggles  
     specifying with DEFINETOG [16-29](#)  
     turning off RESETTOG [16-68](#)  
     turning on with SETTOG [16-82](#)  
     using with IF or IFNOT directive [16-47](#)

**O**

- Object files [11-1](#)
- Object-file content directives  
     ASSERTION [16-8](#)  
     CHECK [16-8](#)  
     COMPACT [16-8](#)  
     CPU [16-8](#)  
     DUMPCONS [16-8](#)  
     FIXUP [16-8](#)  
     INHIBITXX [16-9](#)  
     INT32INDEX [16-9](#)  
     OLDFLTSTDFUNC [16-9](#)  
     OPTIMIZE [16-9](#)  
     PEP [16-9](#)  
     ROUND [16-9](#)

- Object-file content directives (continued)
  - SEARCH [16-9](#)
  - SQL [16-9](#)
  - SQLMEM [16-9](#)
  - summary of [16-8](#)
  - SYNTAX [16-92](#)
- Octal code, listing with CODE directive [16-18](#)
- Odd-byte references, extended (32-bit) pointers [9-3](#)
- Of keyword
  - CASE expression [4-20](#)
- OLD attribute, ENV directive [16-33](#)
- OLDFLTSTDFUNC directive [16-63](#)
- One or more illegal attributes (warning 29) [A-46](#)
- One's complement, obtaining with \$COMP [14-4](#)
- Only allowed with variable (error 40) [A-14](#)
- Only arrays of simple (error 89) [A-24](#)
- Only data may be indexed (error 60) [A-18](#)
- Only initialization with constants (error 14) [A-8](#)
- Only INT index (error 145) [A-34](#)
- Only INT or STRING STRUCT pointers (error 71) [A-21](#)
- Only INT values allowed (error 13) [A-7](#)
- Only INT(32) values (error 45) [A-16](#)
- Only items subordinate to structure (error 70) [A-20](#)
- Only label or USE (error 30) [A-12](#)
- Only one language attribute (error 160) [A-37](#)
- Only PROC or SUBPROC (error 31) [A-12](#)
- Only standard indirection allowed (error 22) [A-10](#)
- Only STRING arrays SQL (error 151) [A-35](#)
- Only STRUCT items allowed(error 121) [A-30](#)
- Only structure identifier (error 75) [A-21](#)
- Only type INT and INT(32) index (error 120) [A-30](#)
- Open failed (error 39) [A-14](#)
- Operands in arithmetic expressions [4-1](#)
- Operating system services [1-2](#)
- Operators
  - arithmetic
    - signed [4-5](#)
    - unsigned [4-5](#)
  - bit-shift [4-27](#)
  - Boolean [4-12](#)
  - logical [4-5](#)
  - precedence of [4-3](#)
  - relational [4-13](#)
  - summary of [2-9](#)
- Optimization level, selecting [16-64](#)
- OPTIMIZE 2 register allocation (error 80) [A-22](#)
- OPTIMIZE directive
  - description [16-64](#)
  - with INNERLIST directive [16-53](#)
- OR operator [4-14](#)
- OTHERWISE keyword
  - CASE expression [4-20](#)
  - CASE statement, labeled [12-12](#)
  - CASE statement, unlabeled [12-13](#)
- Overflow
  - causes [4-17](#)
  - dealing with [4-18](#)
  - testing with \$OVERFLOW [14-35](#)
- Overflow, causes of [14-35](#)

## P

- PAGE directive [16-65](#)
- Page heading, printing with PAGE directive [16-65](#)
- PAGES option SQL (warning 81) [A-55](#)
- Parameter list
  - CALL statement [12-10](#)
- Parameter mismatch (error 1) [A-3](#)
- Parameter pairs
  - passing

Parameter pairs (continued)  
 conditionally with  
**\$OPTIONAL** [14-32](#)

Parameter types [13-9](#)

Parameters  
 actual, CALL statement [12-9](#)  
 checking with \$PARAM [14-36](#)  
 formal specification [13-8](#)  
 location, obtaining with  
**\$BOUNDS** [15-12](#)  
 passing  
 conditionally with \$OPTIONAL [14-4](#)  
 in expressions [4-17](#)

PASCAL (TAL procedure language attribute) [13-10](#)

PEP directive  
 description [16-66](#)  
 with ABSLIST directive [16-13](#)

PEP size estimate (warning 5) [A-41](#)

PEP table  
 description [13-4](#)  
 setting size with PEP directive [16-66](#)

PIN and HIGHPIN directive [16-43](#)

Plus operator  
 binary signed [4-7](#)  
 binary unsigned [4-9](#)  
 unary [4-3](#)

Pointers  
 description [9-1](#)  
 extended (32-bit) [9-1](#)  
 simple pointers [9-2](#)  
 standard (16-bit) [9-1](#)  
 syntax summary  
   bracket-and-brace diagrams [C-12](#)  
   railroad diagrams [B-15](#)  
 system global pointers [15-3](#)  
 'SG'-equivalenced  
   simple pointers [15-8](#)  
   structure pointers [15-9](#)

POP prefix, directives [16-3](#)

POPCHECK directive [16-17](#)  
 POPCODE directive [16-19](#)  
 POPDEFEXPAND directive [16-28](#)  
 POPICODE directive [16-46](#)  
 POPINNERLIST directive [16-53](#)  
 POPINT32INDEX directive [16-55](#)  
 POPLIST directive [16-60](#)  
 POPMAP directive [16-62](#)  
 Precedence of operators [4-3](#)  
 Previous data block not ended (error 93) [A-24](#)  
 Previous errors and warnings (warning 60) [A-51](#)  
 PRINTSYM directive  
 description [16-67](#)

PRIV attribute of procedures [13-6](#)

Private data area, as a TAL feature [1-2](#)

Private global data blocks [11-3](#)

PRIVATE keyword, BLOCK declaration [11-3](#)

Privileged functions  
 \$AXADR [15-11](#)  
 \$BOUNDS [15-11](#)  
 \$SWITCHES [15-11](#)

Privileged mode [15-1](#)

Privileged operations [15-2](#)

Privileged procedures  
 description [15-1](#)  
 syntax summary  
   bracket-and-brace diagrams [C-30](#)  
   railroad diagrams [B-49](#)

PROC keyword, procedure declaration [13-10](#)

PROC not FORWARD (warning 19) [A-44](#)

Procedure calls (CALL statement) [12-10](#)

Procedure code space (error 131) [A-32](#)

Procedure declared FORWARD (warning 40) [A-47](#)

Procedure Entry Point table [13-4](#)

Procedure missing label (error 104) [A-27](#)

Procedure was previously, language (error 164) [A-37](#)

Procedure was previously, public name (error 165) [A-38](#)  
**Procedures** [13-1](#)

- attributes [13-5](#)
- CALLABLE attribute [13-6](#)
- description [13-1](#)
- EXTENSIBLE attribute [13-7](#)
- formal parameter specification [13-8](#)
- FORWARD declaration [13-15](#)
- INTERRUPT attribute [13-6](#)
- LANGUAGE attribute [13-8](#)
- MAIN attribute [13-5](#)
- nonprivileged [15-2](#)
- PRIV attribute [13-6](#)
- privileged [15-1](#)
- RESIDENT attribute [13-6](#)
- standard [14-1](#)
- syntax summary
  - bracket-and-brace diagrams [C-16](#)
  - railroad diagrams [B-23](#)
- typed [13-1](#)
- VARIABLE attribute [13-7](#)
- with RETURN statement [12-31](#)

Process identification number (PIN), HIGHPIN directive [16-43](#)  
**PROC(32)** parameter type [13-10](#)  
**CROSSREF** [A-23](#)  
**Program control**

- ASSERT statement [12-1](#)
- CALL statement [12-1](#)
- CASE statement, unlabeled [12-13](#)
- conditional expressions [4-12](#)
- DO statement [12-19](#)
- FOR statement [12-22](#)
- GOTO statement [12-25](#)
- IF statement [12-26](#)
- RETURN statement [12-31](#)
- statements [12-1](#)
- WHILE statement [12-40](#)

Pseudocodes, CODE statement [12-17](#)

Public name only (error 163) [A-37](#)  
**PUSH** prefix, directives [16-3](#)  
**PUSHCHECK** directive [16-17](#)  
**PUSHCODE** directive [16-19](#)  
**PUSHDEFEXPAND** directive [16-28](#)  
**PUSHICODE** directive [16-46](#)  
**PUSHINNERLIST** directive [16-53](#)  
**PUSHINT32INDEX** directive [16-55](#)  
**PUSHLIST** directive [16-60](#)  
**PUSHMAP** directive [16-62](#)  
P-relative array passed (warning 4) [A-40](#)

## **Q**

Quadrupletword [3-5](#)

## **R**

Railroad syntax summary [B-1](#)  
**Read-only arrays** [7-5](#)  
**REAL** data type
 

- description [3-2](#)
- numeric format [3-14](#)
- obtaining
  - \$FLT [14-19](#)
  - \$FLTR [14-20](#)

REAL parameter type [13-9](#)  
**REAL** underflow or overflow (error 79) [A-22](#)  
**REAL (32)**, alias of REAL [3-4](#)  
**REAL (64)** data type
 

- description [3-2](#)
- numeric format [3-14](#)

REAL(64) data type
 

- obtaining
  - \$EFLT [14-15](#)
  - \$EFLTR [14-15](#)

Records (structures) [8-1](#)  
**Recursion**, as a TAL feature [1-2](#)  
Recursive DEFINE invocation (error 3) [A-4](#)  
Redefinition offset (warning 34) [A-46](#)  
Redefinition size (warning 33) [A-46](#)

Redefinitions  
 arrays [8-19](#)  
 referral substructures [8-11](#)  
 rules for [8-17](#)  
 simple pointers [8-23](#)  
 structure pointers [8-24](#)

Referenced subprocedure FORWARD  
(error 106) [A-27](#)

Referral structures  
 declaring [8-6](#)

Referral substructures  
 redefinitions [8-22](#)

Register stack  
 loading values with STACK  
 statement [12-36](#)  
 removing values with STORE  
 statement [12-37](#)

Register stack not empty (warning 87) [A-55](#)

Relational expressions [4-13](#)

Relational operators  
 condition code indicator, testing [4-13](#)  
 conditional expressions [4-13](#)  
 group comparison expression [4-19](#)  
 signed [4-23](#)  
 unsigned [4-23](#)

RELEASE1 and RELEASE2 (warning 65) [A-52](#)

Relocatable global data block [11-2](#)

RELOCATE directive [16-67](#)

Repetition constant lists [3-16](#)

Reserved toggle name (warning 80) [A-54](#)

RESETTOG directive  
 description [16-68](#)

RESIDENT attribute of procedures [13-6](#)

RETURN not encountered (warning 32) [A-46](#)

RETURN statement [12-31](#)

Right shift emitted (warning 15) [A-43](#)

Right shifts, bit [4-30](#)

ROUND directive  
 description [16-70](#)

ROUND directive (continued)  
 with assignment statement [12-5](#)

Rounding  
 ROUND directive [16-70](#)  
 type-transfer functions [14-5](#)

Routine declared forward (error 26) [A-11](#)

Routines [1-3](#)

RP directive  
 description [16-71](#)  
 in DEFINEs [5-4](#)

RP internal count  
 obtaining with \$RP [14-38](#)

RP or S register (warning 9) [A-42](#)

RP register overflow (warning 10) [A-42](#)

RP value  
 with FOR statement [12-18](#)  
 with STACK statement [12-36](#)  
 with STORE statement [12-37](#)

RSCAN statement [12-34](#)

RUNNAMED directive [16-73](#)

Run-time environment directives  
 DATAPAGES [16-25](#)  
 ENV [16-32](#)  
 EXTENDSTACK [16-37](#)  
 HEAP [16-42](#)  
 HIGHPIN [16-43](#)  
 HIGHREQUESTERS [16-45](#)  
 INSPECT [16-54](#)  
 LARGESTACK [16-57](#)  
 LIBRARY [16-58](#)  
 RUNNAMED [16-73](#)  
 SAVEABEND [16-73](#)  
 STACK [16-87](#)  
 SUBTYPE [16-87](#)  
 summary of [16-11](#)  
 SYMBOLS [16-90](#)

# S

S register

  checking for overflow [15-12](#)

S register underflow (warning 38) [A-47](#)

Save file, generating with SAVEABEND directive [16-73](#)

SAVEABEND directive

  description [16-73](#)

  with INSPECT directive [16-74](#)

  with Inspect directive [16-73](#)

SAVEGLOBALS and USEGLOBALS (warning 57) [A-51](#)

SAVEGLOBALS directive

  description [16-75](#)

  with BEGINCOMPILATION

  directive [16-16](#)

Scale point must be constant (error 64) [A-19](#)

Scaling, FIXED values

  changing fpont with \$SCALE [14-39](#)

  in expressions [4-7](#)

SCAN statements [12-34](#)

Scientific notation, binary [3-15](#)

SEARCH directive

  description [16-79](#)

Second argument \$OPTIONAL (error 176) [A-39](#)

Secondary entry point missing (error 105) [A-27](#)

SECTION directive

  description [16-81](#)

SECTION name not found (warning 8) [A-41](#)

Section name, specifying with SECTION directive [16-81](#)

Segment number lost (warning 35) [A-46](#)

Selector

  CASE expression [4-20](#)

  CASE statement, labeled [12-11](#)

  CASE statement, unlabeled [12-13](#)

SETTOG directive

  description [16-82](#)

Shifts, bit [4-29](#)

Signed arithmetic operators [4-6](#)

Signed left-shift operator [4-30](#)

Simple pointers

  addresses [9-2](#)

  as structure items [8-2](#)

  base-address equivalenced [10-13](#)

  declaring [9-2](#)

  redefinitions [8-23](#)

  @ operator [9-4](#)

Simple variables

  as structure items [8-1](#)

  base-address equivalenced [10-11](#)

  declaring [6-1](#)

  equivalenced [10-1](#)

  redefinitions [8-2](#)

  syntax summary

    bracket-and-brace diagrams [C-7](#)

    railroad diagrams [B-8](#)

  'SG'-equivalenced [15-4](#)

Size

  combined primary global blocks [11-5](#)

  identifiers [2-4](#)

  storage units [3-5](#)

  structures [8-4](#)

Source code, listing with LIST directive [16-59](#)

Source commands nested (error 67) [A-20](#)

SOURCE directive

  description [16-84](#)

  with ABORT directive [16-12](#)

  with COLUMNS directive [16-20](#)

  with FMAP directive [16-40](#)

Source files

  checking syntax with SYNTAX directive [16-92](#)

  correcting with ERRORFILE directive [16-34](#)

Source files (continued)  
 description [11-1](#)

Source line truncated (warning 20) [A-44](#)

Special expressions  
 assignment [4-19](#)  
 CASE [4-19](#)  
 group comparison [4-19](#)  
 IF [4-19](#)

Specified bit extract/deposit (warning 42) [A-48](#)

SQL directive [16-86](#)

SQLMAP and NOSQLMAP (warning 67) [A-53](#)

SQLMEM directive [16-86](#)

SQLMEM STACK cannot (error 150) [A-35](#)

SQL-TAL code, listing with DEFEXPAND directive [16-27](#)

SSV, TACL ASSIGN commands [16-58](#)

STACK directive  
 description [16-87](#)  
 with DATAPAGES directive [16-26](#)

Stack overflow trap with CHECK directive [16-17](#)

STACK statement  
 description [12-36](#)

Standard functions  
 categories [14-1](#)  
 data types of arguments [14-5](#)  
 scope of [14-5](#)  
 See also individual functions  
 by variable data type [3-6](#)

syntax summary  
 bracket-and-brace diagrams [C-23](#)  
 railroad diagrams [B-39](#)

Standard (16-bit) pointers [9-1](#)

Statements  
 categories [12-1](#)  
 compound [12-2](#)  
 syntax summary  
 bracket-and-brace diagrams [C-19](#)  
 railroad diagrams [B-31](#)

Storage format  
 FIXED numbers [3-14](#)

Storage units, by data type [3-5](#)

STORE statement  
 description [12-37](#)  
 with RP directive [16-72](#)

STRING attribute [9-6](#)

STRING data type  
 description [3-2](#)  
 numeric format [3-9](#)

String overflow (error 7) [A-5](#)

String parameter pair expected (error 142) [A-34](#)

String parameter pair not expected (error 143) [A-34](#)

String parameters  
 CALL statement [12-9](#)  
 passing conditionally with \$OPTIONAL [14-33](#)

Strings, character [3-8](#)

STRUCT parameter type [13-10](#)

Structure item rather than (warning 54) [A-50](#)

Structure items [8-2](#)  
 filler bits or bytes [8-10](#)  
 referral substructures [8-11](#)  
 simple pointers [8-13](#)  
 simple variables [8-7](#)  
 structure pointers [8-15](#)

Structure pointers  
 addresses in [9-7](#)  
 declaring [9-6](#)

Structures  
 arrays of arrays [8-1](#)  
 as arrays of structures [8-1](#)  
 as multidimensional arrays [8-1](#)  
 maximum nesting levels [8-9](#)  
 template structures [8-1](#)

Sublocal declarations [13-17](#)

Subprocedure declared FORWARD (warning 44) [A-48](#)

Subprocedures  
 body of [13-17](#)  
 declaring [13-17](#)  
 functions [13-1](#)  
 VARIABLE attribute [13-7](#)

Subprocedures cannot be parameters (error 35) [A-13](#)

Substructures  
 declaring [8-9](#)  
 definition substructures [8-10](#)  
 number of occurrence, obtaining [14-29](#)  
 referral substructures [8-11](#)

Subtraction operator  
 signed [4-3](#)  
 unsigned [4-3](#)

SUBTYPE directive [16-87](#)

SUPPRESS directive  
 description [16-88](#)  
 with ABORT directive [16-12](#)

Switch register content, obtaining with \$SWITCHES [15-13](#)

Symbol table overflow (error 57) [A-18](#)

Symbol table, setting size with SYMBOLPAGES directive [16-90](#)

SYMBOLPAGES directive [16-89](#)

Symbols  
 delimiters [2-7](#)  
 indirection [2-6](#)  
 See also Identifiers  
 base-address [2-7](#)

SYMBOLS directive  
 description [16-90](#)  
 with INSPECT directive [16-54](#)

SYMSERV died (warning 77) [A-54](#)

SYMSERV fatal error [A-57](#)

SYNTAX directive  
 description [16-92](#)

Syntax summary  
 bracket-and-brace diagrams [C-1](#)  
 railroad diagrams [B-1](#)

System clock setting, obtaining with \$READCLOCK [14-38](#)

System global data, accessing [15-2](#)

System global pointers [15-2](#)

System services [1-3](#)

S-register  
 decrementing with DECS directive [16-26](#)

## T

Table overflow (error 42) [A-15](#)

TACL ASSIGN SSV commands [16-4](#)

TACL DEFINE commands [16-5](#)

TAL  
 applications and uses of [1-1](#)  
 features of [1-1](#)

TAL cannot set RP (warning 78) [A-54](#)

TAL run-time library [1-3](#)

TARGET directive  
 description [16-93](#)

Target file [11-1](#)

TARGETSPECIFIED, IF directive option [16-47](#)

Template structure not addressable (warning 94) [A-56](#)

Template structures [8-1, 8-5](#)

Terminating compilation, ABORT directive [16-12](#)

THEN keyword  
 IF expression [4-22](#)  
 IF statement [12-26](#)

TNS\_ARCH  
 IF directive option [16-47](#)  
 TARGET directive option [15-13](#)

TNS\_R\_ARCH  
 IF directive option [16-47](#)  
 TARGET directive option [15-13](#)

TO keyword  
 FOR statement [12-22](#)

## Toggles

- specifying with DEFINETOG [16-29](#)
- turning off with RESETTOG [16-68](#)
- turning on with SETTOG [16-82](#)
- using with IF or IFNOT directive [16-47](#)

Too many ASSIGN commands (warning 75) [A-54](#)

Too many named toggles (warning 83) [A-55](#)

Too many parameters (error 140) [A-33](#)

Type incompatibility (error 32) [A-12](#)

Type mismatch SQL (error 152) [A-36](#)

## Type-transfer functions

- rounding by [14-5](#)
- summary of [14-5](#)
- \$DBL** [14-5](#)
- \$DBLR** [14-5](#)
- \$DFIX** [14-4](#)
- \$EFLT** [14-4](#)
- \$EFLTR** [14-4](#)
- \$FIX** [14-5](#)
- \$FIXD** [14-5](#)
- \$FIXI** [14-5](#)
- \$FIXL** [14-5](#)
- \$FIXR** [14-5](#)
- \$FLT** [14-5](#)
- \$FLTR** [14-5](#)
- \$HIGH** [14-4](#)
- \$IFIX** [14-4](#)
- \$INT** [14-4](#)
- \$INTR** [14-4](#)
- \$LFIX** [14-4](#)
- \$UDBL** [14-4](#)

## U

Unable to process text (error 66) [A-20](#)

Undeclared identifier (error 49) [A-16](#)

Undefined ASSERTION procedure (error 41) [A-15](#)

Undefined option (warning 12) [A-42](#)

Undefined toggle (error 179) [A-39](#)

Unlabeled CASE statement [12-13](#)

Unsigned arithmetic operators [4-9](#)

UNSIGNED data type [3-2](#)

Unsigned less than [4-4](#)

Unsigned less than or equal to [4-4](#)

UNSIGNED parameter type [13-12](#)

UNSPECIFIED (TAL procedure language attribute) [13-8](#)

## UNTIL

SCAN statement [12-35](#)

Use a DUMPCONS directive (error 168) [A-38](#)

USE register overwritten (warning 24) [A-45](#)

Use relational expression (error 135) [A-32](#)

## USE statement

description [12-38](#)

## USEGLOBALS directive

description [16-93](#)

USEGLOBALS file created (error 112) [A-28](#)

## User code segment

word content, \$USERCODE [14-42](#)

User data area size, setting with DATAPAGES directive [16-25](#)

## User data stack

increasing with EXTENDSTACK directive [16-37](#)

setting size with STACK directive [16-87](#)

User heap, setting with HEAP directive [16-42](#)

## V

Value assigned to USE (error 134) [A-32](#)

Value out of range (error 149) [A-35](#)

Value out of range (warning 13) [A-43](#)

Value passed by reference (warning 16) [A-43](#)

VARIABLE attribute [13-7](#)

Variable attribute ignored (warning 45) [A-48](#)

Variable may not be target (error 130) [A-31](#)

Variable needs subscript (error 129) [A-31](#)

Variable size error (error 23) [A-10](#)

## Variables

kinds of [2-6](#)

VARIABLE-to-EXTENSIBLE procedure conversions [13-7](#)

## W

WARN directive [16-95](#)

### Warning Messages

general description [A-1](#)

### Warning messages

printing with WARN directive [16-95](#)

specific descriptions [A-40](#)

WHENEVERLIST and NOWHENEVERLIST (warning 66) [A-52](#)

WHILE statement [12-40](#)

### Width

variables [3-4](#)

Width of UNSIGNED array (error 118) [A-29](#)

Word [3-5](#)

Word addressable items (error 74) [A-21](#)

### WORDS keyword

group comparison expression [4-24](#)

## X

XBNDSTEST procedure [15-12](#)

XEP table [13-4](#)

XOR operator [4-3, 4-11](#)

XREF option, LMAP directive [16-61](#)

XSTACKTEST procedure [15-12](#)

### XX instructions

with INHIBITXX directive [16-51](#)

with NOINHIBITXX directive [16-51](#)

## Z

ZZSAnn save file, SAVEABEND directive [16-74](#)

# Special Characters

" (character string delimiter) [3-9](#)

# (DEFINE delimiter) [5-4](#)

#GLOBAL (implicit global data block) [11-5](#)

\$ABS Function [14-6](#)

\$ALPHA function [14-7](#)

\$AXADR function [15-11](#)

\$BITLENGTH function [14-8](#)

\$BITOFFSET function [14-9](#)

\$BOUNDS function [15-12](#)

\$CARRY function

carry indicator, testing [4-17, 14-10](#)

description [14-10](#)

\$COMP function [14-11](#)

\$DBL function [14-11](#)

\$DBLL function [14-12](#)

\$DBLR function [14-13](#)

\$DFIX function [14-14](#)

\$EFLT function

description [14-15](#)

preventing scaling of FIXED(n)

arguments [16-63](#)

\$EFLTR function

description [14-15](#)

preventing scaling of FIXED(n)

arguments [16-63](#)

\$FIX function [14-16](#)

\$FIXD function [14-16](#)

\$FIXI function [14-17](#)

\$FIXL function [14-18](#)

\$FIXR function [14-18](#)

\$FLT function

function [14-19](#)

preventing scaling of FIXED(n)

arguments [16-63](#)

\$FLTR

preventing scaling of FIXED(n)

arguments [16-63](#)

\$FLTR function

description [14-20](#)

\$HIGH function [14-20](#)

- \$IFIX function [14-21](#)
- \$INT function [14-21](#)
- \$INTR function [14-22](#)
- \$LADR function [14-23](#)
- \$LEN function [14-24](#)
- \$LFIX function [14-25](#)
- \$LMAX function [14-26](#)
- \$LMIN function [14-26](#)
- \$MAX function [14-27](#)
- \$MIN function [14-27](#)
- \$NUMERIC function [14-28](#)
- \$OCCURS function [14-29](#)
- \$OFFSET function
  - and structure pointers [9-7](#)
  - description [14-30](#)
- \$OPTIONAL function [14-32](#)
- \$OPTIONAL only allowed (error 175) [A-38](#)
- \$OVERFLOW function [14-35](#)
- \$PARAM function [14-36](#)
- \$POINT function [14-37](#)
- \$READCLOCK function [14-38](#)
- \$RP function [14-38](#)
- \$SCALE function [14-39](#)
- \$SPECIAL function [14-40](#)
- \$SWITCHES function [15-13](#)
- \$TYPE function [14-41](#)
- \$UDBL function [14-41](#)
- \$USERCODE function [14-42](#)
- \$XADR function [14-43](#)
- % (prefix for octal constants) [3-9](#)
- %B (prefix for binary constants) [3-9, 3-11](#)
- %D
  - (suffix for INT(32) hexadecimal constants) [3-12](#)
  - (suffix for INT(32) nonhexadecimal constants) [3-12](#)
- %F
  - (suffix for FIXED hexadecimal constants) [3-13](#)
- %F (suffix for FIXED nonhexadecimal constants) [3-13](#)
- %H
  - (prefix for hexadecimal constants) [3-9](#)
- %H (prefix for hexadecimal constants) [3-9](#)
- ( )
  - invoking DEFINEs [5-7](#)
- \* (asterisk)
  - constant-list repetition factor [3-16](#)
  - \* (asterisk) multiplication, binary signed [4-7](#)
- +
  - binary signed addition [4-7](#)
  - unary plus [4-5](#)
- 
- .
  - binary signed subtraction [4-7](#)
  - unary minus [4-5](#)
  - . (dereferencing operator)
    - operation [2-9](#)
  - . (period)
    - bit extractions [4-27](#)
  - .EXT (extended indirection symbol)
    - arrays [7-2](#)
    - formal parameters [13-11](#)
    - simple pointers [9-2](#)
    - structure pointers [9-6](#)
  - .SG structure not allocated (error 9) [A-5](#)
  - .SG (system global pointers) [15-3](#)
  - .(standard indirection symbol)
    - arrays [7-2](#)
    - simple pointers [9-2](#)
    - structure pointers [9-6](#)
  - /signed division [4-3](#)
  - :(colon), bit extractions [4-27](#)
  - := (left-to-right move) [12-7](#)
  - :=(assignment operator), assignment expression [4-19](#)
  - <, signed less than [4-3](#)
  - <=signed less than or equal to [4-3](#)
  - >, signed not equal to [4-4](#)
  - =
    - (right-to-left move) [12-27](#)
  - =signed equal to [4-15](#)
  - >signed greater than [4-15](#)

>=signed greater than or equal to [4-15](#)  
 >>(signed right-shift operator) [4-30](#)  
 @ operator  
   pointers [9-4](#)  
 @ prefix not allowed (error 114) [A-28](#)  
 ^ (circumflex)  
   in identifiers [2-4](#)  
 ^(circumflex)  
   location of the error in source file [A-2](#)  
 \_ (underscore), in identifiers [2-4](#)  
 '\*' (unsigned multiplication) [4-3](#)  
 '+' (unsigned addition) [4-3](#)  
 '-' (unsigned subtraction) [4-3](#)  
 '/' (unsigned division) [4-3](#)  
 '=' (unsigned equal to) [4-4](#)  
 '>=' (unsigned greater than or equal to) [4-4](#)  
 '>' (unsigned greater than) [4-4](#)  
 '' (unsigned modulo division) [4-3](#)  
 '' (unsigned not equal to) [4-4](#)  
 ' [4-29](#)  
   ='(left-to-right move) [12-27](#)  
 'G' (global addressing symbol), base-address equivalenced  
   variables [10-12](#)  
 'G' (global addressing symbol), base-address equivalenced variables [10-12](#)  
 'L' (local addressing symbol), base-address equivalenced  
   variables [10-12](#)  
 'P' (read-only array symbol) [7-5](#)  
 'SG' (system global base-address symbol) [15-4](#)  
 'SG'-equivalenced variables  
   definition structures [15-5](#)  
   description [15-5](#)  
   referral structures [15-6](#)  
   simple pointers [15-8](#)  
   simple variables [15-4](#)  
   structure pointers [15-9](#)  
 'S' (sublocal addressing symbol), base-address equivalenced variables [10-12](#)  
 '>>' (unsigned right-shift operator) [4-29](#)