NonStop™ Systems
NonStop 1+™ System

Transaction Application Language (TAL™) Reference Manual

Languages Library

82581

NOTICE

Effective with the B00/E08 software release, Tandem introduced a more formal nomenclature for its software and systems.

The term "NonStop 1+™ system" refers to the combination of NonStop 1+ processors with all software that runs on them.

The term "NonStop™ systems" refers to the combination of NonStop II™ processors, NonStop TXP™ processors, or a mixture of the two, with all software that runs on them.

Some software manuals pertain to the NonStop 1+ system only, others pertain to the NonStop systems only, and still others pertain both to the NonStop 1+ system and to the NonStop systems.

The cover and title page of each manual clearly indicate the system (or systems) to which the contents of the manual pertain.

NonStop™ Systems
NonStop 1+™ System

Transaction Application Language (TAL™) Reference Manual

Abstract

This manual provides reference information for TAL and the TAL compiler for system and application programmers.

Product Version
TAL B00
TAL E08

Operating System Version
GUARDIAN B00 (NonStop Systems)
GUARDIAN E08 (NonStop 1+ System)

Part No. 82581 A00

March 1985

Tandem Computers Incorporated
19333 Vallco Parkway
Cupertino, CA 95014-2599

DOCUMENT HISTORY

Edition	Part Number	Operating System Version	Date
1st Edition	82081 A00	GUARDIAN A00/E01	April 1981
TAL Addendum	82182	GUARDIAN A04/E05	October 1982
2nd Edition	82581 A00	GUARDIAN B00/E08	March 1985

New editions incorporate all updates issued since the previous edition. Update packages, which are issued between editions, contain additional and replacement pages that you should merge into the most recent edition of the manual.

NonStop™ Systems
NonStop 1+™ System

Transaction Application Language (TAL™) Reference Manual

Abstract
This manual provides reference information for TAL and the TAL compiler for system and application programmers.

Product Version
TAL B00
TAL E08

Operating System Version
GUARDIAN B00 (NonStop Systems)
GUARDIAN E08 (NonStop 1+ System)

Part No. 82581 A00

March 1985

Tandem Computers Incorporated
19333 Vallco Parkway
Cupertino, CA 95014-2599

DOCUMENT HISTORY

Edition	Part Number	Operating System Version	Date
1st Edition	82081 A00	GUARDIAN A00/E01	April 1981
TAL Addendum	82182	GUARDIAN A04/E05	October 1982
2nd Edition	82581 A00	GUARDIAN B00/E08	March 1985

New editions incorporate all updates issued since the previous edition. Update packages, which are issued between editions, contain additional and replacement pages that you should merge into the most recent edition of the manual.

NonStop™ Systems
NonStop 1+™ System

Transaction Application Language (TAL™) Reference Manual

Abstract
This manual provides reference information for TAL and the TAL compiler for system and application programmers.

Product Version
TAL B00
TAL E08

Operating System Version
GUARDIAN B00 (NonStop Systems)
GUARDIAN E08 (NonStop 1+ System)

Part No. 82581 A00

March 1985

Tandem Computers Incorporated
19333 Vallco Parkway
Cupertino, CA 95014-2599

DOCUMENT HISTORY

Edition	Part Number	Operating System Version	Date
1st Edition	82081 A00	GUARDIAN A00/E01	April 1981
TAL Addendum	82182	GUARDIAN A04/E05	October 1982
2nd Edition	82581 A00	GUARDIAN B00/E08	March 1985

New editions incorporate all updates issued since the previous edition. Update packages, which are issued between editions, contain additional and replacement pages that you should merge into the most recent edition of the manual.

NEW AND CHANGED INFORMATION

This manual is the second edition of the TAL Reference Manual. It incorporates the TAL Reference Manual Addendum, Part Number 82182.

The manual is reorganized and rewritten and includes the following new information:

- EXTENSIBLE procedure description
- Compiler directives--ABORT, DEFEXPAND, LINES, GMAP, PRINTSYM, and WARN
- Additional error messages

CONTENTS

FIGURES

TABLES

PREFACE

This manual provides reference information for the Transaction Application Language (TAL) used on Tandem systems. This manual is intended for:

- Systems programmers writing operating system components, compilers, interpreters, special subsystems, drivers for non-standard input/output devices, and special routines that support data communications activities.

- Applications programmers writing code for server processes used with the PATHWAY transaction processing system and other data management software supplied by Tandem, conversion routines that facilitate transfer of data between Tandem software products and various applications, specialized procedures callable from COBOL or FORTRAN programs, and other applications software where optimal performance has high priority.

The following manuals provide additional information:

- Introduction to Tandem Computer Systems for an overview of the system hardware and software.

- System Description Manual for the NonStop or NonStop 1+ system for details about the hardware aspects of the system and the process oriented organization of the GUARDIAN operating system.

- System Procedure Calls Reference Manual for the syntax for calling operating system procedures.

- GUARDIAN Operating System Programmer's Guide for information about using the operating system procedures.

- BINDER Manual for information about binding modules.

SYNTAX CONVENTIONS IN THIS MANUAL

The following list summarizes the conventions for syntax notation in this manual.

Notation	Meaning
UPPERCASE LETTERS	Uppercase letters represent keywords and reserved words; you must enter these items exactly as shown.
<lowercase letters>	Lowercase letters within angle brackets represent variables that you supply.
Brackets []	Brackets enclose optional syntax items. A vertically aligned group of items enclosed in brackets represents a list of selections from which you can choose one or none.
Braces {}	Braces enclose required syntax items. A vertically aligned group of items enclosed in braces represents a list of selections from which you must choose only one.
Vertical Bar |	Two horizontally aligned items separated by a vertical bar represent a pair of selections surrounded by either brackets or braces.
Ellipsis ...	An ellipsis immediately following a pair of brackets or braces indicates that you can repeat the enclosed syntax items any number of times.
Percent Sign %	Precedes a number in octal notation.
Spaces	If two items are separated by a space, that space is required between the items. If one of the items is a punctuation symbol, such as a parenthesis or a comma, spaces are optional.
Punctuation	Parentheses, commas, semicolons, and other symbols or punctuation not described above must be entered precisely as shown. If any of the punctuation above appears enclosed in quotation marks, that character is not a syntax descriptor but a required character and you must enter it as shown.

SECTION 1

INTRODUCTION

The Transaction Application Language (TAL) is a high-level, block-structured language used to write systems software and routines that support transaction-oriented applications. The TAL compiler compiles source programs written in TAL into executable object programs. The TAL compiler and the object programs it generates execute under control of the GUARDIAN operating system.

APPLICATIONS AND USES

TAL is most often used for writing systems software or transaction-oriented applications where optimal performance has high priority. You can, for example, use TAL to write:

- Operating system components, command interpreters, source-language compilers and interpreters, and special subsystems
- Input/output processes, drivers, and protocols that support nonstandard devices and integrate them into the operating system
- Data communications routines for interfacing with the ENVOY data communications manager or message-switching functions
- Special procedures callable by COBOL or FORTRAN programs
- Server processes used with the PATHWAY transaction processing system and other data management software produced by Tandem

TAL works efficiently with the hardware to provide optimal object program performance. Many software products supplied by Tandem are written in TAL.

PROCESSES

Object programs execute as individual processes. While a program is a static group of machine instructions and initialized data residing in a file, a process is a dynamically running program. Thus, the same program can execute concurrently many times, and each execution comprises a different process.

Each process has its own user code space and user data space. The code space consists of one or more code segments; the data space consists of a data segment and one or more extended segments. For each executing process, the system maintains two physical areas in memory, the current user code segment and the current user data segment, as shown in Figure 1-1.

The code segment is not modifiable by a user process but is sharable among processes. The data segment is modifiable but private to the running process. Thus, many processes can execute the same code, but the data on which the code operates remains exclusive to each process.

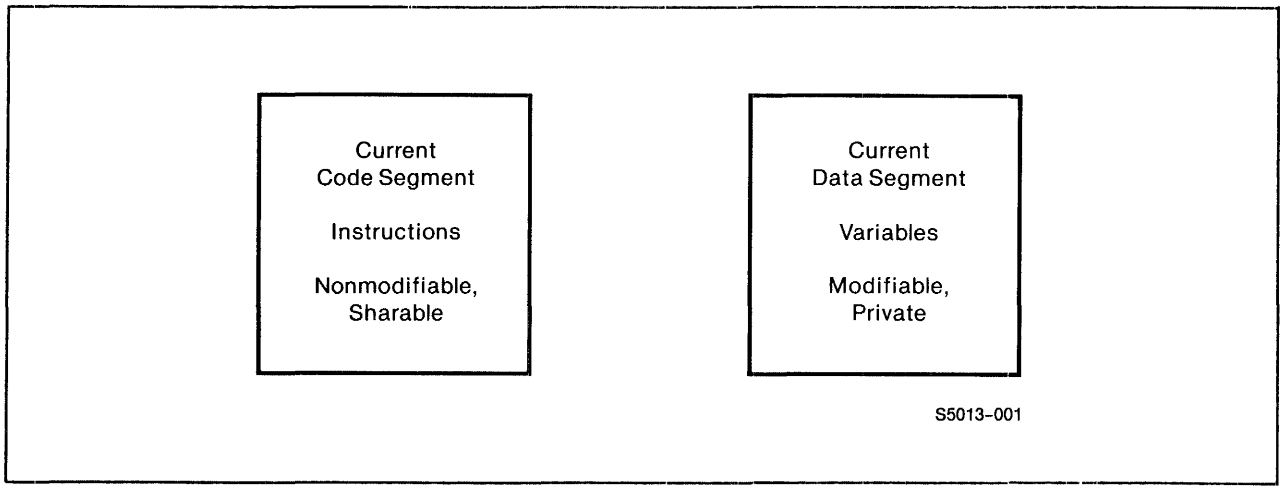


Figure 1-1. Code and Data Segments of a Process

MAJOR FEATURES

The major features of TAL are:

- Procedures--The code space for each program contains one or more procedures. A procedure is a block of machine instructions that performs a specific task. It exists once in the program but is callable from anywhere in the program.

  When the current procedure invokes another procedure, the system automatically saves the current process environment. When the called procedure terminates, the system restores the environment of the previous procedure. Thus, each procedure executes in its own environment and is not affected by the actions of other procedures.

  Each activation of a procedure has its own local data area. That is, the system allocates and initializes a new local data area each time a procedure is entered. When each activation completes execution, it relinquishes its local data area. Thus, the memory space that a program requires is continuously held to a minimum.

- Recursion--Because each activation of a procedure has its own local data area, a procedure can call itself. This feature, called recursion, can enhance programming efficiency for certain applications.

- Parameter Passing--You can declare optional or required parameters for procedures.

- Subprocedures--A procedure can contain subprocedures, callable only from within the same procedure. Since each activation of a subprocedure has its own private data, subroutines can be recursive.

- Six Data Types--You can declare and reference six types of data:

--STRING	8-bit integer byte
--INT	16-bit integer word
--INT(32)	32-bit integer doubleword
--FIXED	64-bit fixed-point quadword
--REAL	32-bit floating-point doubleword
--REAL(64)	64-bit floating-point quadword

- Data Structures--You can describe and reference sets of related data variables such as records and arrays.

- Data Operations--You can move a contiguous group of words or bytes and compare one group with one 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 deposit, bit extraction, and bit shift operations.
- Pointers--Pointer variables can contain byte addresses or word addresses. You can use pointers to access locations throughout memory. You can initialize them when you declare them or at any time during program execution.
- Modular Programming--TAL supports modular programming with separate compilation and relocatable global data blocks. You can compile each module that contains one or more procedures as a separate compilation unit. The compile-time binder cooperates with the TAL compiler to build a bound object file from each module.

INTERFACE WITH OPERATING SYSTEM

Object programs run under the control of the GUARDIAN operating system. It provides an environment that allows your program to ignore many things such as the presence of other programs and whether your program fits into memory. For example, the operating system loads programs into memory, brings absent pages from disc into memory as needed, and allocates CPU time.

The operating system performs all file system functions for programs. It treats all devices as files including disc files, disc packs, terminals, printers, and processes running on the system. Programs can reference a file by its symbolic name without regard for its physical address or configuration status. File system procedures provide a single, uniform file access method that masks the peculiarities of devices from applications.

Process control system procedures let processes activate and terminate other processes in any processor on the system. Processes can monitor the operation of any process or processor. If a process stops or a processor fails, your program can determine this fact.

Operating system procedures are described in the System Procedure Calls Reference Manual and GUARDIAN Operating System Programmer's Guide.

MACHINE DEPENDENCIES

The TAL compiler is a disc-resident program on each Tandem system and runs under the control of the GUARDIAN operating system.

For previous versions of the operating system, the same version of the TAL compiler executes on all Tandem systems. Operating system version B00 requires TAL compiler version B00. Operating system version E08 requires TAL compiler version E08.

Certain features, such as extended pointers, extended data segments, user library segments, and multiple user code segments, are not available on the NonStop 1+ system. A summary of machine dependencies appears in Appendix A.

SYSTEM REQUIREMENTS

Some object programs require optional microcode such as:

- Decimal arithmetic option for operations with quadword operands and arithmetic operations
- Floating-point option for doubleword and quadword (extended) floating-point arithmetic and related operations

Some object programs require other software products such as the PATHWAY transaction processing system.

PROGRAM DEVLOPMENT TOOLS

Other Tandem utilities that provide additional program development features are:

- EDIT--a full text editor with screen and conversational editing features, described in the EDIT Manual, that can help you create TAL source programs
- CROSSREF--a process that creates a cross-reference listing of variables, functions, and keywords in a program, either as an interactive process described in the CROSSREF Manual or a compiler-driven process as described in this manual
- INSPECT--an interactive debugger that lets you stop and start program execution and display and modify program values symbolically as described in the INSPECT Interactive Symbolic Debugger User's Guide

- DEBUG--an interactive debugger that lets you stop and start program execution and display and modify program values by location as described in the DEBUG Manual
- BINDER--an interactive binder that lets you examine, modify and combine object files and produce optional load maps and cross-reference listings as described in the BINDER Manual

COMPILATION CYCLE

The object file is the output of the compiler or BINDER. The output of each compilation is an object program that is either an entire executable program or a part of a modular program. You can compile each part (module) of a program separately, then bind the resulting object files into a new object file called the target file.

Figures 1-2 and 1-3 show the compilation cycle of a nonmodular program and of a modular program, respectively.

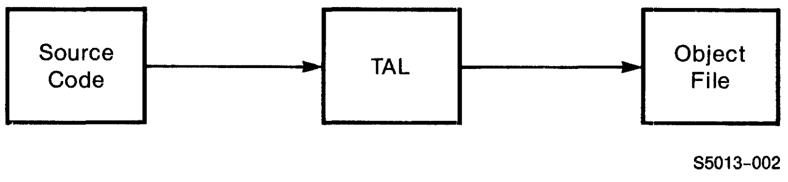


Figure 1-2. Compilation Cycle of Nonmodular Program

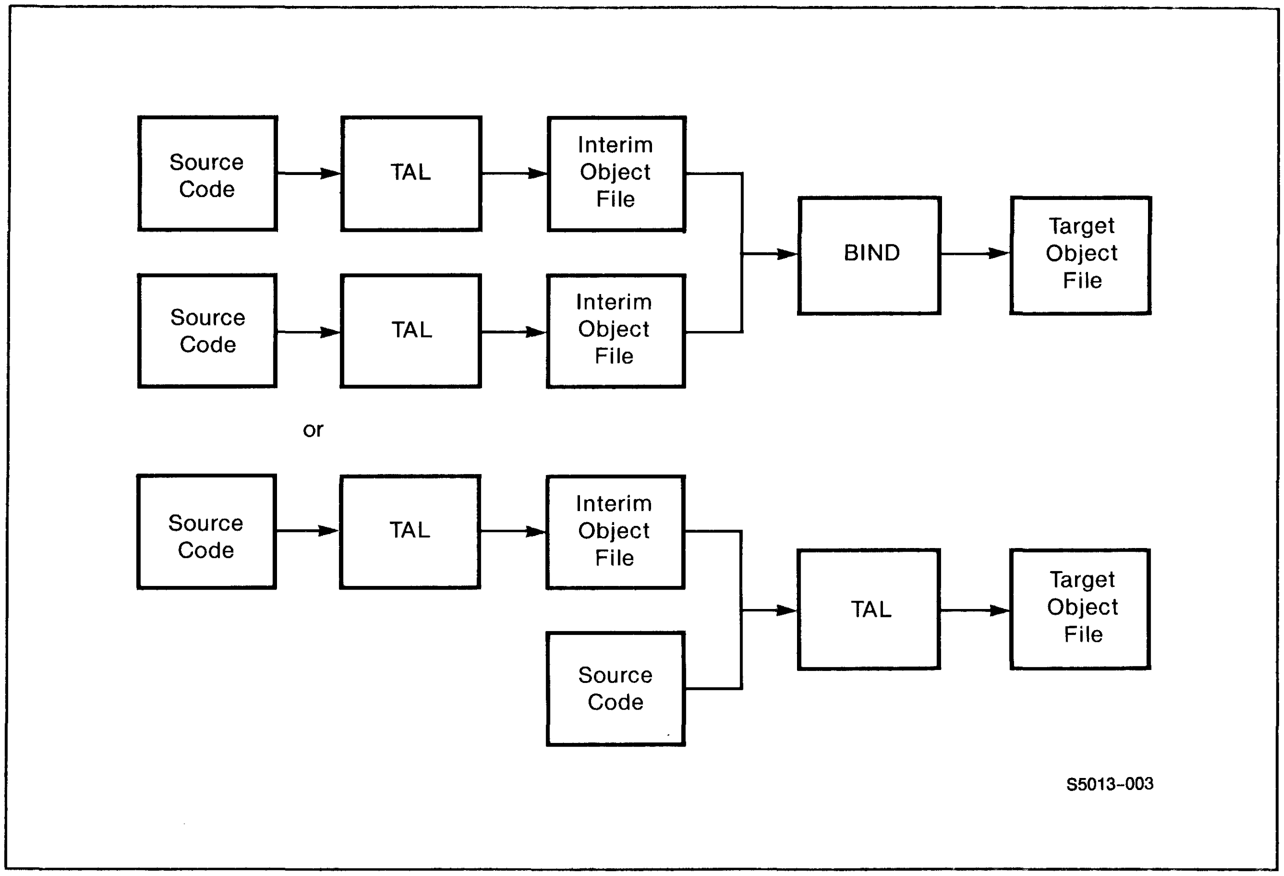


Figure 1-3. Compilation Cycle of Modular Program

EXAMPLE PROGRAM

Figure 1-4 shows an example of a TAL source program. The program opens the home terminal, then loops forever. Each iteration of the loop consists of the following actions:

1. The program displays the prompt "ENTER STRING" and accepts a character string of up to 72 characters.

2. The program scans the input string for an asterisk. If one occurs, it displays a circumflex at the position of the asterisk.

```
INT hometerm,      !File number of home terminal
    left^side,     !sbuffer address of 1st character after prompt
    num^xferred,   !Number of bytes transferred by file system
    count,         !General-purpose variable
    asterisk,      !Location of asterisk
    buffer[0:40];  !Input/output (I/O) buffer

STRING
    .sbuffer := @buffer '<<' 1,     !STRING pointer to I/O buffer
    blanks[0:71] := 72 * [" "];     !Blanks for initialization

?SOURCE $SYSTEM.SYSTEM.EXTDECS(MYTERM,OPEN,WRITEREAD,WRITE,STOP)
    ! Operating system procedure declarations

PROC main^proc MAIN;
BEGIN
  CALL MYTERM(buffer);                    !Get name of home terminal
  CALL OPEN(buffer, hometerm);            !Open home terminal

  WHILE 1 DO                              !Infinite loop
    BEGIN
      sbuffer ':=' "ENTER STRING" -> left^side;
      CALL WRITEREAD(hometerm, buffer, 12, 68, num^xferred);
      sbuffer[num^xferred] := 0;                  !Delimit the input
      SCAN sbuffer UNTIL "*" -> asterisk;         !Scan for asterisk
      IF NOT $CARRY THEN                          !Asterisk found
        BEGIN
          sbuffer ':=' blanks FOR
             (count := asterisk '-' @sbuffer +
                     (left^side '-' @sbuffer));
          sbuffer[count] := "^";
          CALL WRITE(hometerm, buffer, count + 1);
        END;                                      !End of IF
    END;                                          !End of WHILE
  END;                                            !End of procedure
```

Figure 1-4. Example Source Program

SECTION 2

PROGRAM STRUCTURE

This section summarizes the structure of a TAL source program. The source code for a program consists of one or more compilation units. Each compilation unit contains all declarations, statements, and compiler directives needed for a single compilation but does not necessarily contain everything needed for an executable program.

The overview describes:

- The components and structure of a nonmodular source program
- Additional components and the structure of a module of a modular source program

PROGRAM COMPONENTS

Program components are parts of the source program that define objects and specify operations on these objects. The primary components of a nonmodular program are:

- Global Declarations
- Procedure Declarations
- Local Declarations
- Subprocedure Declarations
- Sublocal Declarations
- Statements

Each primary component in turn can contain other components such as variables, pointers, numeric constants, character strings, reserved words, operators, delimiters, and other symbols. These are discussed in later sections.

Global Declarations

Global declarations define identifiers you can reference throughout the program. Global identifiers are accessible for the duration of the compilation.

Declarations that can have global scope are:

- Data Declarations--These associate identifiers with memcry locations and allocate memory for storing values and the results of computations.
- LITERAL Declarations--These associate constant values with identifiers.
- DEFINE Declarations--These associate text with identifiers.
- FORWARD Procedure Declarations--These specify that the declaration for the procedure body occurs later in the source file.
- EXTERNAL Procedure Declarations--These specify that the declaration for the procedure body occurs in another compilation.

Procedure Declarations

Procedure declarations specify discrete portions of source code within a program. They define the executable parts of the program.

A procedure can contain local declarations and subprocedure declarations.

Local Declarations

Local identifiers are accessible only during execution of the encompassing procedure. They can be accessed only by statements and subprocedures within the procedure in which they are declared, unless the procedure passes them as parameters to another procedure.

Declarations that can have local scope are:

- Data Declarations
- LITERAL Declarations
- DEFINE Declarations
- Label Declarations--These reserve identifiers for later use as names of locations in the procedure.
- Entry-Point Declarations--These specify additional entry points into a procedure or subprocedure body.
- FORWARD Subprocedure Declarations--These specify that the declaration for the subprocedure body occurs later in the same procedure.

The system allocates and initializes a separate local data area for each activation of a procedure. When each activation completes execution, the system deallocates its local data area.

Subprocedure Declarations

Subprocedure declarations specify discrete blocks of source code within a procedure. A procedure can contain any number of subprocedures, all nested at the same level.

A subprocedure can contain sublocal declarations, but it cannot contain other subprocedures.

Sublocal Declarations

Sublocal declarations define identifiers that are accessible only during execution of the encompassing subprocedure. Sublocal identifiers can be accessed only by statements within the subprocedure, unless the subprocedure passes them as parameters to another subprocedure or procedure.

Declarations that can have sublocal scope are:

- Data Declarations
- LITERAL Declarations
- DEFINE Declarations
- Label Declarations
- Entry-Point Declarations

The system allocates and initializes a separate sublocal data area for each activation of a subprocedure. When each activation completes execution, the system deallocates its sublocal data area.

Statements

Statements request specific actions. Local statements appear within a procedure. Sublocal statements appear within a subprocedure.

Local statements in a procedure can invoke any procedure previously declared in the program and any subprocedure previously declared within the same procedure. They can reference global identifiers and local identifiers in this procedure but not those in other procedures or in subprocedures.

Sublocal statements in a subprocedure can invoke any procedure previously declared in the program, or any subprocedures previously declared within the same procedure. They can reference global identifiers, local identifiers in the encompassing procedure, and sublocal identifiers in this subprocedure but not those declared in other subprocedures.

PROGRAM STRUCTURE

The TAL compiler expects source declarations and statements in the following order:

1. All global declarations must appear before the first procedure declaration.

2. A procedure declaration comes next.

3. All local declarations for this procedure come next.

4. A subprocedure declaration, if any, appears next, followed in order by:

   --All sublocal data declarations for this subprocedure

   --All sublocal statements for this subprocedure

5. For each subsequent subprocedure, the primary components listed in item 4, if present, appear in the order given

6. All local statements for the encompassing procedure follow the last subprocedure contained in this procedure. If no subprocedures appear in this procedure, all local statements follow the local data declarations for this procedure.

7. For each subsequent procedure, the primary components listed in items 2 through 6, if present, must appear in the order given.

You must declare procedures and subprocedures before you reference them in statements unless you use FORWARD declarations. For further information, see Section 16, "Procedures and Subprocedures."

Figure 2-1 shows the structure of a nonmodular program that has three procedures, one of which contains a subprocedure.

In the figure, the scope of the declarations in each box is inward only. That is, global data is accessible to all items in the program. Local data is accessible only to items in the procedure in which it appears. Sublocal data is accessible only to items within the subprocedure in which it appears.

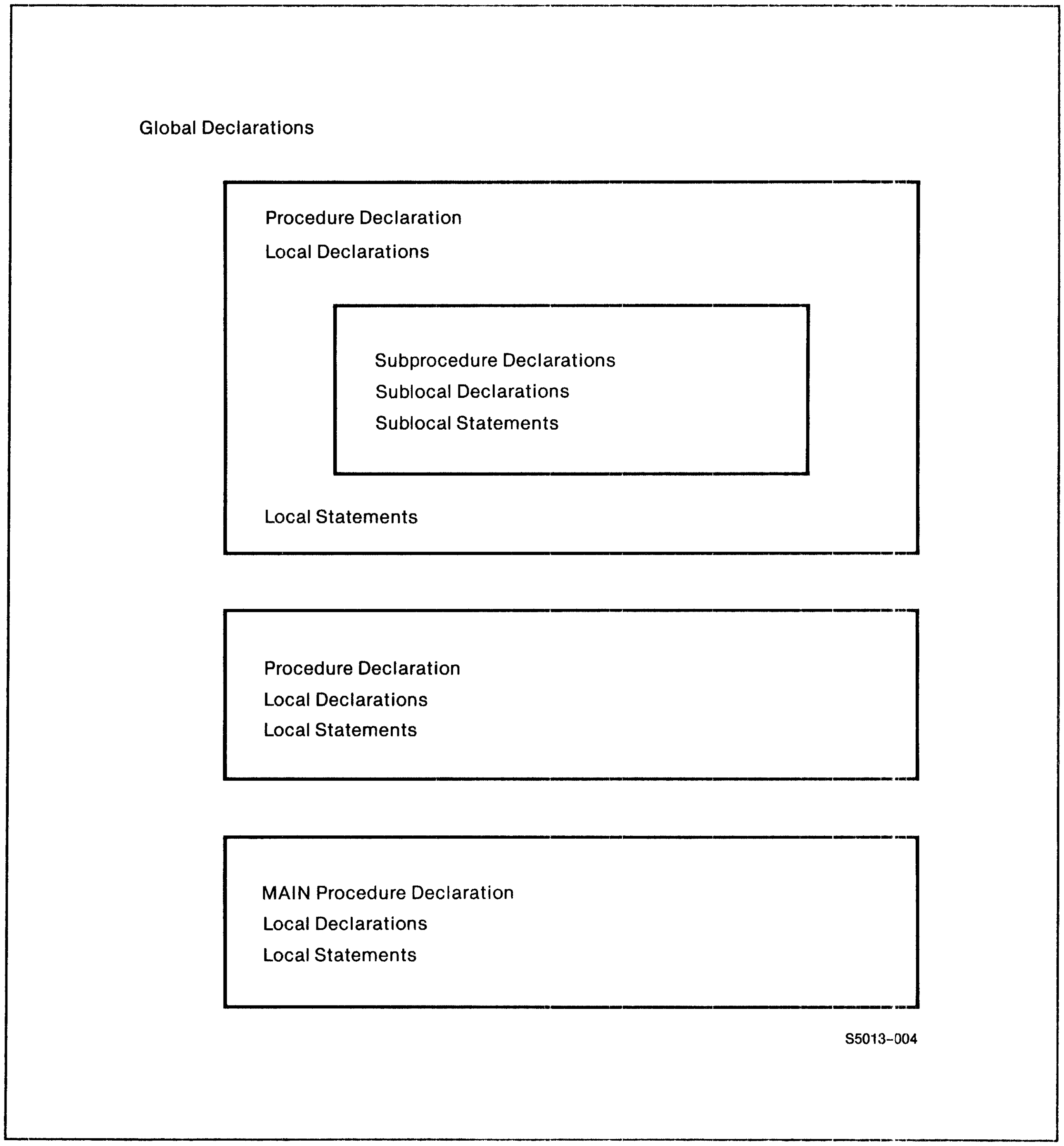


Figure 2-1. Structure of a Nonmodular Source Program

MODULAR PROGRAMMING

Modular programming provides several advantages. For example, it allows you:

- To divide a large program into smaller, more manageable modules
- To work independently on a module, while other programmers work on other modules
- To bind new code to existing debugged object code including general-purpose library routines
- To code different procedures for the same program in different languages

Compiler and binder support for modular programming is described in Section 22, "Separate Compilation." The differences between modular programs and nonmodular programs are summarized below.

Modules can have the following additional components:

- NAME Declaration--This declaration assigns a name to the module.
- BLOCK Declarations--These group global data declarations into relocatable global data blocks. Each module can have one private data block and any number of user-named data blocks. The private block is global only to that module. The named blocks are global to all modules in the program.

Any global data declarations not contained in a BLOCK declaration must appear before the first BLOCK declaration. TAL treats the unblocked declarations as an implicit data block that is global to all modules in the program.

Modular Structure

The structure of a source module is shown in Figure 2-2. The NAME, unblocked, and BLOCK declarations, if present, must appear in the order shown in the figure.

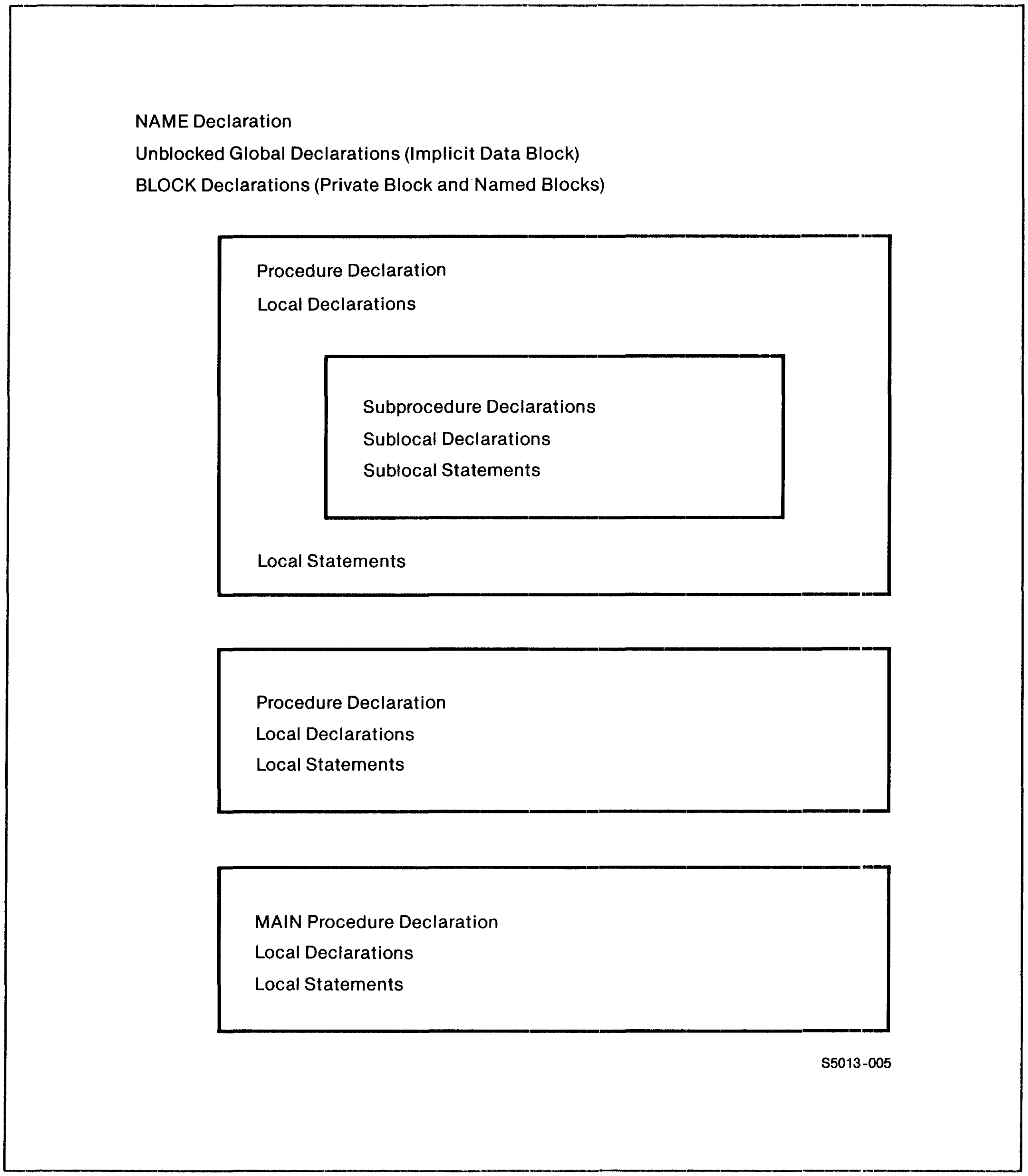


Figure 2-2. Structure of a Source Module

SECTION 3

LEXICAL ELEMENTS

This section describes the format you can use for source code and lists the lexical elements that make up the TAL language.

Elements include the character set supported, components, reserved words, identifiers, constants, variables, indirection symbols, address base symbols, delimiters, and operators.

FORMAT OF SOURCE CODE

The maximum line length is 132 characters.

TAL allows almost a free format for source code. This flexibility lets you design a format that is readable and maintainable. The following example shows a legal format:

```
INT    a,
       b,
       c;
STRING char1,
       char2,
       char3;

PROC format^example MAIN;
BEGIN
  a := 1;
  b := 2;
  c := a + b;
  char1 := "A";
  char2 := "B";
  char3 := "C";
END;
```

BEGIN-END Construct

The BEGIN-END construct is an integral part of the TAL language. For example:

- It encloses the body of a procedure, as in the following example:

```
PROC a;
BEGIN
  .
  .
END;
```

- It forms a compound statement, as in the following example:

```
IF a < b THEN
  BEGIN
    .
    .
  END
ELSE
  BEGIN
    .
    .
  END;
```

Comments

Comments begin with an exclamation point (!) and terminate with either another exclamation point or the end of the line. Valid examples are:

```
CALL calc;    !Comment
CALL calc;    ! Comment !
! Comment
!Comment! CALL !Comment! calc; !Comment!
```

CHARACTER SET

TAL supports the complete ASCII character set including uppercase and lowercase alphabetics, numerics 0 through 9, and special characters. The ASCII character set appears in Appendix E.

COMPONENTS

TAL program components consist of declarations and statements.

- Declarations associate identifiers with data variables and other declarable objects in a program:

  --Variable objects such as simple variables, arrays, structures, pointers, and equivalenced variables

  --Other objects such as procedures, literals, defines, labels, and entry points

- Statements specify operations to be performed on declared objects. Statements are summarized in Table 3-1 and described in Section 15.

Table 3-1. TAL Statements

Statement	Meaning
ASSERT	Conditionally calls error-handling procedure.
Assignment	Stores value in variable.
CALL	Invokes procedure or subprocedure.
CASE	Executes statement based on index value.
CODE	Specifies machine codes for inclusion in object code.
DO-UNTIL	Executes posttest loop until true condition.
DROP	Frees index register or removes label from symbol table.
FOR-DO	Executes pretest loop for <n> times.
GOTO	Unconditionally branches to label within procedure or subprocedure.
Move	Moves group of elements from one location to another.
IF-THEN-ELSE	Executes THEN statement for true state or ELSE statement for false state.
RETURN	Returns from procedure or subprocedure to caller. For functions, also can specify returned value.
RSCAN	Searches scan area, right to left, for test character.
SCAN	Searches scan area, left to right, for test character.
STACK	Loads value on register stack.
STORE	Stores register stack element in variable.
USE	Reserves index register for user manipulation.
WHILE-DO	Executes pretest loop during TRUE condition.

RESERVED WORDS

Reserved words are keywords that have predefined meanings when you use them in declarations and statements. Table 3-2 lists the reserved words in alphabetic order. You cannot use reserved words for user-defined identifiers unless noted otherwise below.

Table 3-2. Reserved Words

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

* NAME is reserved only when used in the first declaration in a compilation unit. BLOCK and PRIVATE are reserved in a named compilation unit. In an unnamed compilation unit, you cannot declare data blocks using BLOCK declarations, but you can use BLOCK and PRIVATE as user-defined identifiers. For details, see Section 22, "Separate Compilations."

** EXTENSIBLE is a procedure attribute, as described in Section 16, "Procedures and Subprocedures." However, you can also use EXTENSIBLE as a user-defined identifier.

*** FILLER is a reserved word only within the scope of a structure declaration, as described in Section 11, "Structures."

IDENTIFIERS

Identifiers are symbolic names you use for objects in declarations and statements. The following rules apply when forming identifiers:

- They can be up to 31 characters in length.
- They must begin with an alphabetic character or a circumflex (^).
- They can consist only of alphabetics, numerics, and circumflexes.
- You can use lowercase characters, but TAL treats them as uppercase.

The following examples show valid identifiers:

```
a2
number^of^bytes
^
TANDEM
^23456789012^00
Name^with^exactly^31^characters
```

The following examples show invalid identifiers:

```
2abc                                 !Begins with number
ab%99                                !Illegal symbol
Variable                             !Reserved word
This^name^is^too^long^so^it^is^invalid   !Too long
```

Identifier Classes

Each identifier is a member of an identifier class. TAL determines the identifier class based on the declaration of the identifier and stores the information in the symbol table.

Table 3-3 summarizes the identifier classes and the sections in this manual in which each class is described.

Table 3-3. Identifier Classes

Class	Meaning	Section
Block	Global data block	22
Code	Read-only (P-relative) array	9
Constant	Unnamed numeric or character string constant	4
Variable	Simple variable, array, pointer, structure, substructure, or structure data item	8-11
DEFINE	Named text	6
Function	Procedure or subprocedure with a return value	16
Label	Statement label	7
LITERAL	Named constant	6
PROC	Procedure or subprocedure with no return value	16
Register	Index register (R5, R6, or R7) (See USE statement)	15
Template	Structure template	11

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 Section 4, "Data Representation."

A constant expression is an arithmetic expression that contains no variables. You can use a constant expression anywhere a single constant is allowed.

The following are examples of constants and constant expressions:

```
   255          !Numeric constant
   "xyz"        !Character string constant
   2 * 5        !Constant expression
```

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 Section 4. Examples are:

Binary:	%B101111
Octal:	%57
Decimal:	47
Hexadecimal:	%H2F

VARIABLES

A variable is a symbolic representation of an item or a group of elements. It stores data that can change during program execution. Table 3-4 summarizes variables.

Table 3-4. Variables

Variable	Meaning	Section
Simple Variable	A variable that contains one item of a specified data type	8
Array	A variable that contains multiple elements of the same data type, all accessible by one identifier	9
Structure	A variable that contains multiple elements of one or more data types, all accessible by one identifier	11
Substructure	A structure declared within another structure or substructure	11
Structure data item	An array or simple variable declared within a structure or substructure	11
Pointer	A variable that contains the address of another item of a specified data type; referencing a pointer accesses the item to which the pointer points	10

SYMBOLS AND OPERATORS

Symbols are indirection symbols, address base symbols, prefix symbols, and delimiters (punctuation symbols):

- Indirection symbols are the period (.), .EXT, .SG, and @, as summarized in Table 3-5.
- Address base symbols are 'SG', 'P', 'G', 'L', and 'S', as summarized in Table 3-6.
- Delimiters start or end a field of information as summarized in Table 3-7.
- Other symbols are "$" and "?", as follows:

  $ --specifies a standard function, such as $ABS and $DBL, as described in Section 17.

  ? --specifies a directive line that contains one or more compiler directives, as described in Section 20.

Operators specify assignment, move, bit shift, arithmetic, boolean, and relational operations, as summarized in Table 3-8.

Table 3-5. Indirection Symbols

Symbol	Meaning	Section
.	Declares indirect array (standard indirection)	9
	Declares indirect structure (standard indirection	11
	Declares 16-bit standard pointer	10
	Declares 16-bit standard structure pointer	11
	Uses direct INT variable as a temporary pointer	10
@	Removes indirection (accesses address contained in pointer or address of any other item)	10
.EXT	Declares 32-bit extended pointer	10
	Declares 32-bit extended structure pointer	11
.SG	Declares 16-bit system global pointer	18
	Declares 16-bit system global structure pointer	18

Table 3-6. Address Base Symbols

Symbol	Meaning	Section
'P'	P-register addressing (read-only array declaration)	9
'G'	Base-address equivalencing, global user data area	12
'L'	Base-address equivalencing, local user data area	12
'S'	Base-address equivalencing, sublocal user data area	12
'SG'	Base address, system global space (privileged procedures)	18

Table 3-7. Delimiters

Symbol	Meaning	Section
!	Begins and optionally ends a comment	3
,	Separates fields of information	
	Constant lists	4
	Declarations	6-12
	Parameters (DEFINEs, procedures,	6,16
	standard functions, CALL statements)	17,15
;	Terminates declarations	6-12
	Separates statements	15
.	Word.<bit> specification	14
	Structure name qualification	11
<:>	Bit field	4,14
:	Label, ASSERT statement, entry point	7,15,16
( )	Expression precedence	13
	CODE statement	15
	Parameters (DEFINEs, procedures,	6,16
	standard functions, CALL statements)	17,15
	Structure pointer referral mode	11
	FIXED (<fpoint>)	8
(*)	FIXED (*) formal parameter specification	16
	Template structure declaration	11
*	Repetition factor	4
[ ]	Constant list; index; array element	4,5,9
[:]	Array bounds	9
	Structure or substructure bounds	11
->	<next-addr> in SCAN, RSCAN, move statements	15
	<next-addr> in group comparison expression	13
" "	Begins and ends character strings	4
""	Embedded quotation mark in character strings	4
#	Terminates DEFINE declaration text	6
','	Embedded comma in DEFINE parameter	6

Table 3-8. Operators

Operation	Operator	Meaning	Section
Assignment	:=	Data declaration initialization	8-11
		Assignment and FOR statements	15
		Assignment form of arithmetic expression	13
Representation	=	LITERAL or DEFINE declaration	6
		Equivalenced variable declaration	12
		Redefinitions inside structures	11
Move	':='	Left-to-right move	15
	'=:'	Right-to-left move	
Bit Shift	<<	Signed left shift	14
	>>	Signed right shift	
	'<<'	Unsigned left shift	
	'>>'	Unsigned right shift	
Arithmetic	+	Signed addition	13
	-	Signed subtraction	
	*	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	
Boolean	AND	Logical conjunction	13
	OR	Logical disjunction	
	NOT	Logical negation	
Relational	<	Signed less than	13
	=	Signed equal to	
	>	Signed greater than	
	<=	Signed less than or equal to	
	>=	Signed greater than or equal to	
	<>	Signed 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	

SECTION 4

DATA REPRESENTATION

Data is the information on which a program operates.

Variables store data that can change during program execution. When you declare a variable, you specify a data type, which determines its storage, range of values and precision, and the way it can be used in a program.

This section describes the following:

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

DATA UNITS

Data units are the formats in which you can access data stored in memory. The system stores all data in 16-bit word units, but you can access this data as any of the five units listed in Table 4-1.

Table 4-1. Data Units

Data Unit	Number of Bits	Description
Bit field	1-16	One or more contiguous bits within a word
Byte	8	Two bytes comprise a word, with byte 0 (most significant) in the left half and byte 1 (least significant) in the right half
Word	16	Basic addressable unit of memory
Doubleword	32	Four contiguous bytes or two contiguous words
Quadword	64	Eight contiguous bytes or four contiguous words

Bit Fields

A bit field specifies one or more contiguous bits in a data unit by bit number. For a word unit, the bit numbers are 0 through 15 from left to right, as shown in Figure 4-1.

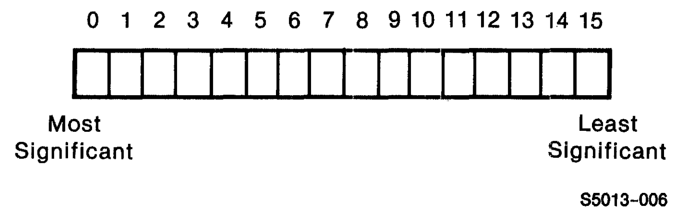


Figure 4-1. Bit Field

For a one-bit field, specify the bit number enclosed in angle brackets, as in <0>, <7>, or <14>.

For a multiple-bit field, specify the leftmost and rightmost bit numbers of the field separated by a colon and enclosed in angle brackets, as in <2:3>, <0:7>, or <4:15>.

DATA TYPES

The data type of a variable determines the values it can represent, the operations you can perform on it, byte or word addressing and alignment, data length, indexing offsets, and kind of machine instructions generated.

Data can be character strings or numbers. Table 4-2 shows the six data types and the numeric range each represents.

Table 4-2. Data Types

Data Type	Data Unit	Number Representation
STRING	Byte	ASCII character or 8-bit integer in the range 0 through 255 unsigned
INT	Word	16-bit integer in the range 0 through 65,535 unsigned or -32,768 through 32,767 signed
INT(32)	Doubleword	32-bit integer in the range -2,147,483,648 through +2,147,483,647
FIXED	Quadword	64-bit fixed-point number in the range -9,223,372,036,854,775,808 through +9,223,372,036,854,775,807
REAL	Doubleword	32-bit floating-point number
REAL(64)	Quadword	64-bit floating-point number
		REAL and REAL(64) data are in the range $+/-8.62 * 10^{-78}$ through $+/-1.16 * 10^{78}$

Address Modes

The data type of a variable determines byte or word addressing, alignment, and indexing, as discussed in Section 5, "Addressing Modes."

Operations and Functions

The data type of a variable determines the operations you can perform on it and the standard functions you can use with it, as shown in Table 4-3.

Table 4-3. Operations and Functions

	STRING	INT	INT(32)	FIXED	REAL	REAL(64)
Operations						
Unsigned arithmetic	●	●				
Signed arithmetic	●	●	●	●*	●**	●**
Logical operations	●	●				
Relational operations	●	●	●	●*	●**	●**
Bit shifts	●	●	●			
Byte scans	●					
Standard Functions						
Type transfer	●	●	●	●	●	●
Character test	●					
Minimum/Maximum	●	●	●	●*	●**	●**
Scaling				●		
Structure	●	●	●	●	●	●
Address conversion	●	●	●	●	●	●
Miscellaneous	●	●	●			

* Fixed-point optional microcode required on the NonStop 1+ system
** Floating-point optional microcode required

STRING Operands

In expressions, the system treats STRING variables and constants as if they were 16-bit quantities. For more information on expressions, see Section 13.

SYNTAX FOR CONSTANTS

The remaining pages of this section give the following syntax definitions for specifying constants in your program:

- 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 (All Data Types)

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

The syntax for specifying a character string constant is:

```
"<string>"

<string>

   is a sequence of one or more ASCII characters enclosed in
   quotation mark delimiters.  If a quotation mark is a
   character within the string, use two quotation marks (in
   addition to the quotation mark delimiters).  TAL does not
   upshift lower case characters.
```

Each ASCII character in the character string requires one byte of storage. Thus, the number of characters that each element can accommodate depends on its data type:

STRING = 1 byte	INT(32) or REAL = 1 to 4 bytes
INT = 1 to 2 bytes	REAL(64) or FIXED = 1 to 8 bytes

In initializations, a character string can contain up to 128 characters. The character string must be on one line unless enclosed in a constant list, described later in this section. The system left justifies the character string. For examples initializing simple variables and arrays with character strings, see Sections 8 and 9.

In expressions, a character string can contain one to four characters, as in "a" or "abcd". The system right justifies the character string. For examples, see "Assignment Statement" in Section 15.

Example of Character String Constant

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

```
   FIXED fix^num := "ABCD1234";
```

STRING Numeric Constants

Representation: Unsigned 8-bit integer

Range: 0 through 255

The syntax for specifying a STRING numeric constant is:

```
[ <base> ] <integer>


<base>

   is one of:

      %   =  Octal
      %B  =  Binary
      %H  =  Hexadecimal

   The default base is decimal.


<integer>

   is one or more digits.  The digits allowed are:

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

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

The syntax for specifying an INT numeric constant is:

```
[ + ] [ <base> ] <integer>
[ - ]
```

<base>

is one of:

%	=	Octal
%B	=	Binary
%H	=	Hexadecimal

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 for each base are:

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

Examples of INT Numeric Constants

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

Storage Format

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

```
 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

The syntax for specifying an INT(32) numeric constant is:

```
[ + ] [ <base> ] <integer> { D  }
[ - ]                      { %D }


<base>

   is one of:

      %   =  Octal
      %B  =  Binary
      %H  =  Hexadecimal

   The default base is decimal.


<integer>

   is one or more digits.  The digits allowed for each base are:

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


 D and %D

    are suffixes that specify INT(32) constants:

        D  =  Decimal, octal, or binary
       %D  =  Hexadecimal
```

Examples of INT(32) Numeric Constants

Decimal:	0D +14769D -327895066d
Octal:	%1707254361d -%24700000221D
Binary:	%B00010010110001000101000100ld
Hexadecimal:	%h096228d%d -%H99FF29%D

Storage Format

The system stores signed numbers 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

The syntax for specifying a FIXED numeric constant is:

```
[ + ] [ <base> ] <integer> [.<fraction>] { F  }
[ - ]                                   { %F }
```

<base>

is one of:

```
%   =  Octal base
%B  =  Binary base
%H  =  Hexadecimal base
```

The default base is decimal.

<integer>

is one or more digits. The digits allowed for each base are:

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

<fraction>

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

F and %F

are suffixes that specify FIXED constants:

```
 F =  Decimal, octal, or binary
%F =  Hexadecimal
```

Examples of FIXED Numeric Constants

Decimal:	1200.09F 0.1234567F 239840984939873494F -10.09F
Binary:	%B1010111010101101010110F
Octal:	%765235512F
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 information on scaling of FIXED values in declarations, see Section 8, "Simple Variables." For information on scaling of FIXED values in expressions, see Section 13, "Expressions."

REAL and REAL(64) Numeric Constants

Representation:	Signed 32-bit REAL or 64-bit REAL(64) floating-point number
Range:	$\pm 8.62 * 10^{-78}$ through $\pm 1.16 * 10^{77}$
Precision:	REAL--to approximately seven significant digits REAL(64)--to approximately 17 significant digits

The syntax for specifying a REAL or REAL(64) numeric constant is:

```
[ + ] <integer>.<fraction> { E } [ + ] <exponent>
[ - ]                      { L } [ - ]
```

<integer>

is one or more decimal digits comprising the integer part.

<fraction>

is one or more decimal digits comprising the fractional part.

E and L

are suffixes that specify floating-point constants:

E = REAL constant
L = REAL(64) constant

<exponent>

is one or two decimal digits comprising the exponential part.

Examples of REAL and REAL(64) Numeric Constants

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. Since the integer part of X is always 1, only the fractional part of X is stored.

The value y is an exponent in the range 0 through 511 (%777). The system adds 256 (%400) to y before storing it. Thus, the exponent is the stored value minus 256. This provides for exponents from -256 (represented by %0) through 255 (represented by %777).

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

	Sign Bit	Fraction	Exponent
REAL	<0>	<1:22>	<23:31>
REAL(64)	<0>	<1:54>	<55:63>

Examples of Storage Formats

1. For the REAL constant shown, 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

2. For the REAL constant shown, 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

3. For the REAL(64) constant shown, the sign bit is 0, the fraction bits contain the octal representation of .333333..., and the exponent bits contain %400 - 2, or %376:

   $1/3 = .333333... * 2^{-2}$ 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:

- Array declarations not in structures (Section 9)
- Group comparison expressions (Section 13)
- Move statements but not assignment statements (Section 15)

The syntax of the constant list is:

```
[ <repetition-factor> * ] "[" <constant-list> "]"


<repetition-factor>

   is an INT constant that specifies the number of times
   <constant-list> occurs


<constant-list>

   is a list of elements stored on an element boundary.  It
   has the form:

   <constant> [ , <constant ] ...


   <constant>

      is a character string, a number, or a LITERAL.  For INT
      arrays only, the constants can be different types.  The
      range and syntax for specifying constants depends on the
      data type.
```

Examples of Constant Lists

1. The two examples in each pair below are equivalent:

```
[ "A", "BCD" , "...", "Z" ]
[ "ABCD...Z" ]

10 * [0];
[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. These examples declare arrays and initialize them using constant lists:

```
STRING  a[0:99] := ["A constant list that is a single ",
                     "character string can continue on ",
                                  "more than one line."];

INT     b[0:79] := 80 * [" "];              !Repetition factor

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

SECTION 5

ADDRESSING MODES

This section summarizes the process environment, the user data space, and the addressing modes used in this environment. The addressing modes described are:

- Byte and word addressing
- Direct and indirect addressing
- Standard and extended addressing
- Indexing

For more information than is given in this section, see the System Description Manual for your system.

PROCESS ENVIRONMENT

Figure 5-1 shows the current process environment. The following registers are shown in this figure:

- Program Counter (P) Register--Contains the address of the next instruction in the code area
- Instruction (I) Register--Contains the instruction that is currently executing
- Local (L) Register--Contains the address of the beginning of the local data area for the most recently called procedure.
- Stack (S) Register--Contains the address of the last allocated word in the dynamic data stack (see also Figure 5-2)

- Register Stack--Eight registers (R0 through R7) for computation; R5, R6, and R7 double as index registers; the register pointer (RP) points to the top of the register stack

- Environment (ENV) Register--Contains information about the current process such as the current RP pointer and whether traps are enabled

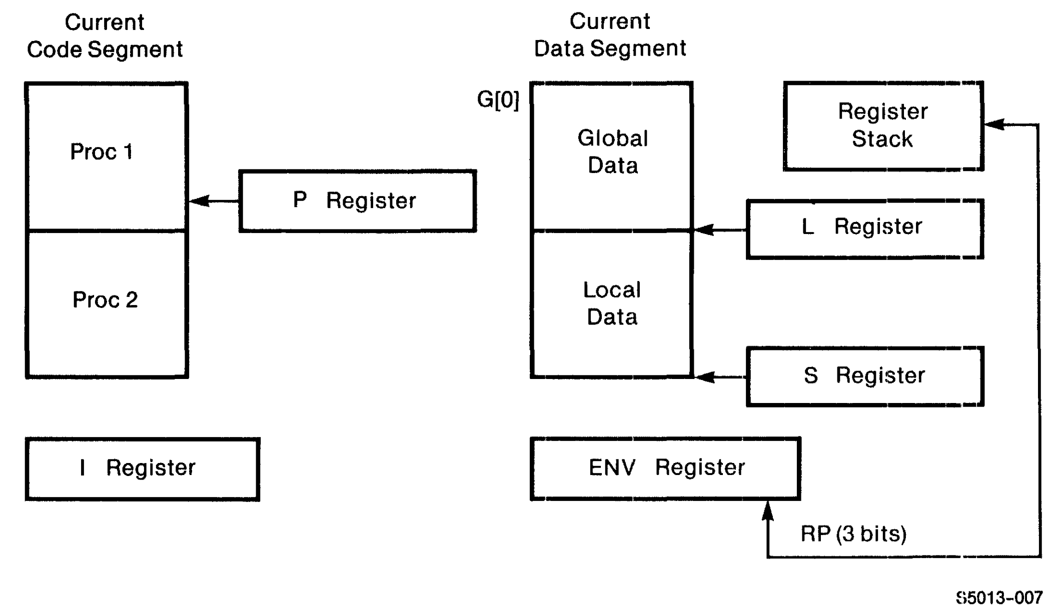


Figure 5-1. Process Environment

USER DATA SPACE

The user data space consists of the current user data segment and extended data segments, if any. (A segment is a non-extended segment except where the word "extended" is specifically used.)

The organization of the current data segment is shown in Figure 5-2.

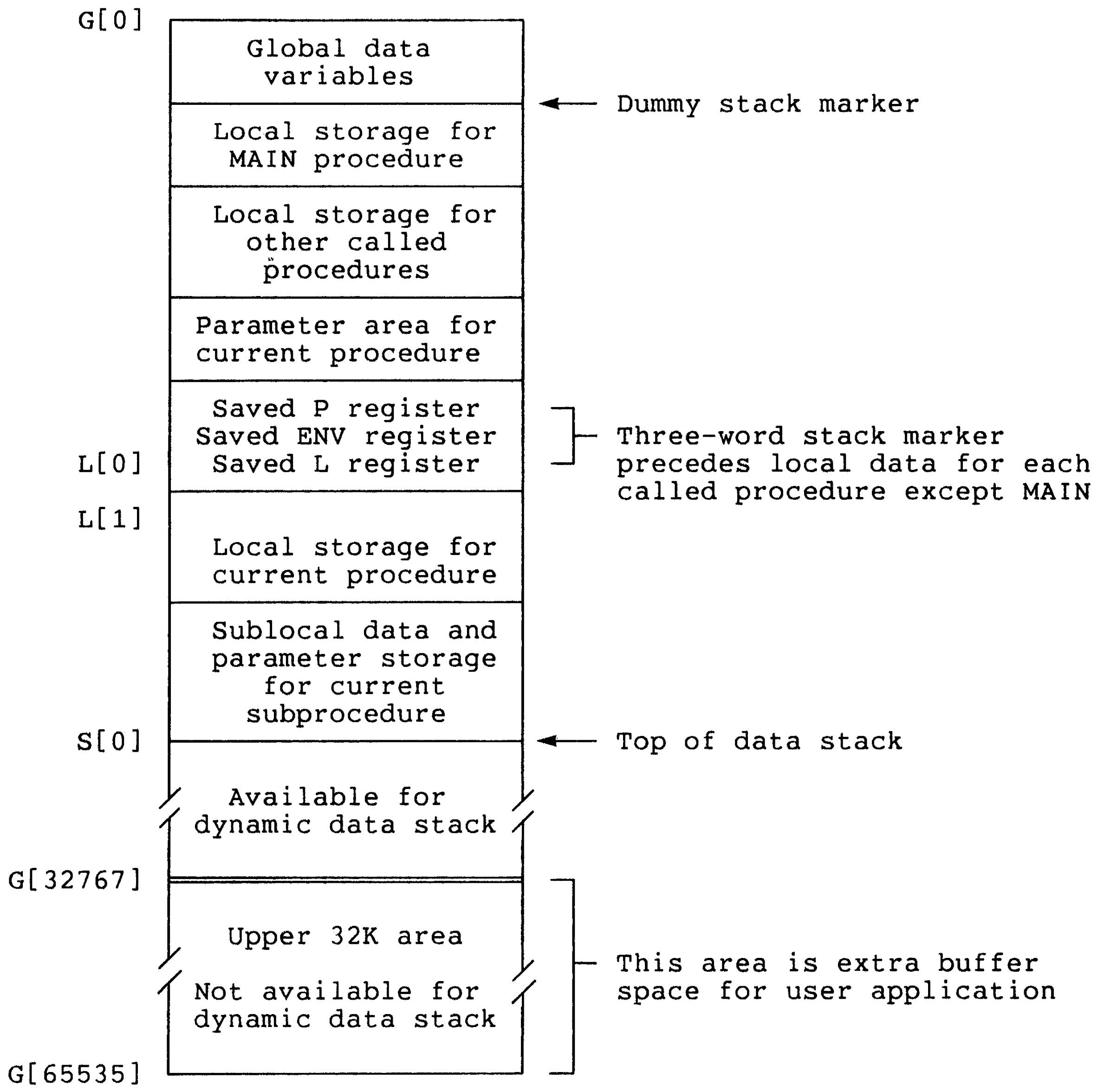

G[0]
Global data variables
Dummy stack marker
Local storage for MAIN procedure
Local storage for other called procedures
Parameter area for current procedure
Saved P register
Saved ENV register
Saved L register
Three-word stack marker precedes local data for each called procedure except MAIN
L[0]
L[1]
Local storage for current procedure
Sublocal data and parameter storage for current subprocedure
S[0]
Top of data stack
Available for dynamic data stack
G[32767]
Upper 32K area
Not available for dynamic data stack
This area is extra buffer space for user application
G[65535]

Figure 5-2. Organization of Current Data Segment

ADDRESSING MODES

Addressing modes are byte and word addressing, direct addressing, standard and extended indirection, and indexing.

Byte and Word Addressing

Figure 5-3 shows byte and word addresses in the data segment.

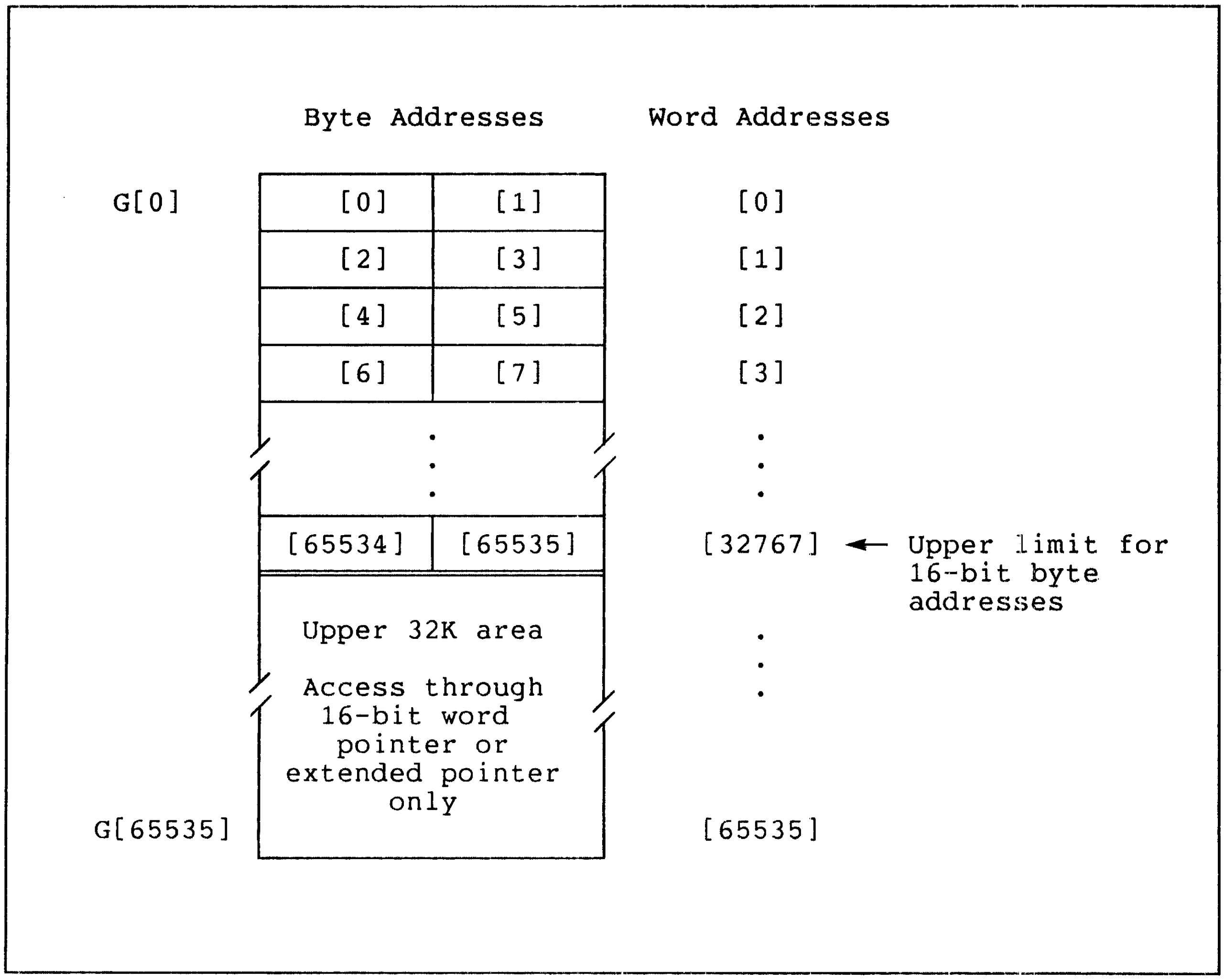


Figure 5-3. Byte and Word Addressing

Except for structures and substructures, the data type of a variable determines whether it has a byte or a word address. Variables of type STRING have byte addresses; variables of any other data type have word addresses.

Structures always have a word address; substructures always have a byte address. (Variables contained in structures and substructures have byte or word addresses based on the data type of the variable.)

For examples specific to simple variables, arrays, and structures, see Sections 8, 9, and 11.

Direct Addressing

Direct addressing is data access that requires only one memory reference. Direct addressing is not absolute but is relative to the base of the global, local or sublocal area of the current data segment.

The range for direct addressing is limited to the lower 32K words of memory. The upper 32K always requires indirect addressing (described next) since it is not part of the dynamic data stack. That is, the upper 32K is not directly addressable using the L or S register.

Indirect Addressing

Indirect addressing is data access through a pointer (a data element that contains the memory address of another data element). Indirect addressing requires two memory references, one to get the pointer contents and the second to get the data element to which the pointer points. Indirect addressing is standard or extended.

Standard Indirection

Standard 16-bit addresses allow access to the current data segment (byte or word addresses in the lower 32K area and word addresses in the upper 32K area). Standard indirection is data access through either:

- Standard pointers and structure pointers you declare and initialize yourself
- Standard pointers TAL provides and initializes when you declare indirect arrays and structures

Extended Indirection

Extended 32-bit addresses allow access to byte addresses in the entire data segment, code segment, and extended data segment. Extended indirection is data access through extended pointers and structure pointers you declare and initialize.

For examples showing standard and extended indirection, see the following sections: Section 9 (indirect arrays), Section 10 (standard or extended pointers), Section 11 (indirect structures and standard or extended structure pointers).

Primary and Secondary Storage

The global and local areas in the data segment each have a primary and secondary storage area. The sublocal area has only primary storage.

The primary areas are directly addressable; they contain pointers and direct variables based on global, local, or sublocal scope. The size of each primary area is:

Global primary area:	256 words
Local primary area:	127 words
Sublocal primary area:	31 words

The secondary areas are indirectly addressable; they contain the data for indirect arrays and structures depending on global or local scope. The secondary areas have no explicit size limit, except that the total data storage cannot exceed the lower 32K area.

Figure 5-4 shows the global and local primary and secondary storage areas and the sublocal primary area.

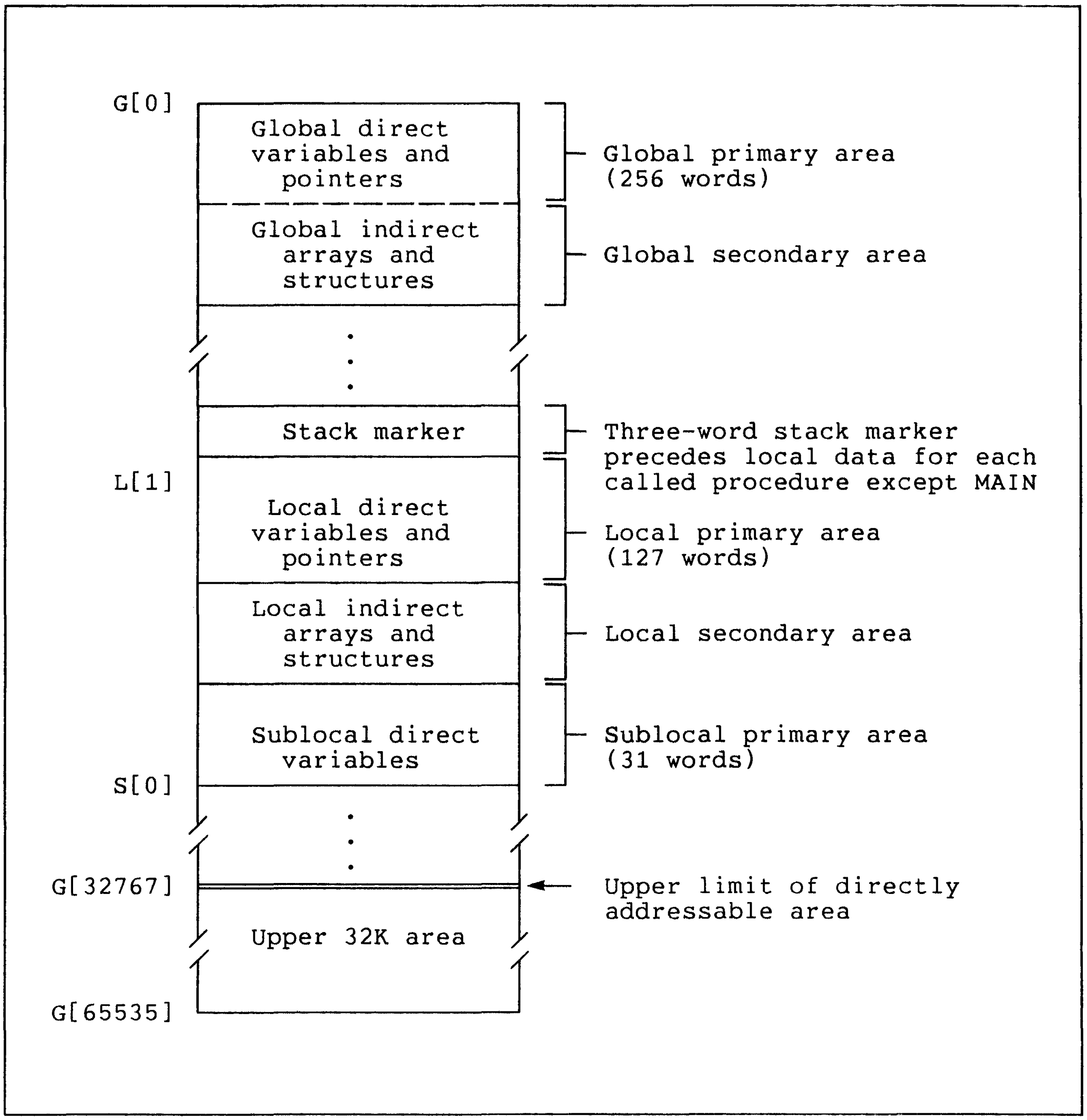

G[0]
Global direct variables and pointers
Global primary area (256 words)
Global indirect arrays and structures
Global secondary area
Stack marker
Three-word stack marker precedes local data for each called procedure except MAIN
L[1]
Local direct variables and pointers
Local primary area (127 words)
Local indirect arrays and structures
Local secondary area
Sublocal direct variables
Sublocal primary area (31 words)
S[0]
G[32767]
Upper limit of directly addressable area
Upper 32K area
G[65535]

Figure 5-4. Primary and Secondary Storage in User Data Segment

Storage Allocation

TAL allocates space for each variable in the order in which you declare them as follows:

- Global Variables--TAL allocates space at compilation for each variable at an offset from the beginning of the data block in which it appears. The data blocks are relocatable at bind time.

- Local and Sublocal Variables--TAL allocates space for each variable when a call to a procedure or subprocedure occurs.

Primary Storage. For global or local variables, TAL allocates primary storage for each direct variable and each pointer. Allocation starts at G[0] (global scope) or L[1] (local scope). Each successive variable or pointer is allocated space at an increasingly higher offset.

For sublocal variables, TAL allocates storage starting at S[0]. Each successive sublocal variable is allocated storage at a negative offset from S[0].

Secondary Storage. TAL allocates storage for the data in each indirect array and structure in the global or local secondary area. The secondary area begins immediately after the last direct variable or pointer.

Examples specific to simple variables, arrays, pointers, structures, and equivalenced variables are given in Sections 8 through 14.

Indexing

You can access data by appending an index to the name of a variable.

The syntax for indexing a variable is:

```
<identifier> "[" <index> "]"
```

<identifier>

is the name of a previously declared variable (simple variable, array, structure, substructure, structure data item, or pointer). The variable can be direct or indirect.

<index>

is one of the following values:

- For standard addressing, it is a signed INT arithmetic expression that represents either:

  --an element offset from the address of a simple variable or array (when appended to a simple variable, array, pointer, or structure data item)

  --an occurrence offset from the address of a structure (when appended to a structure or structure pointer) or from the address of a substructure (when appended to a substructure). An occurrence is one copy of a structure or substructure.

- For an extended pointer, it is a signed INT or INT(32) arithmetic expression.

- For an extended structure pointer, it must be a signed INT arithmetic expression.

The following example shows use of indexes:

```
INT var[0:4];           !Declares array
INT .ptr := %100000;    !Declares pointer
var[2] := 5;            !Assigns 5 to third element of "var"
ptr ':=' [1, 2, 3];     !Moves constant list to location to which
                        ! "ptr" points
var[3] := ptr[2];       !Assigns 3 to fourth element of "var"
```

Indexes and Data Type

The data type impacts the amount of offset yielded by an index. For type STRING, the index yields a byte offset from the variable base. For INT, a word offset; for INT(32) and REAL, a doubleword offset; for REAL(64) and FIXED, a quadword offset.

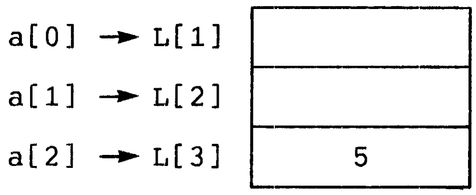
In the following example, "var" contains five doubleword elements:

```
INT(32) var[0:4];   !Declares array
var[3] := 2;        !Accesses the fourth element of "var"
```

Indexing Examples

1. The following example shows an indexed direct variable:

```
PROC x MAIN;
BEGIN
  INT a[0:2];

  a[2] := 5;
END;
```



2. This example of an indexed pointer initializes an INT pointer with the address of an INT(32) array, then assigns a value to the last word of the array via the indexed pointer:

```
PROC z MAIN;
BEGIN
  INT(32) d[0:4] := [1D, 2D, 3D, 4D, 0D];
  INT .p := @d[0];        !View "d" as single words

  p [9] := 5;             !Last word of "d" is a
END;                      ! nine-word offset from p[0]
```

Figure 5-5 shows the array before and after the assignment. (L[0] contains the third word of the 3-word stack marker.)

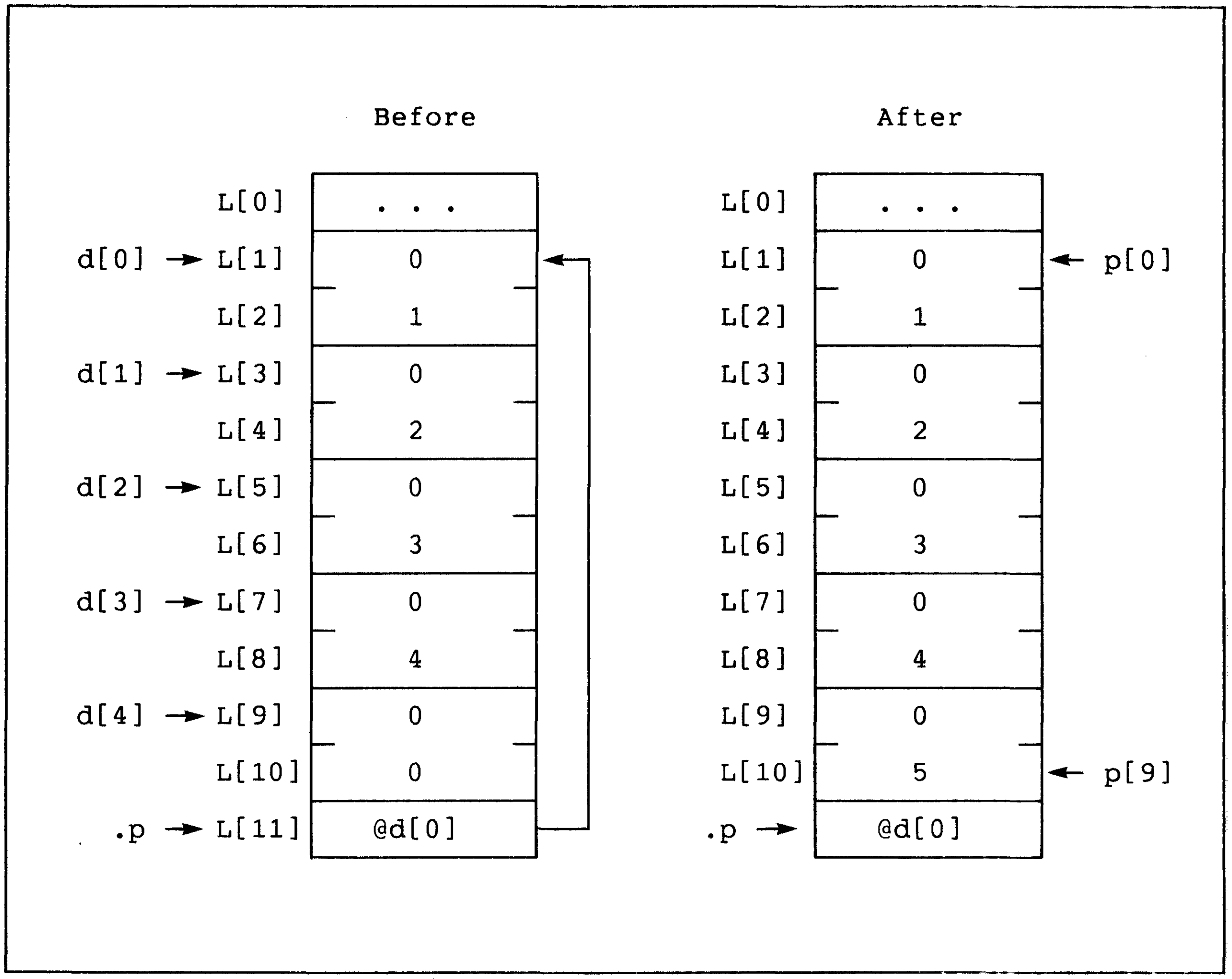


Figure 5-5. Indexing a Pointer

SECTION 6

LITERALS AND DEFINES

This section describes the following declarable objects:

- LITERALS--Named constants
- DEFINES--Named text with or without parameters

For each, the following information is given:

- Declarations
- Compiler action
- Data access

LITERAL and DEFINE declarations let you define constants and text once in a program, then reference them by name 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

The LITERAL declaration associates an identifier with a constant.

The syntax for the LITERAL declaration is:

```
LITERAL <identifier> = <constant>

     [ , <identifier> = <constant> ] ... ;


<identifier>

   is an identifier associated with <constant>.


<constant>

   is an INT, INT(32), FIXED, REAL, or REAL(64) constant
   expression or a character string of one to two characters.

   It must not be the address of a global variable because all
   global variables are relocatable.
```

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

TAL allocates no storage for LITERAL constants. It substitutes the associated value at each occurrence of the identifier.

LITERAL identifiers make the source code more readable. In the example shown on the next page, identifiers such as "buffer^length", "table^size", "table^base", and "entry^size" are more readable than their corresponding constant values (80, 128, %1000, and 4).

Examples

1. The following example shows various LITERAL declarations:

```
LITERAL true          =        -1,
        false         =         0,
        buffer^length =        80,
        table^size    =       128,
        table^base    =     %1000,
        entry^size    =         4,
        timeout       = %100000D,
        CR            =       %15,
        LF            =       %12;
```

2. The following example declares the length of an array as a LITERAL constant, then references the LITERAL identifier in an array declaration:

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

3. The following example declares LITERAL constants, then references their identifiers in subsequent LITERAL declarations:

```
LITERAL second      =             1,
        minute      = second * 60,
        hour        = minute * 60,
        over^time   =   hour + 30,
        double^time =    2 * hour;
```

DEFINE DECLARATION

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

The syntax for the DEFINE declaration is:

```
 DEFINE <identifier> [ ( <param> [ , <param> ] ...) ] = <text> #

    [ , <identifier> [ ( <param> [ , <param> ] ...) ] =

                                                <text> # ] ... ;
```

<identifier>

is the name associated with <text>.

<param>

is the name of a formal parameter.

<text>

is all characters between = and #. Enclose character strings in quotation marks ("). To use # in the <text>, enclose it in quotation marks or embed it in a character string.

#

terminates a definition.

When specifying <text>, the following rules apply:

- The expanded text must produce legal TAL constructs.
- The text must not be recursive; that is, it must not call itself.

Examples of DEFINE Declarations

1. This example shows a DEFINE declaration with no parameters:

```
    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:

```
    DEFINE word^val (a, b) = (a '<<' 12) LOR b #;
```

Compiler Operation

TAL allocates no storage for defined text. When TAL encounters a DEFINE identifier in a statement, it replaces the identifier with the text, compiles it, and emits any machine instructions needed.

Accessing Defined Text

You access defined text by using its identifier in a statement.

If you use a DEFINE identifier in an expression, make sure that proper evaluation occurs. For example, if the DEFINE identifier represents an expression to be evaluated first, you must enclose the text in parentheses:

```
   DEFINE expr = (5 + 2) #;
   j := expr * 4;          !Means (5 + 2) * 4 and assigns 28 to "j"
```

Without parentheses, the same example has a different result:

```
   DEFINE expr = 5 + 2 #;
   j := expr * 4;          !Means 5 + 2 * 4 and assigns 13 to "j"
```

Passing Parameters

If the DEFINE declaration has formal parameters, you supply the actual parameters when you reference the DEFINE identifier in a statement. The following rules apply to actual parameters:

- If an actual parameter requires commas, enclose the 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 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 Accessing Defined Text

1. The following example shows a DEFINE declaration and the statement that references it:

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

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

2. This example provides incrementing and decrementing utilities and a statement that references one of them:

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

increment(index);           !Means "index := index + 1;"
```

3. The following 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. The following example displays a message, checks the condition code, and returns an error if one occurs:

```
INT error;
INT file;
INT .buffer[0:50];
INT count^written;

DEFINE emit (filenum, text, bytes, count) =
               BEGIN
                 CALL WRITE (filenum, text, bytes, count);
                 IF < THEN
                   BEGIN
                     CALL FILEINFO (filenum, error);
                     RETURN error;
                   END;
                END #;
    .
    .
IF i = 1 THEN emit (file, buffer, 80, count^written);
```

SECTION 7

LABELS

This section describes how to declare and use labels. A label is an identifier you use with the GOTO statement.

LABEL DECLARATION

The LABEL declaration reserves an identifier for later use as a label.

The syntax of the LABEL declaration is:

```
LABEL <identifier> [ , <identifier> ] ... ;

<identifier>

   is the name of the label.  It cannot be a global declaration.
```

Labels are the only declarable objects you do not need to declare before using them. However, declaring them ensures that you access the label rather than a variable in the event they have the same name. (See Examples 4 and 5.)

Local Labels

The steps for declaring, using, and referencing local labels are:

1. Declare the label name inside a procedure.

2. Place the label name and a colon (:) preceding a statement in the same procedure (not in a subprocedure).

3. Reference the label in another statement located in the same procedure or in any subprocedure contained in that procedure.

Sublocal Labels

The steps for declaring, using, and referencing sublocal labels are:

1. Declare the label name inside a subprocedure.

2. Place the label name and a colon (:) preceding a statement in the same subprocedure.

3. Reference the label in another statement located in the same subprocedure.

Referencing Labels

Statements you can use for referencing labels include:

- A GOTO statement to branch to the label

  A GOTO statement in a procedure can reference a label in the same procedure, but not in any subprocedure.

  A GOTO statement in a subprocedure can reference a label in either the same subprocedure or the encompassing procedure.

- An assignment statement to store the address of the label in a variable

Examples

1. This example shows valid placements of undeclared local labels:

```
PROC a;
INT a, b;
   .
   .
label^a : IF a>b               !Valid placement of labels
          THEN
label^b :   <statement>        !
          ELSE
label^c :   <statement>;       !
```

2. This example is not a legal use of labels because a label cannot have global scope, and you must place it at the start of a statement:

```
LABEL label^a;                 !Invalid label declaration; a
                               ! label cannot be global
PROC b;
INT a, b;
   .
   .
          IF a>b
          THEN
            <statement>
label^a : ELSE                 !Invalid placement of label;
            <statement>;       ! ELSE does not begin a statement
```

3. This example declares a label and makes two branches to it:

```
INT op1, op2, result;                    !Global declarations

PROC p;
BEGIN
       LABEL addr;                       !Label declaration
       op1 := 5;
       op2 := 28;
       GOTO addr;                        !Branches to the label
            .
            .
addr : result := op1 + op2;              !Labeled location
       op1 := op2 * 299;
            .
            .
       IF result < 100 GOTO addr;        !Branches to the label
```

4. This example uses an undeclared label name that is also the name of a global variable. Using the name accesses the address of the variable, not the address of the label as intended.

```
INT loop, data;                         !Global variables

PROC p;
BEGIN
           .
           .
      data := @loop;                    !Stores address of variable
           .                            ! instead of label
           .
loop : a := 0;                          !Uses undeclared label
```

5. This example corrects example 4 by declaring the label. It stores the address of the label in a variable:

```
INT loop, data;                         !Global variables

PROC p;
BEGIN
      LABEL loop;                       !Label declaration
           .
           .
      data := @loop;                    !Stores label address
           .                            ! in "data"
           .
loop : <statement>                      !Labeled location
END;
```

SECTION 8

SIMPLE VARIABLES

A simple variable is a single-element variable of a specified data type. You allocate storage for it through a data declaration, then use it in statements to access or change its data.

This section gives information on simple variables:

- Declaration
- Initialization
- Storage allocation
- Data access

SIMPLE VARIABLE DECLARATION

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

The syntax for the simple variable declaration is:

```
<type>  <identifier> [ := <initialization> ]

       [ , <identifier> [ := <initialization> ] ] ... ;
```

<type>

is one of the following data types:

```
STRING
INT
INT(32)
FIXED [ ( <fpoint> ) ]
REAL
REAL(64)
```

<fpoint>

is the position of the decimal point. It is a value in the range -19 through 19. The default value is 0 (no decimal places). A positive value is the number of decimal places. A negative value is the number of integer places between the least significant digit and the decimal point.

If <initialization> has a different decimal setting than <fpoint>, the system scales <initialization> to match <fpoint>. If the value is scaled down, some precision is lost.

<identifier>

is the name of the simple variable in the form described in Section 3 under "Identifiers."

<initialization>

is a constant expression (global data) or an arithmetic expression (local or sublocal data).

Initializing Simple Variables

The data type of the initializing value must match that of the variable, except for character strings. If a character string is smaller than the space allocated, TAL left justifies the characters in the variable and sets the extra bytes to 0. If it is too large, TAL truncates the excess characters and emits a warning.

Examples of Simple Variable Declarations

1. The following examples declare simple variables:

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

2. The following examples declare and initialize simple variables:

```
STRING y := "A";                        !Character string
STRING z := 255;                        !Unsigned number
INT a := "AB";                          !Character string
INT b := 5 * 2;                         !Expression
INT c := %B110;                         !Binary value
INT(32) dblwd2 := %B1011101D;           !Doubleword value
REAL flt2 := 365335.6E-3;               !Real value
REAL(64) b := 2718.2818284590452L-3;    !Quadword value
```

3. The following examples show FIXED declarations and how the <fpoint> affects storage (and scaling):

```
FIXED(-3) f := 642000F;  !Stores 642
FIXED(3)  g := 0.642F;   !Stores 642 (three implicit decimal
                         ! places)
FIXED(2)  h := 1.234F;   !Scales rightmost digit; stores 123
                         ! (two implicit decimal places)
```

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

```
INT global := 34;                 !Constants allowed at global,
                                  ! local, or sublocal levels
PROC mymain MAIN;
BEGIN
  INT local := global + 10;       !Variables allowed at
  INT local2 := global * local;   ! local or sublocal levels
  FIXED local3 := $FIX(local2);   ! but not at global level
  .
  .
END;                              !End of "mymain" procedure
```

STORAGE ALLOCATION

Figure 8-1 shows simple variable declarations and the offset allocation that results.

For a simple variable of type STRING, TAL allocates one word of storage. The initializing value is stored in the left byte and a zero is stored in the right byte.

```
STRING a;
INT(32) b;     !Global data

PROC proc^a;
BEGIN
  STRING c;    !Local data
  REAL d;
     .
     .
     .
  SUBPROC subproc^a;
  BEGIN
    INT e;     !Sublocal data
    FIXED f;
  END;
END;
```

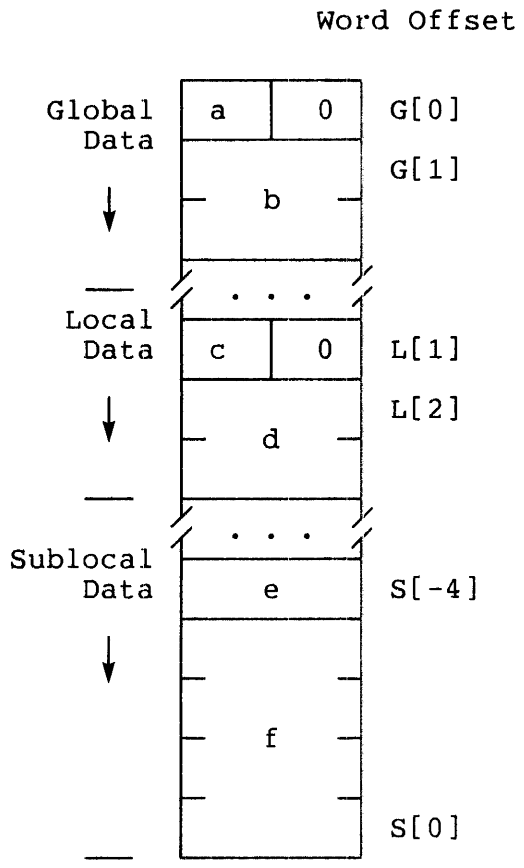


Figure 8-1. Storage Allocation for Simple Variables

ACCESSING SIMPLE VARIABLES

To access a declared simple variable, you use its name in a statement, with or without an index.

Examples of Accessing Simple Variables

1. The following example declares and initializes a simple variable, then assigns a new value to it:

```
INT count := 0;          !Declaration and initialization

count := count + 1;      !Assignment
```

2. This example shows how initialization left justifies a one-byte character string, whereas an assignment right justifies it (unless you assign a character and a space):

```
INT v := "A";       !Declares "v"; initializes
                    ! it with character
INT x, z;           !Declares "x" and "z"

x := "A";           !Assigns character to "x"
z := "A ";          !Assigns character and
                    ! space to "z"
```

v	"A"	0
x	0	"A"
z	"A"	

3. This example shows indexed access to a simple variable:

```
INT i;
INT j;
INT k;

i[2] := 0;      !"k" gets 0
```

SECTION 9

ARRAYS

An array is a collectively stored set of elements of the same data type. You can use the same identifier to access the elements individually or as a group.

This section describes one-dimensional arrays:

- Arrays stored in the current user data segment
- Read-only arrays stored in a user code segment

Information discussed includes:

- Array declarations
- Storage allocation
- Data access

Arrays within structures and multidimensional arrays are described in Section 11, "Structures."

ARRAY DECLARATION

An array declaration associates an identifier with a set of elements of the same data type collectively stored in the user data segment.

The syntax for the array declaration is:

```
<type> [ . ] <identifier> "[" <lower-bound> : <upper-bound> "]"
                                     [ := <initialization> ]
   [ , [ . ] <identifier> "[" <lower-bound> : <upper-bound> "]"
                                     [ := <initialization> ] ] ... ;
```

<type>

is one of the following data types:

```
STRING
INT
INT(32)
FIXED [ ( <fpoint> ) ]
REAL
REAL(64)
```

<fpoint> is as described for simple variables in Section 8.

. (period)

is the indirection symbol for standard addressing.

<identifier>

is the name of the array in the form shown in Section 3 under "Identifiers."

<lower-bound>

is an INT constant expression in the range -32768 through 32767 that defines the first array element. Both lower and upper bounds are required.

⟶

<upper-bound>

is an INT constant expression in the range -32768 through 32767 that defines the last array element. For arrays outside of structures, <upper-bound> must be equal to or greater than <lower-bound>.

<initialization>

is a numeric or character string constant or a constant list to assign to the array elements.

Direct Versus Indirect Arrays

In the global and local areas, you can declare direct or indirect arrays. In the sublocal area, arrays must be direct.

Because the global and local primary areas are limited to 256 and 127 words of direct data each, you should declare most arrays by using the indirection symbol. TAL manages indirection for you by providing a pointer and initializing it with the location of the data. To access an indirect array, you reference it by name as if it were a direct array.

Array Base

To the TAL compiler, the base of an array is element [0] regardless of the lower and upper bounds specified. For example, if you declare array bounds of [-5:5] or [3:7], TAL allocates space only for the specified range, but the array base is still element [0].

For direct arrays, the array base must be addressable. The base must reside between 'G' relative word addresses [0:32767]. For example:

- If the first global array is direct, its lower bound must be a 0 or negative value, since the global area has 'G' plus addressing only.
- The upper bound of the last sublocal array must be a 0 or larger value, since the sublocal area has 'S' minus addressing only.

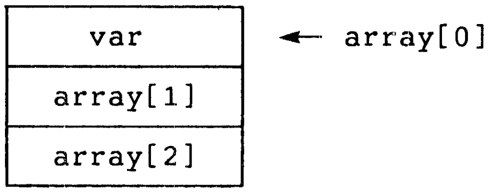
Examples of Array Declarations

1. This example declares indirect arrays with various bounds:

```
INT    .array^a[0:0];                  !One-element array
INT    .array^b[-1:0];                 !Two-element array
FIXED  .array^c[0:3];                  !Four-element array
INT    .array^d[0:49];                 !Fifty-element array
```

2. In this example, the simple variable and the array base (logical element [0]) are located at the same address:

```
INT var;
INT array[1:2];
```



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

```
INT .b^array[0:9] := [1,2,3,4,5,6,7,8,9,10];      !Constant list

STRING .buffer[0:108] := [ "You can use a constant list when ",
                          "a character string constant is too ",
                                "long to fit on one line." ];

INT(32) .mixed[0:4] := ["abcd", 1D, %B0101011D, %20D];  !Mixed
                                                        ! constant list

LITERAL len = 80;                                    !Length of array
STRING .buffer[0:len - 1] := len * [" "];          !Repetition factor

INT .rec[0:11] := ["$RECEIVE", 8*["  "]];        !GUARDIAN file name

FIXED .f[0:20] := 3*[2*[1F,2F], 4*[3F,4F]]; !Repetition factors

LITERAL cr = %15,
        lf = %12;
STRING .err^msg[0:15] := [cr, lf, "ERROR", cr, lf, 0];
```

Storage Allocation

The data type and number of elements determine the amount of storage TAL allocates for array data. Direct or indirect addressing determines if the data is allocated in primary or secondary storage.

Direct Array Allocation

For global direct arrays, TAL allocates primary storage at offsets from the beginning of the global data block that contains the arrays.

For local or sublocal arrays, TAL allocates primary storage at offsets from the base of the local or sublocal storage area.

Figure 9-1 shows an example of direct array declarations and the offset storage that results.

```
INT(32) a[0:1];      !Global
INT b[1:2];          ! arrays

PROC proc^a;
BEGIN
  STRING c[0:2];     !Local
  FIXED d[0:3];      ! arrays
     .
     .
  SUBPROC subproc^a;
  BEGIN
    INT e[0:1];      !Sublocal
    STRING f[0:3];   ! arrays
  END;
END;
```

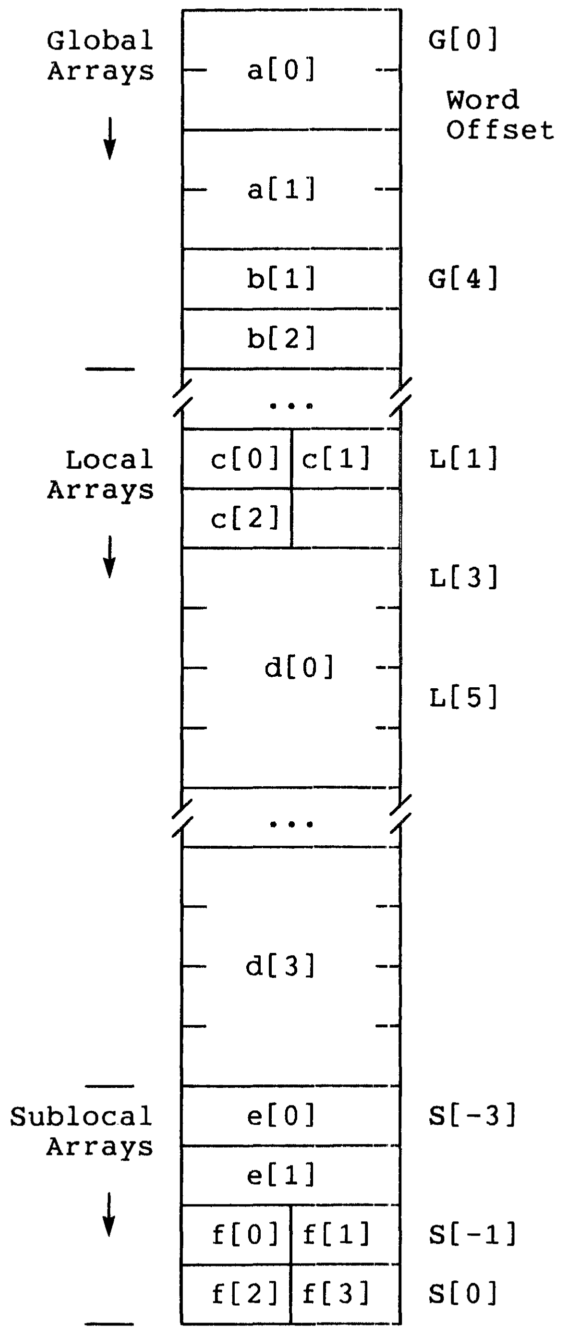


Figure 9-1. Storage Allocation for Direct Arrays

Indirect Array Allocation

For each indirect array, TAL allocates storage for a 16-bit pointer in the global or local primary area. It then allocates the array data in the corresponding global or local secondary area. Finally, the system initializes the pointer with the base address of the array. For a STRING array, the pointer contains a byte address. For any other type of array, the pointer contains a word address.

Figure 9-2 shows allocation for global indirect arrays. In this example, the global secondary storage area begins at location G[4].

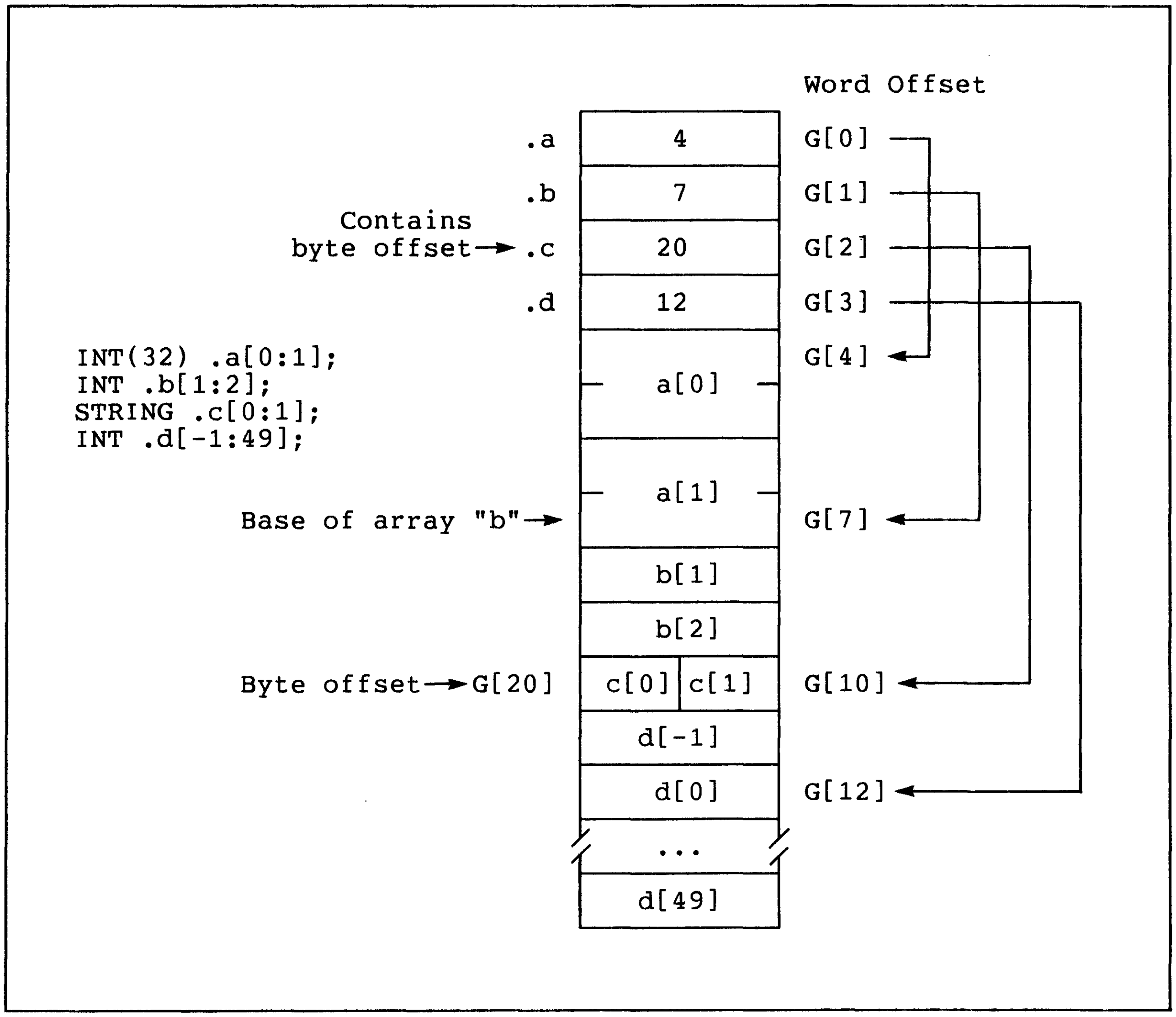


Figure 9-2. Storage Allocation for Indirect Arrays

Data Access

The method for accessing data in direct and indirect arrays is the same. You reference the array name in a statement.

To access a particular element, you reference the array name suffixed with an index value, as in "buffer[2]". If you reference the array name with no index, you access element [0]. Thus, the references "buffer" and "buffer[0]" are equivalent.

Because TAL does no bounds checking, you access an address outside the array if the index value is outside the upper and lower bounds declared for the array.

To access byte elements in a word-aligned array, you must convert the word address of the array element to a byte address. You can use a bit shift operation for address conversions. Operating system procedures, for example, require INT arrays, but the SCAN statement requires byte elements. (See Example 3.)

Array operations include:

- Assigning values to elements one at a time using assignment statements
- Moving values into multiple elements using a move statement
- Scanning multiple elements using an SCAN or RSCAN statement
- Comparing multiple elements using a group comparison expression

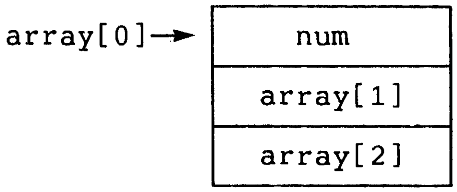
Examples

1. The following example shows how accessing of direct and indirect arrays is the same:

```
INT dir^array[0:2];
INT .ind^array[0:2];

dir^array[2] := 5;
ind^array[2] := 5;
```

2. The following example assigns a value to an out-of-bound address:

```
INT num;
INT array[1:2];

array[0] := 4;    !"num" gets 4
```



3. This example uses a bit shift operation ('>>' 1) to convert the word address of an INT array to a byte address. It loads the byte address into a STRING pointer to scan bytes in the array:

```
INT .array[-1:8] := [0,"Doe, J",0];  !Declares INT array

STRING .s^ptr := @array[0] '<<' 1;   !Declares STRING pointer;
                                     ! initializes it with byte
                                     ! address of array

SCAN s^ptr[0] UNTIL ",";             !Scans bytes in array
```

4. This example accesses an array element by using index variables:

```
INT .b[0:10];                        !Declares arrays
INT .c[0:9];
INT x;                               !Declares indexes
INT y;
INT z;

!Code to manipulate indexes
  .
  .
b[x] := c[y-z];                      !Accesses array element
```

5. This example compares the contents of two arrays and fills the first array with zeros if the contents match:

```
LITERAL count = 99;                  !Declares array length
INT .array[0:count - 1];             !Declares arrays
INT .text[0:count - 1];

!Code to manipulate arrays
  .
  .
IF array[0] = text[0] FOR count      !Compares arrays
THEN array[0] ':=' count * [0];      !Fills "array" with
                                     ! zeros if contents
                                     ! match
```

READ-ONLY ARRAY DECLARATION

A read-only array declaration allocates storage for a nonmodifiable array in a user code segment.

The syntax for the read-only array declaration is:

```
<type> <identifier> [ "[" <lower-bound> : <upper-bound> "]" ]

                          = 'P' := <initialization>

   [ , <identifier> [ "[" <lower-bound> : <upper-bound> "]" ]

                          = 'P' := <initialization> ] ... ;
```

<type>

is one of the following data types:

STRING
INT
INT(32)
FIXED [(<fpoint>)]
REAL
REAL(64)

<fpoint> is as described in Section 8 for simple variables.

<identifier>

is the name of the read-only array.

<lower-bound>

is an INT constant expression defining the first array element. The default value is [0].

<upper-bound>

is an INT constant expression defining the last array element. 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 numeric or character string constant or a constant list to assign to the array elements. Initialization at declaration is mandatory.

Because code segments have no primary or secondary areas, read-only arrays must be direct.

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

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

Data Access

You access global read-only arrays in the same manner as any other array, except that you cannot modify read-only arrays. That is, you cannot specify them on the left side of an assignment operator (:=).

Procedures can access any global read-only array in the same 32K of the code segment.

Procedures in the upper 32K of the code segment can access global STRING read-only arrays in the lower 32K words only by using extended pointers (described in Section 10, "Pointers"). You declare and load an extended pointer with the address of the read-only array, then use the pointer in a procedure to access the array in the same code segment.

You can pass the data of a read-only array by reference to a procedure only if the read-only array, the called procedure, and the calling procedure all reside in the same code segment.

Examples

1. The following example declares read-only arrays using default lower and upper bounds:

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

2. The following example moves a read-only array into a data array:

```
STRING message = 'P' := ["** LOAD MAG TAPE #00144"];
STRING .array[0:22];

array ':=' message FOR 23;
```

SECTION 10

POINTERS

A pointer is a variable that contains the address of a data item. When you reference a pointer, you access the variable whose address is stored in the pointer.

Pointers are standard or extended:

- Standard pointers can access data in the current data segment (word-addressed data in the entire data segment; byte-addressed data in the lower 32K area).
- Extended pointers can access data in the current data segment, in an extended data segment created as described in Appendix A, or in the current user or system code segments (read access only).

This section describes pointers you declare and manage yourself:

- Declaring and initializing pointers
- Assigning values to pointers
- Accessing data by using pointers

It also tells how to get addresses of other items and how to use INT variables as temporary pointers.

This section does not describe the following kinds of pointers:

- Pointers that TAL provides when you declare indirect arrays (see Section 9) or indirect structures (see Section 11)
- Structure pointers (see Section 11)
- System global pointers (see Section 18)

POINTER DECLARATION

The pointer declaration associates an identifier with a memory location containing the user-initialized address of another variable or buffer area.

The syntax for the pointer declaration is:

```
<type>  { .    } <identifier> [ := <initialization> ]
        { .EXT }

    [ , { .    } <identifier> [ := <initialization> ] ] ... ;
        { .EXT }
```

<type>

is one of the following data types and specifies the type of value to which the pointer points:

```
STRING
INT
INT(32)
FIXED [ ( <fpoint> ) ]
REAL
REAL(64)
```

. (period)

is the indirection symbol for standard addressing.

.EXT

is the indirection symbol for extended addressing. It is a reserved word only when followed by <identifier>. At least one space must precede and follow the symbol.

<identifier>

is the name of the pointer.

⟶

<initialization>

is a constant expression (global scope) or an arithmetic expression (local or sublocal scope) as follows:

- If <identifier> is a standard STRING pointer, use a 16-bit byte address in the lower 32K area.
- If <identifier> is a standard non-STRING pointer, use a 16-bit word address in the 64K area.
- If <identifier> is an extended pointer of any type, use a 32-bit byte address. For details, see Appendix A.

If <initialization> represents the contents of another pointer or the address of an array or structure, the form for <initialization> is:

@<previous-identifier>

@

is the symbol for removing indirection.

<previous-identifier>

is the name of a previously declared pointer, array, or structure, with or without an index.

Before you reference a declared pointer, be sure you have assigned a value to it, either in the pointer declaration or in a subsequent statement (see "Pointer Assignments" in this section). References to uninitialized pointers cause undefinable program execution.

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

Extended pointer declarations should precede other global or local declarations. TAL emits more efficient machine code if it can store extended pointers between G[0] and G[63] or between L[0] and L[63].

Examples of Standard Pointer Declarations

All examples apply to global, local, and sublocal pointer, unless otherwise noted.

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

```
INT(32) .ptr;                   !Declares pointer
```

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

```
STRING .bytes[0:3];             !Declares indirect array
STRING .s^ptr := @bytes[3];     !Declares pointer; initializes
                                ! it with location of "bytes[3]"
```

3. This example declares a standard pointer and initializes it with the starting address of the upper 32K area of the data segment:

```
FIXED .ptr := %100000;          !Declares pointer; initializes
                                ! it with first address in upper
                                ! 32K area
```

4. This example declares standard pointers and initializes them with the contents of another pointer:

```
INT .ptr1 := %100000;           !Contains first word of upper 32K
INT .ptr2 := @ptr1;             !Contains same address
INT .ptr3 := @ptr1 [2];         !Contains third word of upper 32K
```

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

```
INT .i[0:39];                   !Declares INT array
STRING .pt := @i[0] '<<' 1;     !Declares STRING pointer;
                                ! initializes it with array byte
                                ! address that results from bit
                                ! shift operation ('<<' 1)
```

6. This example declares an INT pointer and initializes it with the converted word address of a STRING array. This allows word access to the byte-addressed array. Any indexes appended to this pointer must be even.

```
STRING .b [0:4];                !Declares STRING array
INT .ptr := @b[0] '>>' 1;       !Declares INT pointer; initializes
                                ! it with array word address that
                                ! results from bit shift operation
```

7. This example declares a direct array and local or sublocal standard pointers and initializes them with values derived from the array declaration:

```
INT var[0:1] := [%100000, %110000];      !Declares array

INT .int^ptr1 := var[0];     !Declares pointer; initializes it
                             ! with value of first array element

INT .int^ptr2 := var[1];     !Declares pointer; initializes it
                             ! with value of second array
                             ! element
```

Examples of Extended Pointer Declarations

1. This example declares an extended pointer and initializes it with the first location in the upper 32K of the current data segment:

```
INT .EXT ptr := %200000D;    !Declares extended pointer;
                             ! initializes it with first
                             ! location of upper 32K area
```

2. This example declares a local or sublocal extended pointer and initializes it with the 32-bit address returned by the $XADR standard function for array "a", which has a standard address:

```
INT .a[0:1];                 !Declares INT array
STRING .EXT s := $XADR(a);   !Declares exended pointer;
                             ! initializes it with 32-bit
                             ! address retruned for array "a"
```

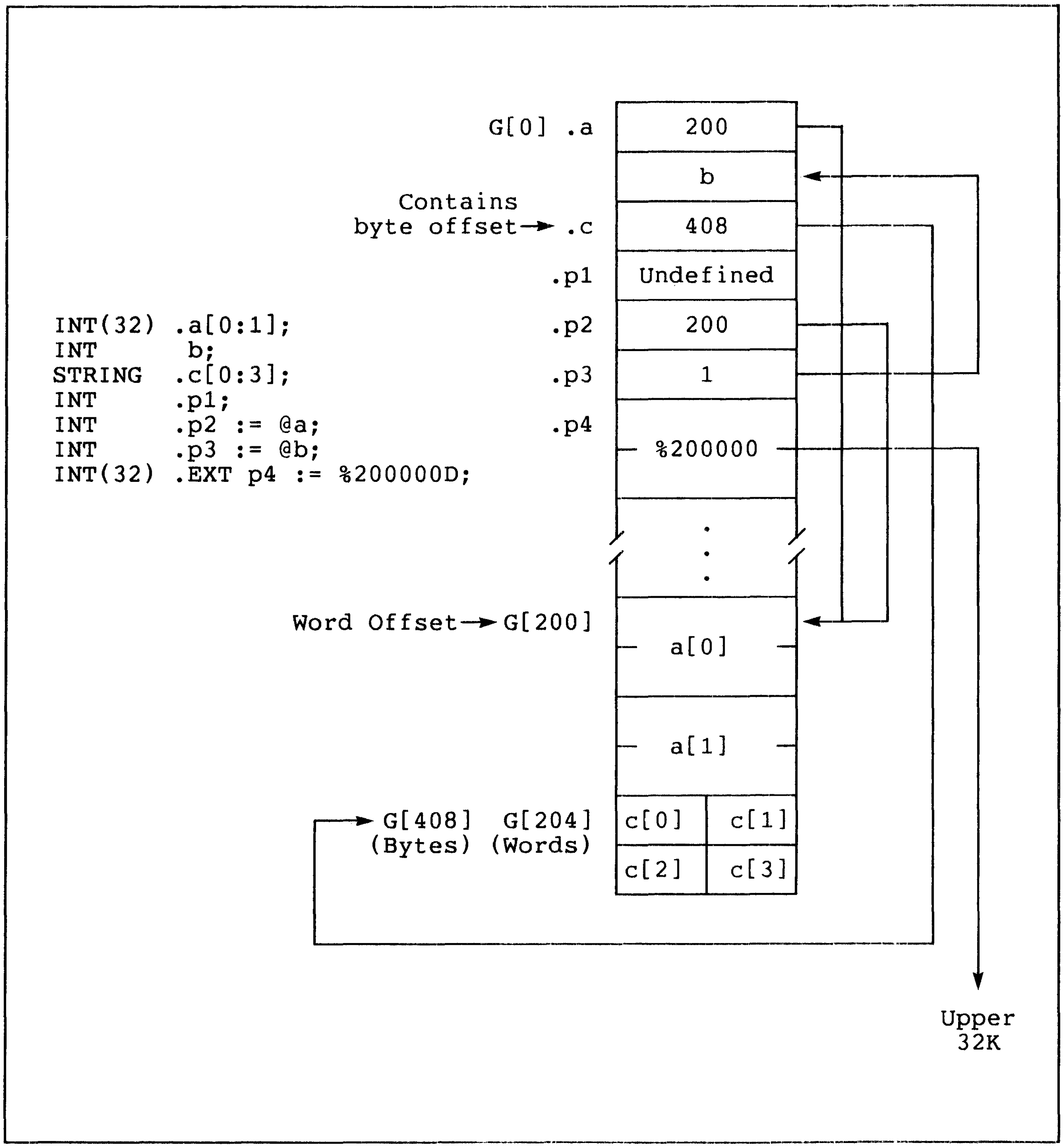
3. This example declares an extended pointer and initializes it with the first address in a previously allocated extended data segment:

```
INT .EXT ptr := %2000000D;   !Declares extended pointer;
                             ! initializes it with first address
                             ! in extended data segment
```

For additional examples using extended pointers to access data in extended segments, see Appendix A.

STORAGE ALLOCATION

TAL allocates primary storage for each pointer in the order in which you declare them. For a standard pointer, TAL allocates one word. For an extended pointer, it allocates two words. Figure 10-1 shows example pointer declarations and the resulting storage allocation.



```
INT(32) .a[0:1];
INT      b;
STRING  .c[0:3];
INT     .p1;
INT     .p2 := @a;
INT     .p3 := @b;
INT(32) .EXT p4 := %200000D;
```

Figure 10-1. Pointer Storage Allocation

POINTER ASSIGNMENTS

The syntax for a pointer assignment is:

```
@<pointer-name> := <arithmetic-expression> ;
```

@

is the symbol for removing indirection. It means get the contents of <pointer-name>, not the item pointed to.

<pointer-name>

is the name of a previously declared standard or extended pointer.

<arithmetic-expression>

is an arithmetic expression that evaluates to one of the following values:

- If <pointer-name> is a standard STRING pointer, use a 16-bit byte address in the lower 32K area.
- If <pointer-name> is a standard non-STRING pointer, use a 16-bit word address in the 64K area.
- If <pointer-name> is an extended pointer of any type, use a 32-bit byte address. For details, see Appendix A.

If the value represents the contents of another pointer or the address of an array or structure, use the following form as described under "Pointer Declaration" in this section:

```
@ <previous-identifier>
```

Examples of Standard Pointer Assignments

1. This example assigns the address of an INT array to standard pointers of different types. The FIXED pointer allows viewing of the array four words at a time; the INT(32) pointer allows viewing two words at a time.

```
INT .array[0:99];             !Declares INT array
FIXED .quad^pt;               !Declares FIXED pointer
INT(32) .dbl^pt;              !Declares INT(32) pointer

@quad^pt := @array[0];        !Assigns array address to
                              ! FIXED pointer
@dbl^pt  := @array[0];        !Assigns array address to
                              ! INT(32) pointer
```

2. This example assigns the converted byte address of an INT array element to a STRING pointer, allowing byte access to the word element:

```
STRING .s^ptr;                !Declares STRING pointer
INT .word[0:5];               !Declares INT array

@s^ptr := @word[3] '<<' 1;    !Assigns byte address of
                              ! "word[3]" (converted by
                              ! bit shift operation)
```

Examples of Extended Pointer Assigments

1. This example uses the $XADR standard function to return a 32-bit address for a STRING array, then assigns the address to an extended pointer:

```
INT .EXT ext^ptr;             !Declares extended pointer
STRING s^array[0:1];          !Declares STRING array

@ext^ptr := $XADR(s^array);   !Assigns 32-bit address of
                              ! array returned by $XADR
```

2. This example uses the $XADR standard function to return the 32-bit address of an INT item to which a standard pointer points, then assigns the address to an extended pointer:

```
INT .EXT ext^ptr;             !Declares extended pointer
INT     .std^ptr := %100000;  !Declares INT standard pointer

@ext^ptr := $XADR(std^ptr);   !Assigns 32-bit address of INT
                              ! item returned by $XADR
```

3. The following example assigns the first byte address in the upper 32K of the current data segment to an extended pointer:

```
INT .EXT top^ptr;            !Declares extended pointer

@top^ptr := %200000D;        !Assigns first byte address in
                             ! upper 32K area to extended
                             ! pointer
```

4. This example shows how to build your own address in the user code space. The $DBLL standard function returns an INT(32) value from two INT values, the first becoming the upper 16 bits and the second becoming the lower 16 bits. After the assignment, the pointer can point to the fourteenth byte or seventh word of the code space.

```
INT .EXT ext^ptr;            !Declares extended pointer

@ext^ptr := ($DBLL (2, 7)) '<<' 1;
                             !Assigns user-code-segment address
```

DATA ACCESS THROUGH POINTERS

To access the data to which a pointer points, you simply use its name in statements. You can use standard and extended pointers in any statement, except that an extended pointer cannot be the object of a SCAN or RSCAN statement.

Examples of Data Access Through Pointers

1. This example assigns a new value to the item to which a standard pointer points:

```
INT .addr[0:2] := [1,2,3];  !Declares and initializes array
INT .sp := @addr[0];        !Declares and initializes standard
                            ! pointer with address of "addr[0]"
sp := 4;                    !Assigns 4 to "addr[0]"
```

2. This example assigns a value to the location to which an extended pointer points:

```
INT .EXT ep := %200000D;    !Declares and initializes extended
                            ! pointer with address of first
                            ! word in upper 32K area
ep := 5;                    !Assigns 5 to location %200000D
```

3. The following example shows data being accessed through extended pointers in various statements:

```
INT var^a;                              !Declares variables
INT var^b;
INT .ptr;                               !Declares standard pointer
INT .EXT ptr^a;                         !Declares extended pointers
INT .EXT ptr^b;

var^a := ptr^a;                         !Assignment statements
ptr^a := var^a;
ptr^a := ptr^b;
var^a ':=' ptr^a FOR 10;                !Move statements
ptr^a ':=' var^a FOR 10;
ptr^a ':=' ptr^b FOR 10;
IF var^a = ptr^a FOR 10 THEN . . .      !IF-THEN-ELSE statement

SCAN ptr^a WHILE " ";                   !Invalid SCAN statement
var^a ':=' var^b FOR 10 -> @ptr^a;      !Invalid move; see Note 1
ptr^a ':=' var^b FOR 10 -> @ptr;        !Invalid move; see Note 2
```

Note 1. Since "var^a" and "var^b" have 16-bit addresses, the variable to the right of the -> symbol must also be a 16-bit variable.

Note 2. Since "ptr^a" is a 32-bit extended address, the variable to the right of the -> symbol must also be a 32-bit variable; "ptr" is a 16-bit variable.

ADDRESSES OF OTHER ITEMS

In addition to its use with pointers, the @ symbol lets you obtain the addresses of other items.

The syntax for getting addresses of other items is:

```
@<item-name>
```

@

is the symbol for removing indirection. It means get the address of <item-name>.

<item-name>

is the name of an existing variable, label, subprocedure, or procedure.

Table 10-1 summarizes the address yielded by the @ symbol for each item. This table does not apply to pointers.

Table 10-1. Addresses of Items

Item	16-Bit Value
STRING variable	Byte address of variable
Non-STRING variable	Word address of variable
Structure	Word address of structure occurrence
Substructure	Byte address of substructure occurrence
Label	Word address of label in current user code segment
Subprocedure	Word address of entry point in current user code segment
Procedure	Procedure entry point (PEP) number of the procedure LORed with code space information

Examples

1. This example returns the address of a simple variable:

```
INT a;                            !Declares simple variables
INT b;

b := @a;                          !Returns address of "a"
```

2. This example returns the addresses of array elements:

```
INT .m[0:2]                       !Declares array
INT n1;                           !Declares simple variable
INT n2;                           !Declares simple variable

n1 := @m[0];                      !Returns address of "m[0]"
n2 := @m[1];                      !Returns address of "m[1]"
```

3. This example returns the address of a label:

```
LABEL loop;                       !Declares label
INT address;                      !Declares variable

loop : <statement>;               !Labels statement
   .
   .
address := @loop;                 !Returns label address
```

4. The following example returns the PEP table number in bits 7 through 15 of the address. (For more information on the PEP, see the System Description Manual for your system).

```
PROC main^proc MAIN;              !Declares procedure
BEGIN
  INT pepnum;                     !Declares variable
   .
   .
  pepnum := @main^proc.<7:15>;    !Returns PEP information
   .
   .
 END;
```

TEMPORARY POINTERS

A temporary pointer is a direct INT variable whose contents become the address of another data item.

The syntax for specifying a temporary pointer is:

```
.<direct-int-variable>
```

. (period)

is the indirection symbol for standard addressing. It causes the contents of <direct-int-variable> to be used as a word address.

<direct-int-variable>

is a previously declared direct variable of type INT located in the current data segment.

You can specify a temporary pointer in any INT arithmetic expression.

Referencing the variable without the period accesses the variable. Using the period accesses the item to which the variable points.

Example

In this example, the direct variable "a" becomes a temporary pointer:

```
INT b;                  !Declares "b"
INT a := 5;             !Declares "a" and initializes it with 5

b := a;                 !"b" equals "a" now
.a := 0;                !Temporary pointer; assigns 0 to G[5]
b := .a;                !"b" equals 0 now
```

SECTION 11

STRUCTURES

This section describes structure and structure pointer declarations, storage, and data access.

A structure is a collectively stored set of data items that you can access individually or as a group. Structures can contain simple variables, arrays, and other structures (called substructures).

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.

They can contain multidimensional arrays, each consisting of any number of arrays.

Global or local structures can be direct or indirect. Sublocal structures must be direct. Since the primary storage areas are limited in size, you should declare indirect global and local structures. TAL manages indirection for you by providing a standard pointer and initializing it with the location of the structure data. You access structure items by referencing the qualified structure name.

A structure pointer associates a previously declared structure with the location to which the pointer points. You manage indirection by declaring a standard or extended structure pointer and initializing it with a value. You access structure items by referencing the qualified pointer name.

STRUCTURE FORMS

A structure declaration can have one of three forms:

- Definition--This form declares a structure, describes its body, and allocates storage for it.
- Template--This form declares a structure template. It describes the structure body but allocates no storage for it.
- Referral--This form declares a structure and allocates storage for it. It describes the structure body by referencing a previously declared structure or structure pointer.

The structure body contains declarations for arrays, simple variables, substructures, FILLER bytes, or redefinitions.

STRUCTURE DECLARATIONS

Definition, template, and referral structures and structure body entities are described separately on the following pages. This discussion is for global, local, and sublocal data, not for formal parameters.

Definition Structure Declaration

The definition form declares a structure, describes its body, and allocates storage for it.

The syntax for the definition structure declaration is:

```
STRUCT [ . ] <identifier>

                [ "[" <lower-bound> : <upper-bound> "]" ] ;

<structure-body>
```

. (a period)

   is the indirection symbol for standard addressing.

<identifier>

   is the name of the structure.

<lower-bound>

   is a constant expression in the range -32768 through 32767 that specifies the first structure occurrence for which to allocate storage. The default value is 0 (one occurrence). Each occurrence is one copy of the structure.

<upper-bound>

   is a constant expression in the range -32768 through 32767 that specifies the last structure occurrence for which to allocate storage. The default value is 0 (one occurrence).

<structure-body>

   contains declarations for data, substructures, FILLER bytes, or redefinitions, as described under "Structure Body" in this section.

   The size of one occurrence of a structure must not exceed 32,767 bytes.

The following example declares 50 occurrences of a definition structure:

```
   STRUCT .inventory1[0:49];
     BEGIN
     INT item;
     INT price;
     INT quantity;
     END;
```

Structure Storage Allocation

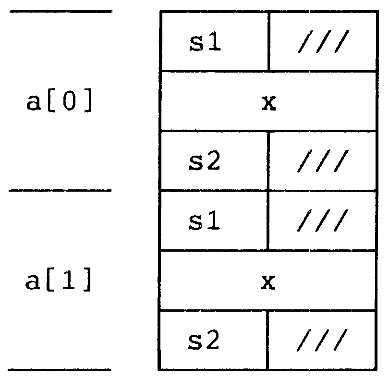
For direct structures, TAL allocates storage for each occurrence of the structure in a primary global or local area or in the sublocal area. Sublocal structures must be direct.

For indirect structures, TAL allocates primary global or local storage for a 16-bit standard pointer. It then allocates storage in the corresponding secondary area for each structure occurrence.

Structures are word addressed. That is, TAL starts each structure occurrence on a word boundary. Within each structure occurrence, TAL allocates storage for each item and adds a pad byte as needed to fill an unused byte caused by the need for INT structure items to be aligned on word boundaries.

The following example shows storage allocation for two occurrences of a structure (slants denote a pad byte):

```
   STRUCT a[0:1];
     BEGIN
     STRING s1;
     INT x;
     STRING s2;
     END
```



Template Structure Declaration

The template form describes a structure body but allocates no space for it. The syntax for the template structure declaration is:

```
STRUCT <identifier> (*) ;

<structure-body>
```

`<identifier>`

is the name of the template structure.

`(*)`

is the symbol that identifies the structure as a template.

`<structure-body>`

contains declarations for data, substructures, FILLER bytes, or redefinitions, as described under "Structure Body" in this section.

Templates have meaning only when you reference them in subsequent referral structure declarations or structure pointer declarations. The subsequent declarations allocate space for a structure body identical to that of the template.

TAL ignores the indirection symbol if specified.

An example of a template structure declaration is:

```
STRUCT inventory2 (*);
  BEGIN
  INT item;
  INT price;
  END;
```

Referral Structure Declaration

The referral form declares and allocates storage for a structure described in a previously declared structure or structure pointer. The referral form has no body of its own.

The syntax for the referral structure declaration is:

```
STRUCT [ . ] <identifier> ( <referral> )
                        [ "[" <lower-bound> : <upper-bound> "]" ] ;
```

. (a period)

is the indirection symbol for standard addressing.

<identifier>

is the name of the new structure.

<referral>

is the name of a previously declared structure or structure pointer.

<lower-bound>

is a constant expression in the range -32768 through 32767 that specifies the first structure occurrence for which to allocate storage. The default value is 0 (one occurrence). Each occurrence is one copy of the structure.

<upper-bound>

is a constant expression in the range -32768 through 32767 that specifies the last structure occurrence for which to allocate storage. The default value is 0 (one occurrence).

TAL allocates storage for a structure with the addressing mode and number of occurrences specified in the referral declaration, not those specified in the previous declaration. TAL uses only the body of the previous declaration for the new structure.

The following example declares a template structure and a referral structure that references the template structure:

```
STRUCT record (*);                !Declares template structure
  BEGIN
  INT name;
  INT addr;
  INT acct;
  END;

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

STRUCTURE BODY

The structure body is a BEGIN-END construct that can contain declarations for:

- Data Items--Arrays and simple variables
- Substructures--Structures nested within the primary structure
- FILLER Bytes--Place-holding bytes
- Redefinitions--Items that redefine data items or substructures

Data Declarations

Syntax for data declarations is described in Section 8, "Simple Variables," and Section 9, "Arrays," with the following differences:

- You cannot initialize any variables.
- You cannot declare read-only arrays.
- You cannot use indirection.
- You can specify array bounds of [0:-1].

  Bounds of [0:-1] place the identifier in the symbol table so you can reference it, but allocates no storage for the array.

Storage Allocation

TAL allocates storage for data within structures by aligning word-addressed items on word boundaries and STRING items on byte boundaries, adding a pad byte where needed to fill an unused byte.

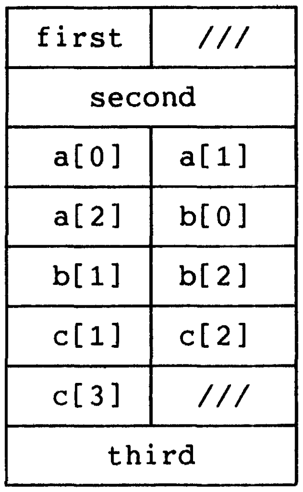
Examples of Data Declarations

1. The following example shows data declarations in a structure body:

```
LITERAL len = 100;          !Number of array elements

STRUCT .str1;
  BEGIN                     !Begins structure body
  STRING s[0:len-1];        !Declares array
  INT index;                !Declares simple variable
  INT count;                !Declares simple variable
  END;                      !Ends structure body
```

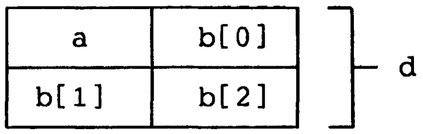
2. The following example shows storage allocation for data inside a structure (slants denote a pad byte):

```
STRUCT .padding;

  BEGIN

  STRING first;

  INT second;

  STRING a[0:2];

  STRING b[0:2];

  STRING c[1:3];

  INT third;

  END;
```



3. This example declares an array with bounds of [0:-1], which allows access to subsequent structure items using the array identifier:

```
STRUCT x;
  BEGIN
  INT(32) d[0:-1];
  STRING a;
  STRING b[0:2];
  END;

x.d := 0D;        !Sets "a" and "b[0:2]" to 0
```



Substructure Declaration

A substructure is a structure embedded within another structure or substructure.

The syntax for the substructure declaration is the same as the syntax previously defined under "Definition Structure Declaration," except that you cannot use the indirection symbol.

Substructures differ from structures as follows:

- Substructures must be directly addressed.
- You can nest substructures to any practical level; that is, you can declare a substructure within a substructure within a substructure, and so on.
- You can specify lower and upper bounds of [0:-1]. This places the substructure in the symbol table but allocates no storage; the substructure is addressable but uses no memory.
- Substructures are byte addressed. Structures are word addressed.
- TAL allocates storage for substructures starting on byte boundaries, if possible. Structures always start on word boundaries.

Examples of Substructure Declarations

1. This example constructs a two-dimensional array. It consists of two occurrences of a structure, each of which contains 50 occurrences of a substructure:

```
LITERAL last = 49;                !Last item in inventory

STRUCT .warehouse[0:1];           !Two warehouses
  BEGIN
  STRUCT inventory [0:last];      !Fifty items in each warehouse
    BEGIN
    INT item^number;
    INT price;
    INT on^hand;
    END;
  END;
```

2. The following example shows substructures used for the Command Interpreter start-up message:

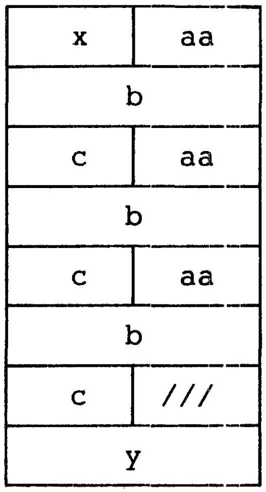
```
STRUCT .startup;
  BEGIN
  INT msgcode;
  STRUCT default;           !Substructure declaration
    BEGIN
    INT volume[0:3];
    INT subvol[0:3];
    END;
  STRUCT infile;            !Substructure declaration
    BEGIN
    INT volume[0:3];
    INT subvol[0:3];
    INT  fname[0:3];
    END;
  STRUCT outfile;           !Substructure declaration
    BEGIN
    INT volume[0:3];
    INT subvol[0:3];
    INT  fname[0:3];
    END;
  STRING param[0:131];      !Program parameters
  END;
```

3. The following example shows nested substructure declarations:

```
STRUCT .mil^branch;
  BEGIN
  STRUCT div[0:3];                 !Substructure
    BEGIN
    STRUCT reg[0:3];               !Nested substructure
      BEGIN
      STRUCT batt[0:1];            !Nested substructure
        BEGIN
        STRUCT comp[0:3];          !Nested substructure
          BEGIN
          STRUCT plat[0:3];        !Nested substructure
            BEGIN
            INT infantry;
            END;  !Of "plat"
          END;  !Of "comp"
        END;  !Of "batt"
      END;  !Of "reg"
    END;  !Of "div"
  END;  !Of "mil^branch"
```

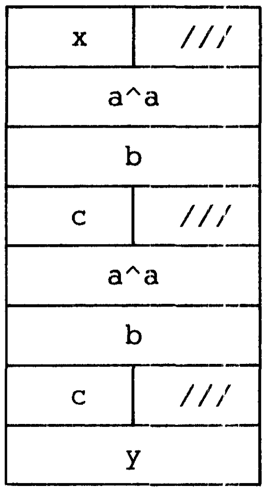
4. This example shows storage for substructure occurrences that begin on byte boundaries because the substructure not only follows a STRING item ("x") and but also starts with a STRING item ("aa"):

```
STRUCT s;
  BEGIN
  STRING x;
  STRUCT sub[0:2];    !Substructure
    BEGIN             ! declaration
    STRING aa;
    INT b;
    STRING c;
    END;
  INT y;
  END;
```



5. This example shows storage for substructure occurrences that begin on word boundaries because the substructure starts with an INT item ("a^a"):

```
STRUCT t1;
  BEGIN
  STRING x;
  STRUCT t2 [0:1]; !Substructure
    BEGIN          ! declaration
    INT a^a;
    INT b;
    STRING c;
    END;           !Of substructure
  INT y;
  END;             !Of structure
```



FILLER Declaration

A FILLER byte provides a place holder for structure data or space that your program does not use.

The syntax for the FILLER declaration is:

```
FILLER <constant-expression> ;


<constant-expression>

   is a positive INT constant value that specifies the number of
   bytes of FILLER.
```

The word FILLER is a reserved word only within the scope of a structure declaration. You cannot reference FILLER byte locations.

FILLER declarations contribute to clearer source code. For example, you can use FILLER bytes:

- To define data that appears in a structure but is not used by your program
- To document word-alignment pad bytes inserted by TAL
- To provide place holders for unused space

The following example shows FILLER declarations:

```
LITERAL last = 11;          !Last occurrence

STRUCT .filler[1:last];
  BEGIN
  STRING byte[0:2];
  FILLER 1;                 !Documents word-alignment pad byte
  INT word1;
  INT word2;
  INT(32) integer32;
  FILLER 30;                !Place holder for unused space
  END;
```

For a FILLER example defining unused data, see "Substructure Redefinition" (example 4) in this section.

Redefinitions

A redefinition declares a new name and description for an existing data item or substructure within a structure.

Data Item Redefinition

The syntax for the data item redefinition declaration is:

```
<type> <identifier> [ "[" <lower-bound> : <upper-bound> "]" ]

                                        = <previous-identifier> ;
```

<type>

is one of the following data types:

```
STRING
INT
INT(32)
FIXED [ ( <fpoint> ) ]
REAL
REAL(64)
```

<identifier>

is the name of the new data item that redefines an existing data item in the structure. A data item is a simple variable or an array.

<lower-bound>

is an INT constant expression in the range -32768 through 32767 that defines the first array element. The default value is 0 (one element).

⟶

<upper-bound>

is an INT constant expression in the range -32768 through 32767 that defines the last array element. The default value is 0 (one element).

<previous-identifier>

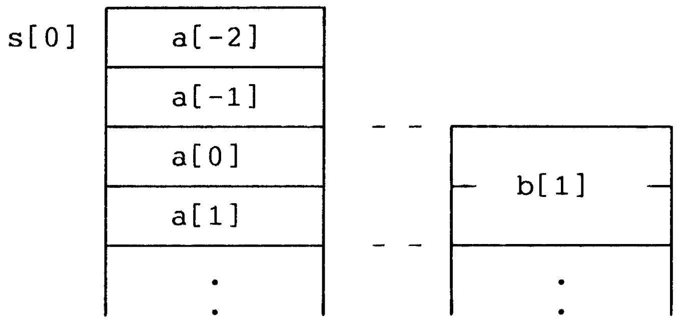
is the name of a data item previously declared in the same structure. You cannot specify an index with this name.

When you redefine data items, the following rules apply:

- The new item must be on the same level as the previous item.
- The new item must have the same, or shorter, length as the previous item.
- You can redefine arrays contained in structures and substructures. For arrays outside structures, see Section 12, "Equivalenced Variables."
- The redefinition must start at element [0] of the previous identifier.
- You cannot redefine the data type of a STRING item that begins on an odd-byte address.

The following example redefines an INT array as an INT(32) array. The redefinition begins at "a[0]":

```
STRUCT .s;
  BEGIN
  INT a[-2:3];
  INT(32) b[1:2] = a;
  END;
```



Substructure Redefinition

The syntax for the substructure redefinition declaration is:

```
STRUCT <identifier> [ "[" <lower-bound> : <upper-bound> "]" ]

                                  =  <previous-identifier> ;

<structure-body>
```

<identifier>

is the name of the new substructure that redefines a previously declared substructure.

<lower-bound>

is a constant expression in the range -32768 through 32767 that defines the first substructure occurrence. The default value is 0 (one occurrence). Each occurrence is one copy of the substructure.

<upper-bound>

is a constant expression in the range -32768 through 32767 that defines the last substructure occurrence. The default value is 0 (one occurrence).

<previous-identifier>

is the name of a substructure that was previously declared in the same structure. No index is allowed with this name.

<structure-body>

contains declarations for data, substructures, FILLER bytes, or redefinitions.

If you do not specify lower and upper bounds, or if the upper bound is equal to 0, the new substructure and the previous substructure occupy the same space and have the same offset from the beginning of the structure.

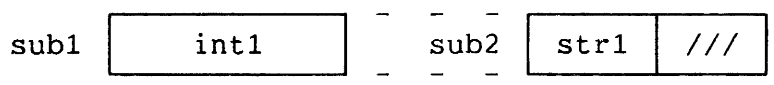
Rules for redefining substructures are:

- The new substructure must be on the same level as the previous substructure.
- The new substructure should have the same, or shorter, length as the previous substructure.
- Both substructures must have the same alignment. If the previous substructure starts on an odd byte, the first data item in the new substructure must be a STRING item.

Examples for redefinition declarations are shown below.

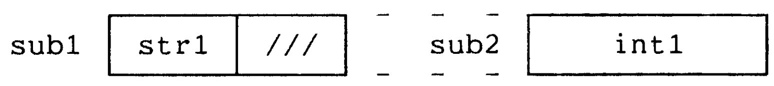
1. In this example, the new substructure is smaller than the previous substructure; the redefinition is proper:

```
STRUCT str1;
  BEGIN
  STRUCT sub1;          !Declares "sub1"
    BEGIN
    INT int1;
    END;
  STRUCT sub2 = sub1;   !Redefines "sub1" as "sub2"
    BEGIN
    STRING str1;
    END;
  END;
```



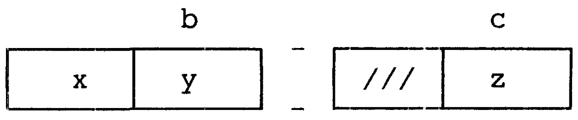
2. In this example, the new substructure is larger than the previous substructure; TAL issues a warning:

```
STRUCT str1;
  BEGIN
  STRUCT sub1;          !Declares "sub1"
    BEGIN
    STRING str1;
    END;
  STRUCT sub2 = sub1;   !Redefines "sub1" as "sub2"
    BEGIN
    INT int1;
    END;
  END;
```



3. In this example, both substructures ("b" and "c") have the same alignment as required. In this case, both begin on an odd-byte boundary:

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



4. This example redefines the format of a substructure record:

```
STRUCT .name^record;
  BEGIN
  STRUCT whole^name;                !Declares "whole^name"
    BEGIN
    STRING first^name[0:10];
    STRING middle^name[0:10];
    STRING last^name[0:15];
    END;
  STRUCT initials = whole^name;     !Redefines "whole^name" as
    BEGIN                           ! "initials"
    STRING first^initial;
    FILLER 10;
    STRING middle^initial;
    FILLER 10;
    STRING last^initial;
    FILLER 15;
    END;
  END;
```

ACCESSING STRUCTURED DATA

To access a definition or referral structure (whether direct or indirect), you specify its identifier in a statement. For a move, SCAN, or RSCAN statement or a reference parameter, specify the unqualified structure or substructure identifier.

For an assignment statement, specify the fully qualified identifier of the structure item, using the following form, with or without indexes:

```
<struct-name> [ [.<substruct-name>]...] .<item-name>
```

All indexes must be signed INT arithmetic expressions. An example of an indexed structure identifier is:

```
record[i].table[2].item[x]
```

Examples of Accessing Structured Data

1. The following example shows how nesting affects the qualification level. In the declaration on the left, the full qualification for "item" is "outer.inner^3.item." In the declaration on the right, it is "outer.inner^1.inner^2.inner^3.item."

```
STRUCT .outer                 STRUCT .outer;
  BEGIN                         BEGIN
  STRUCT inner^1;               STRUCT inner^1;
    BEGIN                         BEGIN
    .                             STRUCT inner^2;
    END;                            BEGIN
  STRUCT inner^2;                   STRUCT inner^3;
    BEGIN                             BEGIN
    .                                 INT item;
    END;                                  .
  STRUCT inner^3;                         .
    BEGIN                             END;
    INT item;                       END;
    END;                          END;
  END;                          END;
```

2. The following example shows how to access an item in a definition structure:

```
STRUCT .d;            !Declares definition structure "d"
  BEGIN
  INT a;
  STRING b;
  REAL c[0:2];
  END;

d.a := 2;             !Assigns value to "a" in structure "d"
```

3. The following example shows how to access an item in a referral structure that references a template structure:

```
STRUCT t (*);         !Declares template structure "t"
  BEGIN
  INT a;
  STRING b;
  REAL c[0:2];
  END;

STRUCT .r (t);        !Declares referral structure "r"

r.a := 2;             !Assigns value to "a" in structure "r"
```

4. These code fragments access a three-dimensional array structure:

```
INT s;                  !Index for store sales
INT d;                  !Index for department sales
INT c;                  !Index for each clerk's sales

STRUCT .chain;
  BEGIN
  INT(32) chain^tot;
  STRUCT store[0:2];
    BEGIN
    INT(32) store^tot;
    STRUCT dept[0:2];
      BEGIN
      INT(32) dept^tot;
      STRUCT clerk[0:1];
        BEGIN
        INT clk;
        INT amt;
        END; !Ends "clerk"
      END; !Ends "dept"
    END; !Ends "store"
  END; !Ends "chain"

!The following code updates each clerk's records using the
! clerk number and amount entered from terminal:

FOR s := 0 TO 2 DO
  FOR d := 0 TO 2 DO
    FOR c := 0 TO 1 DO
    IF chain.store[s].dept[d].clerk[c].clk = entered^clk^no
    THEN chain.store[s].dept[d].clerk[c].amt  :=  entered^amt;

!The following code updates department, store, and chain
! totals:

FOR s := 0 TO 2 DO
BEGIN
  FOR d := 0 TO 2 DO
  BEGIN
    FOR c := 0 TO 1 DO
      chain.store[s].dept[d].dept^tot :=
                   chain.store[s].dept[d].dept^tot +
                      $DBL(chain.store[s].dept[d].clerk[c].amt);
    chain.store[s].store^tot := chain.store[s].store^tot +
                                chain.store[s].dept[d].dept^tot;
  END;
  chain.chain^tot := chain.chain^tot +
                                      chain.store[s].store^tot;
END;
```

Structure Functions

TAL provides the following standard functions for processing of structured data:

- $LEN--Returns the length in bytes of one occurrence of an item.
- $OFFSET--Returns an item's offset in bytes from the structure base.
- $OCCURS--Returns the number of occurrences of an item.
- $TYPE--Returns the data type of an item.

The following example uses the $OCCURS and $LEN functions to read structured data:

```
INT record^num;

STRUCT emp^data(*);                         !Template structure
  BEGIN
  INT number;
  INT dept;
  STRING ssn[0:11];
  FIXED(2) salary;
  END;

PROC main^proc MAIN;
  BEGIN
   .
   .
  STRUCT .job^data (emp^data) [0:5];     !Referral structure

  FOR record^num := 0 TO $OCCURS (job^data) - 1 DO
    CALL READ(discfile,
             job^data[record^num],      !Buffer
             $LEN(job^data),            !Record length
             num^read);
              .
              .
  END;
```

For more information on these functions, see Section 17, "Standard Functions."

STRUCTURE POINTER DECLARATION

The structure pointer declaration associates a structure with the memory location to which the pointer points. Therefore, you can access the location to which the pointer points by referencing a structure item.

The syntax for the structure pointer declaration is:

```
{ INT    } { .    } <identifier> ( <referral> )
{ STRING } { .EXT }

                                          [ := <initialization> ]

        [ , { .    } <identifier> ( <referral> )
            { .EXT }

                                     [ := <initialization> ] ] ... ;
```

INT

indicates the pointer contains a word address.

STRING

indicates the pointer contains a byte address.

. (period)

is the indirection symbol for standard addressing.

.EXT

is the indirection symbol for extended addressing. It is a reserved word only when followed by <identifier>. At least one space must precede and follow the symbol.

⟶

<identifier>

is the name of the structure pointer.

<referral>

is the name of a previously declared structure or structure pointer.

<initialization>

is a constant expression (global scope) or an arithmetic expression (local or sublocal scope), as follows:

- If <identifier> is a standard STRING pointer, use a 16-bit byte address in the lower 32K area.
- If <identifier> is a standard INT pointer, use a 16-bit word address in the 64K area.
- If <identifier> is an extended pointer of any type, use a 32-bit byte address. For details, see Appendix A.

Before referencing a structure pointer, be sure you have assigned a value to it, either in the declaration or in a subsequent statement (see "Structure Pointer Assignments" in this section). References to uninitialized pointers cause undefinable program execution.

Standard STRING structure pointers can access STRING structure items only. Standard INT pointers and extended STRING or INT pointers can access structure items of any type. However, if an INT pointer contains an address in the upper 32K area, you cannot access STRING items with that pointer.

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

Examples of Structure Pointer Declarations

1. This example declares a template structure and a structure pointer that references the template and initializes the pointer with a location in the upper 32K area:

```
STRUCT names (*);                  !Declares template structure
  BEGIN
  INT filename[0:11];
  END;

INT .struc^ptr (names) := %100000; !Declares structure pointer
```

2. This example declares an extended structure pointer that references the structure pointer declared in Example 1 and initializes it with a location in the upper 32K area:

```
STRING .EXT ex^strc^ptr (struc^ptr) := %200000D;
```

Storage Allocation

TAL allocates primary storage for the structure pointer. A standard pointer gets one word of primary storage; an extended pointer gets a doubleword. You must allocate the memory location to which the pointer points.

TAL emits more efficient machine code if it can store extended pointers between G[0] and G[63] or between L[0] and L[63]. Thus, extended pointers should precede other global or local declarations.

Structure Pointer Assignments

The syntax for a structure pointer assignment is:

```
@<pointer-name> := <expression> ;
```

@

is the symbol for removing indirection. It means get the contents of <pointer-name>, not the item to which it points.

<pointer-name>

is the name of a previously declared standard or extended structure pointer.

<expression>

is an arithmetic expression:

- If <pointer-name> is a standard STRING structure pointer, use a 16-bit byte address in the lower 32K area.
- If <pointer-name> is a standard INT structure pointer, use a 16-bit word address in the 64K area.
- If <pointer-name> is an extended structure pointer of any type, use a 32-bit byte address. For details, see Appendix A.

The following example assigns the address of the third occurrence of a structure to a standard structure pointer:

```
STRUCT .struc[0:2];           !Declares structure "struc"
  BEGIN
  INT i;
  STRING s;
  END;

INT .str^ptr (struc);         !Declares structure pointer

@str^ptr := @struc[2];        !Assigns address of "struc[2]" to
                              ! structure pointer
```

Accessing Data Using Structure Pointers

To access a structure item, you reference the pointer name in a statement. In move, SCAN, or RSCAN statements or reference parameters, specify the unqualified pointer name. Extended pointers cannot be the object of SCAN or RSCAN operations.

In assignment statements, specify the fully qualified pointer name using the following form, with or without indexes:

```
<pointer-name> [ [.<substruct-name>]...] .<item-name>
```

An example of a qualified structure pointer name is:

```
struc^ptr.records.customer.name
```

For both standard and extended structure pointers, the index must be a signed INT arithmetic expression.

Standard Structure Pointer Accessing Examples

1. The following example uses standard structure pointers to access INT structure items in the upper 32K of data space:

```
?DATAPAGES 64                         !Gets maximum data stack

STRUCT names (*);                     !Declares template structure
  BEGIN
  INT filename[0:11];
  END;

INT .name^ptr1(names) := %100000;     !Points to beginning of
                                      ! upper 32K area
INT .name^ptr2(names) := %110000;     !Points to upper half of
                                      ! upper 32K area
PROC main^proc MAIN;
     .
     .
  name^ptr1.filename[0] ':=' "$SYSTEM SYSTEM  EDIT    ";
  name^ptr2.filename[0] ':=' "$DATA   OFFICE  PRODUCT ";
     .
     .                                !Accesses structure items
END;
```

2. In the following example, a structure pointer points into an existing structure:

```
STRUCT .data2[0:2];                   !Declares definition structure
  BEGIN
  INT    i1;
  INT    i2;
  INT    i3;
  STRING s1;
  END;

INT .pnt2 (data2) := @data2[1];       !Declares and initializes
                                      ! structure pointer

pnt2.i2 := %1414;
pnt2.s1 := %3;                        !Accesses structure items
```

3. In the following example, a structure pointer points to the beginning of a buffer, thereby imposing the structure on top of the buffer:

```
INT .recbuf[0:7] :=[1,%22,%23,%24,%25,"ABCDE"];   !Buffer
INT num1;

STRUCT data (*);                  !Declares template structure
  BEGIN
  INT    code1;
  INT    i1[0:3];
  STRING s1[0:4];
  END;

INT .pnt2 (data) := @recbuf;      !Declares and initializes
                                  ! structure pointer

num1 := pnt2.i1[2];               !Accesses structure item
```

4. In the following example, a STRING standard structure pointer accesses a STRING item. You must convert the word address of the structure to a byte address before assigning it to the pointer:

```
STRUCT .data[0:1];               !Declares definition structure
  BEGIN
  STRING s1;
  STRING s2;
  STRING s3;
  END;

STRING .pnt (data) := @data[1] '<<' 1;      !Declares and
                                ! initializes structure pointer

pnt.s2 := %4;                   !Accesses structure item
```

Extended Structure Pointer Accessing Examples

1. In this example, extended INT structure pointers access byte-addressed variables. This example assumes previous allocation of an extended segment as described in Appendix A.

```
STRUCT name^rec (*);                         !Declares template
  BEGIN
  STRING name[0:25];
  END;

INT .EXT ext^ptr(name^rec) := %200000D; !Points to beginning of
                                        ! upper 32K area

INT .EXT ext^seg(name^rec) := %2000000D; !Points to beginning
                                         ! of extended segment

ext^ptr.name[0] ':=' "Anastasia L. Malatorious";
ext^seg.name[0] ':=' "Octavious Q. Pumpernickle";
```

Additional examples for extended structure pointers are given at the end of Appendix A.

SECTION 12

EQUIVALENCED VARIABLES

Equivalencing lets you use more than one name to describe a location in a primary storage area. Variables made equivalent to previously allocated locations do not allocate additional memory space.

The variables that represent a location can have different data types and byte or word addressing attributes. For example, you can reference an INT(32) variable as two separate words or four separate bytes, or you can use an INT array and a STRING array to access the same buffer.

This section describes how to declare and access:

- Equivalenced variables--Variables made equivalent to a previously declared variable.
- Base-address equivalenced variables--Variables made equivalent to a global, local, or top-of-stack address base.

The new variable can be a simple variable, pointer, structure, or structure pointer. The previous variable can be a simple variable, a direct array element, pointer, structure, structure pointer, or another equivalenced variable that you previously declared as described in Sections 8 through 12.

For equivalenced items within structures, see "Redefinitions" in Section 11.

For equivalenced system global variables, see Section 18, "Privileged Procedures."

EQUIVALENCED VARIABLE DECLARATION

The equivalenced variable declaration associates a new variable with a previously declared variable.

Equivalenced variables (simple variables, pointers, and structure pointers) are described first, followed by equivalenced structures.

The syntax for the equivalenced variable declaration is:

```
          { { .EXT } { <structure-pointer> ( <referral> ) } }
          { { .    } { <pointer>                          } }
   <type> {                                                 }
          { <simple-variable>                               }

                      = <previous-identifier> [ "[" <index> "]" ]
                                              [ {+|-} <offset>  ]


          { { .EXT } { <structure-pointer> ( <referral> ) } }
          { { .    } { <pointer>                          } }
      [ , {                                                 }
          { <simple-variable>                               }

                 = <previous-identifier> [ "[" <index> "]" ]
                                         [ {+|-} <offset>  ] ] ... ;
```

<type>

For <structure-pointer>, <type> must be STRING or INT.
For <simple-variable> or <pointer>, <type> is any data type.

. (period)

is the indirection symbol for standard addressing.

.EXT

is the indirection symbol for extended addressing.

⟶

<structure-pointer>

is the identifier of a structure pointer to be made equivalent to <previous-identifier>.

<pointer>

is the identifier of a pointer to be made equivalent to <previous-identifier>.

<simple-variable>

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

<referral>

is the identifier of a previously declared structure or structure pointer.

<previous-identifier>

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

<index>

is an INT constant that specifies a number of elements of the type declared. <index> is permitted only with direct variables. <index> must end on a word boundary.

<offset>

is an INT constant that specifies a word offset. <offset> is permitted with direct or indirect variables. For indirect variables, the offset is from the location of the pointer, not from the location of the data pointed to.

The syntax for the equivalenced structure declaration is:

```
STRUCT [ . ] <structure> [ ( <referral> ) ]

                   = <previous-identifier> [ "[" <index> "]" ]
                                           [ {+|-} <offset>  ] ;


[ <structure-body> ]
```

. (period)

   is the indirection symbol for standard addressing.

<structure>

   is the identifier of a definition or referral structure to be made equivalent to <previous-identifier>.

<referral>

   is the identifier of a previously declared structure or structure pointer. Its presence means <structure> is a referral structure and <structure-body> cannot be specified.

<previous-identifier>

   is the name of a previously declared simple variable, direct array element, structure, structure pointer, or equivalenced variable.

<index>

   is an INT constant that specifies a number of elements of the type declared. <index> is permitted only with direct variables. <index> must end on a word boundary.

⟶

<offset>

is an INT constant that specifies a word offset. <offset> is permitted with direct or indirect variables. For indirect variables, the offset is from the location of the pointer, not from the location of the data pointed to.

<structure-body>

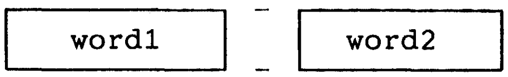
is a BEGIN-END construct that contains declarations as described in Section 11. Its presence means <structure> is a definition structure and <referral> cannot be specified.

Examples of Equivalenced Declarations

The leftmost box in each diagram represents the previously declared variable to which the new variable is made equivalent.

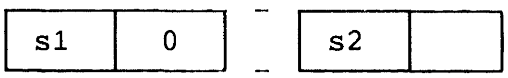
1. This example makes an INT variable equivalent to a previous INT variable:

```
INT word1;
INT word2 = word1;
```



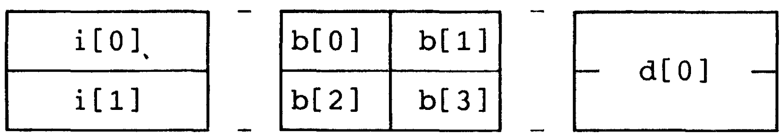
2. This example makes a STRING variable equivalent to another STRING variable:

```
STRING s1 := "A";
STRING s2 = s1;
```



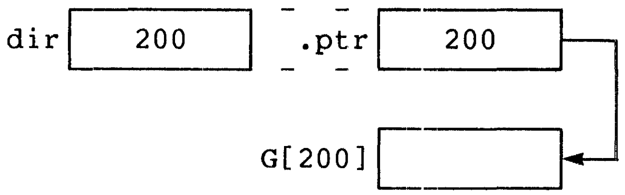
3. This example makes STRING and INT(32) variables equivalent to an INT array:

```
INT i[0:1];
STRING b = i[0];
INT(32) d = b;
```



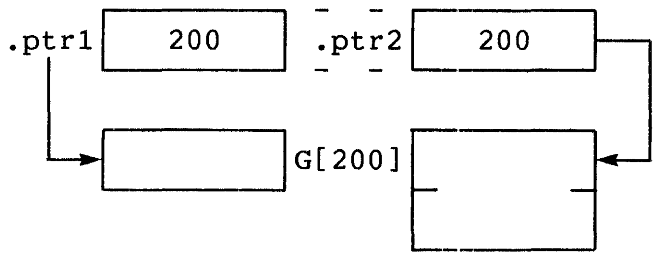
4. This example makes an pointer equivalent to a direct variable:

```
INT dir := 200;
INT .ptr = dir;
```



5. This example makes a word-addressed pointer equivalent to another word-addressed pointer of a different type:

```
INT     .ptr1 := 200;
INT(32) .ptr2 = ptr1;
```

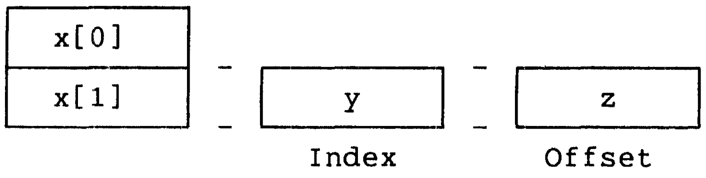


6. This example tries to make a byte-addressed pointer equivalent to a word-addressed pointer. However, the pointers point to different locations, since one pointer contains a word address and the other contains a byte address:

```
INT     .ptr1 := 200;
STRING  .ptr2 = ptr1;
```

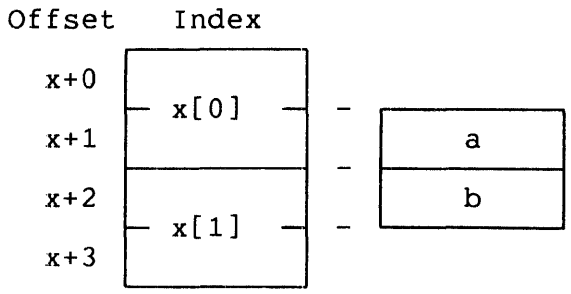
7. For INT variables, indexes and offsets are equivalent:

```
INT x[0:5];
INT y = x[1];  !Index
INT z = x + 1; !Offset
```



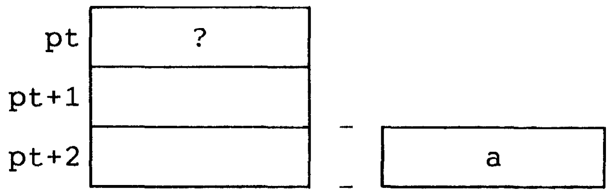
8. For non-INT variables, indexes and offsets are not equivalent:

```
INT(32) x;
INT a = x + 1; !Offset
INT b = x [1]; !Index
```



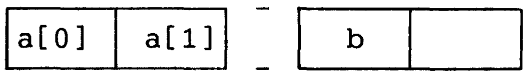
9. You can make a variable equivalent to an offset pointer but not to an indexed pointer:

```
INT .pt;
INT a = pt + 2; !Offset
                ! allowed

INT b = pt [2]; !Index
              ! not allowed
```



10. This example tries to make a STRING variable equivalent to an odd-byte array element. The system ignores the index and and issues a warning.

```
STRING a[0:1];
STRING b = a[1];
```



11. This example tries to make arrays equivalent to other variables, which is not allowed:

```
INT a[0:5];
INT b;
INT c[0:5] = a;               !Not allowed
INT d[0:5] = b;               !Not allowed
```

12. This example makes a referral structure equivalent to a structure pointer:

```
STRUCT record (*);            !Declares template structure
  BEGIN
  INT name[0:14];
  INT address[0:49];
  END;

INT .p (record) := %100000;   !Declares structure pointer

STRUCT .empl (record) = p;    !Makes new structure equivalent
                              ! to structure pointer "p"
```

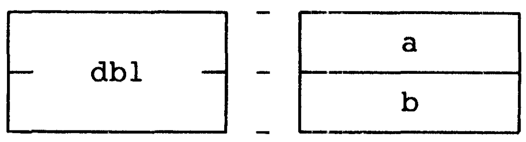
Accessing Equivalenced Variables

You access an equivalenced variable in the same way as any other variable, by specifying its identifier in a statement.

Examples

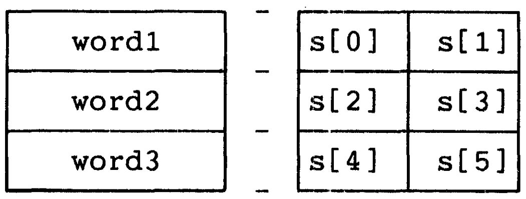
1. This example makes an INT variable equivalent to each word of an INT(32) variable, then accesses the location as an INT variable and as an INT(32) variable:

```
INT(32) dbl;
INT a = dbl,
    b = a + 1;

a := 2 * 2; !Access first
            ! word of "dbl"

dbl := -1D; !Accesses "dbl" as a doubleword
```



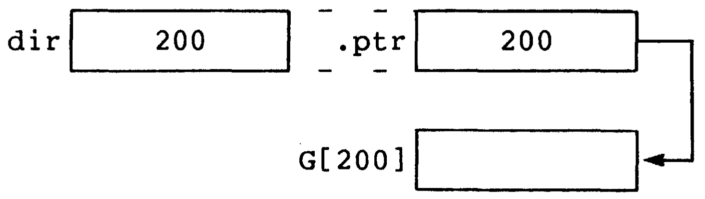
2. This example makes a STRING variable equivalent to the first of three INT variables, then accesses byte items by indexing the STRING variable:

```
INT word1;
INT word2;
INT word3;
STRING s = word1;

s[3] := 0;
IF s[4] > 2 THEN ...;
```



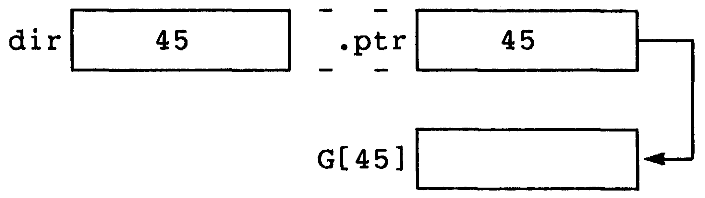
3. These examples make a pointer equivalent to a direct variable, then accesses them in different ways:

```
INT dir := 200;
INT .ptr = dir;
```



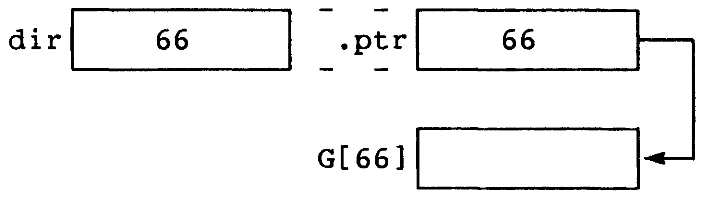
An assignment to the direct variable changes the contents of both the direct variable and the pointer:

```
dir := 45;
```



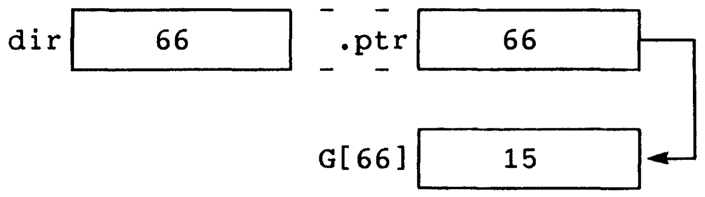
An assignment to the pointer (using the @ symbol) changes the contents of both the direct variable and the pointer:

```
@ptr := 66;
```



An assignment to the pointer (without the @ symbol) changes the contents of only the variable to which the pointer points:

```
ptr := 15;
```



BASE-ADDRESS EQUIVALENCED VARIABLE DECLARATION

Base-address equivalencing lets you declare variables relative to the global, local, and sublocal base addresses.

Equivalenced variables (simple variables, pointers, and structure pointers) are described first, followed by equivalenced structures.

The syntax for the base-address equivalenced variable declaration is:

```
         { { .EXT } { <structure-pointer> ( <referral> ) } }
         { { .    } { <pointer>                        } }
  <type> {                                               }
         { <simple-variable>                             }

                            = <base-address>  [ "[" <index> "]" ]
                                              [ {+|-} <offset>  ]

         { { .EXT } { <structure-pointer> ( <referral> ) } }
         { { .    } { <pointer>                        } }
     [ , {                                               }
         { <simple-variable>                             }

                       = <base-address> [ "[" <index> "]" ]
                                        [ {+|-} <offset>  ] ] ... ;
```

<type>

For <structure-pointer>, <type> must be STRING or INT.
For <simple-variable> or <pointer>, <type> is any data type.

. (period)

is the indirection symbol for standard addressing.

.EXT

is the indirection symbol for extended addressing.

⟶

<structure-pointer>

is the identifier of a structure pointer to be made equivalent to <base-address>.

<pointer>

is the identifier of a pointer to be made equivalent to <base-address>.

<simple-variable>

is the identifier of a simple variable to be made equivalent to <base-address>.

<referral>

is the identifier of a previously declared structure or structure pointer.

<base-address>

is one of:

'G'	Global addressing relative to G[0]
'L'	Local addressing relative to L[0]
'S'	Top-of-stack addressing relative to S[0]

<index> and <offset>

are equivalent INT values giving a location in the following ranges:

'G' addressing:	[0:255]
'L' addressing:	[-31:127]
'S' addressing:	[-31:0]

The syntax for the base-address equivalenced structure declaration is:

```
STRUCT [ . ] <structure> [ ( <referral> ) ]

                              = <base-address> [ "[" <index> "]" ]
                                               [ {+|-} <offset>  ] ;


[ <structure-body> ]
```

. (period)

is the indirection symbol for standard addressing.

<structure>

is the identifier of a definition or referral structure to be made equivalent to <base-address>.

<referral>

is the identifier of a previously declared structure or structure pointer. Its presence means <structure> is a referral structure and <structure-body> cannot be specified.

<base-address>

is one of:

'G'	Global addressing relative to G[0]
'L'	Local addressing relative to L[0]
'S'	Top-of-stack addressing relative to S[0]

<index> and <offset>

are equivalent INT values giving a location in the following ranges:

'G' addressing:	[0:255]
'L' addressing:	[-31:127]
'S' addressing:	[-31:0]

⟶

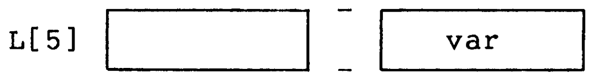
<structure-body>

is a BEGIN-END construct that contains declarations as described in Section 11. Its presence means <structure> is a definition structure and <referral> cannot be specified.

Example

1. This example makes an INT simple variable equivalent to 'L' relative addressing:

```
INT var = 'L'[5];
```



For another example of base-address equivalencing, see the ARMTRAP procedure in the System Procedure Calls Reference Manual.

SECTION 13

EXPRESSIONS

This section gives information about expressions:

- Operators--Arithmetic and conditional (relational and boolean)
- Precedence of Operators--The order in which the system evaluates operators in an expression
- Arithmetic Expressions--General form, assignment form, CASE form, IF-THEN-ELSE form
- Conditional Expressions--General form and group comparison form

An expression is a combination of operands and operators that make up an arithmetic or conditional expression. The operands can be data or constants. The operators specify an arithmetic or conditional operation on the operands. Expressions can be type INT, INT(32), FIXED, REAL, or REAL(64), but not type STRING. The system treats STRING operands as 16-bit quantities.

An arithmetic expression specifies a rule (formula) for computing a numeric value. It consists of one or more operands and arithmetic operators such as:

```
3 + 5
```

A conditional expression specifies a rule for establishing the relationship between values and results in a true or false state. It consists of one or more conditions and conditional operators such as:

```
vary > 5
```

OPERATORS

An operator is a reserved word or a symbol that directs TAL to perform an arithmetic or conditional (relational or boolean) operation on values in the program.

Arithmetic Operators

Arithmetic operators provide signed arithmetic, unsigned arithmetic, and logical operations. You can mix signed and unsigned arithmetic and logical operations in an expression.

Signed Arithmetic Operators

Signed arithmetic operators are +, -, *, and /. They can operate on operands of any data type. All operands in an expression must be of the same type, except that an INT expression can include INT and STRING operands. When the system evaluates an INT expression, it right justifies STRING operands in word units and treats them as 16-bit quantities.

INT expressions produce INT results, even if they contain STRING operands. Expressions of other types produce results of the same data type as their operands. For example, expressions that contain FIXED operands produce FIXED results, and expressions that contain REAL(64) operands produce REAL(64) results.

Signed arithmetic operations affect the condition code and carry indicators. The overflow indicator is set when you divide by 0 or when a result exceeds the bits allowed by the operand type (INT, 15 bits; INT(32) and REAL, 31 bits; REAL(64) and FIXED, 63 bits). If an overflow occurs, the results will have unpredictable values.

Examples of signed arithmetic are:

```
word1 * word2 + word1    !INT operands produce INT result
word2 / word1            !INT operands produce INT result
double1 + double2        !INT(32) operands produce INT(32) result
byte1 + byte2            !STRING operands produce INT result
word1 + byte1            !INT and STRING operands produce INT result
```

Unsigned Arithmetic Operators

Unsigned arithmetic operators are '+', '-', '*', '/', and '\'. They can operate on operands of certain data types, as follows:

- Unsigned add and subtract allow STRING or INT operands in an expression and produce INT results.

  These operations do not set the overflow indicator, but do affect the condition code and carry indicators.

- Unsigned multiplication allows STRING or INT operands and produces INT(32) results.

- An unsigned division operation or an unsigned modulo operation (which returns the remainder) requires an INT(32) dividend and an INT divisor that produces an INT quotient.

  If the quotient exceeds 16 bits, an overflow condition occurs and the results will have unpredictable values.

  For example, the modulo operation "100000D '\' 2" (which should result in a remainder of 0) causes an overflow because the quotient (50000) exceeds 16 bits.

Typically, you use unsigned arithmetic on operands with values in the range 0 through 65,535. An example is pointer variables that contain standard addresses.

Examples of unsigned arithmetic are:

```
word1 '+' word2    !Unsigned addition produces INT result
word1 '-' byte1    !Unsigned subtraction produces INT result
word1 '*' byte1    !Unsigned multiplication produces INT(32) result
dbword '/' word1   !Unsigned division produces INT result
dbword '\' word1   !Unsigned mod division produces INT result
```

Logical Operators

The LOR, LAND, and XOR operators perform bit-by-bit operations on INT and STRING operands only. They return 16-bit results as follows:

Operator	Truth Table	Example
LOR (Logical OR)	$\begin{array}{c\|cc} & 1 & 0 \\ \hline 1 & 1 & 1 \\ 0 & 1 & 0 \end{array}$	10 LOR 12 = 14 $\begin{array}{rr} 10 & 1\ 0\ 1\ 0 \\ 12 & 1\ 1\ 0\ 0 \\ \hline 14 & 1\ 1\ 1\ 0 \end{array}$
LAND (Logical AND)	$\begin{array}{c\|cc} & 1 & 0 \\ \hline 1 & 1 & 0 \\ 0 & 0 & 0 \end{array}$	10 LAND 12 = 8 $\begin{array}{rr} 10 & 1\ 0\ 1\ 0 \\ 12 & 1\ 1\ 0\ 0 \\ \hline 8 & 1\ 0\ 0\ 0 \end{array}$
XOR (Exclusive OR)	$\begin{array}{c\|cc} & 1 & 0 \\ \hline 1 & 0 & 1 \\ 0 & 1 & 0 \end{array}$	10 XOR 12 = 6 $\begin{array}{rr} 10 & 1\ 0\ 1\ 0 \\ 12 & 1\ 1\ 0\ 0 \\ \hline 6 & 0\ 1\ 1\ 0 \end{array}$

The logical operators set the condition code indicator.

Summary of Arithmetic Operators

Table 13-1 summarizes the arithmetic operators and the data types of operands on which each can operate.

TAL does not provide automatic type conversions on operands; instead, it provides built-in type-transfer functions for converting an operand from one type into another. (See Section 17.)

Table 13-1. Arithmetic Operators and Operand Types

		*Data Type of Operand					
Operator	Function	**STRING	INT	INT(32)	FIXED	REAL	REAL(64)
+	Signed Addition	●	●	●	●	●	●
-	Signed Subtraction	●	●	●	●	●	●
*	Signed Multiplication	●	●	●	●	●	●
/	Signed Division	●	●	●	●	●	●
'+'	Unsigned Addition	●	●				
'-'	Unsigned Subtraction	●	●				
'*'	Unsigned Multiplication	●	●		(See Note 1)		
'/'	Unsigned Divison	●	●	●	(See Note 2)		
'\'	Unsigned Modulo Division (remainder)	●	●	●	(See Note 2)		
LOR	Logical OR	●	●				
LAND	Logical AND	●	●				
XOR	Exclusive OR	●	●				

* Except as noted, operand types in an expression must match and the expression yields results of the same type as its operands. To convert an operand type, use a type-transfer standard function described in Section 17.

**The system treats STRING operands as 16-bit quantities; there is no STRING expression. INT expressions can have STRING or INT operands, but always yield INT results.

Note 1: Unsigned multiplication always yields an INT(32) result.

Note 2: Unsigned division and modulo operations require an INT(32) dividend and an INT divisor that produce an INT quotient. See also "Unsigned Arithmetic Operators" in this section.

Scaling of FIXED Operands

FIXED operands in an arithmetic expression need not have the same <fpoint> value. The system makes adjustments as follows:

- In addition or subtraction, the system scales the smaller <fpoint> up to match the larger <fpoint>. The <fpoint> of the result matches the larger <fpoint>. For example, the system scales the smaller <fpoint> in "3.005F + 6.01F" up by a factor of one, and the result is 9.015F.

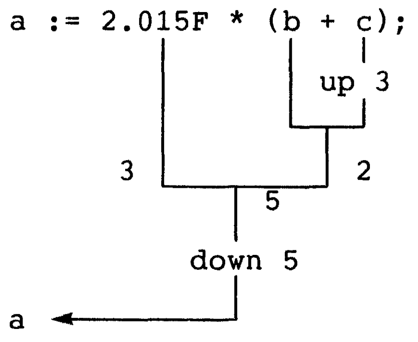
- In multiplication, the <fpoint> of the result is the sum of the <fpoint> values of the two operands. For example, "3.091F * 2.56F" results in the FIXED(5) value 7.91296F.

- 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 of 1.

  To retain precision when dividing operands having nonzero <fpoint> values, use the $SCALE function to scale up the <fpoint> of the dividend by a factor equal to the <fpoint> of the divisor. $SCALE is described in Section 17, "Standard Functions."

The following example shows scaling of FIXED operands having different <fpoint> values and scaling of the result to match the variable to which it is assigned:

```
FIXED      a;                 !Data declarations
FIXED(2)   b;                 !
FIXED(-1)  c;                 !
```



← "c" is scaled up by a factor of 3 to match "b"

← Result of multiplication is an implied <fpoint> of 5

← Result of expression is scaled down by 5 to match "a", with some loss of precision

Conditional Operators

Conditional operators are either relational or boolean. You can combine them with conditions to form conditional expressions. The result of a conditional expression is a true or false state.

You usually use conditional expressions to direct program execution. For example, in an IF-THEN-ELSE statement, if the IF condition is true, the THEN clause executes, or if it is false, the ELSE clause executes. Conditions are described under "Conditional Expressions" in this section.

Relational Operators

Signed relational operators are <, =, >, <=, >=, <>, and unsigned relational operators are '<', '=', '>', '<=', '>=', '<>', as defined in Table 13-2. They perform:

- Signed comparison of two INT, INT(32), FIXED, REAL, or REAL(64) operands
- Unsigned comparison of two INT operands

The operands in a relational expression must have the same data type, except that an INT expression can have STRING and INT operands.

Relational operations set the condition code indicator.

The following example controls program execution based on signed and unsigned comparisons:

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

IF  a  '<' c  THEN x := 314;    !False; "x" still contains 271

IF  a  <   c  THEN x := 313;    !True;  "x" is assigned 313

IF  a  <>  c  THEN              !True, but this is an arithmetic
         IF < THEN x := 314;    ! comparison; since -2 < 3,
                                ! CCL is set

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

Table 13-2. Relational Operators and Operand Types

		*Data Type of Operand					
Operator	Function	**STRING	INT	INT(32)	FIXED	REAL	REAL(64)
<	Signed Less Than	●	●	●	●	●	●
=	Signed Equal To	●	●	●	●	●	●
>	Signed Greater Than	●	●	●	●	●	●
<=	Signed Less Than or Equal to	●	●	●	●	●	●
>=	Signed Greater Than or Equal to	●	●	●	●	●	●
<>	Signed 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	●	●				

* You cannot mix operand types in an expression except STRING and INT. To convert an operand type, use a type-transfer standard function described in Section 17.

**The system treats STRING operands as 16-bit quantities. INT expressions can contain STRING and INT operands.

Boolean Operators

Boolean operators have the following meanings:

- NOT tests a condition for a false state.
- OR produces a true state if either adjacent condition is true.
- AND produces a true state if both adjacent conditions are true.

  Conditions connected by AND are evaluated from left to right until a false state occurs. The second condition is evaluated only if the first condition is true.

A condition is one or more syntactic elements that represent a single state. It can consist of a relational operator, a relational expression, a conditional expression, or an arithmetic expression, as described under "Conditional Expressions" beginning on page 13-18.

If a condition is an arithmetic expression, it must evaluate to an INT value. Thus, the operands in the condition must be type STRING or INT. If the arithmetic expression evaluates to a value of any other type, use a relational expression instead.

Boolean operations set the condition code indicator.

Examples of boolean operators are:

1. In this example, the conditions are arithmetic expressions, so "a" and "b" must be type STRING or INT. The expression is true if either condition is true; that is, if "a" or "b" contains a nonzero value:

   ```
   INT a, b:
   IF a OR b THEN . . .
   ```

2. In this example, the conditions are relational expressions, so "a" and "b" can be any data type. The expression is true if either condition is true; that is, if "a" or "b" contains a nonzero value:

   ```
   FIXED a, b:
   IF  a <> 0F  OR  b <> 0F  THEN . . .
   ```

3. The conditions in this expression are arithmetic expressions that evaluate to INT values. The expression is true if either "a" is false or both "b" and "c" are true:

   ```
   STRING a, b, c;
   IF NOT a OR b AND c . . .
   ```

Table 13-3 summarizes boolean operators and the data types of operands on which they can operate.

Table 13-3. Boolean Operators and Operand Types

		Data Type of Operand**					
Operator	Function	*STRING	INT	INT(32)	FIXED	REAL	REAL(64)
AND	Logical Conjunction	●	●				
OR	Logical Disjunction	●	●				
NOT	Logical Negation	●	●				

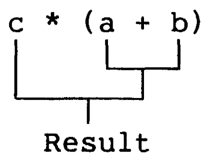
* The system treats STRING operands as 16-bit quantities. An expression can contain INT and STRING operands.

**This table applies to operands in arithmetic expressions. For types of operands allowed in relational expressions, see Table 13-2.

Precedence of Operators

TAL evaluates operations in expressions from left to right according to standard rules of precedence. Table 13-4 shows the level of precedence for each operator, from highest (0) to lowest (9).

To override the standard order of operations, place parentheses around the operation to be performed first. Examples are:



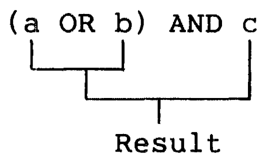


Table 13-4. Precedence of Operators

Operator	Operation	Order of Precedence	Section
.	Indirection	0	10
@	Address of Identifier	0	10
<...>	Bit Extraction	1	14
<...>	Bit Deposit	1	14
<<	Bit Shift	2	14
>>	Bit Shift	2	14
'<<'	Bit Shift	2	14
'>>'	Bit Shift	2	14
*	Signed Multiplication	3	13
/	Signed Division	3	13
'*'	Unsigned Multiplication	3	13
'/'	Unsigned Division	3	13
'\'	Unsigned Modulo Division	3	13
+	Signed Addition	4	13
-	Signed Subtraction	4	13
'+'	Unsigned Addition	4	13
'-'	Unsigned Subtraction	4	13
LOR	Logical OR	4	13
LAND	Logical AND	4	13
XOR	Exclusive OR	4	13
<	Signed Less Than	5	13
=	Signed Equal to	5	13
>	Signed Greater Than	5	13
<=	Signed Less Than or Equal to	5	13
>=	Signed Greater Than or Equal to	5	13
<>	Signed Not Equal to	5	13
'<'	Unsigned Less Than	5	13
'='	Unsigned Equal to	5	13
'>'	Unsigned Greater Than	5	13
'<='	Unsigned Less Than or Equal to	5	13
'>='	Unsigned Greater Than or Equal to	5	13
'<>'	Unsigned Not Equal to	5	13
NOT	Boolean Negation	6	13
AND	Boolean Conjunction	7	13
OR	Boolean Disjunction	8	13
:=	Assignment	9	15

ARITHMETIC EXPRESSIONS

An arithmetic expression is a rule for computing a single numeric value of a specific data type. It has a general, assignment, CASE, or IF-THEN-ELSE form.

General Form

The general form of an arithmetic expression is:

```
[ + ] <primary> [ [ <arith-operator>  <primary> ] ... ]
[ - ]
```

+ -

are unary plus and minus, indicating the sign of the leftmost <primary>. Plus is the default sign.

<primary>

is one or more items that represent a single value. <primary> can consist of the following as described under "Primaries" in this section:

Constant
Variable
Function reference
Bit shift
Bit extraction
(<arithmetic-expression>)
Code space item

<arith-operator>

is an arithmetic operator: signed (+, -, *, /); unsigned ('+', '-', '*', '/', '\'); logical (LOR, LAND, XOR).

Examples of arithmetic expressions are:

```
vary1                ! <primary> only
- vary1              ! - <primary>
+ vary1 * 2          ! + <primary> <arith-operator> <primary>
vary1 + vary2        ! <primary> <arith-operator> <primary>
vary1 * (-vary2)     ! <primary> <arith-operator> <primary>
```

Primaries

A <primary> is one or more syntactic elements that represent a single value. It can be any of the following:

- Constant--A character string or numeric constant as described in Section 4
- Variable--A direct or indirect variable as described in Sections 8 through 12 for simple variables, arrays, pointers, structures, substructures, structure data items and equivalenced variables (with or without an indirection symbol (. or @) and index)
- Function reference--A reference to a procedure that returns a value, including standard functions listed in Section 17.
- Bit shift or bit extraction--As described in Section 14.
- Arithmetic expression--The general, assignment, CASE, or IF-THEN-ELSE form described in this section, enclosed in parentheses.
- Code space item--A procedure, subprocedure, or label name prefixed with the @ symbol or a read-only array optionally prefixed with the @ symbol, with or without an index.

Examples of primaries are:

```
10                             !Constant
vary[10]                       !Variable
( IF vary THEN 1 ELSE 2 )      !( <arithmetic-expression> )
```

Assignment Form

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

The syntax for the assignment form is:

```
<variable> := <expression>


<variable>

   is a declared data variable.  (It can have an optional bit
   deposit field).


<expression>

   is an arithmetic or conditional expression that represents a
   value of the same type as <variable>.  This value is the value
   of the assignment expression form.
```

Examples

1. This example increments "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; "a" is incremented 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 Form

The CASE form of arithmetic expression selects one of several expressions for assignment to a variable.

The syntax for the CASE form is:

```
CASE <index> OF
  BEGIN
    <expression> ;     !For <index> = 0
    <expression> ;     !For <index> = 1
        .
        .
        .
    <expression> ;     !For <index> = n
    [ OTHERWISE <expression> ; ]
  END
```

<index>

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

<expression>

is an arithmetic or conditional expression.

OTHERWISE <expression>

indicates the expression to evaluate if <index> does not select an expression within the index range 0 through <n>. If you omit the OTHERWISE clause and an out-of-range case occurs, results are unpredictable.

The CASE expression form resembles the CASE statement except that:

- It selects one of several expressions instead of statements
- The selected expression must be assigned to a variable

Example

This example selects and assigns the value resulting from one of several expressions and assigns it to a variable:

```
   i := CASE a OF
          BEGIN
            b;
            c;
            d;
            OTHERWISE -1;
          END;

   !If the value of "a" is 0, the value of "b" is assigned to "i".
   !If the value of "a" is 1, the value of "c" is assigned to "i".
   !If the value of "a" is 2, the value of "d" is assigned to "i".
   !If "a" has any other value, the value of -1 is assigned to "i".
```

IF-THEN-ELSE Form

The IF-THEN-ELSE form of arithmetic expression conditionally selects one of two expressions, usually for assignment to a variable.

The syntax of the IF-THEN-ELSE form is:

```
IF <conditional-expression> THEN <expression> ELSE <expression>

<conditional-expression>

   is evaluated to determine the <expression> to compute.

<expression>

   is an arithmetic or conditional expression.
```

If <conditional-expression> is true, the THEN clause is computed; otherwise, the ELSE clause is computed. The IF-THEN-ELSE expression resembles the IF-THEN-ELSE statement except that:

- Both the THEN and ELSE clauses are required
- The THEN and ELSE clauses contain expressions, not statements
- The IF-THEN-ELSE form is typically part of an assignment statement

Examples

1. This example assigns one of two arithmetic expressions:

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

2. You can mix this form, enclosed in parentheses, with other forms:

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

CONDITIONAL EXPRESSIONS

A conditional expression specifies a rule for establishing the relationship between values. It has a general form and a group comparison form.

General Form

The general form of conditional expression is:

```
[ NOT ] <condition> [ [ { AND } [ NOT ] <condition> ] ... ]
                        { OR  }
```

<condition>

is one or more syntactic elements that represent a single state. <condition> can consist of the following as described under "Conditions" in this section:

Relational operator
Arithmetic expression
Relational expression
(<conditional-expression>)
Group comparison form

AND, OR, and NOT are boolean operators:

AND produces a true state if both <conditions> are true.
OR produces a true state if either <condition> is true.
NOT tests <condition> for a false state.

Examples of conditional expressions are:

```
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 one of the following:

- Relational operator--An operator (<, =, >, <=, >=, <>, '<', '=', '>', '<=', '>=', or '<>') that tests a condition code (see "Testing Hardware Indicators" in this section)
- Arithmetic expression--general, assignment, CASE, or IF-THEN-ELSE form discussed previously in this section

  Any arithmetic expression used as a condition must evaluate to an INT value. If it evaluates to a value of any other type, use a relational expression. (See examples of conditions below.)

  The condition is true if the value of the arithmetic expression contains a nonzero value.
- Relational expression--Two general arithmetic expressions connected by a relational operator
- Conditional expression--The general form enclosed in parentheses
- Group comparison form of conditional expression--See "Group Comparison Form" in this section

Examples of conditions are:

<condition>	Example	Description
Relational operator	IF < THEN ...	Expression is true if condition code setting is CCL
Arithmetic expression	IF a THEN ...	Expression is true if condition "a" contains nonzero value; "a" must be type INT or STRING
Relational expression	IF a <> 0F THEN ...	"a" is type FIXED; expression is true if "a" contains a nonzero value
	IF a = b THEN ...	Expression is true if "a" equals "b"
(<conditional-expression>)	IF NOT (b OR c) THEN ...	Expression is true if both "b" and "c" are false; the parenthesized condition evaluates first, then NOT is applied

Testing Hardware Indicators

The state of hardware indicators (condition code, carry, and overflow) are affected by arithmetic and conditional operations and most file system calls. If you are checking a hardware indicator, do so before another arithmetic operation occurs in the program.

The condition code setting indicates if the result of an operation is a negative value (CCL), a 0 (CCE), or a positive value (CCG). After an assignment statement, the indicator reflects the new value in the variable. To check this indicator, use a relational operator in a conditional expression, as in the example "IF < THEN . . . ."

The carry setting indicates if a carry out of bit 0 occurred. To check this indicator, use the standard function $CARRY in a conditional expression, as in the example "IF $CARRY THEN . . . ."

The overflow setting indicates if a division by 0 occurred or if the result of a signed arithmetic operation exceeds the number of bits allowed by the data type. An overflow causes an interrupt to the operating system overflow trap handler. To check the overflow indicator, turn off the overflow trap bit (bit 8) in the ENV register, then use the standard function $OVERFLOW in a conditional expression, as in the example "IF NOT $OVERFLOW THEN . . . ."

Assigning Conditional Expressions

Usually conditional expressions direct program execution without returning a value as shown in previous examples. However, if you assign a conditional expression to a variable, TAL returns a -1 for the true state and a 0 for the false state.

1. This example assigns the result of a comparison to a variable:

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

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

2. This example produces a -1 if either "x" or "y" is a nonzero value (true), or a 0 if both "x" and "y" are zeros (false):

```
INT x, y, answer;
answer := x OR y;        !Assigns -1 or 0 to "answer"
```

Group Comparison Form

The group comparison form of conditional expression performs an unsigned comparison of a group of contiguous bytes or words with another group of contiguous bytes or words or with a constant.

The syntax for the group comparison form is:

```
<var1> <rela-operator> { <var2> FOR <count> [ -> <next-addr> ] }
                       { <constant>                           }
```

<var1>

is the name of a variable, with or without an index, to compare to <var2> or <constant>. <var1> can be a simple variable, array, pointer, structure, substructure, structure item, or structure pointer, but not a read-only array.

<rela-operator>

is a relational operator (<, =, >, <=, >=, <>, '<', '=', '>', '<=', '>=', '<>') as defined in Table 13-2.

<var2>

is the name of a variable, with or without an index, to which <var1> is compared. It can be a simple variable, array, read-only array, pointer, structure, substructure, structure item, or structure pointer.

<count>

is a positive INT arithmetic expression of the general form that specifies the number of bytes or words in <var2> to compare. <count> is in bytes if <var2> is a STRING variable or pointer or a substructure. It is in words if <var2> is a non-STRING pointer or variable or a structure.

⟶

<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 address returned is:

- a 32-bit byte address if either <var1> or <var2> has an extended address
- a 16-bit byte address if <var1> and <var2> have standard byte addresses
- a 16-bit word address if <var1> and <var2> have standard word addresses

<constant>

is a numeric or character string constant or a constant list to which <var1> is compared.

The system treats the elements being compared as unsigned values. After a comparison, the condition code setting is:

```
<       (CCL) if <var1> '<' <var2>
=       (CCE) if <var1>  =  <var2>
>       (CCG) if <var1> '>' <var2>
```

The following rules apply:

- If neither <var1> or <var2> are extended, both must have 16-bit byte addresses or both must have 16-bit word addresses.
- If <var1> and <var2> are word addressed, they can be different data types. The number of elements compared depends on the data type of <var2>.
- You can compare byte-addressed data only with byte-addressed data or with constants. However, you can compare data pointed to by an extended STRING pointer with data of any type.

For INT(32) or FIXED variables, the system performs a word comparison, and <next-addr> might not point to an element boundary.

Examples

1. The following example compares two arrays:

```
   in^array = file^name FOR 9
```

2. This example compares an array to a constant list:

```
   IF file^name = [ "$RECEIVE" , 8 * ["  "]] THEN . . .
```

3. This example uses an arithmetic expression for <count>:

```
   IF in^array <> compare^mask FOR (2 * some^vary / 3) THEN . . .
```

4. The following example is a group comparison using the optional <next-addr> variable:

```
   INT .s^array [0:11] := "$SYSTEM  SYSTEM   MYFILE  ",
       .d^array [0:11] := "$SYSTEM  USER     MYFILE  ",
       .pointer;

   IF d^array = s^array FOR 12 -> @pointer THEN . . .
```

The comparison stops with element [4]; "pointer" contains the address of "d^array[4]", as follows:

```
                   0 1 2 3 4 5 . . .
   s^array --->    $SYSTEM SYSTEM  MYFILE
   d^array --->    $SYSTEM USER    MYFILE
```

You can then use the address in "pointer" to determine the number of array elements that matched:

```
   n := @pointer '-' @d^array;    !"n" gets 4 (fifth element)
```

5. These examples mix group comparisons with other conditions:

```
   IF length > 0 AND name = user FOR 8 AND NOT abort THEN, . . .
   IF (file = "TERM" OR file = "term") AND mode = 5 THEN . . .
```

6. This example compares two arrays then tests the condition code setting to see if the element in "d^array" that stopped the comparison is less than the corresponding element in "s^array":

```
   IF d^array = s^array FOR 10 -> @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"
```

SECTION 14

BIT OPERATIONS

TAL allows you to access bit fields of arbitrary size and location. You can access individual bits or groups of bits to perform the following operations:

- Bit extraction--Accesses a bit field
- Bit deposit--Assigns a value to a bit field
- Bit shift--Shifts a bit field to the left or right

For information on the precedence of bit operations, see Table 13-4 in Section 13, "Expressions."

BIT EXTRACTION

Bit extraction lets you access individual bits or groups of bits.

The syntax for the bit extraction form is:

```
<primary> . "<" <left-bit> [ : <right-bit> ] ">"
```

`<primary>`

is as described in Section 13 under "Arithmetic Expressions," except that it must be a STRING or INT value. Bit extraction does not alter <primary>.

`<left-bit>`

is an INT constant specifying the left bit of the bit field.

If <primary> is type STRING, bit <8> is the leftmost bit you can extract, because the system right justifies STRING values as if they were 16-bit quantities.

`<right-bit>`

is an INT constant specifying the right bit of the bit field. <right-bit> must be equal to or greater than <left-bit>.

Examples of Bit Extractions

1. The following example shows an assignment where the bits are extracted from an array element:

```
    LITERAL len = 8;
    INT vary;
    INT array[0:len - 1]

    vary := array[8].<8:15>;
```

2. The following example shows an assignment where bits are extracted from an arithmetic expression. Two numbers are added together, and bits <4> through <7> of the total are assigned to "result".

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

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

3. The following example shows bit extraction used in a conditional expression. It checks bits <0> through <7> for "A":

   ```
   INT word;

   IF word.<0:7> = "A" THEN ... ;
   ```

4. The following example shows bit extraction used in a conditional expression. It checks bit <15> for a nonzero value:

   ```
   STRING var;

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

BIT DEPOSIT

Bit deposit lets you assign a value to an individual bit or a group of bits using an assignment statement.

The syntax for the bit deposit form is:

```
<variable> . "<" <left-bit> [ : <right-bit> ] ">"
                                          := <expression> ;
```

<variable>

is a STRING or INT variable.

<left-bit>

is an INT constant specifying the left bit of the bit field. For STRING variables, the leftmost bit you can specify is <8>.

<right-bit>

is an INT constant specifying the right bit of the bit field. <right-bit> must be equal to or greater than <left-bit>.

<expression>

is an INT arithmetic or conditional expression.

The bit deposit field is on the left side of the assignment operator (:=). The bit deposit operation 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.

Examples of Bit Deposit

1. The following example replaces bits <10> and <11> with zeros:

```
INT old := -1;                 !"old" = 1111111111111111

old.<10:11> := 0;              !"old" = 1111111111001111
```

2. This example sets bit <8>, the leftmost bit of "strng", to 0:

```
STRING strng;

strng.<8> := 0;
```

3. In this example, the value %577 is too large to fit in bits <7:12> of "vary". The system truncates %577 to %77 before performing the bit deposit:

```
INT vary := %125252;           !"vary" = 1010101010101010

vary.<7:12> := %577;           !%577   = 0000000101111111
                                                  /     /
                                                 /     /
                                                /     /
                               !"vary" = 1010101111111010
```

4. This example replaces bits <7:8> of "new" with bits <8:9> of "old":

```
INT new := -1;                 !"new" = 1111111111111111
INT old :=  0;                 !"old" = 0000000000000000
                                                //
new.<7:8> := old.<8:9>;        !"new" = 1111111001111111
```

BIT SHIFT

The bit-shift operation shifts a bit field a specified number of positions to the left or right.

The syntax for the bit-shift form is:

```
<primary> <shift-operator> <positions>


<primary>

   is as described in Section 13 under "Arithmetic Expressions,"
   except that it must be type STRING, INT, or INT(32).  The
   system treats STRING variables as 16-bit quantities.  For types
   STRING and INT, the shift occurs on one word; for type INT(32),
   the shift occurs on two words.  Shifts do not alter <primary>.


<shift-operator>

   is an operator shown in Table 14-1.


<positions>

   is an INT <primary> indicating the number of bit positions to
   shift the bit field.  The system uses <positions> mod %400.
```

The following usage considerations apply:

- The bit shift operation sets the condition code indicator.
- To multiply by powers of two, shift the field one position to the left for each power of 2.
- To divide by powers of two, shift the field one position to the right for each power of 2.
- To convert a word address to a byte address, use an unsigned shift operator.

Table 14-1. 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 <1>	Sign bit (bit <0>) unchanged; zeros fill vacated bits from the right
>>	Signed right shift	Sign bit (bit <0>) unchanged; sign bit fills vacated bits from the left

Examples of Bit Shifts

1. This example of an 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 example of an 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 example of a signed left shift shows how zeros fill the vacated bits from the right, while the sign bit remains the same:

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

4. This example of a 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. This example shows 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
    a := b << 5;                    !Multiply by 32
    a := b >> 6;                    !Divide by 64
```

6. This example uses an unsigned bit shift to convert the word address of an INT array to a byte address and loads the byte address in a STRING pointer. This allows byte access to the INT array.

```
    INT a[0:5];                     !Declares INT array
    STRING .p := @a[0] '<<' 1;      !Declares and initializes STRING
                                    ! pointer with array byte address
                                    ! resulting from the bit shift
    p[3] := 0;                      !Assigns 0 to fourth byte of "a"
```

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

```
    INT b;                          !Declares variable

    b := b '<<' 8;                  !Shifts right byte intc left byte
```

8. This example declares and initializes an extended pointer with the lowest address in an extended segment (see also Appendix A):

```
    STRING .EXT esp := 4D '<<' 17;
```

9. This example declares an extended pointer and assigns to it an extended address in the current user code segment (see also the $DBLL function in Section 17):

```
    INT .EXT p;                     !Declares extended pointer

    @p := ($DBLL (2,7)) '<<' 1;     !Assigns address in code segment
```

SECTION 15

STATEMENTS

This section describes executable statements, which control program execution by accessing and modifying the program's data.

This section contains:

- A summary of statements, organized by functional category
- Rules for forming statements
- Syntax for each statement, listed in alphabetic order

SUMMARY OF STATEMENTS BY FUNCTION

Statements are summarized within the following categories:

- Program Control--Directs the flow of program execution
- Data Transfer--Stores or transfers data within a program
- Data Scan--Searches scan area for a test character
- Machine Instruction--Relates to machine instructions

Program Control

ASSERT	conditionally invokes error-handling procedure.
CALL	invokes procedure or subprocedure.
CASE	executes statement based on an index value.
DO-UNTIL	executes posttest loop until true condition occurs.
FOR-DO	executes pretest loop for <n> times.
GOTO	unconditionally branches to label within procedure or subprocedure.
IF-THEN-ELSE	executes THEN or ELSE statement based on true or false state.
RETURN	returns from procedure or subprocedure to caller. For functions, also can specify returned value.
WHILE-DO	executes pretest loop during true condition.

Data Transfer

Assignment	stores a value in a variable.
Move	moves group of items from one location to another.
STACK	loads value on register stack.
STORE	stores register stack element into variable.

Data Scan

RSCAN	searches scan area, right to left, for test character.
SCAN	searches scan area, left to right, for test character.

Machine Instruction

CODE	specifies machine codes for inclusion in object code.
DROP	frees index register or removes label from symbol table.
USE	reserves index register for user manipulation.

RULES FOR FORMING STATEMENTS

An executable statement can be a single statement or a compound statement. A compound statement is a BEGIN-END construct that groups statements to form a single logical statement.

The syntax for a compound statement is:

```
BEGIN
   [ <statement> ; ]  . . .
END [ ; ]
```

BEGIN

indicates the start of the compound statement.

<statement>

is an executable statement.

END

indicates the end of the compound statement.

; (semicolon)

is a statement separator.

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 to control execution of multiple operations.

The following example shows a null compound statement:

```
BEGIN
END;
```

The following example shows a compound statement that contains multiple statements:

```
BEGIN
  a := b + c;
  d := %B101;
  f := d - e;
END;
```

Separating Statements

Rules for using semicolons as separators are:

- A semicolon must separate each pair of statements.
- A semicolon is optional before the reserved word END, if END terminates a compound statement.
- A semicolon must not precede an ELSE or UNTIL keyword.

ASSERT STATEMENT

The ASSERT statement conditionally invokes the procedure named in an ASSERTION compiler control directive.

The syntax for the ASSERT statement is:

```
ASSERT <assert-level> : <expression>


<assert-level>

   is an integer in the range 0 through 32767 that is higher
   than the <assertion-level> specified in an ASSERTION
   directive.  If the <assert-level> is lower than the
   <assertion-level>, the ASSERT statement has no effect.


<expression>

   is a conditional expression that tests a program condition and
   evaluates to a true or false result.
```

The ASSERT statement is a debugging or error-handling tool. You use it with the ASSERTION directive as follows:

- Place an ASSERTION directive in the source code, naming an error-handling procedure and specifying an <assertion-level>.
- Place an ASSERT statement wherever you want to invoke the error-handling procedure if an error occurs, specifying an <assert-level> higher than the <assertion-level> of the ASSERTION directive.
- When the error occurs, the ASSERTION directive invokes the procedure.
- After you debug the program, you can nullify the ASSERT statement by raising the <assertion-level> of the ASSERTION directive higher than the <assert-level> of the ASSERT statements.

If ASSERT statements that specify the same condition have the same <assert-level>, you can nullify certain levels of ASSERT statements.

For more information on the ASSERTION directive, see Section 20, "Compiler Operation."

Example

This example invokes the operating system DEBUG procedure whenever a $CARRY or $OVERFLOW condition occurs:

```
?ASSERTION = 5, DEBUG               !Effective for all ASSERT
   .                                ! statements
   .
?SOURCE $SYSTEM.SYSTEM.EXTDECS (DEBUG)
SCAN array WHILE " " -> @pointer;
ASSERT 10 : $CARRY;
   .
   .
ASSERT 10 : $CARRY;
   .
   .
ASSERT 20 : $OVERFLOW;
```

TAL generates instructions that check the condition code indicators and invoke DEBUG.

In this example, changing <assertion-level> to 15 nullifies the $CARRY condition. Changing it to 30 nullifies all of the ASSERT statements.

ASSIGNMENT STATEMENT

The assignment statement assigns a value to a previously declared variable.

The syntax for the assignment statement is:

```
<variable> := <expression>
```

<variable>

is the identifier of a variable (simple variable, array element, pointer, or structure data item), with or without a bit deposit field and/or index. If <variable> is a pointer, you can use the @ symbol to update its contents as described in Section 10.

<expression>

is an arithmetic or conditional expression of the same type as <variable>, except as noted under "Mixing Types." It can be a bit extraction value, but not a constant list.

For information on assignments to pointers, see Section 10; for assignments to structures and structure pointers, see Section 11.

Mixing Types

ASSIGNMENT STATEMENT

The assignment statement assigns a value to a previously declared variable.

The syntax for the assignment statement is:

```
<variable> := <expression>


<variable>

   is the identifier of a variable (simple variable, array
   element, pointer, or structure data item), with or without a
   bit deposit field and/or index.  If <variable> is a pointer,
   you can use the @ symbol to update its contents as described
   in Section 10.


<expression>

   is an arithmetic or conditional expression of the same type
   as <variable>, except as noted under "Mixing Types."  It can
   be a bit extraction value, but not a constant list.
```

For information on assignments to pointers, see Section 10; for assignments to structures and structure pointers, see Section 11.

Mixing Types

The data type of the value and the variable must match except in the case of INT and STRING types.

If you assign an INT value to a STRING variable, the system left justifies the right byte of the INT value. It discards the left byte of the value.

If you assign a byte character string to an INT variable, the system stores the value in the right byte of the word, with a 0 in the left byte. (To store a character in the left side, assign the character and a space, as in "A ").

To mix types other than INT and STRING, use a type-transfer standard function, described in Section 17.

FIXED Variables

When you assign a value 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. The following example attempts to assign a value with three decimal places to a FIXED(2) variable:

```
FIXED(2) a;
a := 2.348F                   !System scales value to 2.34F
```

If the ROUND directive is on, the system scales the value as needed, then rounds it up or down. For example, if you assign the value 2.3456 to a FIXED(2) variable, the system scales the value by one digit, then rounds it to 2.35.

Examples of Assignment Statements

1. This example shows various assignment statements:

```
STRING a;                     !Declarations
INT b;
REAL c;
FIXED d;

a := 255;                     !Assignment statements
b := a + 10;
c := 36.6E-3;
d := $FIX (c);                !Type-transfer function returns
                              ! FIXED value from REAL value
```

2. In this example, the declaration is equivalent to the three assignment statements below it:

```
INT .b[0:2] := ["ABCDEF"];    !Declaration with constant list

b[0] := "AB";                 !Assignment statements
b[1] := "CD";                 ! cannot use constant lists
b[2] := "EF";
```

3. This example shows what happens when you assign an INT value to a STRING variable:

```
STRING byte1;

byte1 := "AB";
```

"B"	?

```
!"A" is lost; right half
!  retains old value
```

4. This example shows that a character assigned to an INT variable is right justified unless you also assign a space:

```
INT int1;

int1 := "A";
int1 := "A ";
```

0	"A"	!"A"
"A"		!"A "

5. In this example, the multiple assignment statement is equivalent to the three separate assignments below it:

```
INT int1;
INT int2;
INT int3;
INT vary := 16;                    !Declarations

int1 := int2 := int3 := vary;     !Multiple assignment

int1 := vary;                      !Separate assignments
int2 := vary;
int3 := vary;
```

CALL STATEMENT

The CALL statement invokes a procedure, subprocedure, or entry point, and optionally passes parameters to it.

The syntax for the CALL statement is:

```
 CALL <identifier> [ ( <param> [ , <param> ] ... ) ]


<identifier>

   is the name of a previously declared procedure, subprocedure,
   or entry point.


<param>

   is a variable or an expression that defines an actual
   parameter to pass to <identifier>.
```

You invoke procedures and suprocedures using the CALL statement, whereas you invoke functions by using their names in expressions. A CALL statement can also invoke a function. In this case, the caller ignores the returned value of the function.

Actual parameters are value or reference parameters and are optional or required depending on the procedure or subprocedure declaration, as described in Section 16.

If you omit any optional parameters, use a place-holding comma for each omitted parameter except the rightmost ones. TAL does not check for optional parameters.

When you invoke a procedure, the operating system saves the environment of the calling procedure or subprocedure and executes the called procedure. When you invoke a subprocedure, the operating system saves only the location to which control is to return after the subprocedure completes execution.

After the called procedure or suprocedure completes execution, the program returns to the statement following the CALL statement, as shown in Figure 15-1.

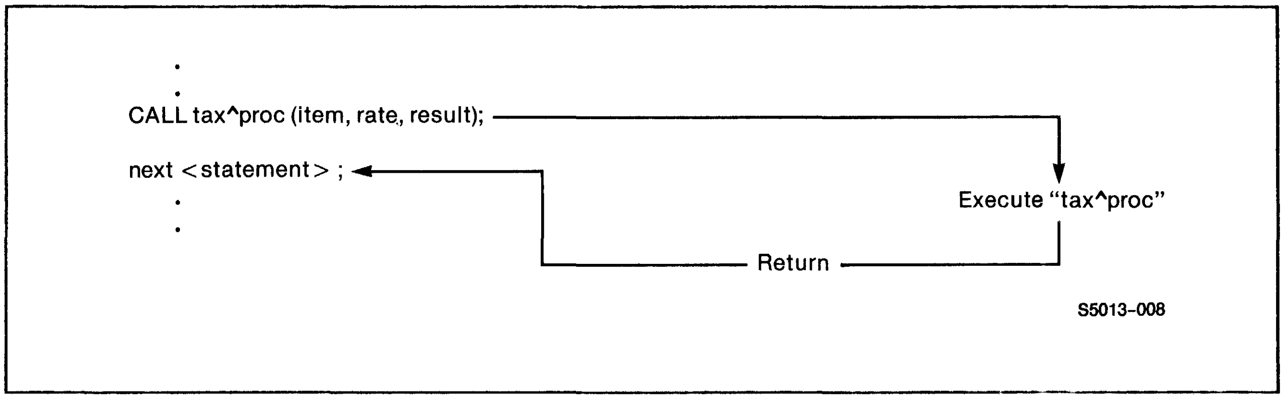


Figure 15-1. CALL Statement Execution

Examples

1. This example invokes a procedure that has no parameters:

```
CALL error^handler;
```

2. This example shows all parameters included:

```
CALL compute^tax (item, rate, result);
```

3. This example shows place-holding commas for omitted optional parameters:

```
CALL FILEINFO (filenum, error, , dev^num, , , eof ) ;
```

4. This example uses place-holding commas and comments in place of omitted parameters:

```
CALL FILEINFO (filenum, error, !filename! , dev^num,
               !dev^type! , !ext^size! , eof ) ;
```

CASE STATEMENT

The CASE statement executes one of a choice of statements, based on an index value.

The syntax for the CASE statement is:

```
CASE <index> OF
   BEGIN
      [ <statement> ] ;      !For <index> = 0
      [ <statement> ] ;      !For <index> = 1
           .
           .
           .
      [ <statement> ] ;      !For <index> = <n>
      [ OTHERWISE [ <statement> ] ; ]
   END
```

<index>

is an INT arithmetic expression that selects the statement to execute.

<statement>

is any executable statement, including a compound or CASE statement.

OTHERWISE

indicates the statement to execute for any case outside the <index> range. If you omit the OTHERWISE clause and an out-of-range case occurs, execution is unpredictable.

The CASE statement lets you make multiple branch decisions in applications where selection is based on a range of index values.

The following rules apply:

- If a case in the <index> range has no action, you must specify either a null statement (a semicolon with no <statement>) or a null compound statement.
- If a <statement> consists of more than one statement, you must use a compound statement.
- If the same <statement> applies to multiple <index> values, you only need to code the <statement>, preceded by a label, for one <index> value. Then you can use GOTO statements to the label for the other <index> values to which the <statement> applies.

Figure 15-2 shows how the CASE statement works.

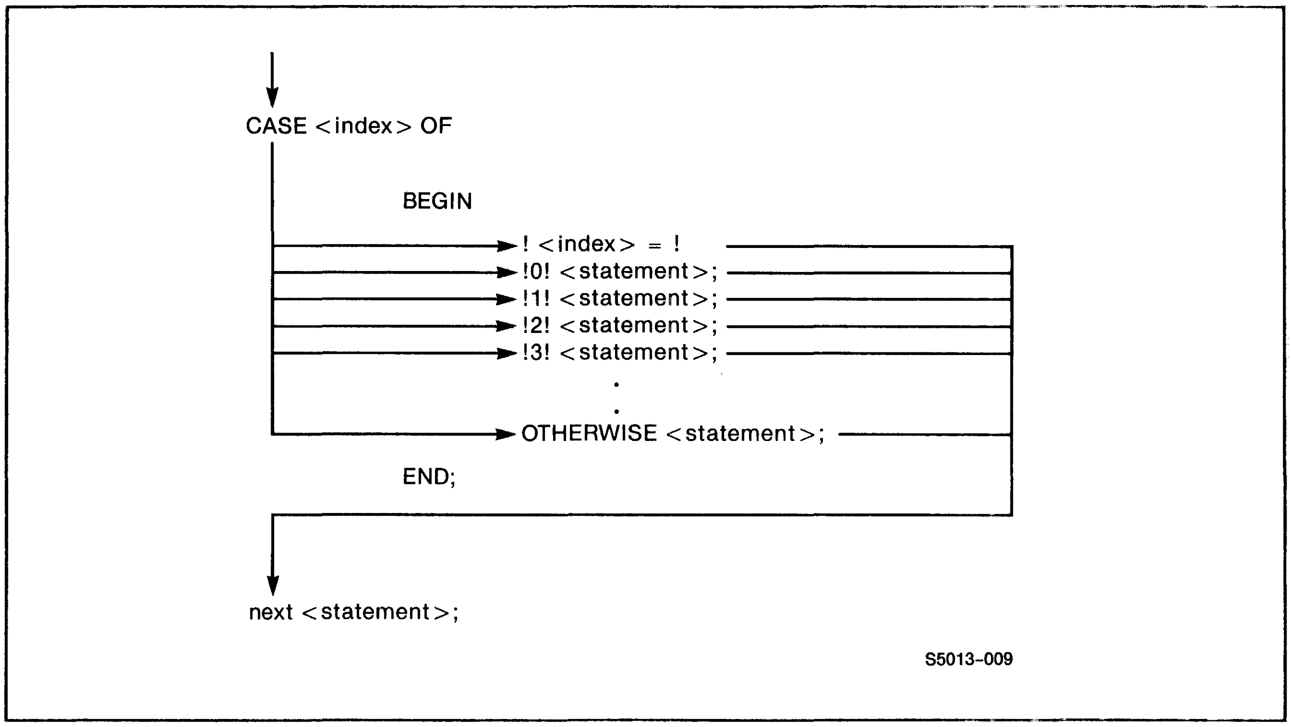


Figure 15-2. CASE Statement Execution

Examples

1. In this example, if "vary" is 0, the first statement executes; if "vary" is 1, the second statement executes. For any other case, the third statement executes.

```
INT vary;
INT vary0;
INT vary1;

CASE vary OF
  BEGIN
    vary0 := 0;                  !First statement
    vary1 := 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 (index);
INT index;                               !Index value
BEGIN
  LITERAL len = 80;                      !Length of array
  STRING .a^array[0:len - 1];            !Destination array

  CASE index 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 lets you specify machine-level instructions to compile into the object program.

The syntax for the CODE statement is:

```
CODE ( <instruction> [ ; <instruction> ] ... )
```

<instruction>

is a machine instruction in one of six forms:

No.	Form
1	<mnemonic>
2	<mnemonic> [. | @] <identifier>
3	<mnemonic> <constant>
4	<mnemonic> <index-register>
5	<mnemonic> [. | @] <identifier> [, <index-register>]
6	<mnemonic> <constant> [, <index-register>]

<mnemonic>

is an instruction code (described in the System Description Manual for your system).

<identifier>

is the name of a previously declared object. For a PCAL, XCAL, or SCAL instruction, it is a procedure name. For a branch instruction, it is a label. (The procedure name must be resolvable by the time the executable object file is created.)

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.

⟶

<index-register>

is an INT constant specifying either:

- the number of an index register in the range 5 through 7
- an identifier associated with an index register in an USE statement

If you omit <index-register>, no indexing occurs.

The form of the CODE statement correlates to the requirements of each instruction code as described in the System Description Manual for your system. You must include all required operands for each machine instruction.

TAL inserts indirect branches around instructions emitted in a CODE statement, if needed. Normally, TAL emits these values after the first unconditional branch instruction occurs.

Pseudocodes

In addition to the instruction codes described in the System Description Manual, TAL recognizes the following pseudocodes as part of the <mnemonic> set:

- CON--This code is a form 3 instruction that emits inline simple or character string constants and indirect branch locations.
- FULL--This code is a form 1 instruction that signals TAL when the register stack is full and sets the TAL RP counter to 7. TAL emits no code for this mnemonic.

Examples

1. The following example turns off traps:

```
CODE ( RDE; ANRI %577; SETE );    !Turn off traps
```

2. The following example scans from a code-relative address to the test character 0, then saves the next address:

```
STRING .ptr;
STACK @ptr, 0;
CODE ( SBU %640 );
STORE @ptr;
```

3. The following are examples of 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
```

4. This example emits %125 in the next instruction location:

```
CODE ( CON %125 );
```

5. This example emits 14 words of constant information starting in the next instruction location:

```
CODE ( CON "the con pseudo operator code" );
```

6. This example emits a code-relative pointer to "labelid" in the next instruction location:

```
CODE ( CON @labelid );
```

DO STATEMENT

The DO statement is a posttest loop that executes a statement until a specified condition becomes true.

The syntax for the DO statement is:

```
DO [ <statement> ] UNTIL <expression>

<statement>

   is any executable statement (including compound, null, and
   nested DO statements).

<expression>

   is an arithmetic or conditional expression.
```

If <expression> is always false, infinite looping occurs unless some event in the DO loop causes an exit (such as a RETURN statement).

Figure 15-3 shows the action of the DO statement.

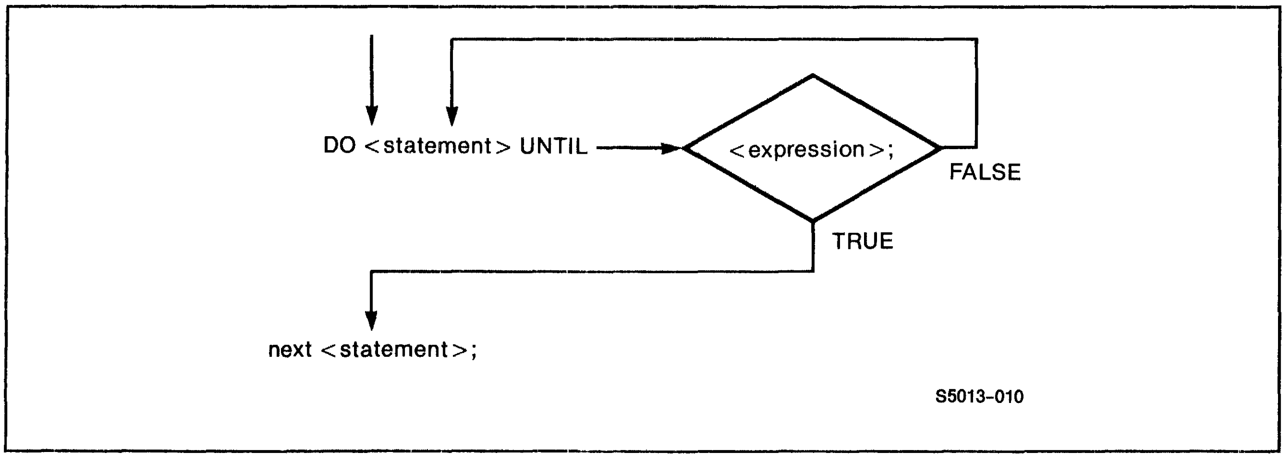


Figure 15-3. DO Statement Execution

<u>Examples</u>

1. This example loops until it clears each array element with a 0:

```
STRING .array[0:49];
DO array [index := index + 1] := 0 UNTIL index = limit;
```

2. This example tests each array element until it finds a character:

```
DO index := index + 1 UNTIL $ALPHA (array[index]);
```

3. This example shows a multiline DO statement:

```
DO
BEGIN
  i := i + 1;
  CALL check^error (error);
END                                    !No semicolon here
UNTIL i > 15 OR error = true;
```

DROP STATEMENT

The DROP statement disassociates an identifier from either (1) a label or (2) an index register that you reserved in a previous USE statement.

The syntax for the DROP statement is:

```
DROP <name>


<name>

   is the identifier of a label or of an index register
   that you reserved in a previous USE statement.
```

Dropping Labels

- You can drop a label only if you have either declared it in a label declaration or used it in a statement.
- Before you drop a label, be sure there are no further references to the label. If a GOTO appears after the drop, an error occurs.

Dropping Registers

- The name must be associated in a USE statement.
- If you reserve an index register for a FOR loop, do not drop the register within the scope of the FOR loop.
- Once you drop a name, you need a new USE statement to reference it.

Examples

1. This example uses and drops a label within a DEFINE declaration:

```
DEFINE loop =
  lab:                     !Uses label name
    .
    .
    IF a = b
    THEN
      GOTO lab;            !Branches to label
    DROP lab; #;           !Frees label name for reuse
```

2. This example reserves, uses, and drops an index register:

```
LITERAL limit = 100;
INT array[0:limit-1];      !Declarations

USE x;                     !Reserves index register; names it "x"
FOR x := 0 TO limit - 1 DO
array[x] := 0;             !Uses index register to clear array
DROP x;                    !Drops index register
```

FOR STATEMENT

The FOR statement is a pretest loop that repeatedly executes a statement while incrementing or decrementing a variable until the variable is greater than or less than a given limit.

The syntax for the FOR statement is:

```
FOR <variable> := <initial> { TO     } <limit> [ BY <step> ] DO
                            { DOWNTO }

   [ <statement> ]
```

<variable>

is the identifier of an INT variable (simple variable, array element, pointer, or structure data item).

<initial>

is an INT arithmetic expression that defines the beginning value of <variable>.

TO

specifies that <step> is added to <variable> each time through the loop until <variable> exceeds <limit>.

DOWNTO

specifies that <step> is subtracted from <variable> each time through the loop until <variable> is less than <limit>.

<limit>

is an INT arithmetic expression. Looping stops when <variable> passes <limit>.

⟶

<step>

is an INT arithmetic expression to add to or subtract from <variable> each time <statement> executes. The default is 1.

<statement>

is any executable statement, including a compound or null statement or a nested FOR statement.

Because the FOR statement tests <variable> before looping, if <variable> passes <limit> on the first test, the loop never executes.

You must enter a FOR statement only at the beginning, not at the <statement>. You can nest FOR loops to any level.

Figure 15-4 shows the action of the FOR statement.

Optimizing FOR Loops

TAL emits more efficient machine code (using the BOX instruction) if you use a reserved index register for <variable> in the FOR statement, as follows:

1. Specify a USE statement to reserve and assign a name to an index register.

2. In the FOR statement:

   --Specify the name of the index register for <variable>.

   --Specify a 1 (the default) for <step>.

   --Specify the TO clause, not the DOWNTO clause.

3. Do not modify the register stack unless you save and restore it before the end of the loop.

4. Do not drop the reserved index register (using the DROP statement) until after the FOR statement completes executing.

5. If you include procedure calls in the FOR loop, TAL does not emit more efficient code with the USE statement. Instead, TAL must emit code to save and restore the registers associated with the BOX instruction before and after the CALL statement.

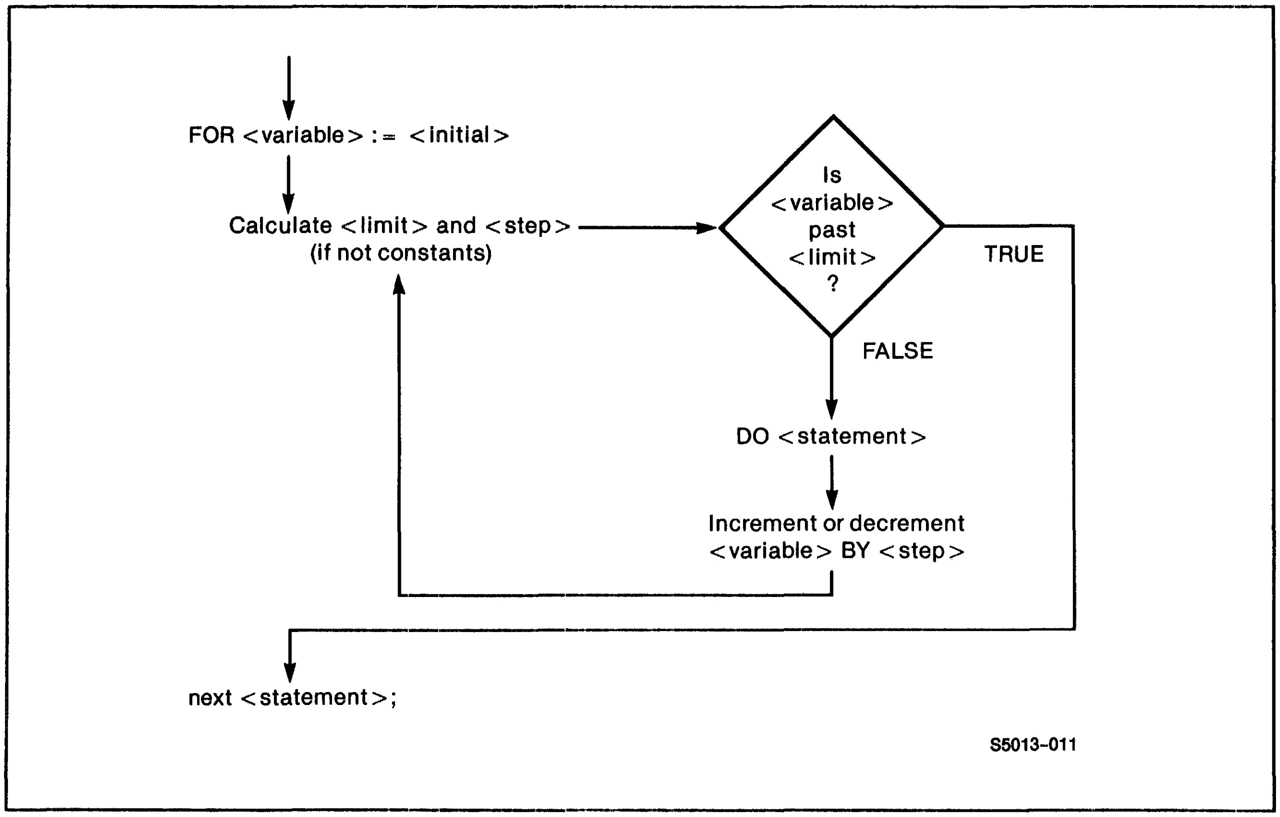

FOR <variable> := <initial>
Calculate <limit> and <step>
(if not constants)
Is <variable> past <limit> ?
TRUE
FALSE
DO <statement>
Increment or decrement <variable> BY <step>
next <statement>;
S5013-011

Figure 15-4. FOR Statement Execution

Examples

1. This FOR loop clears each array element:

```
LITERAL len = 100;
STRING .array[0:len - 1];
INT index;                              !Declarations

FOR index := 0 TO len - 1 DO            !Uses default <step> of 1
  array[index] := " ";
```

2. This example optimizes the FOR loop shown in Example 1:

```
LITERAL len = 100;
STRING .array[0:len - 1];    !Declarations

USE x;                       !Reserve index register
For x := 0 TO len - 1 DO
  array[x] := " ";
DROP x;                      !Release index register
```

3. This example uses the DOWNTO clause and a compound statement:

```
LITERAL len = 200;
INT .array[0:len - 1];
INT index;
INT answer;                                 !Declarations

FOR index := len - 1 DOWNTO 0 BY 5 DO
  BEGIN                                     !Begin compound statement
    answer := answer + index;
    array[index] := answer + index;
  END;                                      !End compound statement
```

4. This nested FOR statement 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);
```

GOTO STATEMENT

The GOTO statement unconditionally transfers program control to a labeled statement.

The syntax of the GOTO statement is:

```
GOTO <label-name>

<label-name>

   is a label you previously associated with a statement.  It
   cannot be an entry point.
```

A GOTO statement in a procedure can branch only to a label in the same procedure; it cannot branch to a label in a subprocedure. A GOTO statement in a subprocedure can branch within the same subprocedure or from the subprocedure to the calling procedure but not to another subprocedure.

Figure 15-5 shows the action of the GOTO statement.

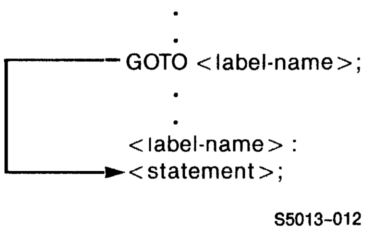


Figure 15-5. GOTO Statement Execution

Example

1. In this example, the GOTO statement transfers program execution to the statement labeled "calc^a":

```
              INT a;
              INT b := 5;

   calc^a : a := b * 2;
                 .
                 .
              GOTO calc^a;
```

IF-THEN-ELSE STATEMENT

The IF-THEN-ELSE statement executes one of a pair of statements based on whether a condition is true or false.

The syntax for the IF statement is:

```
    IF <conditional-expression>
    THEN
      [ <statement> ]
  [ ELSE
      [ <statement> ] ]
```

<conditional-expression>

is a conditional expression.

THEN <statement>

specifies the statement to execute if <conditional-expression> is true. <statement> can be any executable statement, including a compound or IF statement. If you omit <statement>, no action occurs for the THEN clause.

ELSE <statement>

specifies the statement to execute if <conditional-expression> is false. <statement> can be any executable statement, including a compound or IF statement. If you specify ELSE with no <statement>, no action occurs for the ELSE clause.

TAL sets no limit on nested IF conditions.

The IF-THEN form executes as shown in Figure 15-6. The IF-THEN-ELSE form executes as shown in Figure 15-7.

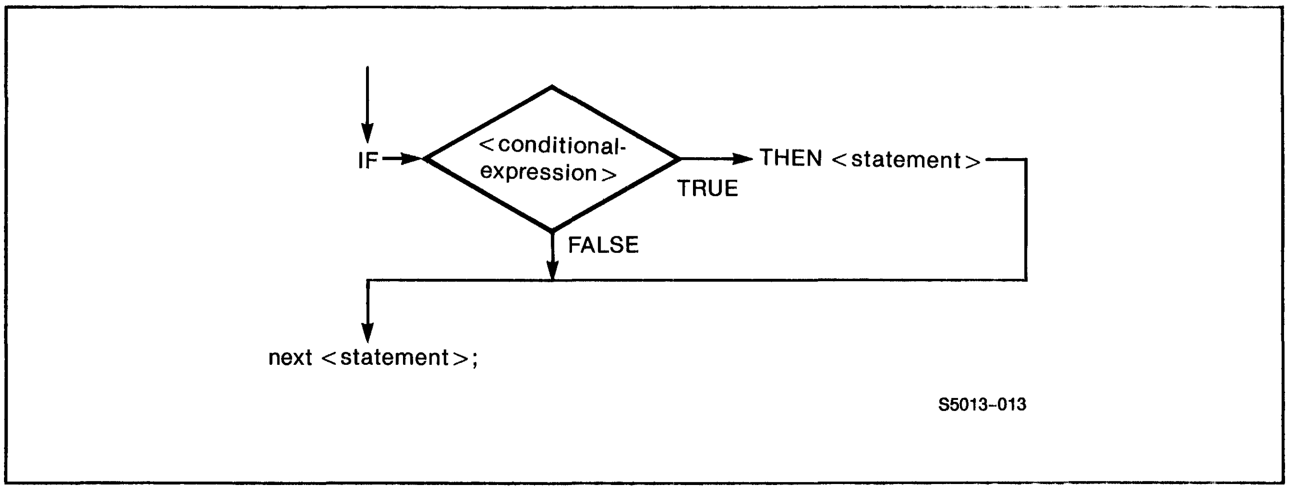


Figure 15-6. IF-THEN Form Execution

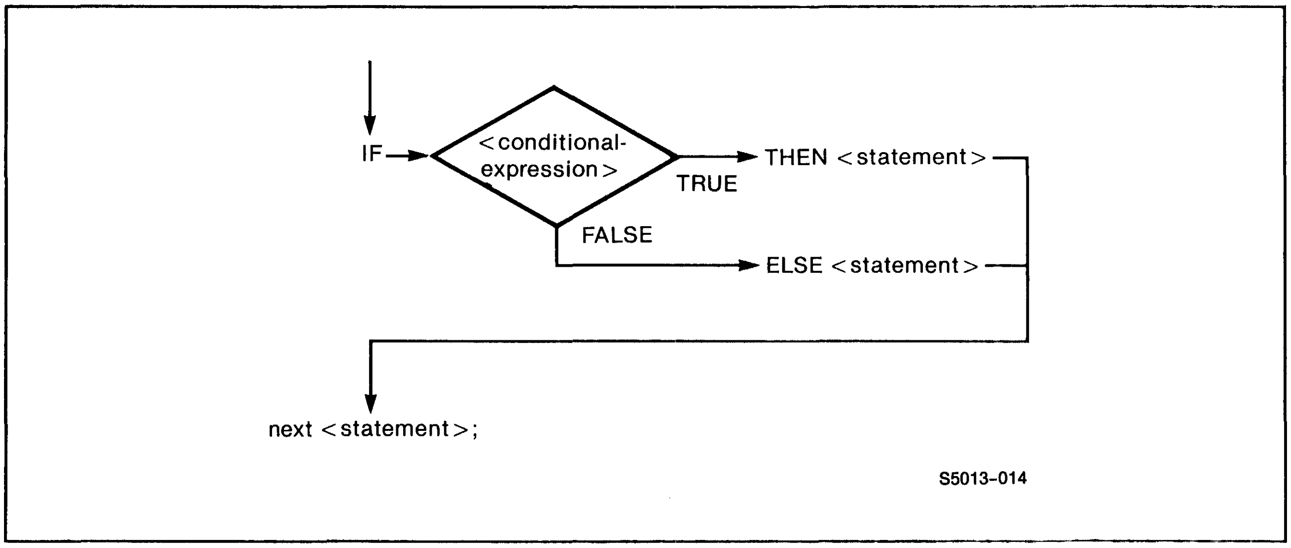


Figure 15-7. IF-THEN-ELSE Form Execution

THEN-ELSE Pairing

The innermost THEN clause pairs with the closest ELSE clause, and pairing proceeds outward. In the following examples, the ELSE clause belongs to the second THEN clause (IF "condition2"). The statements shown are equivalent, but the THEN-ELSE pairing is clearer in the example on the left:

```
Recommended Format                Ambiguous Format

IF condition1                     IF condition1 THEN
THEN                                IF condition2 THEN
  IF condition2                       stmt1
  THEN                            ELSE
    stmt1                           stmt2;
  ELSE
    stmt2;
```

To override the THEN-ELSE pairing, you can use the BEGIN or END keyword in a compound statement. Using the same example, if you insert a BEGIN-END pair as shown below, the ELSE clause belongs to the first THEN clause (IF "condition1"):

```
IF condition1
THEN
  BEGIN                 !Begin compound statement
    IF condition2
    THEN
      stmt1
  END                   !End compound statement (no semicolon here)
ELSE
    stmt2;
```

Examples

1. This example checks a variable for a nonzero value:

```
     INT var^item;

     IF var^item <> 0
     THEN
       CALL error^handler;
```

2. This example checks the hardware condition code setting and calls a message-printing procedure when an error occurs:

```
 .
 .
CALL READ (filenum,...);    !Sets condition code on error
IF <                        !Checks the condition code
THEN
  BEGIN
    CALL print^error;       !Call message-printing procedure
    CALL STOP;
  END;
```

3. This example of the IF-THEN-ELSE form compares two arrays:

```
IF new^array = old^array FOR 10
THEN
  item^ok := 1
ELSE
  item^ok := 0;
```

4. This nested IF statement illustrates THEN-ELSE pairings:

```
   IF a = b
┌──THEN
│     IF c = d
│  ┌──THEN
│  │     IF e = f
│  │  ┌──THEN
│  │  │     IF g <= h
│  │  │  ┌──THEN
│  │  │  │     BEGIN
│  │  │  │     IF (NOT g > 1) OR (m = n)
│  │  │  │  ┌──THEN
│  │  │  │  │    result := 0
│  │  │  │  └──ELSE
│  │  │  │       result := 1
│  │  │  │     END
│  │  │  └──ELSE             !No statement
│  │  └──ELSE
│  │       result := 2
│  └──ELSE
│       result := 3;
└───                         !No corresponding ELSE clause
```

MOVE STATEMENT

The left or right move statement transfers contiguous bytes, words, or elements from one location to another.

The syntax of the move statement is:

```
<destination> { ':=' } { <source> FOR <count> } [ -> <next-addr> ]
              { '=:' } { <constant>           }
```

<destination>

is the name of the variable, with or without an index, to which the move begins. It can be a simple variable, array, pointer, structure, substructure, structure data item, or structure pointer, but not a read-only array.

':='

indicates a left-to-right sequential move.

'=:'

indicates a right-to-left sequential move.

<source>

is the name of the variable, with or without an index, from which the move begins. It can be a simple variable, array, read-only array, pointer, structure, substructure, structure item, or structure pointer.

→

<count>

is a positive INT arithmetic expression that defines the number of bytes, words, or elements in <source> to move, as follows. If omitted, TAL assumes a <count> of 1 and issues a warning.

Simple variable = elements
Array = elements
Structure = words
Substructure = bytes
Structure pointer = bytes if STRING, words if INT
Pointer = elements

<constant>

is a LITERAL, numeric or character string constant, or constant list to move.

<next-addr>

is a variable to contain the location in <destination> that follows the last item moved. <next-addr> is:

- a 32-bit byte address if either <source> or <destination> has an extended address
- a 16-bit byte address if both <source> and <destination> have standard byte addresses
- a 16-bit word address if both <source> and <destination> have standard word addresses

Element Moves

If either <source> or <destination> is extended, the data in either location can be any type (STRING, INT, INT(32), FIXED, REAL, or REAL(64)).

If <source> and <destination> have standard addresses, the data in both locations must be byte addressed, or they must both be word addressed. If both are word addressed, their data types need not match and can be INT, INT(32), FIXED, REAL, or REAL(64).

After an element move, <next-addr> might not point to an element boundary in <destination>.

A concatenated move lets you move more than one <source> or constant list, each separated by an ampersand (&).

Examples

Examples of structure moves follow examples of element moves.

Examples of Element Moves

1. This example shows a left-to-right move from one array to another:

```
LITERAL length = 12;
INT .out^array[0:length - 1];
INT .in^array[0:length - 1];

out^array[0] ':=' in^array[0] FOR length;
```

2. This is a right-to-left quadword element shift by one within an array. It frees element [0] for new data:

```
LITERAL upper = 11;           !Upper bound (same as length - 1)
FIXED .in^array[0:upper];     !Source and destination array

in^array[upper] '=:' in^array[upper - 1] FOR upper;
```

3. This example moves a constant list:

```
LITERAL len = 10;
STRING .p^array[0:len - 1];

p^array[0] ':=' len * ["-"];    !Moves hyphen into each element
```

4. This example moves spaces into the first five elements, then uses <next-addr> as <destination> to move dashes into the next five elements:

```
LITERAL len = 20;                       !Length of array
LITERAL num = 5;                        !Number of elements
STRING .array[0:len - 1];               !Destination array
STRING .next^addr;                      !Pointer for next address

array[0] ':=' num * [" "] -> @next^addr;
next^addr ':=' num * ["-"];
```

5. This concatenated move is a fast way to clear an array:

```
LITERAL length = 100;                           !Length of array
INT .array[0:length - 1];                       !Destination array

array[0] ':=' "  " & array[0] FOR length - 1;  !Clears array
```

6. This concatenates and moves three arrays and some constants:

```
LITERAL line^len = 68;                 !Length of destination array
LITERAL date^len = 11;                 !Length of source array 1
LITERAL id^len = 11;                   !Length of source array 2
LITERAL dp^len = 3;                    !Length of source array 3

STRING .line^array[0:line^len - 1];
STRING .date^array[0:date^len - 1] := "Feb 1, 1985";
STRING .id^number[0:id^len - 1] := "854-70-1950";
STRING .dp^num[0:dp^len - 1] := "107";

line^array ':='   "    DATE: " & date^array FOR date^len
                & "   IDENTIFICATION: " & id^number FOR id^len
                & "   DEPARTMENT: " & dp^num FOR dp^len;
```

After execution, "line^array" contains the following:

```
DATE: Feb 1, 1985   IDENTIFICATION: 854-70-1950   DEPARTMENT: 107
```

Examples of Structure Moves

1. This example moves three occurrences of the source structure to the destination structure:

```
LITERAL copies = 3;                   !Number of occurrences

STRUCT .s[0:copies - 1];              !Source structure
BEGIN
  INT a;
  INT b;
  INT c;
END;

STRUCT .d (s) [0:copies - 1];         !Destination structure

PROC p;
BEGIN
  d ':=' s FOR copies * (($LEN(s) + 1) '>>' 1);
END;                                  !Word move for structures;
                                      ! moves three occurrences
```

2. This right-to-left move makes room for a new occurrence at the beginning of a structure:

```
LITERAL last = 9;               !Last occurrence

STRUCT t(*);                    !Template structure
BEGIN
  INT i;
  INT j;
  INT k;
  INT l;
END;

STRUCT .s (t) [0:last];         !Source and destination structure

PROC p;
BEGIN
  s[last] '=:' s[last-1] FOR last * (($LEN(s) + 1) '>>' 1);
END;
```

3. This example moves 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;
  INT b;
  END;
END;

STRUCT .d (s);                    !Destination substructure
                                  ! is within structure "d"

PROC p;
BEGIN
  d.s^sub ':=' s.s^sub FOR copies * $LEN(s.s^sub);
END;                              !Byte move for substructures;
                                  ! moves three occurrences
```

4. This code moves structure occurrences using structure pointers:

```
STRUCT t (*);                       !Template structure
BEGIN
  INT a;
  STRING b;
END;

INT .EXT ptr0(t) := %200000D;       !Structure pointer to
                                    ! upper 32K
STRING .EXT ptr1(t) := %2000000D;   !Structure pointer to start
                                    ! of extended segment

PROC p;
BEGIN
  ptr1 ':=' ptr0 FOR (($LEN(t) + 1) '>>' 1);   !Word move
                                    ! from upper 32K to start
                                    ! of extended segment
  ptr0 ':=' ptr1 FOR $LEN(t);       !Byte move from extended
END;                                ! segment to upper 32K
```

RETURN STATEMENT

The RETURN statement provides exit points from an invoked procedure or subprocedure body back to the caller. If the invoked procedure or subprocedure is a function, it can return a value.

The syntax for the RETURN statement is:

```
RETURN                              !Untyped procedure

RETURN <expression>                 !Function (typed procedure)


<expression>

   is an arithmetic or conditional expression of the same type
   as the encompassing procedure or subprocedure. <expression>
   is the value to return to the caller. Specify <expression>
   only when returning from functions.
```

A procedure or subprocedure returns to the caller when:

- A RETURN statement occurs.
- The invoked procedure or subprocedure finishes execution by reaching the last END.

In a procedure designated MAIN, a RETURN statement stops execution of the procedure and passes control to the operating system.

If a function does not contain a RETURN or if the TAL RP counter setting is 7 (empty register stack), TAL emits a warning. If a function contains a RETURN, you must specify <expression>. The value of <expression> goes on the register stack.

For untyped procedures and subprocedures, a RETURN statement is optional. If you do use a RETURN statement, you cannot include an <expression> with it.

Examples

1. This example shows RETURN statements in a function:

```
INT PROC other (nuff, more);
    INT nuff;
    INT more;
  BEGIN
    IF nuff < more
    THEN
      RETURN nuff * more        !Function returns a value
    ELSE
      RETURN 0;
  END;
```

2. This example show an untyped procedure with a RETURN statement:

```
PROC another;
BEGIN
  INT a,
      b;
   .
   .
  IF a < b THEN RETURN;         !Returns no value
   .
   .
END;
```

SCAN STATEMENTS

The SCAN or RSCAN statement searches a scan area for a test character from left to right or from right to left, respectively.

The syntax for the SCAN and RSCAN statements is:

```
{ SCAN  } <variable> { WHILE } <test-char> [ -> <next-addr> ]
{ RSCAN }            { UNTIL }
```

SCAN

indicates a left-to-right search.

RSCAN

indicates a right-to-left search.

<variable>

is the name of a variable, with or without an index, at which to start the scan. It can be a simple variable, array, standard pointer, structure, substructure, structure data item, or standard structure pointer. The data must be in the lower 32K area.

WHILE

specifies that the scan continues until a character other than <test-char> occurs. A scan stopped by a 0 sets the hardware CARRY bit.

UNTIL

specifies that the scan continues until <test-char> or a 0 occurs. 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. If the value is larger than eight significant bits, execution errors might result.

<next-addr>

is a 16-bit variable to contain the 16-bit byte address of the character that stopped the scan, regardless of what type <variable> is.

If the test character or a 0 does not occur during a SCAN UNTIL operation, the scan might continue to the 32K boundary. Before doing any scans, you can delimit the scan area as follows:

```
INT .buffer[-1:20] := [0,"  John James Jones  ",0];
                       ↑                          ↑
                       └──── scan delimiters ─────┘
```

A scan that stops on a 0 sets the hardware CARRY bit, which means the test character did not occur. To check the CARRY bit, use the $CARRY function before doing any arithmetic operations, as follows:

```
IF $CARRY                    !If test character not found...
THEN ...;

IF NOT $CARRY                !If test character found...
THEN ...;
```

Examples

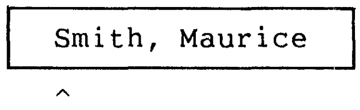
The following declarations apply to the examples:

```
INT .buffer[-1:18] := [0,"  Smith, Maurice  ",0];  !INT buffer
STRING .sptr := @buffer '<<' 1;                    !STRING pointer
                                                   ! to INT buffer
STRING .first1, .first2, .last1, .last2, .comma;   !Pointers
INT offset, length;                                !Variables
```

In the diagrams, a circumflex (^) denotes the character that stopped the scan. Declarations are on the previous page.

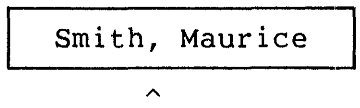
1. This example scans from element [0] for spaces, checks the CARRY bit, and calls a string-handling procedure if a character occurs:

```
SCAN sptr[0] WHILE " " -> @first1;
IF NOT $CARRY THEN
CALL string^handler;
```



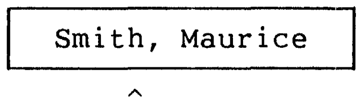
2. This example scans from the first character of the last name for a comma (,), checks the CARRY bit, and calls an error-printing procedure if a comma does not occur:

```
SCAN first1 UNTIL "," -> @comma;
IF $CARRY THEN
CALL invalid^input;
```



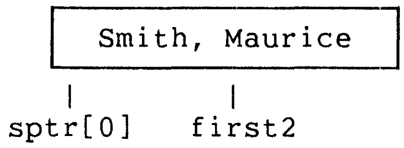
3. This example scans for spaces right to left from the location preceding the comma. In this case, the scan starts and stops at the same location:

```
RSCAN comma[-1] WHILE " " -> @last1;
```



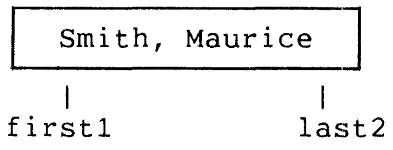
4. This example uses <next-addr> to compute the offset of the last name from the beginning of the array:

```
SCAN comma[+1] WHILE " " -> @first2;
offset := @first2 '-' @sptr;
```



5. This example uses <next-addr> to compute the length of the character string stored in the array:

```
SCAN first2 UNTIL " " -> @last2;
length := @first1 '-' @last2;
```



STACK STATEMENT

The STACK statement loads a value onto the register stack.

The syntax for the STACK statement is:

```
STACK <expression> [ , <expression> ] ...


<expression>

   is a value to load onto the register stack.  If you list
   multiple values, STACK loads them from left to right.  The
   number of registers needed by an <expression> depends on its
   data type.
```

You can use the register stack for temporary storage and for optimizing critical code sections.

TAL loads values on the register stack starting at the current setting of the RP + 1. As TAL loads each value, it increments RP by the number of words required by the type of the value. For example, for an INT(32) value, it increments RP by 2; for a quadword value, it increments RP by 4.

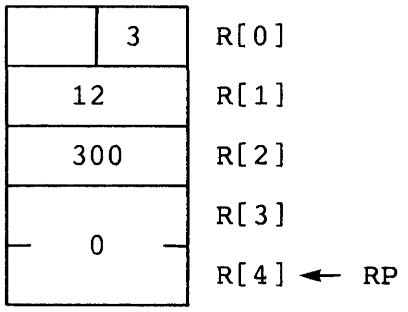
TAL keeps track of the size and type of values being stacked and emits appropriate machine instructions. TAL right justifies byte values; that is, it loads them on the register stack in bits <8:15>.

If the number of registers needed exceeds the number of free registers, TAL transfers the contents of registers R[0] through RP to the data stack, then loads the registers starting at RP[0] with values specified in the STACK statement.

Examples

1. This example loads values of various types onto the register stack:

```
STRING   .b[0:2] := [1,2,3] ;
INT      wrd := 3;
INT(32)  dwrd := 0D;

STACK b[2], wrd * 4, 300, dwrd;
```



Register Stack

2. This example shows two versions of a switch operation commonly used in sorting. The first version needs six memory references; the second needs only four memory references, uses the register stack, and is faster:

```
INT temp;
INT x;
INT y;

temp := x;
x := y;
y := temp;                    !Switch operation version 1

STACK x,y;
STORE x,y;                    !Switch operation version 2
```

STORE STATEMENT

The STORE statement removes values from the register stack and stores them into variables.

The syntax for the STORE statement is:

```
STORE <variable> [ , <variable> ] ...
```

`<variable>`

is the name of a variable (simple variable, array element, pointer, or structure data item), with or without a bit deposit field and/or index. If <variable> is a pointer, you can use the @ symbol to update its contents as described in Section 10.

If the STORE statement specifies multiple variables, storage begins with the leftmost variable.

The data type of each variable specified dictates the number of registers to unload, starting at the current RP. If the RP setting is too small to satisfy the variable type, TAL removes the required number of items from the data stack, places them on the register stack, and stores them in the variable.

Examples

1. The following example stores register contents into variables of various types:

```
LITERAL len = 100;
STRING     .byte[0:len - 1];
INT        word;
INT(32)    twowords;

STORE byte[3], word, twowords;
```

2. The following example stacks two variables, then stores them back into the same variables:

```
    STACK x, y;
      .
      .
    STORE y, x;
```

3. The following example switches the values of two variables:

```
    STACK x, y;
    STORE x, y;
```

USE STATEMENT

The USE statement associates an identifier with an index register and reserves it for your use.

The syntax for the USE statement is:

```
USE <name>

<name>

   is an identifier to associate with an index register.
```

TAL associates each identifier with an index register, starting with R[7] down to R[5]. You can then reference the identifier in statements. For example, you can use a reserved index register to optimize a FOR loop, as described under the FOR statement.

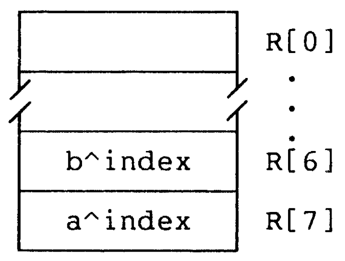
The following rules apply:

- You cannot reserve more than three registers at a time.
- Evaluation of certain expressions might overwrite the value in a reserved register, such as multiplication of two FIXED values.
- If the compiler needs an index register and none is available, a compilation error results.
- You can issue a DROP statement to release a register. (When TAL reaches the END reserved word of a procedure or subprocedure body, all registers are automatically dropped.)
- After you drop an index register, you cannot use its name without a new USE statement.

Examples

1. This example reserves two index registers:

```
USE a^index;
USE b^index;
```



Register Stack

2. This example reserves an index register, then drops it:

```
USE x;                                !Reserve register
  .
  .
DROP x;                               !Free register
```

3. This example shows two versions of a FOR loop, the second of which uses a reserved register and runs faster (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;               !Version 1

USE x;
FOR x := 0 to len - 1 DO                  !Version 2 is faster
  array[x] := array[x] + 5;
DROP x;
```

WHILE STATEMENT

The WHILE statement is a pretest loop that repeatedly executes a statement while a specified condition is true.

The syntax for the WHILE statement is:

```
WHILE <conditional-expression> DO [ <statement> ]


<conditional-expression>

   is a conditional expression.

<statement>

   is any executable statement (including compound, null, and
   WHILE statements).
```

The WHILE statement is useful when the number of loops needed is unknown. It evaluates and tests <conditional-expression> before looping; if <conditional-expression> is false after the first test, <statement> never executes.

If <conditional-expression> is always true, <statement> executes indefinitely unless some event in the WHILE loop causes an exit, such as a RETURN statement.

Figure 15-8 shows the action of the WHILE statement.

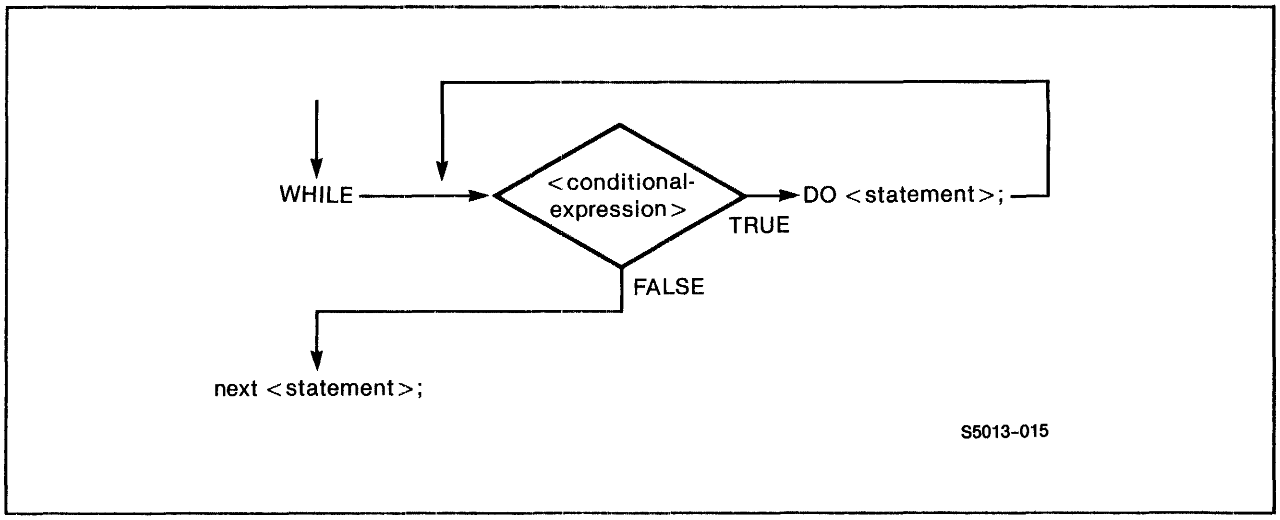


Figure 15-8. WHILE Statement Execution

Examples

1. This loop continues while "item" is not equal to zero:

```
LITERAL len = 100;
INT .array[0:len - 1];
INT item := 1;
INT i := 0;

WHILE item <> 0 DO
  BEGIN
    item := array[i];
    i := i + 1;
  END;
```

2. This WHILE statement 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 . . .
```

SECTION 16

PROCEDURES AND SUBPROCEDURES

Procedures and subprocedures are the executable portions of a TAL program. They compose the block structure of the program. They allow you to segment the program into discrete blocks or subroutines that perform a task.

An executable program contains at least one procedure. Furthermore, one procedure has the attribute MAIN, which identifies it as the first procedure to execute when you run the program. A procedure can contain subprocedures, which execute at various points within that procedure.

The maximum possible size of a single procedure is 32K words minus either the Procedure Entry Point (PEP) Table in the lower 32K area or the External Entry Point (XEP) Table in the upper 32K area. For information on the PEP or XEP table, see the System Description Manual for your system.

This section describes:

- Characteristics of procedures and subprocedures
- Procedure and subprocedure declarations
- Parameters and parameter passing
- Entry-point declarations

CHARACTERISTICS OF PROCEDURES AND SUBPROCEDURES

Procedures and subprocedures share the following characteristics:

- Procedures and subprocedures are parameterized. The same procedure or subprocedure can process different sets of variables.
- Procedures and subprocedures allow all items that have global scope (except procedures) to have local scope (in a procedure) or sublocal scope (in a subprocedure).
- Procedures and subprocedures can be functions and return a value to the caller. You can use the name of a function in an expression as if it were a variable name.
- The system allocates and initializes a private data area for each activation of a procedure or subprocedure. After each activation completes execution, the system deallocates its data area.
- Procedures and subprocedures can receive variables, constant expressions, and procedure names passed as parameters. (The MAIN procedure does not receive parameters.)
- FORWARD declarations let you reference procedures and subprocedures before their bodies occur in the source code. Thus, you can declare their bodies in any order.
- Procedures and subprocedures can call themselves; that is, they can be recursive.

Procedures and subprocedures differ as follows:

- Procedures have global scope; you use procedures for operations needed throughout the program. Subprocedures have local scope; you use subprocedures for operations needed within a procedure.
- Procedures can contain subprocedures; subprocedures cannot contain subprocedures.
- Unlike subprocedures, procedures can be referenced as external procedures by procedures declared in other compilations.
- A procedure has a 127-word primary storage area and a larger secondary area. A subprocedure has a 31-word primary area and no secondary area.
- The system invokes subprocedures more rapidly than procedures. For subprocedures, it uses the BSUB instruction; for procedures, it uses the PCAL instruction. These instructions are described in the System Description Manual for your system.

- When you invoke a procedure, the operating system saves the environment of the calling procedure or subprocedure. It restores the environment when the invoked procedure completes execution.

  When you invoke a subprocedure, the operating system saves only the location to which control is to return when the invoked subprocedure completes execution.

- Within procedures, initializations and statements can refer to global variables or to local variables declared in that procedure.

  Within subprocedures, initializations and statements can refer to global variables, to local variables declared in the encompassing procedure, or to sublocal variables declared in that subprocedure.

- Subprocedures can have the following attribute only:

VARIABLE	Subprocedure parameters are optional.

  Procedures can have the following attributes:

MAIN	This procedure executes first when you run the program.
RESIDENT	Procedure's instruction codes are not swapped in and out of main memory when you run the program.
CALLABLE	Procedure executes in privileged mode, but nonprivileged procedures can call it.
PRIV	Procedure executes in privileged mode, and only privileged procedures can call it.
INTERRUPT	Only operating system interrupt handlers can use this attribute. When returning to its caller, the procedure executes an IXIT (rather than an EXIT) instruction.
VARIABLE	Procedure parameters are optional.
EXTENSIBLE	You can add new parameters to the procedure without recompiling the caller.

PROCEDURE AND SUBPROCEDURE DECLARATIONS

The syntax of a procedure or subprocedure declaration is:

```
[ <type> ] { PROC    } <identifier>
           { SUBPROC }

     [ ( <formal-param-name> [ , <formal-param-name> ] ... ) ]

     [ <attribute> [ , <attribute ] ... ] ;

[ <formal-param-specification>

                     [ , <formal-param-specification> ] ... ; ]

{  <body>    ; }
{  FORWARD   ; }
{  EXTERNAL  ; }  !For procedures only
```

<type>

specifies that the procedure or subprocedure is a function that returns a value and indicates the data type of the returned value. <type> is one of:

```
STRING
INT
INT(32)
FIXED [ ( <fpoint> ) ]
REAL
REAL(64)
```

<identifier>

is the name of the procedure or subprocedure.

<formal-param-name>

is the name of a formal parameter. The number of formal parameters you can declare is limited by space available in the parameter area. See "Parameter Area" in this section.

⟶

<attribute>

For a subprocedure, <attribute> can be VARIABLE only.

For a procedure, <attribute> can be one or more of the following, as defined under "Attributes" in this section:

```
MAIN | INTERRUPT
RESIDENT
CALLABLE
PRIV
VARIABLE | EXTENSIBLE
```

<formal-param-specification>

specifies the data type of a formal parameter and whether it is a value or reference parameter. See "Formal Parameter Specifications" in this section.

<body>

is a BEGIN-END construct that contains declarations and statements. See "Procedure and Subprocedure Bodies" in this section.

FORWARD

means the declaration for the body occurs later in the source file (procedures) or later in this procedure (subprocedures).

EXTERNAL

applies to procedures only and means the procedure body is declared in another compilation such as a part of the operating system or a user library.

Operating system external declarations are contained in a system file that you can specify in a SOURCE directive. The system file is $SYSTEM.SYSTEM.EXTDECS[<n>], where:

```
EXTDECS0 = current release version
EXTDECS1 = current release version minus one
EXTDECS  = current release version minus two
```

Formal Parameter Specifications

A formal parameter specification defines the parameter type of a formal parameter and whether it is a value or a reference parameter.

The syntax for the formal parameter specification is:

```
<param-type> [ .    ] <formal-param-name> [ ( <referral> ) ]
             [ .EXT ]

    [ , [ .    ] <formal-param-name> [ ( <referral> ) ] ] ... ;
        [ .EXT ]
```

`<param-type>`

is the parameter type of the formal parameter:

```
STRING
INT
INT(32)
FIXED [ ( <fpoint> ) ]
FIXED(*)
REAL
REAL(64)
STRUCT
[ <type> ] PROC
```

. (period)

denotes a standard pointer and a reference parameter. The absence of "." or ".EXT" denotes a value parameter.

.EXT

denotes an extended pointer and a reference parameter. The absence of "." or ".EXT" denotes a value parameter.

`<formal-param-name>`

is the identifier of a formal parameter, as defined in "Parameters" in this section.

⟶

<referral>

is the name of a previously declared structure or structure pointer. <referral> is required only if <formal-param-name> is a structure pointer.

Procedure and Subprocedure Bodies

Procedure and subprocedure bodies contain declarations and statements.

Procedure bodies and subprocedure bodies are described separately on the following pages.

Procedure Body

The syntax for the procedure body is:

```
BEGIN

  [ <local-declaration> ]  . . .

  [ <subprocedure-declaration> ]  . . .

  [ <statement> ]  . . .

END ;


<local-declaration>

   is a declaration for one of:

      Simple variable
      Array (direct or indirect)
      Structure (direct or indirect)
      Equivalenced variable
      Pointer
      Structure pointer
      LITERAL
      DEFINE
      Label
      Entry point
      FORWARD subprocedure


<subprocedure-declaration>

   is as previously described under "Procedure and Subprocedure
   Declarations" in this section.


<statement>

   is any executable statement described in Section 15.
```

Subprocedure Body

The syntax for the subprocedure body is:

```
BEGIN

   [ <sublocal-declaration> ]  . . .

   [ <statement> ]  . . .

 END ;


<sublocal-declaration>

   is a declaration for one of:

      Simple variable
      Array (direct only)
      Structure (direct only)
      Equivalenced variable
      Pointer
      Structure pointer
      LITERAL
      DEFINE
      Label
      Entry point


<statement>

   is any executable statement described in Section 15.
```

Sublocal Variables

Data variables declared in subprocedures must be directly addressed, because the sublocal area has no secondary storage. (See "Primary and Secondary Storage" in Section 5.) If you declare a sublocal indirect array, TAL converts it to a direct array and emits a warning.

Invoking Procedures, Subprocedures, and Functions

You invoke procedures or subprocedures by using their names in CALL statements. You can call a procedure from anywhere in the program. You can call a subprocedure from within the encompassing procedure.

You invoke functions (typed procedures or subprocedures) by using their names in expressions.

Statements in the invoked procedure or subprocedure body execute until the last statement or a RETURN statement executes. Program execution then returns to the point following the invocation of the procedure or subprocedure.

The scope of items declared within a procedure or subprocedure is limited to the same procedure or subprocedure. Thus, a local or sublocal item can have the same name as a global item without conflict. In this case, however, you cannot reference the global item.

Examples

1. The following example shows two procedures, the second of which calls the first:

```
INT c;

PROC first;
BEGIN                       !Procedure body
  INT a,
      b;
  !Some code
  IF a < b THEN
    RETURN;
  c := a - b;
END;

PROC second;
BEGIN
  !Lots of code
  CALL first;               !Calls first procedure
  !More code
END;
```

2. The following example shows (1) a function that has two formal value parameters and (2) a procedure that invokes the function and passes actual parameters to it:

```
INT PROC mult (var1, var2);
  INT var1,                         !Formal specifications
      var2;                         ! for value parameters
BEGIN
  RETURN var1 * var2;
END;


PROC myproc;
BEGIN
  INT num1 := 5,
      num2 := 3,
      answer;
  answer := mult (num1, num2);     !Invokes function
END;
```

3. The following example shows a FORWARD declaration for "procb", a procedure that calls "procb" before its body is declared, and the declaration for the body of "procb":

```
INT g2;

PROC procb (param1);                !FORWARD declaration
  INT param1;                       ! for "procb"
FORWARD;

PROC proca;                         !Declares "proca"
BEGIN
  INT i1 := 2;
  CALL procb (i1);                  !Calls "procb"
END;

PROC procb (param1);                !Declares body for "procb"
  INT param1;
BEGIN
  g2 := g2 + param1;
END;

PROC mymain MAIN;
BEGIN
  g2 := 314;
  CALL proca;                      !Calls "proca"
END;
```

4. The following example shows how to include and invoke external operating system procedures:

```
?SOURCE $SYSTEM.SYSTEM.EXTDECS (DEBUG, STOP, . . . )

PROC a MAIN;
BEGIN
  INT x, y, z;

  !Code for manipulating x, y, and z

  If x = 5 THEN CALL STOP;      !Calls external procedure
  CALL DEBUG;
END;
```

5. The following example declares a procedure and a subprocedure that contain local and sublocal items with the same names:

```
PROC main2 MAIN;                !Declares procedure
BEGIN
  INT a := 4;                   !Declares local items
  INT b := 1;
  INT c;

  SUBPROC sub2 (param2);        !Declares subprocedure
    INT param2;
    BEGIN
      INT a := 5;               !Declares sublocal items
      INT b := 2;

      c := a + b + param2;      !Sublocal "a" and "b"
    END;                        !End of subprocedure

  a := a + b;                   !Local "a" and "b"
  CALL sub2 (a);
END;                            !End of procedure
```

ATTRIBUTES

Subprocedures can have only the VARIABLE attribute.

Procedures can have the following attributes:

```
MAIN | INTERRUPT
RESIDENT
CALLABLE
PRIV
VARIABLE | EXTENSIBLE
```

MAIN Attribute

This attribute causes the procedure to execute first when you run the program. When the MAIN procedure completes execution, control passes to the operating system STOP procedure.

If more than one procedure in a compilation has the MAIN attribute, TAL emits a warning and puts the MAIN attribute with the first MAIN procedure it sees. In the following example, "x" and "y" have the MAIN attribute in the source code, but only "x" has the MAIN attribute in the object file:

```
PROC x MAIN;              !This procedure is MAIN in object file
BEGIN
  CALL this^proc;
  CALL that^proc;
END;

PROC y MAIN;              !Second MAIN procedure is not MAIN in
BEGIN                     ! object file
  CALL some^proc;
END;
```

INTERRUPT Attribute

This attribute is used only by operating system interrupt handlers. It causes TAL to generate an IXIT (interrupt exit) instruction instead of an EXIT instruction at the end of execution. An example is:

```
PROC int^handler INTERRUPT;
BEGIN
  !Do some work
END;
```

RESIDENT Attribute

This 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 proca RESIDENT;
  BEGIN
    !Do some work
  END;
```

CALLABLE Attribute

CALLABLE means the procedure can execute privileged instructions, and a nonprivileged procedure can call it. It is the only way a nonprivileged program can become privileged. For information on privileged mode, see the System Description Manual. The following callable procedure calls a privileged procedure (described next):

```
PROC proc2 CALLABLE;
BEGIN
  CALL priv^proc;
END;
```

PRIV Attribute

PRIV means the procedure can execute privileged instructions, and only other privileged procedures can call it. PRIV protects the operating system from unauthorized calls to its internal procedures, as follows:

$$\begin{tikzcd}
\text{Nonprivileged Procedures} \arrow[r] & \text{CALLABLE Procedures} \arrow[r] & \text{PRIV Procedures}
\end{tikzcd}$$

(Application)   (Operating System)

The following privileged procedure is called by the callable procedure declared above:

```
PROC priv^proc PRIV;
BEGIN
  !Privileged instructions
END;
```

VARIABLE Attribute

This attribute means some or all of the procedure or subprocedure parameters are optional. TAL considers all the parameters to be optional, even if some are required by your code. The following example declares a VARIABLE procedure:

```
PROC v (a, b) VARIABLE;
  INT a, b;
BEGIN
  !Lots of code
END;
```

When a call to a VARIABLE procedure or subprocedure occurs, TAL allocates space in the parameter area for all the parameters and generates a parameter mask, which indicates those actually passed. The called procedure or subprocedure can use the $PARAM function to check for receipt of each parameter.

VARIABLE Parameter Mask

The parameter mask for a VARIABLE procedure or subprocedure has the following characteristics:

- Each formal parameter corresponds to one bit. For 16 or fewer parameters, TAL generates a single-word mask. For more than 16 parameters, TAL generates a doubleword mask.
- The mask is right justified. For a single-word mask, bit <15> corresponds to the last parameter. For a doubleword mask, bit <15> of the low-order word corresponds to the last parameter.
- For each passed parameter, TAL sets the corresponding bit to 1. For each omitted parameter, TAL sets the corresponding bit to 0.

For procedures, a single-word mask resides in location L[-3]; a doubleword mask resides in location L[-4:-3]. For subprocedures, either single-word or doubleword mask resides between the last parameter and the caller's return address.

Figure 16-1 shows an example of a single-word parameter mask for a VARIABLE procedure "zz", whose formal parameters correspond to mask bits <10:15>. The mask indicates which parameters are passed by procedure "aa".

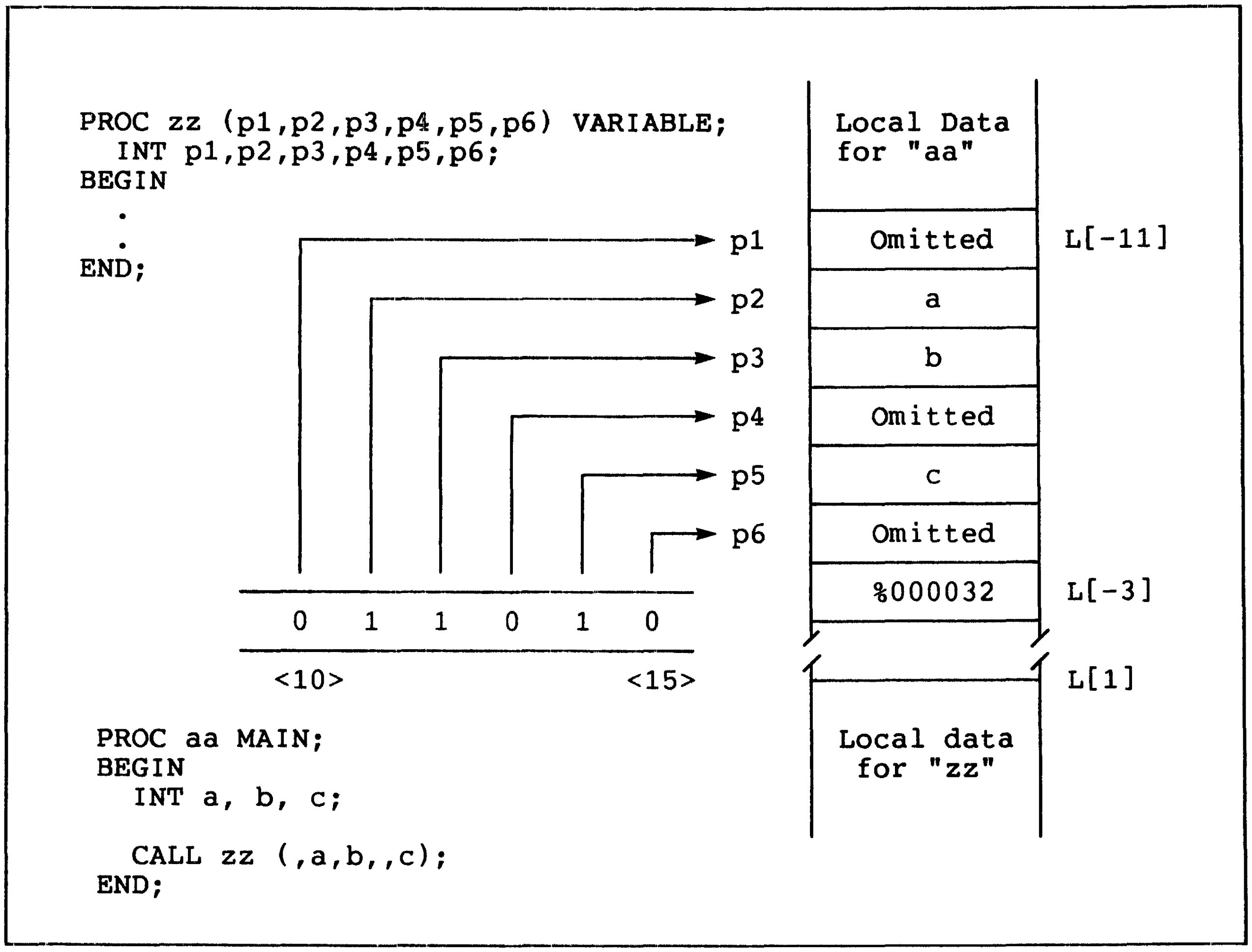


Figure 16-1. VARIABLE Single-Word Parameter Mask

Figure 16-2 shows a doubleword mask for the following example, in which a VARIABLE procedure declares 18 formal parameters, and another procedure passes five actual parameters to it.

```
INT aa, dd, ee, ff, jj;

PROC mask (a,b,c,d,e,f,g,h,i,j,k,l,m,n,o,p,q,r) VARIABLE;
  INT a,b,c,d,e,f,g,h,i,j,k,l,m,n,o,p,q,r;
BEGIN
  !Do more processing
END;

PROC caller;
BEGIN
  !Do processing
  CALL mask (aa,,,dd,ee,ff,,,,jj);
END;
```

```
Bit Numbers:  0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15
              ----------------------------------------------
L [-4]:       0  0  0  0  0  0  0  0  0  0  0  0  0  0  1  0
                                                         aa

L [-3]:       0  1  1  1  0  0  0  1  0  0  0  0  0  0  0  0
                 dd ee ff          jj
```

Figure 16-2. VARIABLE Doubleword Parameter Mask

Parameter Checking

The following example shows a VARIABLE procedure that contains parameter-checking statements:

```
PROC errmsg (msg, count, errnum) VARIABLE;
  INT .msg;                        !Required parameter
  INT count;                       !Required parameter
  INT errnum;                      !Optional parameter
BEGIN
  IF NOT $PARAM (msg) OR
     NOT $PARAM (count) THEN
     RETURN;                       !If required parameters missing
  IF NOT $PARAM (errnum) THEN
  errnum := 0;                     !Default for optional parameter
  !Process the error . . .
END;
```

EXTENSIBLE Attribute

EXTENSIBLE means you can later add new parameters to the procedure without recompiling the caller. An example declaration is:

```
PROC x (a, b) EXTENSIBLE;         !Declares EXTENSIBLE procedure
  INT a, b;
BEGIN
  !Do some work
END;
```

TAL considers all parameters of an EXTENSIBLE procedure to be optional, even if some are required by your code. When a call to an EXTENSIBLE procedure occurs, TAL allocates space in the parameter area for all the parameters and generates a parameter mask, which indicates those actually passed. The called procedure can use the $PARAM function to check for a passed parameter, as was described for VARIABLE procedures.

A new procedure with or without parameters can be extensible. An existing procedure that has no parameters cannot become extensible. An existing VARIABLE procedure can become extensible as follows.

Converting Procedures From VARIABLE to EXTENSIBLE

A VARIABLE procedure can become extensible only if:

- It has 15 or fewer parameters.
- It has 16 or fewer words of parameters.
- All parameters, except the last parameter, are one word long.

When converting a VARIABLE procedure, the required form for the EXTENSIBLE attribute is:

```
EXTENSIBLE ( <param-count> )

<param-count>

   an INT arithmetic expression in the range 1 through 15 that
   defines the number of parameters declared when the procedure
   was VARIABLE.
```

The following example converts an existing VARIABLE procedure to an ENTENSIBLE procedure:

```
PROC errmsg (msg, count, errnum, new^param) EXTENSIBLE (3);
  INT .msg;                      !Required parameter
  INT count;                     !Required parameter
  INT errnum;                    !Optional parameter
  INT new^param;                 !New optional parameter
BEGIN
 !Do something
END;
```

EXTENSIBLE Parameter Mask

The format for EXTENSIBLE parameter masks differs from that of VARIABLE procedure masks, as follows:

- Each formal parameter corresponds to one or more bits, depending on the size of the parameter. Each bit represents one word of a parameter.

- The mask is left justified. For a single-word mask, bit <0> corresponds to the first parameter if it is a single word. For a doubleword mask, bit <0> of the low-order word corresponds to the first parameter.

- For each passed parameter, TAL sets all the bits for that parameter to 1. For each omitted parameter, TAL sets the corresponding bits to 0. The $PARAM function checks only the high-order bit that corresponds to a parameter. (Word parameters have only one corresponding bit.)

Figure 16-3 shows a single-word mask for the following example in which an EXTENSIBLE procedure declares INT, INT(32), and FIXED formal parameters. The seven formal parameters occupy 12 parameter words. Another procedure passes four actual parameters to it.

```
INT aa, ff, gg;
FIXED cc;

PROC baz (a,b,c,d,e,f,g) EXTENSIBLE;
  INT     a,d,f,g;
  INT(32) b,e;
  FIXED   c;
BEGIN
  !Code for processing
END;

PROC maz;
BEGIN
  !Code for processing
  CALL baz (aa,,cc,,,ff,gg);
END;
```

```
Bit Numbers:   0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15
               ----------------------------------------------
L [-3]:        1  0  0  1  1  1  1  0  0  0  1  1  0  0  0  0
               aa       (    cc   )           ff gg
```

Figure 16-3. EXTENSIBLE Single-Word Parameter Mask

Figure 16-4 shows a doubleword mask for the following example in which an EXTENSIBLE procedure declares INT, INT(32), and FIXED formal parameters. The 12 formal parameters occupy 20 parameter words. Another procedure passes five actual parameters to it.

```
INT aa, ff, gg;
FIXED cc;
INT(32) jj;

PROC baz (a,b,c,d,e,f,g,h,i,j,k,l) EXTENSIBLE;
  INT     a,d,f,g,k,l;
  INT(32) b,e,h,i,j;
  FIXED   c;
BEGIN
  !Do more work
END;

PROC maz;
BEGIN
  !Do some work
  CALL baz (aa,,cc,,,ff,gg,,,jj);
END;
```

```
Bit Numbers:   0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15
               ----------------------------------------------
L [-4]:        1  0  0  1  1  1  1  0  0  0  1  1  0  0  0  0
               aa       (    cc   )           ff gg

L [-3]:        1  1  0  0  0  0  0  0  0  0  0  0  0  0  0  0
               (jj)
```

Figure 16-4. EXTENSIBLE Doubleword Parameter Mask

Number of Parameter Words Passed

In addition to the parameter mask, TAL generates a one-word value that represents the number of parameter words passed to the EXTENSIBLE procedure. TAL stores the negative form of that value in the parameter area immediately preceding the three-word stack marker. For example, if four parameter words are passed, TAL stores -4.

Procedure Entry Sequence

On entry to an EXTENSIBLE procedure, the system loads the following values on the register stack:

- For a converted VARIABLE procedure:

  --R[0] = Number of parameters when the procedure was VARIABLE

  --R[1] = Number of parameter words when it was VARIABLE

  --R[2] = Number of parameter words now expected

  RP must be 2.

- For a procedure that was EXTENSIBLE to begin with:

  --R[0] = Number of parameter words expected

  RP must be 0.

The system then executes the ESE instruction, which uses the RP setting to tell the cases apart. ESE sets RP to 7 but does not save the values in R0 through R7.

For a converted VARIABLE procedure, ESE converts the mask format to the EXTENSIBLE format. It adds the needed bits and words and initializes them to 0. It does not initialize any extra words on the register stack caused by the stack movement.

PARAMETERS

Each parameter requires a formal parameter name and a formal parameter specification.

Formal parameter names provide identifiers that have local scope if declared in a procedure body or sublocal scope if declared in a subprocedure body. When a call occurs, each formal parameter assumes the value of the corresponding passed parameter.

A formal parameter specification defines the parameter type of a formal parameter and whether it is a value or reference parameter.

Parameter Area

The calling procedure enters the actual parameter values in the parameter area before transferring control to the called procedure or subprocedure.

For procedures, the parameter area limit is 29 words, less storage required for a single-word or doubleword parameter mask, if present. For EXTENSIBLE procedures, a word value representing the number of parameter words passed also resides in the parameter area. Thus, the space available for the parameters of a procedure is:

	Parameter Words
VARIABLE procedure with single-word mask	28
VARIABLE procedure with doubleword mask	27
EXTENSIBLE procedure with single-word mask	27
EXTENSIBLE procedure with doubleword mask	26
Any other procedure	29

For subprocedures, the parameter area limit is 30 words, less storage required for sublocal variables and for a single-word or doubleword VARIABLE parameter mask, if any.

Figure 16-5 shows an example of parameter storage.

```
INT .buffer[0:20];

PROC b (parm1, parm2);
  INT parm1, parm2;
BEGIN
  INT b^local^array[0:12];

  buffer[0] := parm1 + parm2;
  buffer[1] := parm1 - parm2;
  buffer[2] := parm1 * parm2;
END;  !End of "b"

PROC a MAIN;
BEGIN
  INT first,
      second,
      a^local^array[0:2];

  CALL b (first, second);
END;  !End of "a"
```

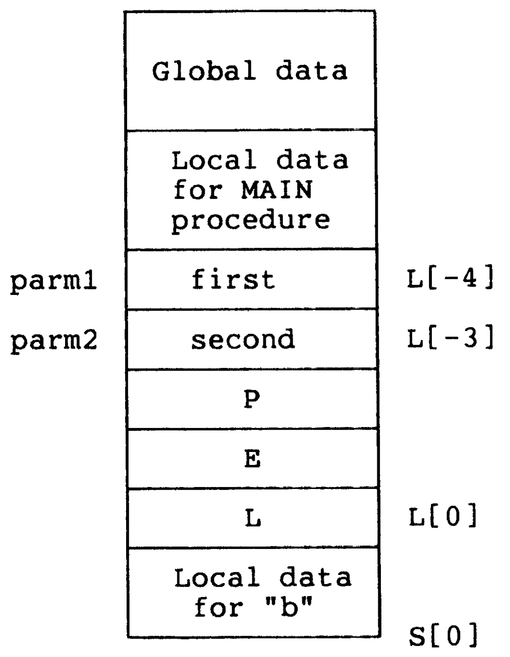


Figure 16-5. Parameter Storage

Value Parameters

If a procedure or subprocedure specifies a formal parameter without an indirection symbol ("." or ".EXT"), you pass a value parameter. (Structures and arrays must be reference parameters.)

Statements in the called procedure body access the passed value parameter directly in the parameter area. They can use a value parameter as working space within the procedure without affecting the actual variables used to generate the value for that parameter.

A passed value parameter can be any arithmetic expression. The formal parameter specification of the called procedure defines the data type and storage allocation for the passed parameter (one word for INT and STRING, two words for INT(32) and REAL, and four words for REAL(64) and FIXED).

The system right justifies STRING value parameters in the parameter area as if they were INT expressions. If you want to left justify a STRING parameter in bits <0:7> of the word, shift the value 8 bits to the left when you call the procedure; for example:

```
   CALL proc1 (byte '<<' 8)
```

FIXED Value Parameters

The system scales FIXED value parameters up or down to match the <fpoint> in the parameter specification. If the <fpoint> of the passed parameter is greater than the <fpoint> in the parameter specification, precision is lost according to the amount of scaling required.

To prevent scaling of the <fpoint> of the passed parameter, you can use a parameter type of FIXED (*). The called procedure treats the parameter as having an <fpoint> of 0.

Procedures as Value Parameters

A procedure can declare a procedure as a formal parameter. TAL treats the identifier associated with a parameter type PROC as a procedure name within the procedure body. TAL allocates one word in the parameter area for the PEP number of the procedure to be passed.

If the passed procedure itself has parameters, you must make certain that all parameters are supplied. TAL cannot perform this check. If the passed procedure is VARIABLE or EXTENSIBLE, you must supply the correct parameter mask. TAL treats any missing parameters in the CALL statement as type INT value parameters.

If the passed procedure has reference parameters, each must be preceded by an @ symbol in the call.

The following example shows a procedure passed as a value parameter:

```
PROC a(t);               !Declares procedure to be passed
  STRING .t;
BEGIN
  t ':=' "NO MAN IS AN ISLAND.";
END;

PROC p(q);               !Declares procedure to be called
  PROC q;                !Formal PROC parameter specification
BEGIN
  STRING .s[0:20];
  CALL q(@s);            !Calls "q" and passes address of array "s";
END;                     ! "s" gets "NO MAN IS AN ISLAND."

PROC m MAIN;
BEGIN
  CALL p(a);             !Calls "p" and passes procedure "a" as a
END;                     ! parameter
```

Reference Parameters

If a procedure specifies a formal parameter with an indirection symbol (. or .EXT), you pass a reference parameter. TAL allocates storage in the parameter area for the address of the variable (one word for a standard pointer and two words for an extended pointer). If required by the procedure, TAL converts standard addresses to extended addresses. Converting extended addresses to standard addresses, however, is an error since the segment information in the extended pointer is lost.

To pass a parameter by reference, place the name of the variable in the CALL statement. TAL generates the address of the variable and places it in the parameter area. Statements within the called procedure access the actual variable indirectly through the parameter location. The called procedure can store values in reference parameters and modify the actual variables.

The caller can change the contents of a pointer by prefixing the pointer name with an @ symbol and passing it by reference. The following example shows how this is done:

```
PROC p ( ptr );
  INT .ptr;
BEGIN
  ptr := %100000;
END;

PROC q;
BEGIN
  INT .upper32k;
  CALL p ( @upper32K );
END;
```

Arrays and structures must be reference parameters. The previous example in "Procedures as Value Parameters" specifies array "s" as a formal reference parameter.

FIXED Reference Parameters

For FIXED reference parameters, the <fpoint> of the passed parameter must match the <fpoint> in the parameter specification. If they do not match, TAL does not perform scaling and issues a warning. The statements in the called procedure then apply the <fpoint> in the formal parameter specification to the passed parameter.

Mixing Data Types of Formal and Actual Parameters

You can pass a non-STRING parameter to a formal reference parameter that has a standard byte address. TAL converts the word address of the actual parameter to a byte address.

You can pass a STRING variable to formal reference parameter that has a word address. However, when TAL converts the byte address of the actual parameter to a word address by right shifting, the byte number is lost. If the actual variable is aligned on an even-byte boundary, this is no problem, but if it is aligned on an odd-byte boundary, you access a byte outside the variable. TAL issues a warning message that right shifting occurred.

ENTRY-POINT DECLARATION

The entry-point declaration associates a name with a secondary location in a procedure or subprocedure where execution can start.

The syntax for the entry-point declaration is:

```
ENTRY  <entry-point-name> [ , <entry-point-name> ] ... ;


<entry-point-name>

   is the name of an entry point in the procedure or subprocedure
   body.  It is an alternate name to use when invoking the
   procedure or subprocedure.
```

Procedure and subprocedure entry points are discussed separately below.

Procedure Entry Points

The following rules apply:

1. Declare all entry-point names for a procedure within the procedure.
2. Place each entry-point name and a colon (:) at a point in the procedure at which execution is to start.
3. To invoke an entry point, reference its name in a CALL statement located in any procedure or subprocedure. Include any actual parameters as if you were calling the procedure name. (See Example 1.)
4. A GOTO statement to an entry point is not allowed.
5. You can declare FORWARD or EXTERNAL procedure entry points. The syntax is the same as for a FORWARD or EXTERNAL procedure declaration, except that you specify the name of the entry point. The declaration must include all formal parameters and parameter specifications declared for the procedure. (See Example 2.)

Each time you invoke a procedure entry point, all local variables receive their initial values.

For a procedure entry point, the reference @<entry-point-name> yields the PEP number of the entry point.

Subprocedure Entry Points

The following rules apply:

1. Declare all entry-point names for a subprocedure within the subprocedure.

2. Place each entry-point name and a colon (:) at a point in the subprocedure at which execution is to start.

3. To invoke an entry point, reference its name in a CALL statement located anywhere in the encompassing procedure, such as in another subprocedure in the same scope. Include any actual parameters as if you were calling the subprocedure name.

4. A GOTO statement to an entry point is not allowed.

5. You can declare FORWARD subprocedure entry points. The syntax is the same as for a FORWARD subprocedure declaration except that you specify the name of the entry point. The declaration must include all formal parameters and parameter specifications declared for the subprocedure.

Each time a you invoke a subprocedure entry point, all sublocal variables receive their initial values.

For a subprocedure entry point, the reference @<entry-point-name> yields the code address of the entry point.

Examples

1. This example illustrates use of entry points:

```
INT to^this := 314;             !Global data declaration

PROC add^3 (g2);
  INT .g2;
BEGIN
        ENTRY add^2, add^1;     !Declares entry points
        INT m2 := 1;
        g2 := g2 + m2;
add^2 : g2 := g2 + m2;          !Entry-point location
add^1 : g2 := g2 + m2;          !Entry-point location
END;

PROC mymain MAIN;               !Main procedure
BEGIN
  CALL add^1 (to^this);         !Calls entry point
END;
```

2. This example shows FORWARD declarations for entry points:

```
INT to^this := 314;

PROC add^1 (g2);                !Declares a FORWARD entry point
  INT .g2;
FORWARD;

PROC add^2 (g2);                !Declares a FORWARD entry point
  INT .g2;
FORWARD;

PROC add^3 (g2);                !Declares a FORWARD procedure
  INT .g2;
FORWARD;

PROC mymain MAIN;               !Main procedure
BEGIN
  CALL add^1 (to^this);         !Calls entry point
END;

PROC add^3 (g2);                !Body for FORWARD procedure
  INT .g2;
BEGIN
        ENTRY add^2, add^1;     !Declares entry points
        INT m2 := 1;
        g2 := g2 + m2;
add^2 : g2 := g2 + m2;          !Entry-point location
add^1 : g2 := g2 + m2;          !Entry-point location
END;
```

SECTION 17

STANDARD FUNCTIONS

TAL provides a variety of standard functions that perform frequently used operations.

This section contains:

- A summary of standard functions, organized by operational group
- The syntax of each function, listed in alphabetic order, and the need for optional microcode, if any

STANDARD FUNCTIONS BY OPERATIONAL GROUP

Functions are summarized within the following groups:

- Type Transfer
- Address Conversion
- Character Test
- Minimum-Maximum
- Carry and Overflow Test
- Fixed-Point Value and Scale
- Structure
- Parameter-Checking and Register Pointer
- Miscellaneous

Type Transfer

The type-transfer functions convert a variable of one data type into a variable of another data type. As indicated, some functions apply rounding to the result. This means if the least significant digit is less than 5, it is truncated; otherwise, the result is rounded up.

$DBL	returns a signed INT(32) value from an INT, FIXED(0), REAL, or REAL(64) expression.
$DBLL	returns an INT(32) value from two INT values.
$DBLR	returns a signed INT(32) value from an INT, FIXED(0), REAL, or REAL(64) expression and applies rounding to the result.
$DFIX	returns a 64-bit integer from a signed doubleword integer (the equivalent of a signed right shift of 32 positions).
$EFLT	returns a REAL(64) value from an INT, INT(32), FIXED, or REAL expression.
$EFLTR	returns a REAL(64) value from an INT, INT(32), FIXED, or REAL expression and applies rounding to the result.
$FIX	returns a FIXED(0) value from an INT, INT(32), REAL, or REAL(64) expression and applies rounding to the value.
$FIXD	returns the INT(32) equivalent of a FIXED expression treated as a 64-bit integer.
$FIXI	returns the signed INT equivalent of a FIXED expression treated as a 64-bit integer.
$FIXL	returns the unsigned INT equivalent of a FIXED expression treated as a 64-bit integer.
$FIXR	returns a FIXED(0) value from an INT, INT(32), REAL, or REAL(64) expression and applies rounding to the result.
$FLT	returns a REAL value from an INT, INT(32), FIXED, or REAL(64) expression.
$FLTR	returns a REAL value from an INT, INT(32), FIXED, or REAL(64) expression and applies rounding to the result.
$HIGH	returns an INT value from the left half of an INT(32) expression.
$IFIX	returns a 64-bit integer from a signed INT expression (the equivalent of a signed right shift of 48 positions).
$INT	returns an INT value from INT(32), FIXED(0), REAL, or REAL(64) expression.

$INTR	returns an INT value from an INT(32), FIXED(0), REAL, or REAL(64) expression and applies rounding to the result.
$LFIX	returns a 64-bit integer from an unsigned INT expression.
$UDBL	returns an INT(32) value from an unsigned INT expression.

Table 17-1 cross-references the type-transfer functions according to data type:

Table 17-1. Type-Transfer Functions by Data Type

	TO				
FROM	INT	INT(32)	FIXED	REAL	REAL(64)
INT	-	$DBL $UDBL	$IFIX $LFIX	$FLT $FLTR	$EFLT $EFLTR
INT(32)	$INT $HIGH	-	$DFIX	$FLT $FLTR	$EFLT $EFLTR
FIXED	$FIXI $FIXL	$FIXD	-	$FLT $FLTR	$EFLT $EFLTR
REAL	$INT $INTR	$DBL $DBLR	$FIX $FIXR	-	$EFLT $EFLTR
REAL(64)	$INT $INTR	$DBL $DBLR	$FIX $FIXR	$FLT $FLTR	-

Address Conversion

These functions convert standard addresses to extended addresses or extended addresses to standard addresses.

$XADR	converts a standard address to an extended address.
$LADR	converts an extended address to a standard address.

Character Test

These functions test for an alphabetic, a numeric, or a special (nonalphanumeric) ASCII character. They return a true value if the character passes the test or a false value if the character fails. You typically use these functions in conditional expressions to direct the flow of program execution.

$ALPHA	tests an expression for an alphabetic character.
$NUMERIC	tests an expression for a numeric character.
$SPECIAL	tests an expression for a special character.

Minimum-Maximum

These functions return the maximum or the minimum of two expressions.

$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, REAL, or REAL(64) expressions of the same type.
$MIN	returns the minimum of two signed INT, INT(32), FIXED, REAL, or REAL(64) expressions of the same type.

Carry and Overflow Test

These functions check the state of the carry or overflow indicator in the ENV register. They return a true value if the indicator is on or a false value if it is off. Typically, you use these functions in conditional expressions to direct the flow of program execution.

$CARRY	tests the state of the carry indicator.
$OVERFLOW	tests the state of the overflow indicator.

Fixed-Point Value and Scale

These functions assist you in manipulating FIXED expressions.

$POINT	returns the <fpoint> value, in integer form, associated with a FIXED expression.
$SCALE	moves the position of the implied decimal point by adjusting the internal representation of the expression.

Structure

These functions return information about previously defined data structures.

$LEN	returns the unit length in bytes of a variable.
$OCCURS	returns the number of occurrences of a STRUCT item.
$OFFSET	returns the offset in bytes of a structure item from the structure base.
$TYPE	returns a value indicating the type of a variable.

Parameter-Checking and Register Pointer

These functions check for the presence or absence of a parameter in a procedure or subprocedure call or return the current setting of the TAL register pointer.

$PARAM	checks for the presence or absence of a parameter in a procedure or subprocedure call.
$RP	returns the current setting of the TAL register pointer.

Miscellaneous

These functions return the absolute value or the one's complement of an expression.

$ABS	returns the absolute value of an expression.
$COMP	returns the one's complement of an INT expression.

$ABS FUNCTION

The $ABS function returns the absolute value of an expression. The returned value has the same data type as the expression.

The syntax for the $ABS function is:

```
$ABS ( <expression> )


<expression>

   is an expression of any type as defined in Section 13 of
   this manual.
```

$ABS sets the overflow indicator if the absolute value of a negative number cannot be represented in two's complement or real format (depending on the type of the expression). For example, $ABS (-32768) causes an arithmetic overflow.

Example

This example assigns the absolute value of "i2" to "j2". Since "i2" is equal to -5, "j2" receives the absolute value of (-5), which is 5.

```
INT i2 := -5,
INT j2;
j2 := $ABS(i2);     !Sets "j2" equal to absolute value of (-5)
```

$ALPHA FUNCTION

The $ALPHA function tests the right half of an INT value for the presence of an alphabetic character.

The syntax for the $ALPHA function is:

```
$ALPHA ( <int-expression> )


<int-expression>

   is an INT expression.  $ALPHA inspects bits <8:15> of
   <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"
```

$ALPHA sets the condition code indicator to "=" if an alphabetic character occurs. If you plan to check the condition code, you must 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).

Example

This example tests for an alphabetic character in expression "some^char":

```
   STRING some^char;
   IF $ALPHA (some^char) THEN . . . ;
```

$CARRY FUNCTION

The $CARRY function checks the state of the carry bit in the ENV register.

The syntax for the $CARRY function is:

```
$CARRY
```

If the carry bit is on, $CARRY returns a -1 (true); otherwise, it returns a 0 (false).

Example

This example tests the state of the carry bit:

```
IF $CARRY THEN . . . ;
```

For additional examples, see the SCAN statement in Section 15.

$COMP FUNCTION

The $COMP function obtains the one's complement of an INT expression.

The syntax for the $COMP function is:

```
$COMP ( <int-expression> )

<int-expression>

   is an INT expression.
```

Example

This example assigns "some^int" a value equal to the one's complement of 10:

```
INT some^int;

some^int := $COMP (10);
```

$DBL FUNCTION

The $DBL function returns a signed INT(32) value from an INT, FIXED(0), REAL, or REAL(64) expression.

The syntax for the $DBL function is:

```
$DBL ( <expression> )

<expression>

   is an INT, FIXED(0), REAL, or REAL(64) expression.
```

$DBL sets the overflow indicator if the expression is too large in magnitude to be represented by a 32-bit two's complement integer.

This function needs the following optional microcode:

System	FIXED	REAL	REAL(64)
NonStop 1+	QLD CQD	CFD	QLD CED
NonStop	CQD	CFD	CED

Example

This example converts the INT variable "i2" into a signed INT(32) value and assigns the result to the INT(32) variable "b32":

```
INT      i2 := %177775;
INT(32) b32;
b32 := $DBL(i2);
```

$DBLL FUNCTION

The $DBLL function returns an INT(32) value from two INT values.

The syntax for the $DBLL function is:

```
$DBLL ( <int-expression> , <int-expression> )

<int-expression>

   is an INT expression.
```

To form the INT(32) value, $DBLL places the first INT value in the upper 16 bits and the second INT value in the lower 16 bits.

Examples

1. This example returns the INT(32) value formed from "first^int" and "second^int":

```
INT first^int, second^int;       !Declares variables
INT(32) some^double;

some^double := $DBLL (first^int, second^int);
```

2. This example returns an extended (32-bit) address in the current user code segment:

```
INT .EXT p;                     !Declares extended pointer

@p := ($DBLL (2, 7)) '<<' 1;  !Assigns address in code segment
```

$DBLR FUNCTION

The $DBLR function returns a signed INT(32) value from an INT, FIXED(0), REAL, or REAL(64) expression and applies rounding to the result.

The syntax for the $DBLR function is:

```
$DBLR ( <expression> )

<expression>

   is an INT, FIXED(0), REAL, or REAL(64) expression.
```

$DBLR sets the overflow indicator if the expression is too large in magnitude to be represented by a 32-bit two's complement integer.

This function needs the following optional microcode:

System	FIXED	REAL	REAL(64)
NonStop 1+	QLD CQD	CFDR	QLD CEDR
NonStop	CQD	CFDR	CEDR

$DFIX FUNCTION

The $DFIX function returns a 64-bit integer from a signed INT(32) expression.

The syntax for the $DFIX function is:

```
$DFIX ( <dbl-expression> , <fpoint> )

<dbl-expression>

   is a signed INT(32) arithmetic expression.

<fpoint>

   is a value in the range -19 through +19 that specifies the
   position of the implied decimal point, as described in
   Section 8 under "Simple Variable Declaration."
```

$DFIX converts a signed INT(32) expression to a 64-bit integer by performing the equivalent of a signed right shift of 32 positions.

This function needs the following optional microcode:

System	INT(32)	FIXED
NonStop 1+	CDQ	QUP QDWN
NonStop	CDQ	

$EFLT FUNCTION

The $EFLT function returns a REAL(64) value from an INT, INT(32), FIXED, or REAL expression.

The syntax for the $EFLT function is:

```
$EFLT ( <expression> )

<expression>

   is an INT, INT(32), FIXED, or REAL expression.
```

This function needs the following optional microcode:

System	INT	INT(32)	FIXED	REAL	REAL(64)
NonStop 1+	CIE	CDE	QLD CQE	CFE	QLD
NonStop	CIE	CDE	CQE	CFE	

$EFLTR FUNCTION

The $EFLTR function returns a REAL(64) value from an INT, INT(32), FIXED, or REAL expression and applies rounding to the result.

The syntax for the $EFLTR function is:

```
$EFLTR ( <expression> )

<expression>

   is an INT, INT(32), FIXED, or REAL expression.
```

This function needs the following optional microcode:

System	INT	INT(32)	FIXED	REAL	REAL(64)
NonStop 1+	CIE	CDE	QLD CQER	CFE	QLD
NonStop	CIE	CDE	CQER	CFE	

$FIX FUNCTION

The $FIX function returns a FIXED(0) value from an INT, INT(32), REAL, or REAL(64) expression.

The syntax for the $FIX function is:

```
$FIX ( <expression> )

<expression>

   is an INT, INT(32), FIXED, REAL or REAL(64) expression.
```

$FIX sets the overflow indicator if the expression is too large in magnitude to be represented by a 64-bit two's complement integer.

This function needs the following optional microcode:

System	INT	INT(32)	FIXED	REAL	REAL(64)
NonStop 1+	CIQ	CDQ	QLD	CFQ	QLD CEQ
NonStop	CIQ	CDQ		CFQ	CEQ

Example

This example initializes a FIXED variable with the value that the $FIX function returns from an INT value:

```
INT local1;
FIXED local := $FIX(local1);
```

$FIXD FUNCTION

The $FIXD function returns an INT(32) value from a FIXED expression.

The syntax for the FIXD function is:

```
$FIXD ( <fixed-expression> )

<fixed-expression>

   is a FIXED expression, which $FIXD treats as a 64-bit integer
   ignoring any implied decimal point.
```

$FIXD sets the overflow indicator if the result cannot be represented in a signed doubleword.

This function needs the following optional microcode:

System	FIXED
NonStop 1+	QLD CQD
NonStop	CQD

$FIXI FUNCTION

The $FIXI function returns the signed INT equivalent of a FIXED expression.

The syntax for the $FIXI function is:

```
$FIXI ( <fixed-expression> )

<fixed-expression>

   is a FIXED expression, which $FIXI treats as a 64-bit integer,
   ignoring any implied decimal point.
```

$FIXI sets the overflow indicator if the result cannot be represented in a signed 16-bit integer.

This function needs the following optional microcode:

System	FIXED
NonStop 1+	QLD CQI
NonStop	CQI

$FIXL FUNCTION

The $FIXL function returns the unsigned INT equivalent of a FIXED expression.

The syntax for the $FIXL function is:

```
$FIXL ( <fixed-expression> )

<fixed-expression>

   is a FIXED expression, which $FIXL treats as a 64-bit integer,
   ignoring any implied decimal point.
```

$FIXL sets the overflow indicator if the result cannot be represented in an unsigned 16-bit integer.

This function needs the following optional microcode:

System	FIXED
NonStop 1+	QLD CQL
NonStop	CQL

$FIXR FUNCTION

The $FIXR function returns a FIXED(0) value from an INT, INT(32), REAL, or REAL(64) expression and applies rounding to the result.

The syntax for the $FIXR function is:

```
$FIXR ( <expression> )

<expression>

   is an INT, INT(32), REAL, or REAL(64) expression.
```

$FIXR sets the overflow indicator if <expression> is too large in magnitude to be represented by a 64-bit two's complement integer.

This function needs the following optional microcode:

System	INT	INT(32)	FIXED	REAL	REAL(64)
NonStop 1+	CIQ	CDQ	QLD	CFQR	QLD CEQR
NonStop	CIQ	CDQ		CFQR	CEQR

$FLT FUNCTION

The $FLT function returns a REAL value from an INT, INT(32), FIXED, or REAL(64) expression.

The syntax for the $FLT function is:

```
$FLT ( <expression> )

<expression>

   is an INT, INT(32), FIXED, or REAL(64) expression.
```

This function needs the following optional microcode:

System	INT	INT(32)	FIXED	REAL(64)
NonStop 1+	CIF	CDF	QLD CQF	QLD CEF
NonStop	CIF	CDF	CQF	CEF

$FLTR FUNCTION

The $FLTR function returns a REAL value from an INT, INT(32), FIXED, or REAL(64) expression and applies rounding to the result.

The syntax for the $FLTR function is:

```
$FLTR ( <expression> )


<expression>

   is an INT, INT(32), REAL, or REAL(64) expression.
```

This function needs the following optional microcode:

System	INT	INT(32)	FIXED	REAL(64)
NonStop 1+	CIF	CDFR	QLD CQFR	QLD CEFR
NonStop	CIF	CDFR	CQFR	CEFR

$HIGH FUNCTION

The $HIGH function returns an INT value from the left half of an INT(32) expression.

The syntax for the $HIGH function is:

```
$HIGH ( <dbl-expression> )

<dbl-expression>

   is an INT(32) expression.
```

Example

This example assigns the high-order word of "a32" to "num":

```
INT num;
INT(32) a32 := 65538D;

num := $HIGH (a32);
```

$IFIX FUNCTION

The $IFIX function returns a FIXED value from a signed INT expression.

The syntax for the $IFIX function is:

```
$IFIX ( <int-expression> , <fpoint> )

<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, as described in
   Section 8 under "Simple Variable Declaration."
```

When $IFIX converts the signed INT expression to a FIXED value, it performs the equivalent of a right shift of 48 positions.

This function needs the following optional microcode:

System	INT
NonStop 1+	CIQ
NonStop	CIQ

$INT FUNCTION

The $INT function returns an INT value from an INT(32), FIXED(0), REAL, or REAL(64) expression.

The syntax for the $INT function is:

```
$INT ( <expression> )

<expression>

   is an INT(32), FIXED(0), REAL, or REAL(64) expression.
```

If <expression> is type INT(32), $INT returns the low-order (least significant) 16 bits, and no overflow occurs.

If <expression> is not type INT(32), $INT sets the overflow indicator if <expression> is too large in magnitude to be represented by a 16-bit two's complement integer.

This function needs the following optional microcode:

System	FIXED	REAL	REAL(64)
NonStop 1+	QLD CQI	CFI	QLD CEI
NonStop	CQI	CFI	CEI

Example

The following example assigns the low-order word of "a32" to "lnum":

```
INT lnum;
INT(32) a32 := 65538D;

lnum := $INT (a32);
```

$INTR FUNCTION

The $INTR function returns an INT value from an INT(32), FIXED(0), REAL, or REAL(64) expression and applies rounding to the result.

The syntax for the $INTR function is:

```
$INTR ( <expression> )

<expression>

   is an INT(32), FIXED(0), REAL, or REAL(64) expression.
```

If <expression> is type INT(32), $INT returns the low-order (least significant) 16 bits, and no overflow occurs.

If <expression> is not type INT(32), $INT sets the overflow indicator if <expression> is too large in magnitude to be represented by a 16-bit two's complement integer.

The following optional mircrocode is required:

System	FIXED	REAL	REAL(64)
NonStop 1+	QLD CQI	CFIR	QLD CEIR
NonStop	CQI	CFIR	CEIR

$LADR FUNCTION

The $LADR function obtains the standard address of a variable that is accessed through an extended pointer.

The syntax for the $LADR function is:

```
$LADR ( <variable> )


<variable>

   is a variable accessed through an extended pointer.  If
   <variable> is type STRING or a substructure, the standard
   address is a byte address; otherwise, it is a word address.
```

When $LADR converts the extended address to a standard address, it loses the segment number in the extended address. For a description of the extended address format, see Appendix A.

Example

This example initializes a standard pointer with the standard address $LADR returns from the extended address:

```
STRING .EXT eptr := %100000D;   !Declares extended pointer
STRING .sptr := $LADR (eptr);   !Declares standard pointer and
                                ! initializes it with converted
                                ! standard address
```

$LEN FUNCTION

The $LEN function returns the unit length (in bytes) of a variable.

The syntax for the $LEN function is:

```
$LEN ( <variable> )

<variable>

   is the name of a STRUCT item (a structure, substructure, or
   STRUCT data item), as defined in Section 11.
```

For a structure or substructure, $LEN returns a unit length that is the sum of the lengths of its subordinate items. Because $LEN always returns a constant value, you can use it in LITERAL expressions.

For a non-STRUCT item, $LEN returns the number of bytes in the item.

Example

This example returns the length of one occurrence of a structure:

```
INT s^len;

STRUCT .s[0:3];              !Declares four occurrences of a
  BEGIN                      ! structure
  STRING  array1[0:49];
  INT(32) array2[0:199];
  END;

s^len := $LEN (s[0]);      !Returns length of first occurrence
```

For other examples, see "Structure Functions" in Section 11 and "Move Statement" in Section 15.

$LFIX FUNCTION

The $LFIX function returns a 64-bit integer from an unsigned INT expression.

The syntax for the $LFIX function is:

```
$LFIX  ( <int-expression> , <fpoint> )

<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, as described in
   Section 8 under "Simple Pointer Declaration."
```

$LFIX places the INT value in the least significant word of the quadword and sets the three most significant words to 0.

This function needs the following optional microcode:

System	INT
NonStop 1+	CLQ
NonStop	CLQ

$LMAX FUNCTION

The $LMAX function returns the maximum of two unsigned INT expressions.

The syntax for the $LMAX function is:

```
$LMAX ( <int-expression> , <int-expression> )

<int-expression>

   is an unsigned INT arithmetic expression.
```

Example

This example returns the maximum of "integer1" and "integer2" and assigns that value to "max":

```
   INT max;
   INT integer1 := 68 '+' 125 '-' 43;
   INT integer2 := 279 '-' 131;       !Data declarations

   max := $LMAX(integer1, integer2);  !Returns maximum value
```

$LMIN FUNCTION

The $LMIN function returns the minimum of two unsigned INT expressions.

The syntax for the $LMIN function is:

```
$LMIN ( <int-expression> , <int-expression> )

<int-expression>

   is an unsigned INT arithmetic expression.
```

Example

This example returns the minimum of "integer1" and "integer2" and assigns that value to "min":

```
INT min;
INT integer1 := 99 '-' 23;
INT integer2 := 41 '+' 19;          !Data declarations

min := $LMIN(integer1, integer2);    !Returns minimum value
```

$MAX FUNCTION

The $MAX function returns the maximum of two signed INT, INT(32), FIXED, REAL, or REAL(64) expressions.

The syntax for the $MAX function is:

```
$MAX ( <expression> , <expression> )


<expression>

   is a signed INT, INT(32), FIXED, REAL, or REAL(64)
   expression.  Both expressions must be the same type.
```

Example

This example returns the maximum of "exp1" and "exp2" and assigns that value to "max":

```
   REAL max;
   REAL exp1 := 8.3E-1;
   REAL exp2 := 8.2E5;            !Data declarations

   max := $MAX(exp1, exp2);       !Returns maximum value
```

$MIN FUNCTION

The $MIN function returns the minimum of two INT, INT(32), FIXED, REAL, or REAL(64) expressions.

The syntax for the $MIN function is:

```
$MIN ( <expression> , <expression> )


<expression>

   is an INT, INT(32), FIXED, REAL, or REAL(64) expression.
   Both expressions must be of the same type.
```

Example

This example returns the minimum of "exp1" and "exp2" and assigns that value to "min":

```
FIXED(3) min;
FIXED(3) exp1 := 129.653F;
FIXED(3) exp2 := 873.381F;                    !Data declarations

min := $MIN(exp1, exp2);                      !Returns minimum value
```

$NUMERIC FUNCTION

The $NUMERIC function tests the right half of an INT value for an ASCII numeric character.

The syntax for the $NUMERIC function is:

```
$NUMERIC  ( <int-expression> )


<int-expression>

   is an INT expression.  $NUMERIC inspects bits <8:15> of the
   expression and ignores bits <0:7>.

   It tests for a numeric character according to the criterion:

      <int-expression> >= "0" AND <int-expression> <= "9"
```

$NUMERIC sets the condition code to "<" if a numeric character occurs. If you plan to test the condition code, you must 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).

Example

This example tests for a numeric character in the expression "char":

```
STRING char;

IF $NUMERIC (char) THEN . . . ;
```

$OCCURS FUNCTION

The $OCCURS function returns the number of occurrences of a variable.

The syntax for the $OCCURS function is:

```
$OCCURS ( <variable> )


<variable>

   is the name of a STRUCT item (a structure, substructure, or
   STRUCT data item), as defined in Section 11.
```

For structures and substructures, $OCCURS returns the number of occurrences. For example, for a bounds specification of [0:3], $OCCURS returns the value 4.

$OCCURS always returns a constant value. You can use $OCCURS in LITERAL expressions.

<variable> can also be a non-STRUCT item, but this has little meaning. For any non-STRUCT item, $OCCURS returns a 1.

Example

This example returns the number of occurrences of "job^data":

```
INT index;

STRUCT .job^data[0:5];   !Declare structure
  BEGIN
  INT i1;
  STRING s1;
  END;

For index := 0 to $OCCURS (job^data) - 1 DO . . . ;
```

$OFFSET FUNCTION

The $OFFSET function returns the number of bytes from the base of the structure to a variable within the structure.

The syntax for the $OFFSET function is:

```
$OFFSET ( <variable> )


<variable>

   is the name of a STRUCT item (a substructure or STRUCT data
   item), as defined in Section 11.
```

The base of a structure has an offset of 0.

When you qualify the name of a STRUCT item, you can use constant indexes but not variable indexes; for example:

```
$OFFSET (struct1.subst[1].item)
```

$OFFSET always returns a constant value. You can use $OFFSET in LITERAL expressions.

For non-STRUCT items, $OFFSET returns a 0.

Example

This example assigns to "c" the offset of the third occurrence of a substructure:

```
STRUCT a;                       !Declares structure
  BEGIN
  INT array[0:40];
  STRUCT ab[0:9];               !Declares substructure "ab"
    BEGIN                       ! with ten occurrences
      .
    END;
  END;

INT c;

c := $OFFSET (a.ab[2]);         !Returns offset of third
                                ! occurrence of "ab"
```

$OVERFLOW FUNCTION

The $OVERFLOW function tests for an arithmetic overflow condition.

The syntax for the $OVERFLOW function is:

```
$OVERFLOW
```

To enable arithmetic overflow testing, you must clear the arithmetic overflow trap bit (bit 8) in the ENV register. If an arithmetic overflow occurs while this bit is set, a trap results.

If the overflow bit is on, $OVERFLOW returns a -1 (true); otherwise, it returns a 0 (false).

Example

This example tests the condition of the overflow indicator:

```
IF NOT $OVERFLOW THEN . . . ;
```

$PARAM FUNCTION

The $PARAM function checks for the presence or absence of a parameter in the call that invoked the current procedure or subprocedure.

The syntax for the $PARAM function is:

```
$PARAM ( <formal-param> )


<formal-param>

   is the name of a formal parameter as specified in the procedure
   or subprocedure declaration (described in Section 16).
```

If the parameter is present, $PARAM returns a 1. If the parameter is absent, $PARAM returns a 0.

You can only use $PARAM in a VARIABLE procedure or subprocedure or in an EXTENSIBLE procedure. The called procedure must check for the presence or absence of each required parameter in CALL statements. It can check for optional parameters in the same way.

Example

This example checks for the absence of each required parameter and for the presence of the optional parameter:

```
PROC var^proc (buffer,length,key) VARIABLE; !Procedure declaration
  INT .buffer, length,         !Required parameters
       key;                    !Optional parameter
BEGIN
  !Some code here
  IF NOT $PARAM (buffer) AND NOT $PARAM (length) THEN RETURN;
                    !Returns 1 or 0 for each required parameter;
                    ! AND results in true or false value
  IF $PARAM (key) THEN . . .        !Returns 1 if optional
END;                                ! parameter is present
```

$POINT FUNCTION

The $POINT function returns the <fpoint> value, in integer form, associated with a FIXED expression.

The syntax for the $POINT function is:

```
$POINT ( <fixed-expression> )

<fixed-expression>

   is a FIXED expression.
```

TAL emits no instructions when evaluating <fixed-expression>. Therefore, you cannot use <fixed-expression> to invoke a function or assign a value to a variable.

Example

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;
FIXED(3) a;
FIXED(3) b;                        !Data declarations

result := $SCALE ( a, $POINT ( b )) / b;
```

$RP FUNCTION

The $RP function returns the current setting of the TAL RP counter.

The syntax for the $RP function is:

```
$RP
```

Example

This example assigns the current RP setting to "index":

```
INT index;              !Data declaration
index := $RP ;          !Returns RP setting
```

$SCALE FUNCTION

The $SCALE function moves the position of the implied decimal point by adjusting the internal representation of a FIXED expression.

The syntax for the $SCALE function is:

```
$SCALE ( <fixed-expression> , <scale> )
```

<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.

$SCALE sets the overflow indicator if the result of the scale exceeds the range of a 64-bit integer. $SCALE adjusts the implied decimal point by multiplying or dividing by 10 to the <scale> power. If it scales the operand down, some precision is lost.

This function needs the following optional microcode:

System	FIXED
NonStop 1+	QLD QUP

Example

This example scales the value of "a" by +3, making "a" a FIXED(6) value. The result of the divide operation is a FIXED(3) value:

```
FIXED(3) result, a, b;        !Data declarations
result := $SCALE(a, 3) / b;
```

$SPECIAL FUNCTION

The $SPECIAL function tests the right half of an INT value for an ASCII special (nonalphanumeric) character.

The syntax for the $SPECIAL function is:

```
$SPECIAL ( <int-expression> )


<int-expression>

   is an INT expression.  $SPECIAL inspects bits <8:15> of
   <expression> and ignores bits <0:7>.

   It checks for a special character according to the following
   criterion:

   <int-expression> <> alphabetic AND <int-expression> <> numeric
```

$SPECIAL sets the condition code to ">" if it finds a special character. If you plan to check the condition code, you must do so before an arithmetic operation or a variable assignment occurs.

If the character passes the test, $SPECIAL returns a -1 (true); otherwise, it returns a 0 (false).

Example

This example tests the expression "char" for the presence of a special character:

```
STRING char;

IF $SPECIAL (char) THEN . . . ;
```

$TYPE FUNCTION

The $TYPE function returns a value that indicates the type of a variable.

The syntax for the $TYPE function is:

```
$TYPE ( <variable> )

<variable>

   is any identifier that has an associated data type or is
   a structure or substructure.
```

$TYPE returns a 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		

$TYPE always returns a constant value. You can use $TYPE in LITERAL expressions.

Example

This example assigns the value returned by $TYPE to "type1":

```
REAL(64) var1;
INT type1;               !Data declarations

type1 := $TYPE (var1);   !Returns a 6
```

$UDBL FUNCTION

The $UDBL function returns an INT(32) value from an unsigned INT expression.

The syntax for the $UDBL function is:

```
$UDBL ( <int-expression> )

<int-expression>

   is an unsigned INT expression.
```

$UDBL places the INT value in the right half of an INT(32) variable and sets the left half to 0.

Example

This example returns the INT(32) value of "a16" and assigns it to "b32":

```
INT a16 := %177775;
INT(32) b32;

b32 := $UDBL (a16);
```

$XADR FUNCTION

The $XADR function returns an extended address for a variable that has a standard address.

The syntax for the $XADR function is:

```
$XADR ( <variable> )

<variable>

   is a variable that has a standard address.
```

For a pointer variable, $XADR returns the extended address of the data to which the pointer points, not the address of the pointer itself.

Examples

1. This example initializes an extended pointer with the extended address of an array:

```
PROC p;
BEGIN
  INT .array[0:49];                        !Declares array
  STRING .EXT ptr := $XADR (array);        !Declares and initializes
                                           ! extended pointer
END;
```

2. This example returns an extended address for an INT variable to which a standard pointer points, then assigns the extended address to an extended pointer:

```
INT .std^ptr := %1000;                 !Declares standard pointer
INT .EXT ext^ptr;                      !Declares extended pointer

@ext^ptr := $XADR(std^ptr);            !Assigns extended address
```

SECTION 18

PRIVILEGED PROCEDURES

This section tells how to access the system global data area using system global pointers, 'SG' equivalencing, and standard functions for privileged operations.

You can access system globals only within procedures that operate in privileged mode. Such procedures can access system data space, call other privileged procedures, and execute certain privileged instructions. Privileged procedures must be specially licensed to operate, since they might (if improperly written) adversely affect the status of the processor in which they are running.

You 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
- To perform privileged operations through calls to operating system procedures
- To execute privileged instructions that affect other programs or the operating system

An extended pointer can also point to system data as described in Appendix A.

SYSTEM GLOBAL POINTER DECLARATION

The system global pointer declaration associates an identifier with a variable that contains the address of a variable located in the system global data area.

The syntax of the system global pointer declaration is:

```
<type> .SG <identifier> [ := <preset-address> ]

            [ , .SG <identifier> [ := <preset-address> ] ] ... ;
```

<type>

is one of the following data types and specifies the type of value to which the pointer points:

STRING
INT
INT(32)
FIXED
REAL
REAL(64)

.SG

is the indirection symbol for system global addressing. At least one space must precede the .SG symbol; the period in the symbol must not appear in column 1.

<identifier>

is the name of the pointer.

<preset-address>

is the address of a variable in the system global data area determined by you or the system during system generation.

TAL allocates one word of local primary storage for the pointer in the current user data segment.

For information about system tables, see the System Description Manual for your system.

Example

The following example declares an INT system global pointer named "newname":

```
INT .SG newname;
```

'SG'-EQUIVALENCED VARIABLE DECLARATION

'SG' equivalencing associates a global, local, or sublocal identifier with a location relative to the base address of the system global area.

Equivalenced variables (simple variables, pointers, and structure pointers) are described first, followed by equivalenced structures.

The syntax for the 'SG'-equivalenced variable declaration is:

```
         { { .EXT } { <structure-pointer> ( <referral> ) } }
         { { .    } { <pointer>                          } }
  <type> {                                                 }
         { <simple-variable>                               }

                                    = 'SG' [ "[" <index> "]" ]
                                           [ {+|-} <offset>  ]


         { { .EXT } { <structure-pointer> ( <referral> ) } }
         { { .    } { <pointer>                          } }
     [ , {                                                 }
         { <simple-variable>                               }

                               = 'SG' [ "[" <index> "]" ]
                                      [ {+|-} <offset>  ] ] ... ;
```

<type>

For <structure-pointer>, <type> must be STRING or INT.
For <simple-variable> or <pointer>, <type> is any data type.

. (period)

is the indirection symbol for standard addressing.

.EXT

is the indirection symbol for extended addressing.

⟶

<structure-pointer>

is the identifier of a structure pointer to be made equivalent to 'SG'.

<pointer>

is the identifier of a pointer to be made equivalent to 'SG'.

<simple-variable>

is the identifier of a simple variable to be made equivalent to 'SG'.

<referral>

is the identifier of a previously declared structure or structure pointer.

'SG'

is the address base of the system global data area and stands for system global addressing; the identifier is addressed relative to SG[0].

<index> and <offset>

are equivalent INT values in the range 0 through 63.

Example

This example makes "item1" equivalent to the location 'SG' + 15:

```
INT item1 = 'SG' + 15;
```

The syntax for the 'SG'-equivalenced structure declaration is:

```
STRUCT [ . ] <structure> [ ( <referral> ) ]

                                        = 'SG' [ "[" <index> "]" ]
                                               [ {+|-} <offset>  ] ;


[ <structure-body> ]
```

. (period)

is the indirection symbol for standard addressing.

<structure>

is the identifier of a definition or referral structure to be made equivalent to 'SG'.

<referral>

is the identifier of a previously declared structure or structure pointer. Its presence means <structure> is a referral structure and <structure-body> cannot be specified.

'SG'

is the address base of the system global data area; <structure> is addressed relative to SG[0].

<index> and <offset>

are equivalent INT values in the range 0 through 63.

<structure-body>

is a BEGIN-END construct that contains declarations as described in Section 11. Its presence means <structure> is a definition structure and <referral> cannot be specified.

FUNCTIONS FOR PRIVILEGED OPERATIONS

TAL provides four functions for performing certain operations that are restricted to programs running in privileged mode:

- $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
- $PSEM--Accesses a counting semaphone for awaiting completion of an operation external to the calling procedure or subprocedure
- $SWITCHES--Returns the current setting of the SWITCH register

These functions are described on the following pages.

$AXADR Function

The $AXADR function returns an absolute extended address.

The syntax for the $AXADR function is:

```
$AXADR ( <variable> )
```

<variable>

is a variable with a standard or relative extended address to convert to an absolute extended address. If <variable> is a pointer, the absolute extended address of the item it points to is returned, not the pointer's address.

Example

This example converts the standard address of "intr" to an absolute extended address:

```
STRING  .EXT str;
INT  intr;
   .
   .
@str := $AXADR (intr);
   .
   .
END;
```

$BOUNDS Function

The $BOUNDS function checks the location of a parameter passed to a system procedure to prevent an incorrect address pointer from overlaying a system procedure stack register with data.

The syntax for the $BOUNDS function is:

```
$BOUNDS ( <param> , <count> )
```

<param>

is a parameter of the procedure from which the $BOUNDS function is callable. It must not be a subprocedure parameter.

<count>

is a value of the same data type as <param>.

$BOUNDS returns an INT result as follows: 0 for false (no bounds error occurred) or 1 for true (bounds error occurred).

Example

This example checks the location of the parameter "buf":

```
PROC example (buf, z);
     .
     .                                !Checks for any part of "buf"
  IF $BOUNDS (buf, count)             ! in the procedure's stack
  THEN                                !If true, generates an error
    CALL error;
     .
     .
END;
```

$PSEM Function

The $PSEM function requests a semaphore on behalf of the caller, allowing the caller to await completion of an external process that uses the system resource represented by the semaphore. When the semaphore becomes available, the caller can continue.

The syntax for the $PSEM function is:

```
$PSEM ( <semaphore-addr> , <interval> )


<semaphore-addr>

   is the address of the semaphore desired.


<interval>

   is an INT(32) value that defines the maximum duration
   the procedure waits for the semaphore before continuing,
   specified in 10-millisecond intervals.
```

For further information about semaphores, see the System Description Manual for your system.

$SWITCHES Function

The $SWITCHES function returns the current setting of the SWITCH register to the caller.

The syntax for the $SWITCHES function is:

```
$SWITCHES
```

Example

The following example stores the current contents of the SWITCH register into "n":

```
n := $SWITCHES;
```

SECTION 19

SAMPLE PROCEDURE

To illustrate some of the coding techniques used in TAL, the source text for a simple procedure appears in Figure 19-1. This procedure performs a conversion function typical of many algorithms in TAL.

The procedure converts a binary INT value to an ASCII (base 10) value with a maximum length of six characters including the sign, then returns the converted character string and its length to the calling procedure.

Significant items in this procedure are keyed to the following discussion:

Item	Discussion
!1!	Comments preceding the procedure declaration describe the purpose of the procedure. For complex procedures, you can also summarize the input/output characteristics and the main features of the algorithm.
!2!	The formal parameter specifications define the parameters of the procedure. Input parameters "v" and "rjust" are value parameters, and output parameter "stg" is a reference parameter.
!3!	This local declaration reserves six bytes of memory for the buffer in which the number is converted. The declaration also initializes the first five bytes in the buffer to blanks (using a repetition factor of 5) and sets the last byte to an ASCII 0. Thus, an input of 0 results in an output of five blanks and a 0, rather than six blank characters.

!4! This IF-THEN statement deals with any negative number passed as a parameter. When it encounters a negative number, it sets the negative value flag to 1 and takes the absolute value of the number passed.

!5! This WHILE loop performs the conversion, character by character, writing each byte to the buffer from right to left.

!6! This assignment statement does the actual conversion. It illustrates an arithmetic expression that uses the standard function $UDBL. The statement performs a residue modulo 10 operation, then biases the value of each byte up into the numeric range by adding an ASCII 0.

!7! This IF-NOT-THEN statement uses the assignment form of an arithmetic expression as the condition.

!8! This IF-THEN-ELSE statement moves the resulting character string from the buffer into the user's target string.

!9! The RETURN statement returns to the calling procedure the number of characters moved.

```
!1!  !INT PROC ASCII converts a binary INT value to an ASCII
     ! (base 10) value with a maximum length of six characters
     ! (including the sign), then returns the converted character
     ! string and its length to the calling procedure.


     INT PROC Ascii(v,rjust,stg);
!2!    INT     v;                       !INT value to convert
       INT     rjust;                   !Right justify result flag
       STRING .stg;                     !Target string
     BEGIN
!3!  STRING    .b[0:5] := [5*[" "],"0"];
     INT        n;                      !Number of digits converted
     INT        sgn := 0;               !Nonzero if 'v' is negative
     INT        k := 5;                 !Index for converted digit

!4!  IF v < 0                           !Value is negative
     THEN
       BEGIN
       sgn := 1;                        !Set negative value flag
       v := -v;                         !Take absolute value
       END;

!5!  WHILE v                            !While a value is left . . .
     DO
       BEGIN
!6!    b[k] := $UDBL(v) '\' 10 + "0"; !Convert a character
       v := v / 10;                     !Compute remainder
       k := k - 1;                      !Count converted character
       END;

     IF sgn                             !Number is negative
     THEN
       BEGIN
       b[k] := "-";                     !Insert the sign
       k := k - 1;                      !Count it as a character
       END;
```

Figure 19-1. Sample Procedure (Continued on Next Page)

```
!7!  IF NOT (n:=5-k)               !Check for an overflow
     THEN
       n := 1;                     !Return 1 character in that case

!8!  IF rjust                      !Move the resultant string to the
     THEN                          ! user's target
       stg[n-1] '=:' b[5] FOR n !Reverse move if right justified
     ELSE
       stg ':=' b[6-n] FOR n;      !Otherwise forward move

!9!  RETURN n;                     !Return the string's length
     END !ascii! ;
```

Figure 19-1--(Continued)

SECTION 20

COMPILER OPERATION

This section describes:

- The compilation process
- The COMINT PARAM commands that TAL accepts
- The TAL run command
- TAL compiler directives

COMPILATION PROCESS

The input for a single run of the TAL compiler is a compilation unit. A compilation unit consists of one or more source files that contain declarations, statements, and compiler directives. Each compilation unit compiles into an object file that consists of relocatable code and data blocks.

You can bind an object file with other object files to build a new object file called the target file. For a description of object files, see the BINDER Manual.

The TAL compiler is integrated with two other processes, BINSERV and SYMSERV. Compiler directives govern all three processes.

TAL Compiler Process

TAL compiles source code, processes compiler directives, and starts BINSERV and SYMSERV for additional processing. TAL produces any listings that result from the three processes.

Compiler directives select compilation options and provide the compile-time interface to the BINDER, CROSSREF, and INSPECT program development tools. For example, the SYNTAX directive provides a syntax check without object-code generation, and the SEARCH directive lets you specify object files for BINSERV or BINDER to use for resolving external references.

BINSERV Process

BINSERV is the compile-time binder process. If the compilation is successful, BINSERV constructs the target file, resolving external references by binding code and data blocks from object files into the target file.

If the SYNTAX directive is not in effect, BINSERV is present throughout the compilation until TAL detects an error in a source file. Thus, the first error prevents construction of an object file. If BINSERV is present, the output listing contains binder statistics.

You can do further binding by using BINSERV or the standalone BINDER, described in the BINDER Manual.

SYMSERV Process

SYMSERV produces symbol tables for the object file. If the CROSSREF directive is in effect, SYMSERV also generates source-level cross-reference information. SYMSERV is present throughout the compilation.

PARAM COMMANDS

TAL accepts three COMINT PARAM commands (SAMECPU, SWAPVOL, and SPOOLOUT). These are summarized here and described further in the GUARDIAN Operating System Utilities Reference Manual. To take effect, these commands must precede the TAL run command.

PARAM SAMECPU Command

The PARAM SAMECPU command specifies that TAL, BINSERV, and SYMSERV all run in the same CPU. Specify a nonzero value with this command, as in the following example:

```
   PARAM SAMECPU 1
```

PARAM SWAPVOL Command

The PARAM SWAPVOL specifies the volume that TAL, BINSERV, and SYMSERV use for temporary files. The form of this command is:

```
PARAM SWAPVOL [ \<system>. ] $<volume>
```

If you do not specify a volume, TAL uses the default volume; BINSERV and SYMSERV use the volume specified to receive the target file, which might be the default volume. Use the PARAM SWAPVOL command when:

- The volumes normally used for temporary files might not have sufficient space.
- The default volume or the volume to receive the object file is on a different system from the compiler.

  On a NonStop system, if the PARAM SWAPVOL command specifies another system, TAL ignores the command and allocates temporary files on the volume on which it resides.

PARAM SPOOLOUT Command

The PARAM SPOOLOUT command causes significant decreases in elapsed time for compilations with listings, because TAL can use the Level 3 interface to the Spooler. The command form is:

```
PARAM SPOOLOUT 1
```

TAL RUN COMMAND

The command to run the TAL compiler is:

```
TAL [ / [ IN <source-file> ] [ , OUT [ <list-file> ] ]

   [ <comint-option-list> ] / ] [ <target-file-name> ]

   [ ; <directive> [ , <directive> ] ... ]
```

<source-file>

is the name of a file (an edit-format disc file, terminal, magnetic tape unit, or process) containing TAL declarations, statements, and compiler directives. It is read as 132-byte records. The default value is the COMINT <command-file>; if COMINT is in interactive mode, this is the home terminal.

<list-file>

is the name of a file (terminal, line printer, magnetic tape unit, process, or disc file) to receive compiler output. In an unstructured disc file, each record has 132 characters; partial lines are blank-filled through column 132.

If you specify OUT with no <list-file>, TAL suppresses output. If you omit OUT, the OUT file is that of the parent process; if you started the process under a COMINT, this is typically the home terminal.

<comint-option-list>

is one of the RUN command options documented in the GUARDIAN Operating System Utilities Reference Manual, such as:

```
NAME [ <process-name> ]
CPU <cpu-num>
PRI <priority>
NOWAIT
```

The MEM option is valid but has no effect; TAL always uses 64 pages.

⟶

<target-file-name>

is the name of the current target file in the form:

```
[\<sysname>.][$<volname>.][<subvolname>.]<discfile-name>
```

If you omit <target-file-name>, the default value is:

```
\<default-system>.$<default-volume>.<default-subvol>.OBJECT
```

<default-system> is the system specified in the current SYSTEM command, if you entered the command, or is the current system you are running on.

BINSERV constructs the object file in a temporary file. If <target-file-name> cannot be purged, BINSERV renames the existing target file with a name in the form ZZBI<nnnn> (where <nnnn> is a random number). BINSERV then assigns the specified name to the current target file.

<directive>

is any compiler directive described in "Compiler Directives" in this section, except ASSERTION, DECS, DUMPCONS, ENDIF, IF, IFNOT, PAGE, RP, SECTION, and SOURCE.

Do not use "?" on the command line.

Examples

1. This example sets PARAM commands, then starts compilation of the source file "mysource". It directs the listing to $SPOOL (a spooler collector), names "myprog" as the target file, suppresses the symbol map, and requests code mnemonics and cross-reference listings:

```
PARAM SAMECPU 1
PARAM SWAPVOL $junk
PARAM SPOOLOUT 1
TAL /IN mysource, OUT $SPOOL/myprog;NOMAP,ICODE,CROSSREF
```

2. This example starts compilation of the source file "talprg", suppresses output by giving a null list file, and sets a compilation toggle to control inclusion or exclusion of parts of the source text:

```
TAL /IN talprg,OUT / ; SETTOG 3
```

COMPILER DIRECTIVES

Compiler directives specify additional input source code and options for listings, code generation, and building of the object file.

Directive Line

A directive line in the source text begins with "?" in column 1. TAL interprets and processes each directive at the point of occurrence.

The general form of a directive line is:

```
? <directive> [ , <directive> ] ...
```

?

indicates a directive line; "?" must be in column 1.

<directive>

is a compiler directive described in this section.

The following rules apply to directive lines:

- The "?" is not part of the directive name; it appears only in column 1.
- A directive and its arguments must be on a single line unless otherwise noted under the directive description.
- Each continuation line for a list of directives begins with "?".

- Each continuation line for a single directive begins with "?". (SOURCE and SEARCH are examples of directives that can continue on multiple lines.)

Summary of Compiler Directives

This summary groups the directives by function and briefly describes each. The functional groups are:

- Input control
- Listing control
- Diagnostic output control
- Code generation control
- Toggle control
- Internal control
- Object file control

In the functional groups that follow, the default is underlined for directives that have a positive and a negative form.

Input Control

SECTION	names part of a source file.
SOURCE	specifies source to read from another input file.

Listing Control

ABSLIST	NOABSLIST	lists C-relative addresses.
CODE	NOCODE	lists instructions in octal for procedures.
CROSSREF	NOCROSSREF	cross references source identifier classes.
DEFEXPAND	NODEFEXPAND	lists invoked DEFINEs.
GMAP	NOGMAP	prints global map.

ICODE	NOICODE	lists mnemonics after each procedure.
INNERLIST	NOINNERLIST	lists mnemonics after each source statement.
LINES		specifies maximum number of lines per page.
LIST	NOLIST	lists source and enables other listings.
LMAP	NOLMAP	selects BINSERV load maps.
MAP	NOMAP	lists identifier map.
PAGE		causes page eject; specifies a header.
PRINTSYM	NOPRINTSYM	selectively lists symbols.
SUPPRESS	NOSUPPRESS	suppresses all but header, diagnostics, and trailer text.

Diagnostic Output Control

ERRORS		sets number of error messages to terminate TAL
RELOCATE		issues warnings for nonrelocatable globals (see "Object-File Control" directives).
WARN	NOWARN	Selectively enables warnings.

Code Generation Control

ASSERTION		generates debugging aids.
CPU		specifies NonStop or NonStop 1+ system.
DUMPCONS		dumps constant table to code.
INHIBITXX	NOINHIBITXX	inhibits extended, indexed instruction emission.
ROUND	NOROUND	specifies scalar rounding.
SYNTAX		checks syntax only; generates no code.

Toggle Control

ENDIF		marks end of conditional source.
IF		allows conditional compilation.
IFNOT		suppresses compilation.
RESETTOG		turns toggles off.
SETTOG		turns toggles on.

Internal Control

DECS		decrements S-register value of TAL.
RP		sets internal RP counter of TAL.

Object-File Control

<u>ABORT</u>	NOABORT	terminates compilation if TAL cannot open source file.
<u>COMPACT</u>	NOCOMPACT	fills 32K gap in code area.
DATAPAGES		defines size of data area.
EXTENDSTACK		defines number of pages to add to existing stack size.
INSPECT	<u>NOINSPECT</u>	selects default debugger (INSPECT or DEBUG).
LIBRARY		specifies NonStop system user library for resolving run-time external reference.
PEP		specifies PEP table size for BINSERV.
RELOCATE		issues messages if reference made to nonrelocatable global data.
SAVEABEND	<u>NOSAVEABEND</u>	directs INSPECT to create save file that contains process state if program ends abnormally.

SEARCH		names object files from which to resolve external references; SEARCH with no file name negates search list.
STACK		sets new stack size.
SYMBOLS	<u>NOSYMBOLS</u>	generates INSPECT symbol table for symbolic debugging.

DIRECTIVE DESCRIPTIONS

The remaining pages of this section give descriptions of directives in alphabetic order. Unless otherwise noted, each directive applies to all Tandem systems.

ABORT Directive

The ABORT directive terminates compilation if TAL cannot open the file you specified in a SOURCE directive; it issues an error message to the out file, stating the name of the file that cannot be opened.

The default is ABORT.

The syntax for the ABORT directive is:

```
[NO]ABORT
```

The ABORT directive is not a feature of the NonStop 1+ software.

NOABORT causes TAL to attempt to prompt the home terminal when the file cannot be opened.

ABSLIST Directive

The ABSLIST directive specifies that TAL lists instruction locations relative to the base of the code area, location C[0]. (LIST must be enabled.)

The default is NOABSLIST (that is, TAL lists addresses relative to the base of the procedure).

The syntax for the ABSLIST directive is:

```
[NO]ABSLIST
```

To use ABSLIST, you must define the size of the PEP table to TAL before it encounters procedure statements in the source program. You can either:

- Include a PEP directive at the beginning of the source program
- Declare each internal procedure FORWARD or EXTERNAL before the first procedure body

Limitations

ABSLIST attempts to maintain an overall code address; however, at least some addresses are invalid if the file:

- Has more than 32K of code
- Has resident procedures after nonresident procedures
- Does not supply enough PEP table space in the PEP directive or does not declare all procedures FORWARD

If the 64K limit is reached, TAL disables ABSLIST, starts printing offsets from the procedure base, and emits a warning.

Because of these limitations, Tandem does not recommend the use of the ABSLIST as a general practice.

ASSERTION Directive

The ASSERTION directive is a program debugging aid; it conditionally invokes a procedure when an event defined in an ASSERT statement occurs.

The syntax for the ASSERTION directive is:

```
ASSERTION [ = ] <assertion-level> , <procedure-name>

 <assertion-level>

    is an integer in the range 0 through 32767 that defines a
    numeric relationship to an ASSERT statement <assert-level>.

 <procedure-name>

    is the name of the procedure to invoke if the event defined
    in a ASSERT statement occurs and <assertion-level> is not
    greater than <assert-level>.  The named procedure must not
    have parameters.
```

The corresponding ASSERT statements have the form:

```
ASSERT <assert-level> : <expression>;
```

<expression> is a conditional expression that tests a program condition.

For an example of the ASSERTION directive, see the ASSERT statement in Section 15.

CODE Directive

The CODE directive lists instruction codes in octal if LIST is also enabled.

NOCODE suppresses the octal code listing. The default is CODE.

The syntax for the CODE directive is:

```
[NO]CODE
```

The CODE listing for each procedure follows it in the out file.

The CODE listing might not show final G-plus addresses for global variables. If a global variable is within a named data block, the G-plus address shown is relative to the start of the data block. At the end of the compilation, BINSERV creates the final G-plus address. To display the final addresses, use BINDER and INSPECT commands.

Other code locations affected by BINSERV are:

- Fix-up cells to global read-only arrays
- PCAL instructions

COMPACT Directive

The COMPACT directive directs BINSERV to move procedures if they fit into any gap below the 32K boundary of the code area.

The default is COMPACT.

The syntax for the COMPACT directive is:

```
[NO]COMPACT
```

You can use this directive any number of times; the last use of the directive sets the option for the compilation unit.

CPU Directive

The CPU directive specifies whether the object code is to run on a NonStop or a NonStop 1+ system.

The syntax for the CPU directive is:

```
CPU { TNS    }
    { TNS/II }
```

TNS

indicates the object code is to run on a NonStop 1+ system. Nonprivileged programs compiled in this mode can also run on a NonStop system.

TNS/II

indicates the object code is to run on a NonStop system.

If you do not use CPU, the default system type is the system on which you compile the code. Guidelines for using this directive are:

- Specify the CPU directive either on the TAL run command line or in the source code before the first declaration.
- Nonprivileged code containing NonStop software features such as extended addressing can compile on either system if you specify CPU TNS/II. Sections of code that use such features run correctly on a NonStop system only. The remaining code runs correctly on either system. To determine system type, see the TOSVERSION procedure in the System Procedure Calls Reference Manual.
- For nonprivileged code that can compile and run on either system, specify CPU TNS as documentation.

The CPU directive also influences BINDER behavior, as described in the BINDER Manual.

CROSSREF Directive

The CROSSREF directive specifies that TAL lists source-level cross-reference information produced during compilation and specifies the identifier classes to process.

The default is NOCROSSREF.

The syntax for the CROSSREF directive is:

```
[NO]CROSSREF [ <class>                          ]
             [ ( <class> [ , <class> ] ... ) ]
```

<class>

is one of:

BLOCKS	named and private data blocks
CONSTANTS	unnamed constants
DEFINES	named text
LABELS	names for use with GOTO statements
LITERALS	named constants
PROCEDURES	
PROCPARAMS	procedures that are formal parameters
SUBPROCS	
TEMPLATES	STRUCT (*) names
UNREF	unreferenced identifiers
VARIABLES	

The default class list includes all classes except CONSTANTS and UNREF. TAL does not support cross references for the CONSTANTS class.

Generating Cross References

To start generation of cross references for the default class list, specify CROSSREF with no parameters. To stop the generation, specify NOCROSSREF with no parameters.

You can use CROSSREF or NOCROSSREF with no parameters for individual procedures or data blocks. These directives take effect at the beginning of the next procedure or data block.

[NO]CROSSREF without parameters is effective for the entire program or until you respecify the directive. Entering [NO]CROSSREF to select a class list has no effect on starting or stopping cross-reference generation.

Selecting Classes

The CROSSREF directive entered to select a class list is effective for the entire program. Although you can respecify the class list, SYMSERV uses only the class list in effect at the end of compilation.

To add classes to the previous list, specify:

```
?CROSSREF, CROSSREF <add-list>
```

To delete classes from the previous list, specify:

```
?CROSSREF, NOCROSSREF <delete-list>
```

CROSSREF Listing

The compilation results in a single cross-reference list that follows the global map and precedes the load maps.

CROSSREF causes cross references to be collected even if NOLIST is in effect for all or part of the compilation. To include the collected cross references in the listings, a LIST directive is required at the end of the source. (This is true only for LIST and CROSSREF.)

The SUPPRESS directive turns off the cross-reference listing.

It is recommended that you use the CROSSREF directive only for simple cross-reference listings. For other CROSSREF options, use the standalone command-driven CROSSREF process. See the CROSSREF Manual.

Examples

1. This example adds unreferenced names to the class lists in the printed output:

   ```
   ?CROSSREF , CROSSREF UNREF
   ```

2. This example deletes LITERALS from the class list and prints the output:

```
?CROSSREF , NOCROSSREF LITERALS
```

3. This example suppresses part of the listing:

```
?CROSSREF
PROC p;
  BEGIN
   .
   .
  END;

?SUPPRESS           !Turn on SUPPRESS to suppress CROSSREF output
PROC q;
  BEGIN
   .
   .
  END;
?NOSUPPRESS         !Turn off SUPPRESS to get CROSSREF output
```

4. This example selectively collects cross references:

```
?CROSSREF, CROSSREF UNREF, NOCROSSREF VARIABLES
NAME test;
  INT i;

?NOCROSSREF             !No cross references collected for BLOCK
BLOCK PRIVATE;
  INT j;
END BLOCK;

?CROSSREF, CROSSREF VARIABLES
                    !Variables shown; prior directive superseded
PROC p MAIN;
  BEGIN
   .
   .
  END;
```

DATAPAGES Directive

The DATAPAGES directive overrides the default number of data pages that BINSERV assigns for the object program.

The syntax for the DATAPAGES directive is:

```
DATAPAGES [ = ] <integer>
```

<integer>

    is an integer in the range 0 through 64; if you specify an out-of-range value, BINSERV sets DATAPAGES to 64.

If you omit DATAPAGES, BINSERV allocates sufficient pages for global data and enough stack space for procedure locals twice over. If you specify an insufficient amount, BINSERV uses the default algorithm.

You can set DATAPAGES after compilation using the BINDER SET command options (DATA, STACK, or EXTENDSTACK).

You can increase data pages at run time using the RUN command MEM parameter or the memory-pages parameter of the NEWPROCESS procedure.

DECS Directive

The DECS directive decrements the TAL internal S-Register counter.

The syntax for the DECS directive is:

```
DECS [ = ] <sdec-value>


<sdec-value>

   is an unsigned integer to subtract from the TAL S-Register
   counter.
```

Use DECS when the source code manipulates the data stack.

Example

This example places the parameters for "proc^name" on the data stack using a PUSH instruction (rather than a CALL statement). ?DECS 3 decrements the TAL internal S-Register setting by 3.

```
SUBPROC sp;
  BEGIN
      .
      .
    STACK param1, param2, param3; !Loads parameters onto
                                  ! register stack
    CODE( PUSH %722);             !Pushes parameters onto
                                  ! memory stack
    CODE( PCAL proc^name);        !Calls the procedure
?DECS 3
      .
      .
  END;
```

DEFEXPAND Directive

The DEFEXPAND directive causes the text of a DEFINE to appear in the listing when TAL translates the DEFINE.

The default is NODEFEXPAND.

The syntax for the DEFEXPAND directive is:

```
[NO]DEFEXPAND
```

When you specify DEFEXPAND, the text of the DEFINE appears in the listing on the lines following the name of the DEFINE. The text in the listing differs from the text in the DEFINE declaration as follows:

- It contains no comments, line boundaries, or extra blanks.
- Parameters to the DEFINE appear as $<number>, where <number> is the sequence number of the parameter, starting at 1.
- Lowercase letters appear as uppercase.

The DEFINE nesting level (starting at 1) appears in the left margin.

DUMPCONS Directive

The DUMPCONS directive causes TAL to dump immediately all constants currently in the TAL constant table into the object code.

The syntax for the DUMPCONS directive is:

```
DUMPCONS
```

TAL generates an unconditional branch around the dumped constants. DUMPCONS can be useful prior to writing CODE statements, since range requirements can force TAL to dump the constants within inline code. DUMPCONS can also avoid overflow of the TAL internal constant table.

If you do not specify DUMPCONS, TAL inserts constants into the generated code after unconditional branches and at the end of procedures, if possible.

ENDIF Directive

The ENDIF directive terminates the range of the IF or IFNOT directive. ENDIF is useful with toggles and CPU type. Refer also to the IF toggle directive.

The syntax for the ENDIF directive is:

```
ENDIF { <toggle-number> }
      { <cpu-type>      }


<toggle-number>

   is an integer from 1 through 15, as specified by a SETTOG or
   RESETTOG directive.


<cpu-type>

   is one of the following, as specified in the CPU directive:

      TNS      The code executes on the NonStop 1+ system.

      TNS/II   The code executes on the NonStop system.
```

If other directives appear on the same line, the ENDIF directive must be last on the line.

For an example, see the IF directive.

ERRORS Directive

The ERRORS directive sets the number of error messages at which to terminate the compilation.

The syntax for the ERRORS directive is:

```
ERRORS [ = ] <nnnnn>

<nnnnn>

   is an integer in the range 0 through 32767 that specifies
   the number of error messages at which to terminate the
   compilation.
```

TAL counts the number of error messages; a single error can cause many messages. If the count exceeds the maximum you specify, TAL terminates the compilation. (Warning messages do not affect the count.)

If you do not specify ERRORS, TAL does not terminate the compilation because of the number of errors.

EXTENDSTACK Directive

The EXTENDSTACK directive specifies the number of pages to add to the BINDER's estimate of the stack size.

The syntax for the EXTENDSTACK directive is:

```
EXTENDSTACK <value>


<value>

   is the number of pages to add to the stack size.
```

If you omit this directive, the default is the stack size estimated by BINDER.

Example

This example extends the stack size by 20 pages.

```
?EXTENDSTACK 20
```

GMAP Directive

The GMAP directive instructs TAL to print a global map at the end of the compilation listing.

NOGMAP suppresses the global map. The default is GMAP.

The syntax for the GMAP directive is:

```
[NO]GMAP
```

The GMAP directive is a not a feature of the NonStop 1+ software.

GMAP is not effective unless the MAP directive is set. GMAP has no effect when the NOMAP option is in effect. However, if MAP is active you can suppress the global map by entering "?NOGMAP".

Examples

1. This example specifies that the global map is printed:

```
?GMAP
```

2. This example disables printing of the global map:

```
?NOGMAP
```

ICODE Directive

The ICODE directive causes listing of instruction code mnemonics if LIST is enabled.

The default is NOICODE.

The syntax for the ICODE directive is:

```
[NO]ICODE
```

The ICODE listing might not show final G-plus addresses for global variables. If a global variable is within a named data block, the G-plus address shown is relative to the start of the data block. At the end of the compilation, BINSERV creates the final G-plus address. To display the final addresses, use BINDER and INSPECT commands.

Other code locations affected by BINSERV are:

- Fix-up cells to global read-only arrays
- PCAL instructions

IF Directive

The IF and IFNOT toggle directives specify selective compilation depending on the indicated condition.

The syntax for the IF directive is:

```
IF[NOT] { <toggle-number> }
        { <cpu-type>      }
```

<toggle-number>

is an integer from 1 through 15, as specified in a SETTOG or RESETTOG directive.

<cpu type>

is specified by a CPU directive as one of the following:

TNS	The code executes on the NonStop 1+ system.
TNS/II	The code executes on the NonStop system.

If other directives appear on the same line, the IF[NOT] directive must be last in the line.

"IF <toggle-number>" directs TAL to ignore subsequent text unless the software toggle switch indicated by <toggle-number> is set by a SETTOG directive.

"IFNOT <toggle-number>" directs TAL to ignore the text unless the toggle is not set by a SETTOG directive.

Once skipping begins, it continues to the matching ENDIF directive. Thus, in the following fragment, TAL skips both parts if <n> is reset:

```
?IF n
  !Statements for true condition
?IFNOT n
  !Statements for false condition
?ENDIF n
```

If you insert another ENDIF directive into this fragment, TAL skips only the first part if <n> is reset:

```
?IF n
  !Statements for true condition
?ENDIF n
?IFNOT n
  !Statements for false condition
?ENDIF n
```

Examples

1. If CPU TNS/II is in effect, TAL compiles the code between IF TNS/II and ENDIF TNS/II and ignores the code between IFNOT and ENDIF:

```
?IF TNS/II                      !If the NonStop system is the
   .                            ! execution system . . .
   .
 CALL WRITE (term, buff , 67);
   .
   .
?ENDIF TNS/II
?IFNOT TNS/II                   !If the NonStop system is not
   .                            ! the execution system . . .
   .
 CALL WRITE (term, buff2, 78);
   .
   .
?ENDIF TNS/II
```

2. This example tests the toggle number, finds it is ON (set by SETTOG), and causes TAL to include the procedure:

```
?SETTOG 1                      !Turns toggle number 1 ON
   .
   .
?IF 1                          !Tests toggle number 1
 PROC some^proc;               !Toggle 1 is ON; executes
 BEGIN                         ! procedure
   .
   .
 END;
?ENDIF 1
```

INHIBITXX Directive

The INHIBITXX directive suppresses generation of the extended, indexed ('XX') instructions (LWXX, SWXX, LBXX, and SBXX) for extended pointers relocated beyond the first 64 words of primary global data.

The default is NOINHIBITXX.

The syntax for the INHIBITXX directive is:

```
[NO]INHIBITXX
```

You should specify [NO]INHIBITXX before the global declarations occur.

The 'XX' instructions assume that the extended pointer is located between G[0] and G[63] of the primary global data area. The 'XX' instructions are described in the System Description Manual for the NonStop system.

INNERLIST Directive

The INNERLIST directive lists the instruction code mnemonics generated by TAL after each statement if LIST is enabled. It also shows the TAL RP setting.

The default is NOINNERLIST.

The syntax for the INNERLIST directive is:

```
[NO]INNERLIST
```

The INNERLIST listing is less complete than the ICODE listing. Since TAL is a one-pass compiler, many instructions appear with skeleton or space-holder images that TAL or BINSERV modifies later.

INSPECT Directive

The INSPECT directive specifies that INSPECT is the default debugger for the object file.

The default is NOINSPECT.

The syntax for the INSPECT directive is:

```
[NO]INSPECT
```

The last [NO]INSPECT directive in a compilation unit takes effect for the object file.

You can also set the default debugger after compilation using:

- The SET INSPECT command of the BINDER
- The COMINT SET INSPECT and RUN commands

You cannot override INSPECT at run time.

The INSPECT, SAVEABEND and SYMBOLS directives are interrelated. BINSERV and BIND automatically set INSPECT ON if the SAVEABEND directive specifies creation of a save file. The NOINSPECT directive causes BINSERV and BIND to set SAVEABEND OFF.

To use the full symbolic debugging features of INSPECT, specify the SYMBOLS directive to generate the symbol table in the object file. You can turn the SYMBOLS directive on and off on a procedure-by-procedure or block-by-block basis. (Even if you do not specify SYMBOLS, INSPECT still recognizes procedure names in code locations.)

Example

This example requests INSPECT and SAVEABEND for the entire object file and SYMBOLS for part of the code:

```
? INSPECT, SYMBOLS, SAVEABEND
PROC a ;
  .
  .
END;
? NOSYMBOLS
PROC b ;
  .
  .
END;
```

LIBRARY Directive

The LIBRARY directive specifies the name of the NonStop software user library to be associated with the object file at run time.

The syntax for the LIBRARY directive is:

```
LIBRARY <file-name>

<file-name>

   specifies a user library to search before the system library
   for satisfying external references.
```

You can also change the library name either in a BIND session or by using the LIB parameter of the COMINT RUN command.

LINES Directive

The LINES directive sets the maximum number of output lines per page.

The syntax for the LINES directive is:

```
LINES <value>

<value>

   is a decimal number in the range 10 through 32767.  The
   default value is 60 lines per page.
```

The LINES directive is not a feature of the NonStop 1+ software.

Example

This example sets the maximum number of lines per page of output listing at 66 lines per page:

```
?LINES 66
```

LIST Directive

The LIST directive specifies that each source image is written to the list file and enables other list options.

The default is NOLIST.

The syntax for the LIST directive is:

```
[NO]LIST
```

You can specify the LIST directive anywhere in the source text.

The ABSLIST, CODE, ICODE, INNERLIST, MAP, LMAP, GMAP, and PAGE directives require the LIST directive.

The SUPPRESS directive overrides LIST.

LMAP Directive

The LMAP directive specifies the types of load-map and cross-reference information requested from BINSERV.

NOLMAP cancels LMAP. The default is LMAP ALPHA.

The syntax for the LMAP directive is:

```
         { <lmap-option>                              }
[NO]LMAP { ( <lmap-option> [ , <lmap-option> ] ... ) }
         { *                                          }
```

<lmap-option>

specifies the type of map; it is one of:

ALPHA

specifies load maps of procedures and data blocks sorted by name.

LOC

specifies load maps of procedures and data blocks sorted by starting address.

XREF

specifies an entry point and data block cross reference for the object file. This differs from source-level cross references produced by the CROSSREF directive.

*

specifies ALPHA and LOC maps and the cross-reference listings. LMAP* is equivalent to LMAP *.

NOLMAP with options specifies that, if LMAP is in effect, the stated options are turned off. NOLMAP without options suppresses the map entirely.

In releases before TAL E01, LMAP (ALPHA, LOC) is equivalent to LMAP *. Now LMAP * means the output listings contain the ALPHA and LOC maps and the cross-reference data that BINSERV collects. The XREF information listed includes an entry-point cross reference and a common data-block cross reference.

Example

This example illustrates the LMAP directive:

```
   ?LMAP (LOC, XREF)    !Adds LOC and XREF to ALPHA default
     .
     .
   ?NOLMAP (XREF)       !Deletes only XREF from the listing
```

MAP Directive

The MAP directive controls the display of identifier maps in the listing, if LIST is enabled.

NOMAP cancels MAP. The default is MAP.

The syntax for the MAP directive is:

```
[NO]MAP
```

MAP displays sublocal identifiers following each subprocedure, local identifiers following each procedure, and global identifiers following the last procedure in the source program.

The MAP directive requires the LIST directive. The GMAP directive requires the MAP directive.

PAGE Directive

The PAGE directive causes a page eject on the listing file after the first PAGE directive, prints the optional heading, then skips two lines before listing continues.

The syntax for the PAGE directive is:

```
PAGE [ " <heading-string> " ]

<heading-string>

   is a character string that contains a maximum of 61 characters
   on a single line, enclosed in quotation marks.
```

PAGE is effective only if you specify the LIST directive.

The first PAGE directive in a source program does not cause a page eject. Rather, it specifies an initial heading string.

A subsequent <heading-string> replaces the previous header.

The quotation marks are required delimiters; they are not printed. If the string is too long, TAL truncates the extra characters.

If the list file is not a line printer or a process, TAL ignores the PAGE directive.

PEP Directive

The PEP directive specifies the anticipated size, in words, of the PEP table.

The syntax for the PEP directive is:

```
PEP [ = ] <pep-table-size>

<pep-table-size>

   is an integer in the range 3 through 512 to use as the size of
   the PEP table.
```

The <pep-table-size> must be at least large enough to contain the PEP, that is, one word per entry point that is not external. It can be a larger value.

You can respecify the PEP size at any time (without causing a warning from TAL), or it can be insufficient for the program; in either case, the ABSLIST addresses produced are invalid.

You should use the PEP directive if you use the ABSLIST directive so that TAL knows how much space BINSERV allocates for the PEP. (ABSLIST means TAL lists code-relative addresses for instruction locations).

Example

The following example illustrates the PEP directive:

```
?PEP 60
```

PRINTSYM Directive

The PRINTSYM directive enables the printing of a symbol or group of symbols as part of the output listing.

The default is PRINTSYM. NOPRINTSYM disables PRINTSYM.

The syntax for the PRINTSYM directive is:

```
[NO]PRINTSYM
```

The PRINTSYM directive is not a feature of the NonStop 1+ software.

You can use the PRINTSYM directive for global, local, or sublocal declarations.

Example

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

The RELOCATE directive directs TAL to list BINSERV warnings for declarations that depend on absolute addresses in the primary global data area.

The syntax for the RELOCATE directive is:

```
RELOCATE
```

The binder process issues warnings for references to nonrelocatable data during the target-file build (whether at compile time or in command-driven mode.)

TAL checks for nonrelocatable data only if RELOCATE appears.

Since RELOCATE is only effective for the source code that follows it, be sure to specify it at the beginning of the compilation.

Use the RELOCATE directive when the primary global data area (the area below word 256) is relocatable. If you are using the separate compilation features of TAL or binding TAL code with code written in other languages, the primary global data must be relocatable.

An example of a nonrelocatable data declaration is:

```
INT i = 'G' + 22;
```

References to "i" produce a warning when RELOCATE is in effect.

RESETTOG Directive

The RESETTOG directive turns the specified toggles to OFF.

The syntax for the RESETTOG directive is:

```
RESETTOG  [ <toggle-number> [ , <toggle-number> ] ... ]


<toggle-number>

   is an integer from 1 through 15.  If you do not specify a
   <toggle-number>, TAL resets all toggles to OFF.
```

If other directives appear on the same line, RESETTOG must be the last directive on the line.

The IF, ENDIF, IFNOT, and SETTOG directives also control the toggles. For more information, refer to the SETTOG directive.

Example

This example tests the toggle, finds it is reset, and causes TAL to skip over the source text between "IF 1" and "?ENDIF 1":

```
?RESETTOG 1              !Turns toggle number 1 OFF
   .
   .
?If 1                    !Tests toggle, finds it turned OFF

 PROC some^proc;         !TAL skips procedure
 BEGIN
   .
   .
 END;

?ENDIF 1
```

ROUND Directive

The ROUND directive specifies that rounding occurs when a FIXED value is assigned to a FIXED variable with a smaller <fpoint> value.

The default is NOROUND.

The syntax for the ROUND directive is:

```
[NO]ROUND
```

ROUND scales the value of the operand, if necessary, to match the <fpoint> of the assignment variable. If the <fpoint> of the operand is greater than that of the variable, the operand is first scaled, if necessary, so that its <fpoint> is one greater than the variable. The scaled operand is rounded as shown below:

```
(IF operand < 0 THEN operand - 5 ELSE operand + 5) / 10
```

That is, if the operand is negative, 5 is subtracted; if positive, 5 is added. Then, an integer divide by 10 rounds the operand and scales it down by a factor of 10. Therefore, if the absolute value of the least significant digit of the operand after initial scaling is 5 or more, one is added to the absolute value of the final least significant digit.

NOROUND specifies that rounding does not occur when a FIXED value is assigned to a FIXED variable with a smaller <fpoint> value. The value of the operand assigned to the variable is scaled, if necessary, to match the <fpoint> value of the variable. If the <fpoint> value of the operand is greater than that of the variable, the operand is scaled down and some precision is lost.

RP Directive

The RP directive sets the register stack RP count that TAL uses as the current value.

The syntax for the RP directive is:

```
RP [ = ] <register-number>

<register-number>

   specifies the number to which TAL sets its internal RP count.
   If you specify 7, TAL considers the register stack to be empty.
```

The RP value is that of the top element in the register stack. Use it to manipulate the register stack within the source text. Refer to the System Description Manual for register stack information.

You can use the RP directive only within a procedure.

Following each high-level statement (not CODE, STACK, or STORE), the TAL internal RP setting is always 7.

Example

This example informs TAL that five elements are loaded and, therefore, the current RP setting is 4:

```
FOR i := 0 TO 4 DO STACK( i );
?RP = 4
```

SAVEABEND Directive

The SAVEABEND directive causes INSPECT to generate a save file if the process abnormally terminates during execution.

The default is NOSAVEABEND.

The syntax for the SAVEABEND directive is:

```
[NO]SAVEABEND
```

For this option to be effective at run time, INSPECT must be available on the system that runs the process.

This directive can appear anywhere in the source program. BINSERV uses the last specification when building the object file.

If you use SAVEABEND, BINSERV automatically sets the INSPECT directive ON. (NOSAVEABEND does not affect the INSPECT directive.)

The save file contains data area and file-status information at the time of failure. You can examine the save file during an INSPECT session. INSPECT 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 program's volume and subvolume. (You can specify a name for the save file using INSPECT.) Refer to the INSPECT Interactive Symbolic Debugger User's Manual for information on the save file.

You can respecify the SAVEABEND option for a process using either the BINDER or RUN options.

SEARCH Directive

The SEARCH directive directs TAL to construct a list of object files from which BINSERV can resolve unsatisfied external references and validate parameter lists at the end of compilation.

The default is SEARCH with no file-name list, which means BINSERV does not attempt to satisfy remaining external references; no search occurs.

The syntax for the SEARCH directive is:

```
SEARCH [ <object-file-name>                                    ]
       [ ( <object-file-name> [ , <object-file-name> ] ... ) ]
```

<object-file-name>

is a valid file name for an object file; TAL provides automatic file name expansion. Specify the names in the order you want the search to take place.

A SEARCH directive can extend to continuation lines, each beginning with "?" in column 1. SEARCH directives can appear anywhere in the source code.

The search list is an ordered list that BINSERV uses to retrieve object code at bind time for inclusion in the object file.

If multiple SEARCH directives with file names occur, BINSERV appends the file names to the search list in the order specified. The order is important if more than one file contains a procedure or entry-point name that resolves an external reference. BINSERV includes the first occurrence and ignores any subsequent occurrences.

A SEARCH directive with no file names clears the search list. BINSERV can only satisfy external references using files that remain on the search list at the end of compilation.

SECTION Directive

The SECTION directive gives a name to a section of a source file for use in a SOURCE directive.

The syntax for the SECTION directive is:

```
SECTION <text-name>


<text-name>

   is a valid TAL identifier to associate with all source text
   that follows the SECTION directive until another SECTION
   directive or the end of the source file occurs.
```

The SECTION directive must be the only directive on the directive line.

Example

This example gives a section name to each procedure in a source library:

```
!File name "appllib"
?SECTION sort^proc
PROC sort^on^key(key1, key2, key3, length);
 INT .key1, .key2, .key3, length;
BEGIN
   .
   .
END;
?SECTION next^procedure
```

Another source file includes the previous file name and a section name in a SOURCE directive:

```
?SOURCE appllib (sort^proc)
```

SETTOG Directive

The SETTOG directive turns on all specified toggles.

The syntax for the SETTOG directive is:

```
SETTOG  [ <toggle-number> [ , <toggle-number> ] ... ]


<toggle-number>

   is an integer from 1 to 15; if you omit <toggle-number>, all
   toggles are turned on.
```

If other directives appear on the same line, SETTOG must be the last directive on the line.

The IF, ENDIF, IFNOT, and RESETTOG directives also control the toggles.

Example

This example tests the toggle, finds it is set, and causes TAL to compile the source text between "IF 1" and "?ENDIF 1":

```
?SETTOG 1                 !Turns toggle number 1 ON
   .
   .
?If 1                     !Tests toggle, finds it turned ON
 PROC some^proc;          !TAL compiles procedure
 BEGIN
   .
   .
 END;
?ENDIF 1
```

SOURCE Directive

The SOURCE directive specifies a file and optional section from which to read source statements.

The syntax for the SOURCE directive is:

```
SOURCE <file-name> [ ( <section-name>

                             [ , <section-name> ] ... ) ]
```

<file-name>

specifies the name of the disc file from which TAL reads source statements.

<section-name>

is a name specified in a SECTION directive within the source file <file-name>. If TAL does not find <section-name> in the specified file, it issues a warning.

TAL processes the source file until an end of file occurs (or until TAL reads all the sections in the section list). TAL then begins reading at the line following the SOURCE directive. The maximum number of source files you can have open at a time (nested SOURCE directives) is four.

If you include other directives on the same line, the SOURCE directive must be last in the line. The list of section names can extend to continuation lines, each of which must begin with a "?" in column 1.

Example

This example of the SOURCE directive includes an entire file:

```
?SOURCE $src.current.routines
```

STACK Directive

The STACK directive specifies the number of pages you want as the stack size instead of the estimated size.

The syntax for the STACK directive is:

```
STACK <value>

<value>

   is the data stack size in pages.
```

If you omit this directive, the default is the space estimated by BINSERV for local storage.

The total number of data pages is equal to the number of pages specified plus the space required for global data blocks.

Example

This example sets the stack size to 20 pages:

```
?STACK 20
```

SUPPRESS Directive

The SUPPRESS directive is a master override of the listing directives.

The default is NOSUPPRESS.

The syntax for the SUPPRESS directive is:

```
[NO]SUPPRESS
```

SUPPRESS overrides the CODE, CROSSREF, GMAP, ICODE, INNERLIST, LIST, LMAP, MAP, and PAGE directives.

It suppresses all compilation listing output except the compiler leader text, diagnostic messages, and the trailer text. That is, TAL and BINSERV produce diagnostic and trailer text, but BINSERV does not produce the load maps.

Specifying SUPPRESS on the TAL run command line suppresses the listing without altering the source text.

Both SUPPRESS and NOSUPPRESS can appear in the source text.

SYMBOLS Directive

The SYMBOLS directive directs TAL to include a symbol table (for INSPECT symbolic debugging) in the object file.

The default is NOSYMBOLS.

The syntax for the SYMBOLS directive is:

```
[NO]SYMBOLS
```

You can specify the SYMBOLS directive on a procedure-by-procedure or a block-by-block basis. For symbols in a procedure, specify SYMBOLS before the PROC declaration. For symbols in a global data block, specify SYMBOLS before the BLOCK declaration or the first global declaration in an implicit block.

After debugging the program, you can delete symbol tables from the object file by using the BINDER. BINDER provides two methods:

1. This method creates a new object file without symbols and copies it to the target file. The old object file remains intact.

   ```
   ADD * FROM oldobj
   SET SYMBOLS OFF
   BUILD newobj
   ```

2. This method deletes both symbol and BINDER tables from the old object file and does not copy it to the target file. You can no longer use a binder process to examine or modify the file. Before deleting the tables, you can save the file by using the BACKUP program described in the GUARDIAN Operating System User's Guide.

   ```
   STRIP oldobj
   ```

Refer to the BINDER Manual for more information.

Example

This example includes symbols in a procedure and a global data block:

```
NAME the^unit;
?SYMBOLS            !Include symbols in implicit block
  INT a;
  STRING b;
?NOSYMBOLS          !Stop symbols

BLOCK global^data;
  FIXED c;
  STRING d;
END BLOCK;

?SYMBOLS            !Include symbols in procedure
PROC uxb;
BEGIN
  .
  .
END;
```

SYNTAX Directive

The SYNTAX directive requests a syntax check of the source text without object code generation.

The syntax for the SYNTAX directive is:

```
SYNTAX
```

Specifying SYNTAX does not affect the CROSSREF directive. TAL can generate a cross-reference listing even if it produces no object file.

TAL automatically starts BINSERV, which is not needed if TAL produces no object file. To prevent TAL from starting BINSERV, specify SYNTAX on the command line, or to stop BINSERV, specify SYNTAX early in the source text.

WARN Directive

For NonStop software, the WARN directive prints a selected warning or all warnings. For NonStop 1+ software, it prints on all warnings.

NOWARN prevents printing of warnings. The default is WARN.

The syntax for the WARN directive is:

```
[NO]WARN [ <value> ]

<value>

   is the number of a warning message; <value> applies only to
   NonStop software.
```

Even if NOWARN is in effect, the total count of warnings that appears in the trailer includes all warnings, whether printed or not.

Using NOWARN to suppress a warning is useful when your compilation produces a warning and you have determined that no real problem exists. Precede the source line that produces the message with NOWARN and the number of the warning message you want suppressed.

To print selected warnings, you must first specify WARN. If you enter NOWARN first, any subsequent WARN <value> diectives have no effect.

Example

1. This example disables the printing of all warning messages:

```
?NOWARN
```

2. This example, which applies only to NonStop software, disables the printing of warning message 12:

```
?NOWARN 12
```

SECTION 21

COMPILER LISTING

This section describes the TAL listing and gives brief samples of the information. A TAL listing can consist of:

- Header
- Banner
- Compiler Messages
- Source Listing
- Local or Sublocal Map
- CODE Listing
- ICODE Listing
- Global Map
- Cross-Reference Listings
- LMAP Listings
- Compilation Statistics

HEADER

The header for each page consists of:

- The listing page number
- The name of the current source file
- The sequence number for the current source file
- The date and time of compilation in the form mm/dd/yy hh:mm:ss (not shown in the examples that follow) in the right-hand corner of all pages after the first
- An optional page heading caused by the PAGE directive or by TAL

In a listing for multiple source files, the pages containing load maps, cross references, and statistics show the name and number of the first file. The sample headers in Figure 21-1 show the case of a multisource file listing.

```
page num  source file name     num  optional heading

 PAGE 1   $VOL.PROG1.SOURCE1S  [1]
 PAGE 2   $VOL.PROG1.SOURCE2S  [2]  MY ROOT SOURCE FILE
 PAGE 3   $VOL.PROG1.SOURCE2S  [2]  MY ROOT SOURCE FILE
 PAGE 4   $SHR.MSGXX.IMSGSHRS  [3]  INTERPROCESS MESSAGES
 PAGE 59  $VOL.PROG1.SOURCE1S  [1]  GLOBAL MAP
 PAGE 66  $VOL.PROG1.SOURCE1S  [1]  LOAD MAPS
 PAGE 70  $VOL.PROG1.SOURCE1S  [1]  BINDER AND COMPILER STATISTICS
```

Figure 21-1. Page Headers

BANNER

The first page of the listing contains a banner with the heading:

- Compiler version
- Date and time at the start of this compilation
- Language and target machine
- Default options

Figure 21-2 shows a two-line sample banner that is folded only for illustration.

```
TAL - T9250B00 - 28JAN85    SOURCE LANGUAGE: TAL -
                       TARGET MACHINE:  TANDEM NONSTOP II SYSTEM

DATE - TIME :  2/11/85 - 13:47:47
  DEFAULT OPTIONS:  ON  (LIST,CODE,MAP, WARN,LMAP) -
                    OFF (ICODE,INNERLIST)
```

Figure 21-2. Banner

COMPILER MESSAGES

When TAL detects unusual conditions, it issues diagnostic messages conditions interleaved with source statements. (See Appendix C for compiler error and warning messages.)

BINSERV diagnostic messages appear during and after the source listing. (See the BINDER Manual for BINSERV messages.)

SOURCE LISTING

If the LIST directive is in effect (the default), the source text for each procedure is listed line by line. Each line consists of:

- The edit-file line number
- The offset from the procedure base of the generated code
- Lexical (nesting) level of source text
- BEGIN-END pair counter
- Text line from source file

Edit-File Line Number

An edit-file line number precedes each line of source text. Directives entered in the command line appear before the contents of the edit file without line numbers. For text read in response to a SOURCE directive, the edit-file line numbers correspond to the file named in the SOURCE directive.

Code-Address Field

The code address is a six-digit octal number. Depending on the line of source text, it represents an instruction offset or a secondary global count.

For a line of data declarations, the code-address value is a cumulative count of the amount of secondary global storage allocated for the program. The count is relative to the beginning of the secondary global storage. The beginning address is one greater than the last address assigned to primary global storage.

For a line of instructions, the code-address value is the address of the first instruction generated from the TAL source statement on the line. Normally, the octal value is the offset from the base of the current procedure; the instruction at the base has an offset of zero. Adding the offset to the procedure base address yields the code-relative address of the instruction. The procedure base address is listed in the entry-point load map (described later in this section).

If the ABSLIST directive is in effect, TAL attempts to list the address for each line relative to location C[0]. The limitations on the use of ABSLIST are given in the description of the directive in Section 20.

If a procedure or subprocedure has initialized data declarations, TAL emits code to initialize the data at the start of the procedure or subprocedure. The offset or address listed for the first instruction is greater than one to allow for the initialization code.

Lexical-Level Counter

The lexical-level counter is a single-digit number. It represents the compiler's interpretation of the current source line, as follows:

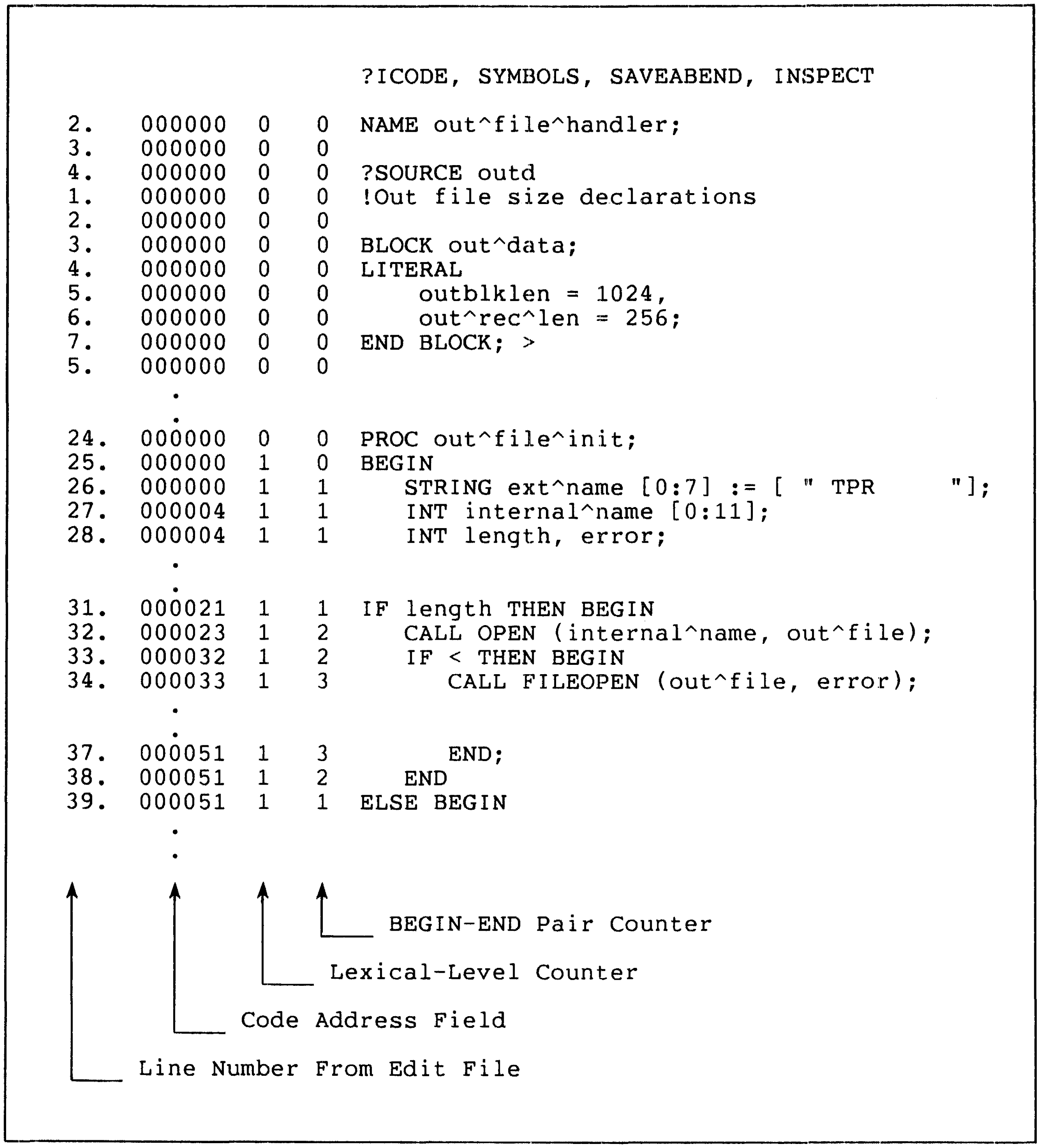
Value	Lexical Level
0	Global level
1	Procedure level
2	Subprocedure level

BEGIN-END Pair Counter

The BEGIN-END pair counter indicates nesting of procedures and subprocedures.

TAL counts BEGIN keywords and matches each BEGIN with an END keyword in STRUCT declarations and in instruction-generating code by incrementing the counter for each BEGIN and decrementing it for each END. TAL displays the value of the counter for each line of source text.

Figure 21-3 is a sample listing page in which TAL reads text from another file (see SOURCE directive in line 4).



```
                         ?ICODE, SYMBOLS, SAVEABEND, INSPECT

   2.   000000  0   0  NAME out^file^handler;
   3.   000000  0   0
   4.   000000  0   0  ?SOURCE outd
   1.   000000  0   0  !Out file size declarations
   2.   000000  0   0
   3.   000000  0   0  BLOCK out^data;
   4.   000000  0   0  LITERAL
   5.   000000  0   0      outblklen = 1024,
   6.   000000  0   0      out^rec^len = 256;
   7.   000000  0   0  END BLOCK; >
   5.   000000  0   0
          .
          .
  24.   000000  0   0  PROC out^file^init;
  25.   000000  1   0  BEGIN
  26.   000000  1   1     STRING ext^name [0:7] := [ " TPR     "];
  27.   000004  1   1     INT internal^name [0:11];
  28.   000004  1   1     INT length, error;
          .
          .
  31.   000021  1   1  IF length THEN BEGIN
  32.   000023  1   2     CALL OPEN (internal^name, out^file);
  33.   000032  1   2     IF < THEN BEGIN
  34.   000033  1   3        CALL FILEOPEN (out^file, error);
          .
          .
  37.   000051  1   3        END;
  38.   000051  1   2     END
  39.   000051  1   1  ELSE BEGIN
          .
          .
```

Figure 21-3. Source Listing

LOCAL OR SUBLOCAL MAP

If the MAP directive in effect (the default), the map of local or sublocal identifiers follows the corresponding source listing. This map gives the following information:

- Class--VARIABLE, SUBPROC, ENTRY, LABEL, DEFINE, or LITERAL

  For STRUCT variables, it is "VARIABLE,<n>" (where <n> is an octal value giving the length in bytes).

- Type--The contents of this field depend on the identifier class:

  --For the VARIABLE class, the type is STRING, INT, INT(32), REAL, REAL(64), FIXED, STRUCT, STRUCT-I, SUBSTRUCT, or TEMPLATE (bytes in octal). STRUCT-I means an INT structure pointer.

  --For the SUBPROC, LABEL, ENTRY, and DEFINE classes, this field is blank.

- The next field is one of:

  --Address Mode (DIRECT or INDIRECT)

  --Offset of SUBPROC, ENTRY, or LABEL in the form "%nnnnnn"

  The offset is relative to the base of the mapped PROC or SUBPROC. For nested subprocedures, the base corresponds to the current map.

  --Value declared for a LITERAL or DEFINE

  TAL prints DEFINE values to the end of the listing line, then truncates the rest.

- Relative Address--For data, it consists of the base (L+, L-, P+, or X) and the offset from the base in octal:

  --L+<nnn> for local variables

  --L-<nnn> for parameters

  --P+<nnn> for read-only (P-relative) arrays

  --X 00<n> for index registers

Figure 21-4 shows a local map corresponding to the following function procedure:

```
INT PROC compute^hash ( name, table^length );
  INT     .name;
  INT(32) table^length;
BEGIN
  INT     int^table^length := $INT(table^length);
  INT     hash^val := 0;
  USE     name^index;
  USE     name^limit;

  name^limit := name.<8:14>;
  FOR name^index  :=  0 TO  name^limit  DO
    hash^val := ((hash^val '<<' 3) LOR hash^val.<0:2>)
                                         XOR name[ name^index ];
  RETURN $UDBL($INT (hash^val '*' 23971)) '\' int^table^length;
END; !compute^hash
```

Identifer Name	Class	Type	Address Mode	Relative Address
HASH^VAL	VARIABLE	INT	DIRECT	L+002
INT^TABLE^LENGTH	VARIABLE	INT	DIRECT	L+001
NAME	VARIABLE	INT	INDIRECT	L-005
NAME^INDEX	VARIABLE	INT	DIRECT	X 007
NAME^LIMIT	VARIABLE	INT	DIRECT	X 006
TABLE^LENGTH	VARIABLE	INT(32)	DIRECT	L-004

Figure 21-4. Local Map

CODE LISTING

If CODE (the default) and LIST are in effect, TAL produces an octal code listing following the local map if one exists.

Figure 21-5 shows a sample CODE listing corresponding to the previous hash procedure. The octal address in the left-hand column is the offset from the procedure base. (If ABSLIST is in effect, TAL attempts to list code-relative addresses.) Each octal address is followed by eight words of instructions to the end of the procedure.

```
Address   Octal Instruction Words

00000    060704 000110 100000 024711 140705 030101 006177 000116
00010    103777 000136 010410 040402 030003 040402 030115 000011

00020    143705 000012 044402 013767 100000 040402 005135 004243
00030    000202 000111 040401 000203 000100 125006
```

Figure 21-5. CODE Listing

ICODE LISTING

If ICODE and LIST are in effect, TAL produces a instruction mnemonic listing. Figure 21-6 shows a sample ICODE listing that is equivalent to the CODE sample.

```
Address    Instruction Mnemonics

000000   1 LDD   L-004     0 STAR      0      1 LDI    +000
         7 PUSH    711     0 LOAD  L-005,I    0 LRS      01

000006   0 ANRI   +177     7 STAR      6      7 LDXI   -001,7
         0 LDRA      6     0 BUN    +010      1 LOAD  L+002

000014   1 LLS      03     2 LOAD  L+002      2 LRS      15
         1 LOR             2 LOAD  L-005,I,7  1 XOR

000022   0 STOR  L+002     7 BOX    -011,7    0 LDI    +000
         1 LOAD  L+002     2 LDLI   +135      2 ORRI    243

000030   2 LMPY            1 STAR      1      2 LOAD  L+001
         1 LDIV            0 STRP      0      0 EXIT     06
```

Figure 21-6. ICODE Listing

GLOBAL MAP

If MAP is in effect, the global map lists all identifiers in the compilation unit. For NonStop software, GMAP must also be in effect. NOMAP specified at the end of the source file suppresses the global map but not the local maps. Figure 21-7 shows sample entries of a global map.

```
Identifier     Class         Type       Class-Specific Information

ABEND          PROC                     EXTERNAL
ABENDPARAM     DEFINE                   OPTIONS.<10:10>
AB^OPENERR     DEFINE                   %B00000000001D
ACCESS^JNK     DEFINE                   ASSIGN.OPTION1.<05:05>
ACCESS^INFO    VARIABLE      TEMPLATE,402
 1  INCL^LEN           0,2   INT
 1  AC                 2,2   INT
AC^INFO^DEF    DEFINE                   BEGIN  INT INCL^LEN; INT AC[0:
ADD^           LITERAL       INT        %000021
ALL^FCB        DEFINE                   INT.$1[0:FSIZE-1]:=[FSIZE,%000
AP^BLOCK       BLOCK
AP^FILE^OK     PROC          INT        EXTERNAL
BLIST^CTL      VARIABLE,4    STRUCT     INDIRECT    BLST^P=001
COD^PTR        VARIABLE      INT(32)    DIRECT      AP^BLOCK+002
COMPRS         VARIABLE      INT        DIRECT      AP^BLOCK+011
DIMEN^INFO     VARIABLE      TEMPLATE,16
 1  NUM                0,2   INT
 1  DOUCE              2,2   INT
 1  DIM^T              4,12  SUBSTRUCT
     2 LOW^C           4,1   STRING
     2 UP^C            5,1   STRING
     2 LOW^B           6,4   INT^(32)
     2 UP^B           12,4   INT^(32)
FILEINFO       PROC                     EXTERNAL
FNAMECOLLAPSE  PROC                     EXTERNAL
```

Figure 21-7. Global Map

CROSS-REFERENCE LISTINGS

If CROSSREF and LIST are in effect, the cross-reference listings follow the global map. These listings are:

- Source-file cross-reference listing (the first page)
- Identifier cross-reference listing (subsequent pages)

Source-File Cross References

Figure 21-8 shows the source-file cross-reference listing. It gives the following information for each source file in the compilation:

- File sequence number in the compilation
- File name from either the IN <source-file> of the TAL run command or from a SOURCE directive
- Name of the source file that contained the SOURCE directive, if one appears
- Edit-file line number of the SOURCE directive, if one appears

```
CROSSREF - CROSS-REFERENCE PROGRAM - T9622A00 - (01OCT82)

FILE NO.   FILENAME.
 [1]       $VOL.PROG1.SOURCE1S
 [2]       $VOL.PROG1.SOURCE2S           SOURCE1S[1]   0.1
 [3]       $SYSTEM.SYSTEM.GPLDEFS        SOURCE2S[2]   2
 [4]       $VOL.PROG1.SOURCE4S           SOURCE1S[1]   7
 [5]       $SYSTEM.SYSTEM.EXTDECS        SOURCE1S[1]   8
```

Figure 21-8. Source-File Cross-Reference Listing

Identifier Cross References

The identifier cross-reference listing gives the following information about each specified identifier class:

- Identifier qualifiers
- Compiler attributes
- Declaring source file
- Reference lines

Identifier Qualifiers

An item declared within a STRUCT, SUBPROC, or PROC can have from none to three levels of qualifiers (listed immediately following the identifier name). The general form shows the ordering of qualifier levels:

```
OF <struct-name> [ OF <subproc-name> ] OF <proc-name>
```

The qualifier field varies according to the following rules:

- If an identifier has no qualifier, it is a global item.
- If an identifier has one qualifier, it is declared in a global STRUCT or a PROC.
- If an identifier has two qualifiers, it is declared in either a STRUCT or a SUBPROC within a PROC
- If an identifier has three qualifiers, it is declared in a STRUCT within a SUBPROC within a PROC.

Examples of identifiers are:

```
1.  GLOBAL^X

2.  ITEM^A      OF GLOB^STRUCT^OR^PROC

3.  ITEM^B      OF LOC^STRUCT^OR^SUBPROC       OF PROC^P

4.  ITEM^C      OF SUBLOC^STRUCT       OF SUBPROC^Q        OF PROC^P
```

Compiler Attributes

Compiler attributes are class (as specified in the CROSSREF directive) and type modifiers:

Class	Modifiers
BLOCK	none
DEFINE	none
ENTRY	type
LABEL	none
LITERAL	type
PROC	type, EXTERNAL
SUBPROC	type
TEMPLATE	none
VARIABLE	type, DIRECT or INDIRECT
UNDEFINED	none

Types that apply to the ENTRY, PROC, SUBPROC, and LITERAL classes are STRING, INT, INT(32), REAL, REAL(64), and FIXED. Type FIXED includes the scale if it is nonzero.

Types for the VARIABLE class are those listed above plus STRUCT, SUBSTRUCT, STRUCT-I, and STRUCT-S.

Declaring Source File

The abbreviated edit-file name of the declaring source file appears on the same line as the identifier name. The sequence number assigned to the source file appears in brackets. The line number where the declaration starts accompanies the file name. An example is:

```
SOURCE1S[23]  137
```

Reference Lines

Reference lines include an entry for each reference in the compilation. Except for read references, an alphabetic code indicates the type of reference. Codes are D (definition), I (invocation), P (parameter), W (write), and M (other). Refer to the CROSSREF Manual for additional information.

Identifier Cross-Reference Example

The identifier cross-reference pages begin with the format shown in Figure 21-9. The header line (only on the first page of references) lists the total number of symbols referenced and the total number of references.

```
152 TOTAL SYMBOLS COLLECTED WITH 61 TOTAL REFERENCES COLLECTED

ALLOCATE^CBS        DEFINE                     GPLDEFS[3]    15
    GPLDEFS[3]    198
ALLOCATE^FCB        DEFINE                     GPLDEFS[3]    27
    SOURCE2S[2]     5
ASSIGN^BLOCKLENGTH  INT LITERAL                GPLDEFS[3]    81
    GPLDEFS[3]    81.1   135
DEFAULT^VOL         INT DIRECT VARIABLE        SOURCE4S[4]    2
    SOURCE1S[1]   14 W
MESSAGE OF STARTUP  INT INDIRECT VARIABLE      SOURCE1S[1]   12
    SOURCE1S[1]   11 D   14
MSG^CLOSE           EXTERNAL PROC              SOURCE4S[4]   10
    SOURCE1S[1]   28 I
RUCB                INT INDIRECT VARIABLE      SOURCE2S[2]    5
    SOURCE1S[1]   18 P
```

Figure 21-9. Identifier Cross-Reference Listing

LMAP LISTINGS

By default, BINSERV produces an alphabetic load map for entry points and another for data blocks, both ordered by name. If LMAP LOC is in effect, BINSERV produces load maps ordered by location in place of the alphabetic maps. For LMAP*, it produces load maps ordered by name and by location and cross-reference listings.

The load maps are different on the NonStop and the NonStop 1+ system. The sample listings shown in the remainder of this section are for the NonStop system. For sample listings produced on another system type or those showing multiple code segments, see the BINDER Manual.

Entry-Point Load Map

Figure 21-10 shows a sample entry-point load map by name. The fields shown for each entry point are:

SP	Code segment (space) number of the entry point
PEP	Sequence number of the entry point in the PEP table
BASE	Base address of the procedure defining the entry point
LIMIT	End address of the procedure defining the entry point
ENTRY	Address of executable code for the entry point
ATTRS	Attributes of the entry point: C (CALLABLE), E (EXTENSIBLE), I (INTERRUPT), M (MAIN), P (PRIVILEGED), R (RESIDENT), V (VARIABLE)
NAME	Entry-point name
DATE	Date of compilation
TIME	Timestamp of the compilation
LANGUAGE	Source language of the procedure
SOURCE FILE	File name of the source code for the procedure

```
ENTRY POINT MAP BY NAME

SP  PEP   BASE     LIMIT    ENTRY     ATTRS  NAME
                      DATE      TIME     LANGUAGE     SOURCE FILE

00  031   010345   043630   0010420          APROC
                      2/11/85   18:13    TAL         $JNK.PRG1.SRCE1S
00  073   032224   032636   032224     V     APROC^VAR^PARAM
                      2/11/85   10:29    TAL         $JNK.PRG1.SRCE2S
00  020   000736   001072   000736     M     MAIN^PROC
                      2/11/85   13:38    TAL         $JNK.PRG1.MAINS
00  367   131432   131441   131432     E     SORT^PROC
                      2/11/85   18:14    TAL         $JNK.PRG1.SORTS
```

Figure 21-10. Entry-Point Load Map by Name

Data-Block Load Maps

On the NonStop system, BINSERV produces a data-block map and a read-only data-block map for primary and secondary global blocks. These maps include information from NAME and BLOCK declarations described in Section 22, "Separate Compilation."

The data-block map lists the following kinds of data blocks:

- Named BLOCK constructs, listed by the declared name
- BLOCK PRIVATE constructs, listed by the name TAL derives from the NAME declaration prefixed with #
- #GLOBAL and .#GLOBAL (compiler-assigned names for global data declared outside the above blocks)

The read-only data-block map lists global read-only arrays, listed by the declared name.

Both maps give the following information for each data block:

BASE	Base address of the block
LIMIT	End address of the block (blank if block is empty)
TYPE	BINDER data-block type (own, common, or special); for TAL code, ony common blocks can occur
MODE	Word or byte addressing
NAME	Data-block name (see above)
DATE	Date of compilation in the form mm/dd/yy
TIME	Timestamp for the compilation in the form hh:mm
LANGUAGE	Source language of the block
SOURCE FILE	Edit-file name of the source file containing the declaration of the block

Figure 21-11 illustrates a data-block map by location. Figure 21-12 shows the corresponding read-only data-block map; it includes the "SP" column, which gives the code segment number specifier for each read-only array.

```
DATA BLOCK MAP BY LOCATION

BASE     LIMIT    TYPE     MODE      NAME
                     DATE     TIME       LANGUAGE  SOURCE FILE
000000   000014   COMMON   WORD      GLOBAL^
                     2/11/85  13:38      TAL       $VOL.PRG.GLBS
000015   000015   COMMON   WORD      LISM^PUB
```

Figure 21-11. Data-Block Load Map by Location

```
READ-ONLY DATA BLOCK MAP BY LOCATION
CODE SPACE 00

SP   BASE     LIMIT    TYPE     MODE     NAME
                          DATE     TIME      LANGUAGE  SOURCE FILE
00   000025   000417   COMMON   WORD     HASH
                          2/11/85  10:48     TAL       $VOL.PRG.SRC1S
00   000055   000442   COMMON   WORD     FIND^TAB
                          2/11/85  10:48     TAL       $VOL.PRG.SRC1S
```

Figure 21-12. Read-Only Data-Block Load Map by Location

COMPILATION STATISTICS

TAL prints compilation statistics at the end of each compilation. If SYNTAX is in effect or if source errors occur, TAL does not print any other statistics. Figure 21-13 shows the statistics emitted when source errors stop the compilation.

```
PAGE 3  $TRMNL [0]           BINDER AND COMPILER STATISTICS

  Number of compiler errors = 5
  Last compiler error on page # 2 IN PROC C
  Number of compiler warnings = 1
  Last compiler warning on page # 1
  Maximum symbol table space used was =      562 bytes
  Number of source lines= 22
  Elapsed time -  00:02:58
```

Figure 21-13. Compiler Statistics

Object-File Statistics

If an object file results from the compilation, TAL prints the following BINSERV statistics preceding the compiler statistics:

- Name of the constructed object file
- Number of binder error messages issued
- Number of binder warning messages issued
- Number of words of primary data area
- Number of words of secondary data area
- Number of resident pages required for total code space allocation
- Minimum number of pages required for data space allocation
- Number of code spaces (segments)

Figure 21-14 shows sample BINSERV statistics:

```
PAGE 91 \SYS.$VOL.SUBV.SRC [1]   BINDER AND COMPILER STATISTICS

BINDER - OBJECT FILE BINDER - T9621B00 - (28JAN85)  SYSTEM \XXX
Object file name is $XVOL.XSUBVOL.OFILE
Number of Binder errors = 0
Number of Binder warnings = 1
Primary data = 184 words
Secondary data = 10026 words
Code area size = 45 pages
Resident code size = 0 pages
Data area size = 64 pages
Number of code spaces = 1 space

The object file will run on a TNS/II, but may not run on a TNS
Number of compiler errors = 0
Number of compiler warnings = 0
Maximum symbol table space used was =      128338 bytes
Number of source lines = 6467
Elapsed time - 00:07:47
```

Figure 21-14. Object-File Statistics

Since the compilation unit includes SEARCH directives that cause previously compiled object code to be bound with the source code, the number of source lines is small compared to the generated code.

If a compilation ends due to a BINSERV error, TAL prints statistics including the BINSERV banner and the number of BINSERV errors and warnings.

SECTION 22

SEPARATE COMPILATION

TAL supports modular programming with separate compilation and relocatable global data blocks. You can compile any module consisting of one or more procedures as a separate compilation unit. You can then bind the separately compiled object files into an executable object file called the target file by using BINSERV (compile-time binder process) or BINDER (stand-alone binder).

This section describes the features that support separate compilation and the data-space image that results. It describes:

- The NAME declaration for naming a compilation unit
- The BLOCK declaration for declaring relocatable global data blocks
- Binding compilation units
- Data-space image
- Sample modules

NAME DECLARATION

The NAME declaration assigns a name to a compilation unit and to its private data block if it has one.

The syntax for the NAME declaration is:

```
NAME <identifier> ;


<identifier>

   is the name of the compilation unit.  If it has a private
   data block, no other compilation unit in the target file can
   use the same name at the global level.  If this compilation
   unit has no private block, the name is global within this
   unit only.
```

If a compilation unit has a BLOCK declaration, the NAME declaration must be the first declaration in the compilation unit. NAME is a reserved word only in the first declaration; you can use "name" elsewhere as an identifier.

A compilation unit that has a NAME declaration as its first declaration is called a named compilation unit.

Example

The following example names a compilation unit:

```
NAME calc^mod;
```

BLOCK DECLARATION

The BLOCK declaration lets you group global data declarations into a named or private relocatable global data block.

The syntax for the BLOCK declaration is:

```
BLOCK   { <identifier> } ;
        { PRIVATE      }

   [ <data-declaration> ; ] ...

END BLOCK;
```

<identifier>

is the name of the data block. The name must be unique among all BLOCK and NAME declarations in the target file. A named data block is accessible to other compilation units.

PRIVATE

indicates a private global data block that is accessible only to this compilation unit.

<data-declaration>

is a global data declaration of any variable described in Sections 8 through 12.

If you use the BLOCK declaration, the first declaration in the compilation unit must be the NAME declaration. In a named compilation unit, BLOCK and PRIVATE are reserved words.

You can declare only one private block in a compilation unit. TAL gives the private block the name you specify in the NAME declaration for this compilation unit.

You can declare any number of named data blocks in a compilation unit.

Examples

1. This example declares a private global data block:

```
BLOCK PRIVATE;
  INT term^num;
  LITERAL msg^buf = 79;
END BLOCK;
```

2. This example declares a named global data block:

```
BLOCK default^vol;
  INT .vol^array [0:7],
      .out^array [0:34];
END BLOCK;
```

Rules for Coding Data Blocks

- The correct order of global declarations is:

  --NAME declaration

  --Unblocked global data declarations

  --Named blocks and the private block

  --PROC declaration

- All unblocked global declarations (those not contained in BLOCK declarations) must appear before the first BLOCK declaration. TAL treats the unblocked declarations as an implicit data block and gives it the name #GLOBAL. A compilation unit can have only one implicit block.

  If more than one compilation unit in a binding session has an implicit block, binding does not combine the implicit blocks. BINSERV binds in only the implicit block in the source code, not those in search lists. You can use BINDER commands to replace the implicit block in the target file.

- The sum of the primary global blocks in the target file must not exceed 256 words.

- All variables referenced in a data block must be declared in the same block. For example, the following declarations must appear in the same block:

```
INT .var;
INT .ptr := @var;
```

- The length of any shared data block must match in all compilation units; use a separate source file for each block.

Sharing Data Blocks

The SECTION and SOURCE directives allows sharing of global data blocks among compilation units. In the compilation unit that declares the data block, use the SECTION directive to give the block a section name. In compilation units that need to include the data block, specify the section name in a SOURCE directive.

If you change any data block declaration, you must recompile all compilation units that use the changed data block.

The following example shows a compilation unit that declares global data blocks and a second compilation unit that specifies the blocks:

```
!Source file name "calc^src"
NAME calc^unit;                         !First compilation unit

?SECTION unblocked^globals              !Name of first section
  LITERAL true   = -1,                  !Implicit data block
          false  =  0;
  STRING read^only^array = 'P' := [ " ","COBOL", "FORTRAN",
                                   "MUMPS", "TAL"];

?SECTION default                        !Name of second section
  BLOCK default^vol;                    !Declares named block
    INT .vol^array [0:7],
        .out^array [0:34];
  END BLOCK;

?SECTION msglits                        !Name of third section
  BLOCK msg^literals;                   !Declares named block
    LITERAL
      msg^eof   = 0,
      msg^open  = 1,
      msg^read  = 2,
  END BLOCK;
?SECTION dummy                          !Ends msglits section

!----------------------------------------------------------!

NAME input^file;                        !Second compilation unit

?SOURCE calc^src(unblocked^globals) !Specifies implicit block
?SOURCE calc^src(default)           !Specifies a named block
```

BINDING COMPILATION UNITS

You can bind compilation units:

- In a compilation session
- After compilation using BINDER commands
- At run time (library binding)

Compile-Time Binding

After a successful compilation, BINSERV binds the new object file with external procedures from the search list that resolve external references. If the external procedure also contains references to other external procedures or to data blocks, BINSERV tries to resolve those from the search list. (A compilation unit must declare all data in that unit.) The object files produced by BINSERV can serve as input to further binding operations by BINSERV, BINDER, or the operating system.

SEARCH directives must list file names in the order in which BINSERV is to search them. Source files that contain additional SEARCH directives can alter the apparent order. The order might be significant if a procedure or entry-point name occurs in more than one search file. BINSERV binds only the first occurrence.

The following example shows SEARCH directives for a search in the order "file1o," "file2o," "file3o," and "file4o":

```
?SEARCH (file1o, file2o)
?SEARCH (file3o, file4o)
```

The following example shows SEARCH directives for external procedures:

```
?SEARCH partxo          !Object file containing proc^x
PROC proc^x;
EXTERNAL;

?SEARCH partyo          !Object file containing proc^y
PROC proc^y;
EXTERNAL;

PROC proc^z;
BEGIN
  CALL proc^x;
  CALL proc^y;
END;
```

Interactive Binding

After compilation, you can bind object files independently of the compiler by using the BINDER. The BINDER Manual describes interactive binding and lists the commands you can use.

For example, you can use the BINDER to build a target file from separate object files, display object-file contents, reorder target-file code blocks, produce optional load maps and cross-reference listings, specify a user run-time library (NonStop systems only), and modify the contents of named global data and code blocks in the target file.

Run-Time Library Binding

On NonStop systems, you can build a run-time library of procedures to share among applications or to extend a single application's code space. Do not bind the procedures in a run-time library with the program file. Instead, store them in a separate file, then associate them with the program file by using any of the following methods:

- A LIBRARY directive in the source file
- A BINDER command
- The COMINT RUN command

The operating system performs run-time binding of a library file to the program file. The BINDER Manual describes user libraries.

DATA-SPACE IMAGE

TAL constructs relocatable blocks of code and data that are bound into the object file. A procedure is the smallest relocatable code block. A global data block is the smallest relocatable unit of data.

Relocatable Global Data Blocks

Data blocks in separate compilations and in mixed-language programs must be relocatable. TAL allocates 'G' relative offsets to relocatable data blocks and places read-only arrays in the user code segment in which you reference them. The primary global space occupies the first 256 words of the data space.

TAL creates the following relocatable blocks, as needed:

Unnamed Compilation Unit	Named Compilation Unit
Implicit primary global data	Implicit primary global data
Implicit secondary global data	Implicit secondary global data
	Explicit primary global data
	Explicit secondary global data

For primary data, TAL names the implicit block #GLOBAL and gives the private block the name used in the NAME declaration. For secondary data, TAL uses names made up of the primary block names plus a period symbol, as in .#GLOBAL.

Data-Space Image Example

Figure 22-1 shows the data-space image resulting from binding two separately compiled units, "unit^1" and "unit^2".

The compilation units contain global declarations for an implicit data block, five named data blocks, and two private data blocks. The figure assumes that G[0] is available for compiled global data.

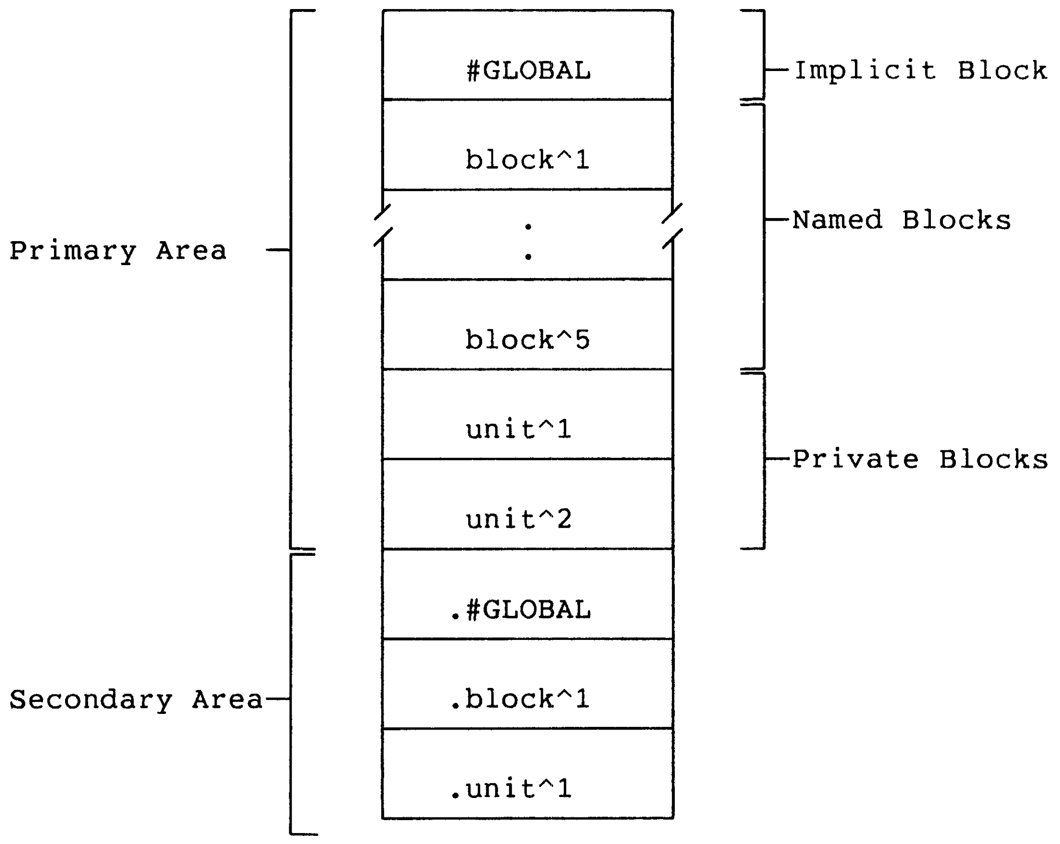


Figure 22-1. Global Data-Space Allocation

Address Assignments

During compilation, TAL assigns each direct variable and pointer an offset from the beginning of each relocatable block. TAL allocates storage according to the type and size of the individual data declarations in each block.

When you bind the object code, the BINDER uses the address of the data block and the offset within the block to construct addresses for indirect data in the secondary data space.

Directives for Relocatable Data

The RELOCATE and INHIBITXX compiler directives help you manage relocatable data:

- RELOCATE--This directive causes TAL warnings if declarations depend on 'G' relative addresses (such declarations might not give the correct results if data relocation occurs); it also causes BINSERV warnings if references to nonrelocatable global data occur.
- INHIBITXX--This directive prevents TAL from generating indexed extended instructions for extended pointers, since relocation of the data blocks can result in an extended pointer outside the first 64 words of primary global space.

For more information on these directives, see Section 20.

SEPARATE COMPILATION SAMPLE PROGRAM

This example is a utility program that converts records in the input file to a different format and length by reordering fields and adding fields to records. The example includes:

- A brief description of program characteristics
- Partial listings of module code
- Load maps for the program file
- Compilation statistics (compile and bind) for the program file

Selected listings show the handling of data and program structure. BLOCK contents appear only in the module that declares them. In modules that reference the blocks, NOLIST directives prevent listing of block contents.

Compilation maps and statistics are not shown for each module. Load maps show entries for blocks that do not exist after compilation such as LITERALs. The mainline load map does not refer to these blocks.

Program Structure

The program consists of five modules, each of which performs a single operation. The structure of the modules and their procedures allows changes to one operation without the need to recompile the others.

Information is accessible across modules on an as-needed basis. They share named global data blocks and pass information as parameters and local data such as a pointer to the locally declared record buffer. The named global data block "default^vol" contains shared run-time data. Other named blocks declare structure templates for record definitions and LITERAL declarations, which use no memory.

Procedures within a module share global data in private blocks.

Table 22-1 summarizes the blocks used by each module. In the table, the symbol (P) means a private block.

Table 22-1. Data Blocks by Module

Module Name	Blocks Defined	Blocks Referenced
tpr^convert	record^defs	msg^literals
initialization^module	default^vol	none
message^module	msg^literals message^module (P)	default^vol
in^file^handler	in^data in^file^handler (P)	msg^literals default^vol
out^file^handler	out^data out^file^handler (P)	default^vol msg^literals

File Naming Conventions

The example uses the following file naming conventions:

- Source file names end with the character "s".
- Object file names correspond to source file names and end with "o". For instance, the object file built from the source file "ins" is named "ino".
- Data file names end in "d" if they belong to a specific module. For instance, "ind" is the source file that contains LITERAL declarations for "ins".
- File names ending with "p" contain EXTERNAL declarations of the procedures in the module with the corresponding name. A module that calls an external procedure includes a SOURCE directive for

the "p" file. For instance, the source for "message^module" is file "msgs", and source file "msgp" declares each EXTERNAL PROC in "msgs". The modules that call "message^module" specify "msgp" in a SOURCE directive.

If any external declarations change, you must recompile both the "p" file and any module that calls a changed external procedure. The "p" file enables compile-time consistency checking between procedure declarations and the corresponding external declarations.

A module also uses a "p" file for its external procedure declarations. Module "xxx^s" uses a SOURCE directive to specify "xxx^p", which contains EXTERNAL declarations for its procedures. (Otherwise, the consistency check is possible only during a later binding.)

Mainline Module

Although the mainline module was compiled last because it contains a search list for the other modules, it is listed first to illustrate the overall logic.

The record-definition STRUCTs are not listed because they are translations of the Data Definition Language (DDL) source code into TAL.

```
NAME tpr^convert;

BLOCK record^defs;
?NOLIST, SOURCE inrec            !STRUCT (*) "in^rec^def"
?NOLIST, SOURCE outrec           !STRUCT (*) "out^rec^def"
END BLOCK;

?NOLIST, SOURCE msglit           !BLOCK "msg^literals"

                                 !EXTERNAL PROC declarations:
?NOLIST,SOURCE inp               ! "in^file^handler"
?NOLIST,SOURCE outp              ! "out^file^handler"
?NOLIST,SOURCE msgp              ! "message^module"
?NOLIST,SOURCE initp             ! "initialization^module"

                                 !Search file list:
?SEARCH ino                      ! "in^file^handler"
?SEARCH outo                     ! "out^file^handler"
?SEARCH msgo                     ! "message^module"
?SEARCH inito                    ! "initialization^module"

PROC out^init (out^rec);         !Initializes output record
  STRUCT .out^rec (out^rec^def);
BEGIN
?NOLIST
END;
```

```
PROC record^convert (in^rec, out^rec);
  STRUCT .in^rec (in^rec^def);     !Converts between two records
  STRUCT .out^rec (out^rec^def);
BEGIN
?NOLIST
END;

PROC convert;
BEGIN
  INT     record^count := 0;
  STRUCT .in^buffer  (in^rec^def);
  STRUCT .out^buffer (out^rec^def);

  WHILE (read^in (in^buffer)) <> 1 DO
  BEGIN                            !Reads record, returns EOF
    CALL out^init (out^buffer);    !Initializes output
    CALL record^convert (in^buffer, out^buffer);
    CALL write^out (out^buffer);
    record^count := record^count + 1;
  END;  !Of WHILE loop
  !EOF
  CALL msg (msg^Eof, record^count);
END;  !Of "convert"

PROC tprconv MAIN;
BEGIN
  CALL file^init;                  !In "initialization^module"
  CALL file^init;
  CALL convert;
  CALL close^all;
END;
?NOMAP
```

Initialization Module

This module defines a primary global data block, default^vol. The block is accessible to all procedures in the modules that declare the block for reference.

```
NAME initialization^module;

?SECTION default
BLOCK default^vol;                 !Default volume, subvolume
  INT default^vol^subvol [0:7];
END BLOCK;

?NOLIST,SOURCE $SYSTEM.SYSTEM.EXTDECS (INITIALIZER)
                                   !EXTERNAL PROC declarations:
?NOLIST,SOURCE outp                !  "out^file^handler"
?NOLIST,SOURCE inp                 !  "in^file^handler"
?NOLIST,SOURCE msgp                !  "message^module"
```

```
   ?NOLIST,SOURCE initp                ! "initialization^module"
                                       ! (for consistency checks)

   PROC startup (rucb, passthru, message, meslen, match) VARIABLE;
   INT .rucb, .passthru, .message, meslen, match;
   BEGIN
     default^vol^subvol ':=' message[1] FOR 8;
   END;

   PROC file^init;
   BEGIN
     CALL INITIALIZER (,,startup);
     CALL msg^init;
     CALL in^file^init;
     CALL out^file^init;
   END;

   PROC close^all;
   BEGIN
     CALL in^close;
     CALL out^close;
     CALL msg^close;
   END;
   ?NOMAP
```

Input File Module

The input file handler contains all procedures that manipulate that file. Therefore, if I/O changes are required, only this module needs to be recompiled. The initialization module, for example, calls a procedure in this module.

This module declares a private block that is accessible only to the procedures in this module. It is allocated in primary global storage.

```
   NAME in^file^handler;

   ?SECTION ind                        !In-file declarations
   BLOCK in^data;
     LITERAL
       inblklen    = 1536,
       in^rec^len  = 555;
   END BLOCK;

   ?NOLIST, SOURCE default
   ?NOLIST, SOURCE msglit

   BLOCK PRIVATE;
     INT in^file;                      !Input file number
   END BLOCK;
```

```
?NOLIST,SOURCE $SYSTEM.SYSTEM.EXTDECS (ABEND, CLOSE, FILEINFO)
?NOLIST,SOURCE $SYSTEM.SYSTEM.EXTDECS (FNAMEEXPAND, OPEN, READ)
                                  !EXTERNAL PROC declarations:
?NOLIST,SOURCE msgp               ! "message^module"
?NOLIST,SOURCE inp                ! "in^file^handler"
                                  ! (consistency checks)
PROC in^file^init;
BEGIN
  STRING ext^name [0:7] := ["OLDTPR  "];
  INT int^name [0:11];
  INT length, error;

  length := FNAMEEXPAND (ext^name, int^name, default^vol^subvol);
  IF length THEN
    BEGIN
      CALL OPEN (int^name, in^file);
      IF < THEN
        BEGIN
          CALL FILEINFO (in^file, error);
          CALL msg (msg^in^open, error);
          CALL ABEND;
        END;
    END  !Of THEN clause
  ELSE
    BEGIN
      CALL msg (msg^in^name, 0);
      CALL ABEND
    END;  !Of ELSE clause
END;  !Of "file^init"

INT PROC read^in (rec);
  INT .rec;
BEGIN
  INT error;

  CALL READ (in^file, rec, in^rec^len);
  IF < THEN
    BEGIN
      CALL FILEINFO (in^file, error);
      CALL msg (msg^read, error);
    END;
  RETURN IF > THEN 1
              ELSE 0;
END;  !Of "read^in"

PROC in^close;
BEGIN
  CALL CLOSE (in^file);
END;
?NOMAP
```

Output File Module

The private block declared in this module is allocated in primary global storage and is accessible only to procedures in this module. Some of the parallel code to the input file handler is not listed.

```
NAME out^file^handler;

?SECTION outd                                  !Out-file declarations
BLOCK out^data;
  LITERAL
    outblklen   = 1024,
    out^rec^len = 256;
END BLOCK;
?NOLIST, SOURCE default                        !BLOCK "default^vol"
?NOLIST, SOURCE msglit                         !BLOCK "msg^literals"

BLOCK PRIVATE;
  INT out^file;
END BLOCK;
?NOLIST

PROC out^file^init;
BEGIN
  STRING ext^name [0:7]  := [ "CURR    " ];
  INT int^name    [0:11];
  INT length,error;
?NOLIST
END;

PROC write^out (rec);
  INT .rec;
BEGIN
  INT error;
  CALL WRITE (out^file, rec, out^rec^len);
  IF < THEN
    BEGIN
      CALL FILEINFO (out^file, error);
      CALL msg (msg^write, error);
    END;
END;  !Of "write^out"

PROC out^close;
BEGIN
  CALL CLOSE (out^file);
END;
?NOMAP
```

Message Module

The terminal number in the private block is allocated in primary global storage and is accessible only to procedures in this module.

```
NAME message^module;

?SECTION msglit                  !Defines BLOCK "msg^literals"
BLOCK msg^literals;
  LITERAL
    msg^Eof      = 0,
    msg^in^open  = 1,
    msg^in^name  = 2,
    msg^read     = 3,
    msg^out^open = 4,
    msg^out^name = 5,
    msg^write    = 6;
END BLOCK;

BLOCK PRIVATE;
  INT term^file^number;
  LITERAL msg^buf^end = 79;
END BLOCK;

?NOLIST,SOURCE $SYSTEM.SYSTEM.EXTDECS (CLOSE, MYTERM, OPEN)
?NOLIST,SOURCE $SYSTEM.SYSTEM.EXTDECS (NUMOUT,WRITE)

PROC msg^init;
BEGIN
  INT .term^name [0:11];

  CALL MYTERM (term^name);
  CALL OPEN (term^name, term^file^number);
END;

PROC msg (mnumber, altnum);
  INT mnumber, altnum;
BEGIN
  STRING .buffer [0:msg^buf^end];
  INT .ibuffer := @buffer '>>' 1;
  STRING .bufptr;
```

```
    CASE mnumber OF
      BEGIN
        !msg^eof!
        buffer ':='  " *** End of File "         -> @bufptr;
        !msg^in^open!
        buffer ':='  " *** In file open failed " -> @bufptr;
        !msg^in^name!
        buffer ':='  " *** Bad in file name "    -> @bufptr;
        !msg^read!
        buffer ':='  " *** Read error "          -> @bufptr;
        !msg^out^open!
        buffer ':='  " *** Out file open failed "-> @bufptr;
        !msg^out^name!
        buffer ':='  " *** Bad out file name "   -> @bufptr;
        !msg^write!
        buffer ':='  " *** Write error "         -> @bufptr;
        OTHERWISE;
      END;

      IF altnum <> 0 THEN
        BEGIN
          CALL NUMOUT (bufptr, altnum, 10, 5);
          @bufptr := @bufptr + 5;
        END;

    CALL WRITE (term^file^number, ibuffer, @bufptr - @buffer);
  END;  !Of "msg"

  PROC msg^close;
  BEGIN
   CALL CLOSE (term^file^number);
  END;
  ?NOMAP
```

Compilation Maps and Statistics

Figures 22-2, 22-3, and 22-4 show the entry-point load map, data-block load map, and statistics for the mainline compilation.

```
ENTRY POINT MAP BY NAME

SP  PEP   BASE     LIMIT    ENTRY    ATTRS   NAME
                      DATE      TIME    LANGUAGE  SOURCE FILE

00  012   000737   000742   000737           CLOSE^ALL
                   2/11/85  13:5    TAL        $VOL.PRG.INITS
00  004   000266   000331   000266           CONVERT
                   2/11/85  13:5    TAL        $VOL.PRG.CONVERTS
00  011   000721   000736   000721           FILE^INIT
                   2/11/85  13:5    TAL        $VOL.PRG.INITS
00  017   001133   001140   001133           IN^CLOSE
                   2/11/85  12:5    TAL        $VOL.PRG.INS
00  015   000767   001050   000773           IN^FILE^INIT
                   2/11/85  12:5    TAL        $VOL.PRG.INS
00  010   000421   000720   000421           MSG
                   2/11/85  12:5    TAL        $VOL.PRG.MSGS
00  021   001147   001154   001147           MSG^CLOSE
                   2/11/85  12:5    TAL        $VOL.PRG.MSGS
00  014   000751   000766   000751           MSG^INIT
                   2/11/85  12:5    TAL        $VOL.PRG.MSGS
00  020   001141   001146   001141           OUT^CLOSE
                   2/11/85  12:5    TAL        $VOL.PRG.OUTS
00  016   001051   001132   001055           OUT^FILE^INIT
                   2/11/85  12:59   TAL        $VOL.PRG.OUTS
00  002   000022   000111   000022           OUT^INIT
                   2/11/85  13:54   TAL        $VOL.PRG.CONVERTS
00  006   000340   000372   000340           READ^IN
                   2/11/85  12:59   TAL        $VOL.PRG.INS
00  003   000122   000265   000122           RECORD^CONVERT
                   2/11/85  13:54   TAL        $VOL.PRG.CONVERTS
00  013   000743   000750   000743   V       STARTUP
                   2/11/85  13:52   TAL        $VOL.PRG.INITS
00  005   000332   000337   000332   M       TPRCONV
                   2/11/85  13:54   TAL        $VOL.PRG.CONVERTS
00  007   000373   000420   000373           WRITE^OUT
                   2/11/85  12:59   TAL        $VOL.PRG.OUTS
```

Figure 22-2. Entry-Point Load Map of Mainline Compilation

```
DATA BLOCK MAP BY NAME

 BASE     LIMIT     TYPE     MODE         NAME
                      DATE     TIME         LANGUAGE  SOURCE FILE

000003   000012    COMMON   WORD         DEFAULT
                     2/11/85  12:59         TAL       $VOL.PRG.INS
000000   000000    COMMON   WORD         IN^FILE^HANDLER
                     2/11/85  12:59         TAL       $VOL.PRG.INS
000002   000002    COMMON   WORD         MESSAGE^MODULE
                     2/11/85  12:58         TAL       $VOL.PRG.MSGS
000000             COMMON   WORD         MESSAGE^LITERALS
                     2/11/85  12:58         TAL       $VOL.PRG.MSGS
000001   000001    COMMON   WORD         OUT^FILE^HANDLER
                     2/11/85  12:59         TAL       $VOL.PRG.OUTS
000000   000000    COMMON   WORD         IN^FILE^HANDLER
                     2/11/85  12:59         TAL       $VOL.PRG.INS
```

Figure 22-3. Data-Block Load Map

```
BINDER - OBJECT FILE BINDER - T9621B00 - (28JAN85)  SYSTEM \XX
Object file name is $VOL.PRG.CONVO
Number of Binder errors = 0
Number of Binder warnings = 0
Primary data = 11 words
Code area size = 1 pages
Resident code size = 0 pages
Data area size = 2 pages
Number of code spaces = 1 space

The object file will run on a TNS/II, but may not run on a TNS
Number of compiler errors = 0
Number of compiler warnings = 0
Maximum symbol table space used was =       9938 bytes
Number of source lines= 221
Elapsed time - 00:01:24
```

Figure 22-4. Compilation Statistics

PROCEDURE REPLACEMENT SAMPLE PROGRAM

This example uses directives to refer to procedures in other files. You need recompile only the main procedure with the source for the replacement procedure. The existing program file is a search file. The compilation unit contains:

- SEARCH directive for file "objlo"
- SOURCE directive for a file of external declarations (for compile-time consistency checks, also provides for FORWARD requirement)
- SOURCE directive for the new procedure "proc^c"
- SOURCE directive for the main procedure "proc^m^main", which contains external references (calls) to the other procedures

TAL passes the search list and the compiled main and replacement procedures to BINSERV, which binds procedures from the search file to resolve the external references from the main procedure.

```
?SEARCH objlo                !BINDER uses this file to
                             ! resolve external references

?SOURCE externp              !"externp" contains only EXTERNALs
PROC proc^a;
EXTERNAL;

PROC proc^b;
EXTERNAL;

PROC proc^c;
EXTERNAL;

PROC proc^m^main MAIN;
EXTERNAL;
?NOLIST, SOURCE proc2cs      !Source file containing replacement
                             ! code for "proc^c"
?SOURCE mainproc
PROC proc^m^main;
BEGIN
  CALL proc^a;
  CALL proc^b;
  CALL proc^c;
END;
```

APPENDIX A

MACHINE DEPENDENCIES

To transport object programs between the NonStop 1+ and other system types, you must modify your program to accommodate certain machine differences. This appendix summarizes the machine differences. These capabilities are not available on the NonStop 1+ system unless otherwise noted.

GENERAL DIFFERENCES

General differences include extended addressing, multiple code segments, absolute addressing, additional machine instructions, system global quadword data, and additional compiler directives.

Extended Addressing

Extended addressing is a nonprivileged feature that allows byte access to any logical segment (current user data segment, user code segment, system data segment, or extended data segment). Two standard functions support this feature. $XADR converts a 16-bit address to a 32-bit address; $LADR converts a 32-bit address to a 16-bit address.

Additional Code Segments

Version B00 of the GUARDIAN operating system allows multiple user code segments, which are described in the System Description Manual for the NonStop system. During the compilation session, BINSERV automatically creates additional code segments as needed. For examples of multiple code-segment listings, see the BINDER Manual.

Absolute Addressing

This is a privileged feature that lets you access absolute extended byte addresses in any segment in virtual memory as described in the System Description Manual for the NonStop system. Two standard functions support this feature. $AXADR converts a 16-bit address to an absolute extended address; $BOUNDS performs a privileged bounds check operation.

Machine Instructions

Many privileged machine instructions, though similar in function, are modified in minor ways to accommodate the NonStop system architecture. For example, the formats of the ENV and Interrupt registers differ from the NonStop 1+ system.

Privileged code included in your source code using the CODE statement executes properly only on the expected host system. Before moving user-written interrupt handlers from one system type to another, examine the instructions and registers used. See the System Description Manual for your system for instruction code lists and definitions and for register formats.

System Global Quadword Data

On the NonStop 1+ system, system global pointers cannot access REAL(64) and FIXED items because no instructions are available for quadword load or store operations from SG data space--even with optional microcode present.

On the NonStop system, system global pointers can access data of any type. This is a privileged feature.

Compiler Directives

The CPU directive generates object code for a system type:

```
?CPU TNS        !Object code to execute on a NonStop 1+ system
?CPU TNS/II     !Object code to execute on a NonStop system
```

The IF, IFNOT, and ENDIF directives allow selective compilation based on the current compilation mode, whether set by the compiler or by a CPU directive. For example:

```
   ?IF TNS/II                 !Compiles intervening code only if the
      .                       ! TNS/II mode is in effect
      .
   ?ENDIF TNS/II
```

The following directives are not features of NonStop 1+ software:

ABORT	terminates compilation if TAL cannot open a source file.
LINES	specifies maximum number of lines for each page.
GMAP	prints the global map if MAP is also on.
PRINTSYM	includes identifier declarations.

EXTENDED POINTERS

This subsection describes the format of extended pointers, address conversions, parameters with extended addresses, indexing, and data operations with extended pointers.

Format of Extended Pointers

The 32-bit format of the extended pointer is:

Bit	Meaning
<0>	Absolute Mode (A) specifier. Nonprivileged use = 0 Privileged use = 1
<1>	Reserved; always 0
<2:14>	Segment specifier Relative extended address = 0:1027 Absolute extended address = 0:8191
<15:20>	Page specifier = 0:63
<21:30>	Word specifier = 0:1023
<31>	Byte specifier = 0:1

Figure A-1 shows the format for extended pointers.

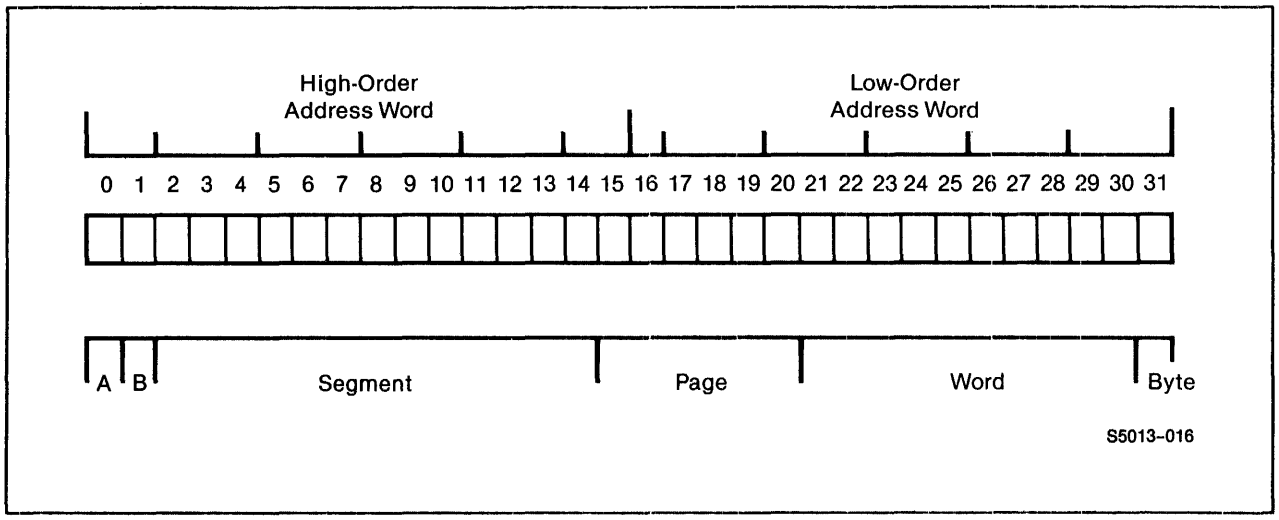


Figure A-1. Format of Extended Pointer

You can use an extended pointer to access any of the four standard (logical) segments. You must use an extended pointer to access an extended data segment. Specify the segment to access in the segment field of the pointer, as follows:

0	User data segment
1	System data segment
2	Current segment (user or system)
3	Currently mapped user code segment (read access only)
4-n	Base address for the current extended data segment

An extended pointer, having 32 bits, can access byte addresses anywhere in a segment. (The page, word, and byte fields together require 17 address bits.) All extended addresses are byte addresses. Word-aligned data items must have even byte addresses.

A standard pointer, having 16 bits, can access byte addresses in only the lower 32K of a segment. To access byte addresses in the upper 32K words of a segment, you must use an extended pointer.

Address Conversions

If a called procedure expects an extended address and the caller passes a nonextended parameter, TAL generates an implicit $XADR function and converts the standard address to extended.

If the caller passes an extended parameter to a nonextended formal parameter, TAL generates an implicit $LADR function and emits a warning. (The segment information is lost and the resultant address (to segment 0) might not be the desired location.)

When converting addresses, TAL assumes the type of the address in a pointer matches the type of the item to which it points. For example, TAL assumes a STRING pointer contains a byte address and an INT(32) pointer a word address. When converting the extended address of a word-aligned item to a nonextended address, the system ignores the byte specifier.

Parameters With Extended Addresses

A formal reference parameter can have an extended address. If you use TNS mode to compile a procedure that declares parameters with extended addresses, TAL flags those as errors.

When a caller passes a reference parameter with an extended address, TAL places a 32-bit pointer to the variable in the called procedure's parameter area. Statements in the procedure access the variable through the extended pointer.

The following example declares formal parameters with extended addresses:

```
PROC new^proc (ext^param, str^param);
  INT .EXT ext^param;        !Extended reference parameter
  STRING .EXT str^param;     !Extended reference parameter
```

You get a warning if you pass a STRING reference parameter with an extended address to a word-aligned formal parameter. You must ensure that the variable is word-aligned.

Indexing With Extended Addresses

You can assign to an extended pointer the address of an indexed element. In the following example, the pointer "name" is assigned the address of "a" minus 5 elements, assuming "a" has an extended address:

```
INT .EXT name := @a[-5];
```

Data Operations With Extended Addresses

In move or group comparison operations, data can have extended addresses. In scan operations, data cannot be extended.

These operations optionally return a <next-addr> value. For a move operation, <next-addr> points to the next word or byte in the destination following the last item moved. For scan and comparison operations, it points to the first word or byte that does not match.

After a move or compare operation, <next-addr> contains an extended byte address if any item has an extended address. If no item is extended, <next-addr> contains a byte address if the location to which the items are moved or compared has a byte address. Otherwise, <next-addr> contains a standard word address.

After a scan operation, <next-addr> contains a standard byte address.

If a standard byte address results for multibyte elements, divide by the number of bytes per element to obtain the number of elements processed, using unsigned arithmetic. (The same is true for multiword data types with word addresses).

After a comparison of multiword items such as FIXED elements, <next-addr> might point into the middle of an element since the comparison is on a word or byte basis, not on an element basis. Round the number of words or bytes up before dividing by the number of elements per word (or byte).

EXTENDED DATA SEGMENTS

You can allocate extended data segments of up to 268 megabytes in size. The extended segments share the same address space, but only one extended segment can be in use at one time.

To create and use extended segments, you must:

1. Declare an extended pointer to an extended segment base address.

2. Allocate an extended segment by invoking the ALLOCATESEGMENT system procedure.

3. Make an extended segment the current extended segment by invoking the USESEGMENT system procedure.

Extended Segment Base Address

The extended segment base address defines the first storage location of any extended segment. The first segment base address you can use is 4. To specify the base address, shift 4 (4D) left 17 places to move 1 from bit 29 to bit 12, as follows:

```
   STRING .EXT ptr := 4D '<<' 17;      !Resulting address = %2000000D
```

The initial byte address in the extended segment is 0. You can access specific locations by indexing from the base or using the base in an arithmetic expression.

Figure A-2 shows the format of the base address.

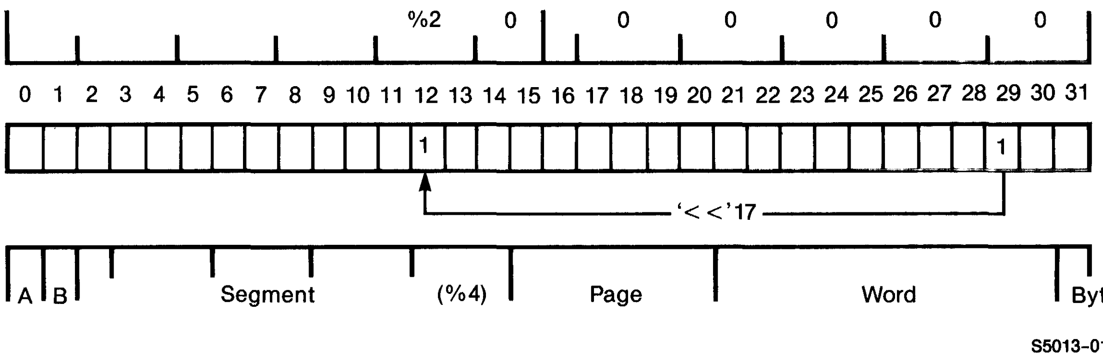


Figure A-2. Format of Extended Segment Base Address

Allocating and Using an Extended Segment

The following example allocates and uses extended segments:

```
     .
     .
   INT .EXT px := 4D '<<' 17;        !Declares and initializes
     .                               ! extended pointer to beginning
     .                               ! of extended data segment
   INT s;

   s := ALLOCATESEGMENT (0,4096D);   !Allocates extended segment 0 and
                                     ! returns status value to "s";
                                     ! requests 2 pages (4K bytes) of
                                     ! extended memory

   IF s <> 0 THEN  error;            !Indicates allocation of
                                     ! extended segment by returning
                                     ! 0; otherwise, returns error

   CALL USESEGMENT (0);              !Makes extended segment 0
                                     ! the current extended segment

   px := 5;                          !Stores a 5 in first word of
                                     ! extended segment 0

   s := ALLOCATESEGMENT (1,4096D);   !Allocates extended segment 1 and
                                     ! returns status value to "s";
                                     ! requests 2 pages (4K bytes) of
                                     ! extended memory

   IF s <> 0 THEN  error;            !Indicates allocation of
                                     ! extended segment by returning
                                     ! 0; otherwise, returns error

   CALL USESEGMENT (1);              !Makes extended segment 1
                                     ! the current extended segment

   px := 2;                          !Stores a 2 in first word of
                                     ! extended segment 1
     .
     .
```

When you no longer need an extended segment, you can delete the storage area by invoking the DEALLOCATESEGMENT system procedure.

Extended Segment Management

TAL does not allocate storage for any extended data segment. You must manage the additional data space. When accessing free space in an extended data segment, you must remember the last storage space assigned to an extended pointer. An extended data segment begins at the byte address %2000000D. All data items in an extended data segment are byte addressed.

The following example shows how to manage extended pointers:

```
                                 !Extended pointer declarations
INT .EXT x;                      !Assume a 435-word array
INT .EXT y;                      !Assume a 1000-word array
INT .EXT z;                      !Assume a 94-word array
 .
 .
BEGIN
 @ x := %2000000D;               !Assigns pointer "x" to the
                                 ! beginning of extended segment

 @ y := @ x + 870D;              !Assigns pointer "y" to the
                                 ! first free space after "x"

 @ z := @ y + 2000D;             !Assigns pointer "z" to the
  .                              ! first free space after "y"
  .
END;
```

The DEFINEPOOL, GETPOOL and PUTPOOL procedures can help you manage large blocks of memory and build proper addresses:

```
LITERAL head^size = 19D;
INT .EXT poolhead := 200000D;         !Pool header
INT .EXT pool := 200000D + head^size; !Points into upper 32K
INT .EXT block;

status := DEFINEPOOL (poolhead, pool, head^size);
@block := GETPOOL (poolhead, 1024D);
  .
  .                                   !Do processing
CALL PUTPOOL (poolhead, block);
```

The DEFINEPOOL, GETPOOL, and USESEGMENT procedures return both a condition code and a value. If you assign a returned value to a variable, the condition code setting is lost. For more information on system procedures, see the System Procedure Calls Reference Manual and the GUARDIAN Operating System Programmer's Reference Manual.

Extended Linked-List Example

The following example illustrates a linked list:

```
INT .EXT pool^head := %2000000D;

STRUCT templ (*);
  BEGIN
  INT(32) link;
  STRING name[0:30];
  STRING address[0:20];
  END;

PROC fill^new^element (d);
  INT(32) d; FORWARD;

PROC get^buffer (current^element);
  STRING .EXT current^element (templ);
BEGIN
  current^element.link :=
            GETPOOL (pool^head, $UDBL($LEN(templ));
  CALL fill^new^element (current^element.link);
       .
       .
END;

PROC fill^new^element (d);
  INT(32) d;
BEGIN
  STRING .EXT new^element := d;

  new^element.name    ':=' . . . ;
  new^element.address ':=' . . . ;
       .
       .
  new^element.link := 0D;
END;
```

Extended Addressing Example Program

The following source code is an example of a complete program that allocates and uses extended segments:

```
?INSPECT, SYMBOLS
?NOCODE
?PAGE "dummy page directive"

LITERAL dealloc^flags = 1;             !For DEALLOCATESEGMENT later

LITERAL seg^id^zero = 0;               !User extended data segment
LITERAL seg^id^two  = 2;               !IDs need not be contiguous

LITERAL seg^id^zero^len = 2048D;
LITERAL seg^id^two^len  = 4096D;

INT    .EXT word^ptr := 0D;
STRING .EXT byte^ptr := 0D;

INT .EXT pool^head  := %2000000D;      !Beginning of 19-word pool
                                       ! header in extended segment

INT .EXT pool^ptr   := %2000046D;      !First byte after pool header
INT .EXT block^ptr1 := 0D;             !Pool block general pointer
INT .EXT block^ptr2 := 0D;             !Pool block general pointer
STRING .byte^array[-1:100];            !Byte array for local scan

                                       !Extended pointer to byte arra
STRING .EXT ba^ptr := 0D;              ! needed for extended move

STRING .offset^ptr := -1;
INT offset^x := 0;

LITERAL str^len = 47;                  !Length of string to move
LITERAL array^len = 102;               !Length of byte array

INT status := 1000;                    !Beyond maximum error range

INT old^seg^num := -1;                 !Not a valid user extended
                                       ! data segment ID

?NOLIST
?SOURCE $system.system.extdecs( ABEND, DEBUG,
?          ALLOCATESEGMENT, USESEGMENT, DEALLOCATESEGMENT,
?          DEFINEPOOL, GETPOOL, PUTPOOL)
?LIST
```

```
?PAGE "Extended Addressing Example Program"

PROC ext^addr^example MAIN;
BEGIN

  status := ALLOCATESEGMENT( seg^id^zero, seg^id^zero^len );
  IF status <> 0 THEN CALL DEBUG;

  status := ALLOCATESEGMENT( seg^id^two, seg^id^two^len );
  IF status <> 0 THEN CALL DEBUG;

  old^seg^num := USESEGMENT ( seg^id^zero );
  IF <> THEN CALL DEBUG;
                                   !Set extended pointer to
  @byte^ptr := %2000000D;          ! first byte of current
                                   ! extended segment

                                   !Put character string into
                                   ! current extended segment
  byte^ptr ':=' "This is a sample string to be scanned for an X.";

                                   !Convert 16-bit address
                                   ! of byte array to 32-bit
  @ba^ptr := $XADR( byte^array[0] ); ! extended pointer "ba^ptr"

  ba^ptr ':=' byte^ptr FOR str^len;  !Extended move of string
                                   ! to user stack

  byte^array[-1]  := 0;            !Set these to 0 to stop
  byte^array[100] := 0;            ! any scans in the array

  SCAN byte^array[0] UNTIL "X" -> @offset^ptr;   !Scan on stack
  IF $CARRY THEN CALL DEBUG;       ! if scan stopped by 0,
                                   ! call DEBUG

  offset^x := @offset^ptr '-' @byte^array[0];
  ! ------------------------------------------------------------ !
  ! USE new extended data segment for more example manipulations !
  ! ------------------------------------------------------------ !

  old^seg^num := USESEGMENT ( seg^id^two );
  IF <> THEN CALL DEBUG;

  status := DEFINEPOOL ( pool^head, pool^ptr, 4000D );
  IF status <> 0 THEN CALL DEBUG;

  @block^ptr1 := GETPOOL ( pool^head , 101D );    !For contents
  IF <> THEN CALL DEBUG;                          ! of "byte^array"

                                              !Move "byte^array" to
                                              ! first pool in
                                              ! extended segment
  block^ptr1 ':=' ba^ptr[-1D] FOR array^len;
?PAGE
                                              !Get a second pool in
                                              ! current extended
                                              ! segment for contents
                                              ! of  "word^array"
  @block^ptr2 := GETPOOL ( pool^head , 1000D );
  IF <> THEN CALL DEBUG;

  @word^ptr := @block^ptr2;                   !Copy extended pointer

                                              !Move a constant list
                                              ! into this pool in
                                              ! extended segment
  word^ptr ':=' [ 8, 16, 32, 40, 48, 56, 64, 128];

  CALL PUTPOOL ( pool^head, block^ptr1 );     !Give first pool back
  IF <> THEN CALL DEBUG;

  CALL PUTPOOL ( pool^head, block^ptr2 );     !Give second pool back
  IF <> THEN CALL DEBUG;

  CALL DEALLOCATESEGMENT ( seg^id^two, dealloc^flags );

  CALL DEALLOCATESEGMENT ( seg^id^zero, dealloc^flags );

END;
```

APPENDIX B

OPTIMAL PERFORMANCE CONSIDERATIONS

Although TAL is a one-pass compiler and is subject to certain limitations inherent in this characteristic, it generates efficient object code for the target computer. However, if optimum run-time speed is important, you can maximize efficiency by following the guidelines given in this appendix.

GENERAL GUIDELINES

The following guidelines describe general practices for achieving efficient code:

- Code programs as cleanly and clearly as possible. Provide structured source code and adequate documentation in the source listing.
- Debug the programs to ensure that they work properly.
- Analyze the programs using performance analysis tools such as XRAY to determine where inefficiencies occur.
- Based on the analysis, change procedures that require modification. Provide comments that describe the changes and why you made them.

SPECIFIC GUIDELINES

The following guidelines apply to addressing, indexing, and arithmetic operations.

Addressing

Although direct addressing is limited in the amount of memory it can reference, it is more efficient than indirect addressing. Thus, you should use direct addressing whenever possible.

For example, suppose a procedure receives a reference parameter that is used heavily in calculations within that procedure before it receives a value to return to the caller. When the procedure begins execution, move the value in the indirectly addressed parameter to a local directly addressed storage area, then use that copy in the calculations. At the end of the procedure, store the result in the original parameter, which is returned. Indirect addressing is used only twice (once in parameter passing and once in returning the value). All other references use direct addressing, which enhances the object-code speed.

Indirect arrays and pointers provide equivalent operation. The advantage of indirect arrays is that TAL provides a pointer for the array, allocates the array data, and initializes the pointer to the base of the array. To use pointers, you must declare and initialize the pointer.

TAL emits shorter instruction sequences if it can place INT and STRING extended pointers in locations G[0] through G[63] or L[1] through L[63]. Thus, you should declare these pointers before other global and local declarations.

STACK and STORE statements do not improve the efficiency of access to data items. These statements are provided primarily for moving operands to and from the register stack when working with the CODE statement.

Indexing

TAL saves index values in index registers so you can refer to them in later statements. For instance, for the operation X[i] := 5, TAL saves the value of "i" in an index register. You can then use it in a reference such as Y[i].

Multiple references to the same index value (using the same data type) promotes efficiency.

For indexed items in structures, TAL optimizes references only to adjacent items within the same substructure.

An index on a 16-bit variable is always a signed INT expression. For a STRING variable, access ranges from 32K bytes below the base to 32K bytes above the base. For a non-STRING variable, access ranges from 32K words below the base to 32K words above the base.

Indexing indirect references is no less efficient than not indexing indirect references, because the hardware requires no extra time to add indexes to address values.

For an INT or STRING extended pointer located below G[63] or L[63] (decimal), a 16-bit index is more efficient than a 32-bit index. A 16-bit index results in a shorter instruction sequence using the LWXX, SWXX, LBXX, and SBXX instructions. (These instructions are described in the System Management Manual for NonStop systems.)

For all other extended pointers, a 32-bit index is more efficient. However, for extended structure pointers, 32-bit indexes are not allowed.

Using a USE register for the 16-bit index of an extended pointer does not provide further efficiency. TAL must still load the index value from the USE register into register "A" for use with the LWXX, SWXX, LBXX, and SBXX instructions. For the less efficient extended access, TAL loads the 16-bit index from the USE register into register "A", then converts it to a 32-bit index.

Arithmetic Expressions

A complex arithmetic expression might cause more memory references than if you make the complex expression into several smaller expressions.

The excessive memory references are triggered by register stack overflow, which is especially likely if indexes are involved. Use of an index might cause part of the computation to be pushed on the stack and later popped off. Doubleword or quadword operands fill the register stack quickly.

For quadword operations, do not nest index calculations in larger arithmetic expressions because register stack overflow is likely to result. Use a separate statement for the index calculations, saving the results in a temporary area. The expression can then reference this area.

The IF-THEN-ELSE and CASE forms of arithmetic expressions do not generate efficient machine code, especially when used to test complex conditions. To evaluate a complex condition, include separate IF-THEN-ELSE or CASE statements that perform proper assignments in all possible branches of the condition.

APPENDIX C

ERROR MESSAGES

This appendix describes:

- TAL Error Messages
- TAL Warning Messages

TAL MESSAGES

TAL scans the source code line by line and notifies you of an error or potential error by displaying one of two types of messages:

- Error message--Indicates an error that must be corrected before TAL can successfully compile the source code
- Warning message--Alerts you to a potential error condition and indicates that you should check an area of the source code

In the source listing, TAL prints a circumflex symbol (^) to indicate the location of the error or potential error. The circumflex 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, TAL displays a message describing the nature of the error. The form of the message is:

```
**** {  ERROR  } **** <message-number> -- <message-text>
     { WARNING }
```

Occasionally, TAL adds a third line for supplemental information, such as "IN PROC <proc-name>" when reference to an earlier procedure is necessary or "PREVIOUS ON PAGE #<page-num>", which refers you to a previous page with an error.

Error messages are described on the following pages in ascending numeric order. Although TAL prints each message on a single line, some messages here are continued on a second line 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.

Compiler Error Messages

The following diagnostic messages identify source errors that prevent correct compilation. No object file is produced for the compilation.

**** ERROR **** 0 -- Compiler error

This error appears only when TAL detects a logic error within its operation. The number following the message is for use by Tandem development personnel. Please report this occurrence to Tandem Computers Incorporated and include a copy of the complete compilation listing (and source, if possible).

**** ERROR **** 1 -- Parameter mismatch

The parameter type of a parameter passed to a procedure does not agree with the parameter type expected by that procedure.

**** ERROR **** 2 -- Identifier declared more than once

A declaration contains an identifier that is already declared within this scope.

**** ERROR **** 3 -- Recursive DEFINE invocation

A DEFINE declaration calls itself or is defined in terms of itself. An example is "DEFINE a = b#, b = a#;". When "a" is expanded, it in turn expands "b", which in turn expands "a",.... TAL checks for this situation and issues the message when it expands the DEFINE.

**** ERROR **** 4 -- Illegal MOVE statement

TAL detects a malformed move for which it cannot generate code.

**** ERROR **** 5 -- INT overflow

A numeric constant represents a value that is too large for its data type, or an overflow occurs while TAL scales a quadword constant up or down.

**** ERROR **** 6 -- Illegal digit

A numeric constant contains a digit that is illegal in the stated base of the constant. For example, an octal constant contains the digit "9".

**** ERROR **** 7 -- String overflow

A character string contains more than 128 characters or does not terminate in the line in which it begins.

**** ERROR **** 8 -- Not defined for INT(32), FIXED, or REAL

An arithmetic operation occurs that is not permissible for the declared data types.

**** ERROR **** 9 -- The compiler does not allocate space for .EXT or .SG STRUCTs

You cannot declare a structure using the .EXT or .SG addressing symbol. You can declare a structure using the standard addressing symbol (.), then declare an extended structure pointer (Section 11) or a system global pointer (Section 18) that refers to the structure.

**** ERROR **** 10 -- Address range violation

This message indicates one of the following conditions:

1. A declaration specifies addresses beyond the allowable range (for example: INT i = 'G' + 300). Only 256 words are directly addressable relative to 'G'.

2. A PROC produces more than 32K of code and causes a code-segment overflow.

3. The total of primary and secondary globals exceeds 32K words.

**** ERROR **** 11 -- Illegal reference

A variable appears in a context where a constant is expected, or an expression appears where a variable is expected.

**** ERROR **** 12 -- Nested routine declaration(s)

One or more PROC declarations are present within the body of another procedure. Procedures can contain SUBPROCs only; no other nesting is permitted.

**** ERROR **** 13 -- Only 16-bit INT value(s) allowed

You specified a STRING, FIXED, REAL, or other data type where only INT values are permitted.

**** ERROR **** 14 -- Only initialization with constant value(s) is allowed

A global data initialization includes variables. Global initializations can include constants only. You can only use variables to initialize identifiers you declare within a PROC or SUBPROC.

**** ERROR **** 15 -- Initialization is illegal with reference specification

You cannot use the same statement to declare an identifier as a reference to another item and to initialize the other item (for example: INT .a = b := <value>). Use separate declarations.

**** ERROR **** 16 -- Item already has an extended address

A parameter to the standard function $XADR is an item that already has an extended address.

**** ERROR **** 17 -- Formal parameter type specification is missing

A declaration for a formal parameter is missing in the PROC or SUBPROC header, and TAL detects a BEGIN.

**** ERROR **** 18 -- Illegal array bounds specification

The upper and lower bounds in an array declaration must be constants or constant expressions. Also, the lower bound must be less than or equal to the upper bound (except when the array is declared within a STRUCT).

This message might also appear if an equivalenced variable is also declared as an array (for example, INT a[0:5] = b). In this case, TAL ignores the bounds specification.

**** ERROR **** 19 -- Global or nested SUBPROC declaration

A SUBPROC declaration appears either outside a procedure or within another subprocedure. You cannot declare global subprocedures or nest them inside another subprocedure.

**** ERROR **** 20 -- Illegal bit field designator

The ending position of a bit field designator must be greater than or equal to the starting position, and both must be INT constants.

**** ERROR **** 21 -- Label declared more than once

This message means an identifier followed by a colon is identical to another label name used in the same procedure. Each label must be unique within a procedure.

**** ERROR **** 22 -- Only standard indirect variables are allowed

You must refer to variables in extended segments by using the extended indirection symbol (.EXT).

**** ERROR **** 23 -- Variable size error

The size field of a data type is invalid, for example, "INT(12)".

**** ERROR **** 24 -- Data declaration(s) must precede PROC declaration(s)

A global data declaration appears after a procedure declaration.

**** ERROR **** 25 -- Item does not have an extended address

The argument to the standard function $LADR does not have an extended address.

**** ERROR **** 26 -- Routine declared forward more than once

More than one forward declaration for the given procedure or subprocedure is present. You can declare a given procedure FORWARD only once.

**** ERROR **** 27 -- Illegal syntax

A statement contains one or more syntax errors. This message can also appear as a result of an error in the previous line.

**** ERROR **** 28 -- Illegal use of code relative variable

You cannot use a read-only array in the present context.

**** ERROR **** 29 -- Illegal use of identifier

The named identifier appears in a PROC or SUBPROC declaration as a formal parameter specification but is not included in the formal parameter list.

**** ERROR **** 30 -- Only label or USE variable allowed

A DROP statement refers to a variable that is not a label or a USE statement variable.

**** ERROR **** 31 -- Only PROC or SUBPROC identifier allowed

A CALL statement can only refer to a PROC, SUBPROC, or ENTRY identifier.

**** ERROR **** 32 -- Type incompatibility

This message indicates one of the following conditions:

1. An expression with identifiers of different types occurs. Use type-transfer standard functions.

2. A procedure without a return type occurs on the right side of an assignment statement.

3. In a comparison operation, the destination and source variables have standard addressing, but both are not either byte or word addressed.

**** ERROR **** 33 -- Illegal global declaration(s)

A declaration occurs for an identifier (such as a label) that cannot exist as a global item.

**** ERROR **** 34 -- Missing variable

A required variable is missing from the current statement.

**** ERROR **** 36 -- Illegal range

A specified value exceeds the allowable range for a given operation.

**** ERROR **** 37 -- Missing identifier

A required identifer is missing from the current statement.

**** ERROR **** 38 -- Illegal index-register specification

You reserved more than three registers for use as index registers. Use a DROP statement to reduce the number of reserved registers.

**** ERROR **** 39 -- ?ABORT active and open failed on <file-name>

The ABORT directive causes TAL to terminate when it cannot open the file you specified in a SOURCE directive.

**** ERROR **** 40 -- Only allowed with a variable

You specified an operation or expression that is valid only when used with a variable (for example, "(a+b).<2:5> := 0;").

**** ERROR **** 42 -- Table overflow

Your source program fills one of the fixed-size tables of TAL. No recovery from this condition is possible. You must modify the source program. The one-digit number identifies the affected table:

0 = Constant Table--Before the overflow occurs, you can place a DUMPCONS in the code 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 Table--Simplify the expression.

2 = Pseudo-Label Table--You might have too many nested IF statements. Simplify the IF statements.

3 = Parametric DEFINE Table--The DEFINE being expanded has parameters that are too long. Shorten the parameters.

4 = Section Table (for SOURCE directives)--You are accessing too many sections at one time. Break the sections into two or more groups.

**** ERROR **** 43 -- Illegal symbol

The current source line contains an invalid character or a character that is invalid in the current context.

**** ERROR **** 44 -- Illegal instruction

The specified mnemonic does not match those for the NonStop 1+ or NonStop system as specified in the CPU directive. Use the CPU directive to define the instruction set TAL is to use.

**** ERROR **** 45 -- Only INT(32) value(s) allowed

You used a non-INT(32) value in a context where only an INT(32) value is legal.

**** ERROR **** 46 -- Illegal indirection specification

The period symbol (.) is used on a variable that is already indirect. Only one level of indirection is legal.

**** ERROR **** 47 -- Illegal for 16-bit INT

The unsigned divide ('/') and unsigned modulo divide ('\') operations require an INT(32) dividend and an INT divisor. You specified an INT value for the INT(32) value.

**** ERROR **** 48 -- Missing <item-specification>

The source code is missing the item specified.

**** ERROR **** 49 -- Undeclared identifier

You made a reference to a data item that is not declared.

**** ERROR **** 50 -- Cannot drop this Label

You specified a DROP statement for a label you did not declare or use. You can drop a label only after you declare it and TAL reads all references to it. You drop labels to save symbol table space or to allow its reuse (as in a DEFINE).

**** ERROR **** 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.

**** ERROR **** 52 -- Missing initialization for code relative array

Initialization is missing from a read-only array declaration. You must initialize read-only arrays when you declare them.

**** ERROR **** 53 -- Edit file has invalid format or sequence <n>

TAL detects an unrecoverable error in the source file; <n> is a negative number that identifies the type of error:

-3 = Text file format error

-4 = Sequence error (the line number of the current source line is less than that of the preceding line)

**** ERROR **** 54 -- Illegal reference parameter

You declared a STRUCT as a formal parameter without specifying indirection. You must declare STRUCT formal parameters as reference parameters.

**** ERROR **** 55 -- Illegal SUBPROC attribute

A SUBPROC declaration contains an EXTERNAL attribute specification. This is not a valid attribute for a subprocedure.

**** ERROR **** 56 -- Illegal use of USE variable

You cannot perform the specified operation on a register. For instance, a USE variable cannot be the target of a move statement.

**** ERROR **** 57 -- Symbol table overflow

The usual cause for this message is lack of space on the disc. TAL issues additional messages for the specific case. An example is:

```
ALLOCATESEGMENT ERROR 43
```

**** ERROR **** 58 -- Illegal branch

If a FOR statement has a USE register as its counter, branching into the FOR loop is not permitted.

**** ERROR **** 59 -- Division by zero

TAL detects an attempt to divide by 0.

**** ERROR **** 60 -- Only a data variable may be indexed

An index is appended to an identifier that does not represent a data variable (such as a label or entry point).

**** ERROR **** 61 -- Actual/formal parameter count mismatch

A call to a procedure or subprocedure supplies more (or fewer) parameters than you defined in the PROC or SUBPROC declaration.

**** ERROR **** 62 -- Forward/external parameter count mismatch

A discrepancy exists between the number of parameters specified in a FORWARD or EXTERNAL declaration and the number you specified in the procedure body declaration.

**** ERROR **** 63 -- Illegal drop of USE variable in context of FOR loop

You specified a USE variable as the index of a FOR loop and then dropped the variable within the scope of that FOR loop. The FOR loop can function correctly only if the register remains reserved. Remove the DROP statement from within the FOR loop.

**** ERROR **** 64 -- Scale point must be a constant

The <fpoint> declaration for a FIXED variable and the <scale> parameter of the $SCALE function must be INT constants in the range -19 through +19. The current source line contains a scale point that is not a constant.

**** ERROR **** 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.

**** ERROR **** 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.

**** ERROR **** 67 -- Source commands nested too deeply

Source coding invoked by the SOURCE directive might contain a SOURCE directive to call in other coding, which, in turn, calls still other coding. The maximum limit for such nesting is four levels; that limit is exceeded.

**** ERROR **** 68 -- This identifier cannot be indexed

A directly addressable identifier was indexed and used in a memory-referencing instruction in a CODE statement.

**** ERROR **** 69 -- Invalid template access

A template structure has meaning only when referred to in subsequent structure declarations; the compiler allocates no storage space for it. The current source line attempts to access a template structure as if it is a normal data item.

**** ERROR **** 70 -- Only items subordinate to a structure may be qualified

A qualified reference of the form <name>.<subname>.<itemname> applies only to data items within a structure. You entered a qualified reference to a data item that is not part of a structure.

**** ERROR **** 71 -- Only INT or STRING STRUCT pointers are allowed

You declared a structure pointer of a data type other than INT or STRING; these are the only acceptable types.

**** ERROR **** 72 -- Indirection must be supplied

In the structure pointer declaration, the indirection symbol (.) must precede the identifier that represents the pointer; the indirection symbol is missing.

**** ERROR **** 73 -- Only structure identifiers may be used as a referral

In a referral structure declaration, the <referral> identifier must be the name of a previously declared definition structure, template structure, or structure pointer.

**** ERROR **** 74 -- Word addressable items may not be accessed through a STRING structure pointer

Although an INT structure pointer can access items of any data type, a STRING structure pointer can only access STRING data items. This restriction does not apply for extended structure pointers on a NonStop system.

**** ERROR **** 76 -- Illegal STRUCT or SUBSTRUCT reference

A structure or substructure reference can appear only in a move, scan, or group comparison operation, or as an actual parameter passed by reference, or as @<primary> in an expression. The current source line violates this restriction.

**** ERROR **** 78 -- Invalid number form

A floating-point constant is entered incorrectly. A REAL constant must be written in the following form:

```
[<sign>]<integer>.<fraction> E [<sign>]<exponent>
```

A REAL(64) constant must be entered in the following form:

```
[<sign>]<integer>.<fraction> L [<sign>]<exponent>
```

**** ERROR **** 79 -- REAL underflow or overflow

Underflow or overflow occurred during input conversion of a REAL or REAL(64) number. Floating-point numbers must be in the following approximate range:

$\pm 8.62 * 10^{-78}$ through $\pm 1.16 * 10^{77}$

**** ERROR **** 81 -- Invoked forward PROC converted to external

The current declaration attempts to redefine as external a PROC that was already called as an internal procedure.

**** ERROR **** 82 -- Not defined for this cpu type - ignored

FIXED or REAL operations involving SG pointers and declarations using extended addressing are not defined for the NonStop system. You must include a CPU TNS/II directive before these operations are accepted.

**** ERROR **** 83 -- CPU type must be set initially

If it is present, the CPU directive must precede any data or procedure declarations.

**** ERROR **** 84 -- There is no SCAN instruction for extended memory

Extended items cannot be the object of a SCAN or RSCAN operation since there is no hardware support for them. Move the array into a temporary location in the normal user data space and perform the operations there.

**** ERROR **** 85 -- Bounds illegal on .SG or .EXT items

Data declarations with .SG or .EXT identifiers define pointers but not data storage. Specifying bounds in these declarations is NOT permitted.

**** ERROR **** 86 -- Constant expected and not found

The compiler expected a constant but found a variable reference.

**** ERROR **** 87 -- Illegal constant format

You specified a constant that does not have a legal form.

**** ERROR **** 88 -- Expression too complex. Please simplify

The current expression is too complex. The compiler's stack overflowed and the compilation terminated.

**** ERROR **** 90 -- Invalid object file name

The name supplied for the target file is not a disc file name.

**** ERROR **** 91 -- Invalid default volume or subvolume

The default volume or subvolume in the startup message was incorrect.

**** ERROR **** 92 -- Branch to entry point not allowed

An entry point cannot be the target of a GOTO statement. In the source code, add a label following the entry point. Use the label as the target of the GOTO statement.

**** ERROR **** 93 -- Previous data block not ended

A BLOCK or PROC declaration appears before an END BLOCK statement for a previous BLOCK declaration. This message occurs only if the compilation begins with a NAME declaration.

**** ERROR **** 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.

**** ERROR **** 95 -- Error reading instruction file

TAL could not open or read the TALINSTR file. This file must be on the same volume and subvolume as the TAL compiler.

**** ERROR **** 96 -- Address references between global data blocks not allowed

In a compilation unit that begins with the NAME declaration, a variable in a global data block cannot be initialized with the address of a variable in another global data block. Because global data blocks are relocatable, such an initialization is invalid.

**** ERROR **** 97 -- Equivalences between global data blocks not allowed

An equivalence declaration in a global data block uses a variable declared in another global data block. Place the equivalence and variable declarations in the same block. This message occurs only if the compilation unit begins with the NAME declaration.

**** ERROR **** 99 -- Initialization list exceeds space allocated

A constant list contains values that exceed the space allocated by the data declaration.

**** ERROR **** 100 -- Nested parametric DEFINE definition encountered during expansion

An invalid nesting was attempted in a DEFINE declaration.

**** ERROR **** 101 -- Illegal conversion to EXTENSIBLE

To be convertible, a VARIABLE procedure must have 15 or fewer parameters, 16 or fewer words of parameters, and all one-word parameters except the last one. You must also specify the number of parameters the procedure had when it was VARIABLE.

**** ERROR **** 103 -- Indirection mode specified not allowed for P-relative variable

A read-only array must be directly addressed.

**** ERROR **** 104 -- This procedure has missing label - <label-name>

A procedure references a label that does not exist within the procedure.

**** ERROR **** 105 -- A declared secondary entry point is missing - <entry-point-name>

The specified entry-point name was declared but not used in the procedure.

**** ERROR **** 106 -- A referenced subprocedure declared FORWARD is missing - <subproc-name>

You declared a FORWARD subprocedure and referenced it but did not declare the subprocedure body.

**** ERROR **** 107 -- This compiler must be run on a Tandem NonStop II or TXP processor.

Version B00 of the compiler must be run on a NonStop system.
Version E08 of the compiler must be run on a NonStop 1+ system.

Compiler Warning Messages

The following messages indicate conditions that might affect program compilation or execution. Recheck the code carefully to determine whether a correction is necessary.

**** WARNING **** 0 -- All index registers are reserved

Three index registers are reserved by USE statements. An attempt to reserve another index register will result in an error message.

**** WARNING **** 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. TAL ignores all excess characters.

**** WARNING **** 2 -- Illegal option syntax

A compiler directive option is entered incorrectly. TAL ignores the option. (This might or might not affect the program itself, depending on the function of the option.)

**** WARNING **** 3 -- Initialization list exceeds space allocated

An initialization list contains more values or characters than can be contained by the variable being initialized. TAL ignores the excess items.

**** WARNING **** 4 -- P-relative array passed as reference parameter

You passed the address of a read-only array to a procedure. This might result in incorrect execution unless the procedure takes explicit action to use the address properly.

**** WARNING **** 5 -- PEP size estimate was too small

Your PEP estimate (from the PEP directive) is not large enough to contain all the entries required. BINSERV has allocated appropriate additional space.

**** WARNING **** 6 -- Invalid ABSLIST addresses may have been generated

ABSLIST addresses might be invalid if you use the ABSLIST directive and have any of the following conditions:

1. Insufficient space for all PEP entries (see warning 5)

2. All procedures not declared FORWARD (and no PEP directive)

3. One or more RESIDENT procedures

Since TAL is a single-pass compiler and cannot adjust addresses for the above conditions, it produces a partially unusable listing. If the program has more than 32K words of code space or if you use the standalone BINDER, do not use ABSLIST.

**** WARNING **** 7 -- Multiply defined SECTION name

The same section name appears more than once in the same SOURCE directive. TAL ignores all occurrences but the first.

**** WARNING **** 8 -- SECTION name not found

A section name listed on a SOURCE directive is not in the specified file.

**** WARNING **** 9 -- RP register mismatch

An operation contains conflicting instructions for the register pointer that cannot be resolved at compilation; for example:

```
IF a
THEN
  STACK 1
ELSE
  STACK 1D;
```

This message can also occur following a large number of errors that result in an RP conflict.

**** WARNING **** 10 -- RP register overflow or underflow

A calculation produced an index register number that is greater than 7 or less than 0.

**** WARNING **** 11 -- Parameter type conflict possible

You are passing a byte-aligned (STRING) extended item as an actual parameter to a procedure that expects a word-aligned (INT, INT(32)...) item's address. If the item's address is not on a word boundary, the hardware ignores the odd-byte number and accesses the entire word.

**** WARNING **** 12 -- Undefined option

You entered a compiler directive option that does not exist. TAL ignores the erroneous directive.

**** WARNING **** 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).

**** WARNING **** 14 -- Index was truncated

This warning occurs when you try to either make a STRING or INT item equivalent to an odd-byte address or make a direct variable equivalent to an indirect variable with an index. TAL truncates the index (for example, INT .s[0:4]; INT s1 = s[1], resulting in INT s1 = s).

**** WARNING **** 15 -- Right shift emitted

A byte address is passed as a parameter when a word address is expected. TAL converted the byte address to a word address. If the STRING item begins on an odd-byte boundary, the word-aligned item also includes the even-byte part of the word.

**** WARNING **** 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.

**** WARNING **** 17 -- Initialization value too complex

An initialization expression is too complicated to evaluate in the current context.

**** WARNING **** 18 -- S register mismatch

A statement contains conflicting instructions for the setting of the S-register. For example, if a subprocedure contains the statement IF A THEN CODE(ADDS 1) ELSE CODE(ADDS 2), the setting of the S-register depends on the evaluation of A, which cannot be resolved at compilation time.

**** WARNING **** 19 -- PROC not declared FORWARD with ABSLIST option on

A PEP directive or a FORWARD declaration is missing. When you use the ABSLIST directive, TAL 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. This warning also results in a WARNING 6 at the end of the compilation.

```
**** WARNING **** 20 -- Source line truncated
```

A source line extends beyond 132 characters. TAL ignores the excess characters.

```
**** WARNING **** 21 -- Attribute mismatch
```

The attributes in a FORWARD declaration do not match those in the procedure body declaration. Change the incorrect set of attributes.

```
**** WARNING **** 22 -- Illegal command list format
```

The format of the list of parameters supplied to a compiler directive is incorrect. TAL ignores the directive.

```
**** WARNING **** 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. TAL obtains the count of items from the length of the constant list. Remove the FOR <count> clause from the group comparison expression.

```
**** WARNING **** 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.

```
**** WARNING **** 25 -- FIXED point scaling mismatch
```

The scale factor of a FIXED value passed as a parameter does not match that of the formal parameter.

```
**** WARNING **** 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. TAL sets the fixed-point value to the maximum limit, either -19 or +19.

```
**** WARNING **** 28 -- More than one MAIN specified. MAIN is still
                        <name>
```

Although more than one procedure can have the MAIN attribute in the source code, only the first MAIN procedure TAL sees retains the MAIN attribute in the object code.

If the program contains any COBOL program units, the main procedure must be written in COBOL. Refer to the COBOL Reference Manual.

**** WARNING **** 29 -- One or more illegal attributes

The only attribute permitted for a subprocedure is VARIABLE. A SUBPROC declaration with other attributes occurs. TAL ignores all attributes but VARIABLE.

**** WARNING **** 31 -- Missing FOR part

The FOR <count> specification is missing from a move statement. TAL assumes the number of items to move is 1.

**** WARNING **** 32 -- RETURN not encountered in typed PROC or SUBPROC

Although a procedure or subprocedure automatically returns control to the calling routine when the last END statement is reached, a typed procedure or subprocedure (function) is expected to return a value. To do so, it must contain at least one RETURN statement with an identifier.

**** WARNING **** 33 -- Redefinition size conflict

When redefining a substructure or structure data item, the redefined item must be of sufficient size to contain the new item.

**** WARNING **** 34 -- Redefinition offset conflict

In the redefinition of a structure data item or substructure, the original item is a STRING item beginning at an odd-byte address, but the redefined item requires word-boundary alignment.

**** WARNING **** 35 -- Segment number information lost

A procedure call passes an actual parameter with an extended address to a procedure that does not expect one. When TAL converts the address, the segment number is lost. If the extended address points into a segment other than the current user data segment, the address that results is invalid.

**** WARNING **** 36 -- Expression passed as reference parameter

A procedure call passes an expression of the form "@<variable>" to a procedure or subprocedure that expects a parameter that is an extended pointer. If the intent is to pass the address of the pointer rather than what it points to, no error is involved.

**** WARNING **** 37 -- Array access changed from indirect to direct

TAL changed an indirect array declared inside a subprocedure to a direct array. All sublocal data must be directly addressed because the sublocal area has no secondary storage for indirect data.

```
**** WARNING **** 40 -- A procedure declared FORWARD is missing -
                        <proc-name>
```

A FORWARD declaration occurs, but the procedure body declaration is missing. TAL converts all references to this procedure into EXTERNAL references to the same name.

```
**** WARNING **** 42 -- Specified bit extract/deposit may be invalid
                        for strings
```

Bit extraction or deposit operations on STRING items use bit numbers 8 through 15 only. You specified bit numbers in the range 0 through 7, which have no effect on the operation.

```
**** WARNING **** 43 -- A default OCCURS count of 1 is returned
```

An $OCCURS function used on a non-STRUCT item returns the default value of 1.

```
**** WARNING **** 44 -- A subprocedure declared FORWARD is missing -
                        <subproc-name>
```

The named SUBPROC is declared FORWARD, but is not referenced and its body is not declared.

```
**** WARNING **** 45 -- Variable attribute ignored - no parameters
```

The VARIABLE attribute appears for a procedure or subprocedure that has no parameters. TAL ignores the VARIABLE attribute.

```
**** WARNING **** 46 -- Non-relocatable global reference
```

The RELOCATE directive is in effect, and all primary global data is relocatable. However, a declaration that refers to a G-relative location appears. This reference might not be valid 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.

```
**** WARNING **** 47 -- Invalid file or subvolume specification
```

An invalid file or subvolume appears in a TAL directive. Respecify the directive.

```
**** WARNING **** 48 -- This directive not allowed in this part
                        of program
```

A directive occurs in an inappropriate place. For instance, an IF directive is not effective on the command line.

```
**** WARNING **** 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 prior to TAL E01 contains the expression @ep-1 to calculate the entry-point location, change it to @ep for correct execution.

**** WARNING **** 50 -- Literal initialized with address reference

Using the address of a global variable as the value of a LITERAL is invalid since global data is now relocatable.

**** WARNING **** 51 -- Instruction will be deleted in the near future

This message gives advance notice that the instruction indicated is to be removed from the TAL instruction set in the near future.

**** WARNING **** 52 -- . or @ in move or array comparison may be invalid

When you specify the source or destination variable in a move statement or group comparison expression, you can use the period symbol (.) only with INT direct variables located in the current user data segment. Do not use the @ symbol in a move or group comparison operation.

**** WARNING **** 53 -- This statement has caused an optional instruction to be generated

A statement occurs that requires optional microcode such as the fixed-point and floating-point optional microcode. For operations that require optional microcode, see Section 4. For standard functions that require optional microcode, see Section 17.

**** WARNING **** 54 -- The structure item rather than the define will be referenced

A DEFINE and a structure data item have the same identifier. When a reference to the qualified identifier occurs, TAL looks for the structure item first. If the structure item does not exist, TAL expands the DEFINE. To ensure proper references, use unique identifiers for all declarations.

**** WARNING **** 55 -- The length of this structure exceeds 32767 bytes at item ** <item-name>

A structure occurrence must not exceed 32767 bytes in length. The message flags the item that caused the structure to exceed the legal length; the next item is the one TAL cannot access. Reduce the length of the structure.

```
**** WARNING **** 56 -- Format of ENV register on data stack has
                        changed as of GUARDIAN Release B00
```

You have made a variable equivalent to L '-' 1, which the operating system now uses for saving the current code segment number when a procedure is invoked. Your program might be in error. For more information on the ENV register, see the System Description Manual for the NonStop system.

```
**** WARNING **** 58 -- Code space exceeds 64k, ABSLIST has been
                        disabled
```

TAL version B00 and later supports up to 16 * 64K words of source code. When the code exceeds 64K words, TAL disables ABSLIST for the remainder of the listing.

OTHER ERROR MESSAGES

The following message might appear during compilation:

```
*** INSPCI ERROR AT:  P = %<nnnnn>, <nnnn>,,<nnnnn>
```

This error appears only when TAL detects a logic error within its operation. The number following the message is for use by Tandem development personnel. Please report this occurrence to Tandem Computers Incorporated and include a copy of the complete compilation listing (and source, if possible).

BINSERV MESSAGES

For BINSERV diagnostic messages, see the BINDER Manual.

APPENDIX D

SYNTAX SUMMARY

This appendix provides a syntax summary for specifying:

- Constants
- Access Forms
- Bit Operations
- Declarations
- Expressions
- Statements
- Standard Functions
- Compiler Directives

CONSTANTS

Character String Constants (All Data Types)

```
   "<string>"
```

STRING Numeric Constants

```
   [ <base> ] <integer>
```

INT Numeric Constants

```
   [ + ] [ <base> ] <integer>
   [ - ]
```

INT(32) Numeric Constants

```
   [ + ] [ <base> ] <integer> { D  }
   [ - ]                      { %D }
```

FIXED Numeric Constants

```
   [ + ] [ <base> ] <integer> [ .<fraction> ] { F  }
   [ - ]                                      { %F }
```

REAL and REAL(64) Numeric Constants

```
   [ + ] <integer>.<fraction> { E } [ + ] <exponent>
   [ - ]                      { L } [ - ]
```

Constant List

```
   [ <repetition-factor> * ] "[" <constant-list> "]"
```

ACCESS FORMS

Address of Nonpointer Item

```
@<item-name>
```

Contents of Pointer

```
@<pointer-name>
```

Indexing

```
<identifier> "[" <index> "]"
```

Temporary Pointer

```
.<direct-int-variable>
```

BIT OPERATIONS

Bit Deposit

```
<variable> . "<" <left-bit> [ : <right-bit> ] ">" := <expression> ;
```

Bit Extraction

```
<primary> . "<" <left-bit> [ : <right-bit> ] ">"
```

Bit Shift

```
<primary> <shift-operator> <positions>
```

DECLARATIONS

Array Declaration

```
<type> [ . ] <identifier> "[" <lower-bound> : <upper-bound> "]"
                                              [ := <initialization> ]
   [ , [ . ] <identifier> "[" <lower-bound> : <upper-bound> "]"
                                      [ := <initialization> ] ] ... ;
```

Array Declaration, Read-Only

```
<type <identifier> [ "[" <lower-bound> : <upper-bound> "]" ]
                                            = 'P' := <initialization>
  [ , <identifier> [ "[" <lower-bound> : <upper-bound> "]" ]
                                    = 'P' := <initialization> ] ... ;
```

Block Declaration

```
BLOCK { <identifier> }
      { PRIVATE      } ;

   [ <data-declaration> ; ] ...

END BLOCK;
```

DEFINE Declaration

```
DEFINE <identifier> [ ( <param> [ , <param> ] ... ) ] = <text> #
 [, <identifier> [ ( <param> [, <param> ] ...) ] = <text> # ] ... ;
```

Entry-Point Declaration

```
ENTRY <entry-point-name> [ , <entry-point-name> ] ... ;
```

Equivalenced Variable Declaration
(Simple Variable, Pointer, or Structure Pointer to
Previous Variable, Base Address, or 'SG')

```
          { { .    } { <structure-pointer> ( <referral> ) } }
          { { .EXT } { <pointer>                          } }
   <type> {                                                 }
          { <simple variable>                               }

               { <previous-identifier> } [ "[" <index> "]" ]
             = { 'G' | 'L' | 'S'       } [                 ]
               { 'SG'                  } [ {+|-} <offset>  ]

          { { .    } { <structure-pointer> ( <referral> ) } }
          { { .EXT } { <pointer>                          } }
      [ , {                                                 }
          { <simple-variable>                               }

               { <previous-identifier> } [ "[" <index> "]" ]
             = { 'G' | 'L' | 'S'       } [                 ] ] ... ;
               { 'SG'                  } [ {+|-} <offset>  ]
```

Equivalenced Variable Declaration
(Structure to Previous Variable, Base Address, or 'SG')

```
   STRUCT [ . ] { <structure> [ ( <referral> ) ]

                  { <previous-identifier> } [ "[" <index> "]" ]
                = { 'G' | 'L' | 'S'       } [                 ] ;
                  { 'SG'                  } [ {+|-} <offset>  ]

   [ <structure-body> ]
```

Label Declaration

```
   LABEL <identifier> [ , <identifier> ] ... ;
```

LITERAL Declaration

```
   LITERAL <identifier> = <constant>

       [ , <identifier> = <constant> ] ... ;
```

Name Declaration (Compilation Unit Name)

```
   NAME <identifier> ;
```

Pointer Declaration

```
<type> { .    } <identifier> [ := <initialization> ]
       { .EXT }

   [ , { .    } <identifier> [ := <initialization> ] ] ... ;
       { .EXT }
```

Procedure or Subprocedure Declaration

```
[ <type> ] { PROC    } <identifier>
           { SUBPROC }

     [ ( <formal-param-name> [ , <formal-param-name> ] ... ) ]

     [ <attribute> [ , <attribute> ] ... ] ;

[ <formal-param-specification>
                     [ , <formal-param-specification> ] ... ; ]

{ <body>    ; }                    !Procedure or subprocedure
{ FORWARD   ; }                    !Procedure or subprocedure
{ EXTERNAL  ; }                    !Procedure only
```

Procedure Formal Parameter Specification

```
<param-type> [ .    ] <formal-param-name> [ ( <referral> ) ]
             [ .EXT ]

      [ , [ .    ] <formal-param-name>  [ ( <referral> ) ] ] ... ;
          [ .EXT ]
```

Procedure Body

```
BEGIN

  [ <local-declaration> ] ...        !Direct or indirect data

  [ <subprocedure-declaration> ] ...

  [ <statement> ] ...

END;
```

Subprocedure Body

```
   BEGIN

     [ <sublocal-declaration> ] ...     !Direct data only

     [ <statement> ] ...

   END;
```

Simple Variable Declaration

```
   <type> <identifier> [ := <initialization> ]

      [ , <identifier> [ := <initialization> ] ] ... ;
```

Structure Declaration, Definition Form

```
   STRUCT [ . ] <identifier>

                                   [ "[" <lower-bound> : <upper-bound> "]" ] ;

   <structure-body>
```

Structure Declaration, Referral Form

```
   STRUCT [ . ] <identifier> ( <referral> )

                                    [ "[" <lower-bound> : <upper-bound "]" ] ;
```

Structure Declaration, Template Form

```
   STRUCT <identifier> (*) ;

   <structure-body>
```

Structure Body FILLER

```
   FILLER <constant-expression> ;
```

Structure Data Item Redefinition

```
   <type> <identifier> [ "[" <lower-bound> : <upper-bound> "]" ]

                                                 = <previous-identifier> ;
```

Structure Pointer Declaration

```
 { INT    } {.   } <identifier> ( <referral> ) [ := <initialization> ]
 { STRING } {.EXT}

 [, {.   } <identifier> ( <referral> ) [ := <initialization> ] ] ... ;
    {.EXT}
```

Substructure Redefinition

```
   STRUCT <identifier> [ "[" <lower-bound> : <upper-bound> "]" ]

                                                   = <previous-identifier> ;

   <structure-body>
```

System Global Pointer Declaration

```
   <type> .SG <identifier> [ := <preset-address> ]

      [ , .SG <identifier> [ := <preset-address> ] ] ... ;
```

EXPRESSIONS

Assignment Form of Arithmetic Expression

```
   <variable> := <expression>
```

CASE Form of Arithmetic Expression

```
   CASE <index> OF
     BEGIN
       <expression> ;      !For index = 0
       <expression> ;      !For index = 1
             .
             .
       <expression> ;      !For index = <n>
       [ OTHERWISE <expression> ; ]
     END
```

General Form of Arithmetic Expression

```
   [ + ] <primary> [ [ <arith-operator>  <primary> ] ... ]
   [ - ]
```

General Form of Conditional Expression

```
   [ NOT ] <condition> [ [ { AND } [ NOT ] <condition> ] ... ]
                           { OR  }
```

Group Comparison Form of Conditional Expression

```
   <var1> <rela-operator> { <var2> FOR <count> [ -> <next-addr> ] }
                          { <constant>                           }
```

IF-THEN-ELSE Form of Arithmetic Expression

```
   IF <conditional-expression> THEN <expression> ELSE <expression>
```

STATEMENTS

ASSERT--Conditionally invokes procedure named in ASSERTION directive

```
ASSERT <assert-level> : <expression>
```

Assignment--Assigns value to variable

```
<variable> := <expression>
```

Assignment (Pointer)--Assigns address to pointer or structure pointer

```
@<pointer-name> := <arithmetic-expression>
```

Assignment (Structure Item)--Assigns value to structure item

```
{ <struct-name>     } [ [.<substruct-name> ] ... ] .<item-name>
{ <struct-ptr-name> }

                                                   := <expression>
```

Compound Statement--Forms a single logical statement from multiple statements

```
BEGIN
  [ <statement; ] ...
END [ ; ]
```

CALL--Invokes procedure, subprocedure, or entry point

```
CALL <identifier> [ ( <param> [ , <param> ] ... ) ]
```

CASE--Executes one of several statements based on index values

```
CASE <index> OF
  BEGIN
    <statement> ;       !For index = 0
    <statement> ;       !For index = 1
          .
          .
    <statement> ;       !For index = <n>
    [ OTHERWISE <statement> ; ]
  END
```

CODE--Specifies machine-level instructions

```
   CODE ( <instruction> [ ; <instruction> ] ... )
```

DO--Executes statement until specified condition becomes true

```
   DO [ <statement> ] UNTIL <expression>
```

DROP--Disassociates identifier from label or index register

```
   DROP <name>
```

FOR--Executes statement until variable increments or decrements past limit

```
   FOR <variable> := <initial> { TO     } <limit> [ BY <step> ] DO
                               { DOWNTO }

      [ <statement> ]
```

GOTO--Unconditionally transfers program control to labeled statement

```
   GOTO <label-name>
```

IF-THEN-ELSE--Executes one of two statements based on true or false condition

```
      IF <conditional-expression>
      THEN
        [ <statement> ]
   [ ELSE
        [ <statement> ] ]
```

MOVE--Transfers contiguous bytes, words, or elements from one location to another, left to right (':=') or right to left ('=:')

```
   <destination> { ':=' } { <source> FOR <count> }  [ -> <next-addr> ]
                 { '=:' } { <constant>           }
```

RETURN--Returns from procedure or subprocedure to caller; for functions also returns value

```
   RETURN                        !Untyped procedure

   RETURN <expression>           !Function (typed procedure)
```

SCAN, RSCAN--Searches scan area for test character, left to right (SCAN) or right to left (RSCAN)

```
   { SCAN  } <variable> { WHILE } <test-char> [ -> <next-addr> ]
   { RSCAN }            { UNTIL }
```

STACK--Loads values onto register stack

```
   STACK <expression> [ , <expression> ] ...
```

STORE--Transfers values from register stack to variables

```
   STORE <variable> [ , <variable> ] ...
```

USE--Associates identifier with, and reserves, index register

```
   USE <name>
```

WHILE--Executes statement while specified condition is true

```
   WHILE <conditional-expression> DO [ <statement> ]
```

STANDARD FUNCTIONS

In the following summary, the symbol "(P)" denotes functions that perform privileged operations.

$ABS--Returns absolute value of same type as expression

```
   $ABS ( <expression> )
```

$ALPHA--Tests right half of INT value for alphabetic character

```
   $ALPHA ( <int-expression> )
```

$AXADR (P)--Returns absolute extended address of variable

```
   $AXADR ( <variable> )
```

$BOUNDS (P)--Checks location of parameter passed to system procedure

```
   $BOUNDS ( <param> , <count> )
```

$CARRY--Checks carry bit in ENV register

```
   $CARRY
```

$COMP--Returns one's complement of INT expression

```
   $COMP ( <int-expression> )
```

$DBL--Returns signed INT(32) value from any expression

```
   $DBL ( <expression> )
```

$DBLL--Returns INT(32) value from two INT expressions

```
   $DBLL ( <int-expression> , <int-expression> )
```

$DBLR--Returns rounded signed INT(32) value from any expression

```
   $DBLR ( <expression> )
```

$DFIX--Returns 64-bit integer from signed INT(32) expression

```
   $DFIX ( <dbl-expression> , <fpoint> )
```

$EFLT--Returns REAL(64) value from any expression

```
   $EFLT ( <expression> )
```

$EFLTR--Returns rounded REAL(64) value from any expression

```
   $EFLTR ( <expression> )
```

$FIX--Returns FIXED(0) value from any expression

```
   $FIX ( <expression> )
```

$FIXD--Returns INT(32) value from FIXED expression

```
   $FIXD ( <fixed-expression> )
```

$FIXI--Returns signed INT value from FIXED expression

```
   $FIXI ( <fixed-expression> )
```

$FIXL--Returns unsigned INT value from FIXED expression

```
   $FIXL ( <fixed-expression> )
```

$FIXR--Returns rounded FIXED(0) value from any expression

```
   $FIXR ( <expression> )
```

$FLT--Returns REAL value from any expression

```
   $FLT ( <expression> )
```

$FLTR--Returns rounded REAL value from any expression

```
   $FLTR ( <expression> )
```

$HIGH--Returns INT value from left half of INT(32) expression

```
   $HIGH ( <dbl-expression> )
```

$IFIX--Returns FIXED value from signed INT expression

```
   $IFIX ( <int-expression> , <fpoint> )
```

$INT--Returns INT value from any expression

```
   $INT ( <expression> )
```

$INTR--Returns rounded INT value from any expression

```
   $INTR ( <expression> )
```

$LADR--Returns standard address of item accessed via extended pointer

```
   $LADR ( <variable> )
```

$LEN--Returns byte length of variable

```
   $LEN ( <variable> )
```

$LFIX--Returns 64-bit integer from unsigned INT expression

```
   $LFIX ( <int-expression> , <fpoint> )
```

$LMAX--Returns maximum of two unsigned INT expressions

```
   $LMAX ( <int-expression> , <int-expression> )
```

$LMIN--Returns minimum of two unsigned INT expressions

```
   $LMIN ( <int-expression> , <int-expression> )
```

$MAX--Returns maximum of two signed expressions

```
   $MAX ( <expression> , <expression> )
```

$MIN--Returns minimum of two signed expressions

```
   $MIN ( <expression> , <expression> )
```

$NUMERIC--Tests right half of INT value for ASCII numeric character

```
   $NUMERIC ( <int-expression> )
```

$OCCURS--Returns number of occurrences of variable

```
   $OCCURS ( <variable> )
```

$OFFSET--Returns byte offset of structure item from structure base

```
   $OFFSET ( <variable> )
```

$OVERFLOW--Tests for arithmetic overflow condition

```
   $OVERFLOW
```

$PARAM--Checks for presence or absence of actual parameter

```
   $PARAM ( <formal-param> )
```

$POINT--Returns <fpoint>, in integer form, of FIXED expression

```
   $POINT ( <fixed-expression> )
```

$PSEM (P)--Requests semaphore on behalf of caller

```
   $PSEM ( <semaphore-addr> , <interval> )
```

$RP--Returns current setting of TAL RP counter

```
   $RP
```

$SCALE--Moves implied decimal point position in FIXED expression

```
   $SCALE ( <fixed-expression> , <scale> )
```

$SPECIAL--Tests right half of INT value for nonalphanumeric ASCII character

```
   $SPECIAL (int-expression)
```

$SWITCHES (P)--Returns current setting of SWITCH register

```
   $SWITCHES
```

$TYPE--Returns value indicating data type of variable

```
   $TYPE ( <variable> )
```

$UDBL--Returns INT(32) value from unsigned INT expression

```
   $UDBL ( <int-expression> )
```

$XADR--Returns extended address from standard address of variable

```
   $XADR ( <variable> )
```

COMPILER DIRECTIVES

A directive line begins with a "?" in the first column and has the form:

```
? <directive> [ , <directive> ] ...
```

In the following summary, a "*" following the directive name means the directive is not a feature of NonStop 1+ software.

ABORT*--Terminates compilation if TAL cannot open SOURCE file

```
[NO]ABORT
```

ABSLIST--Lists C-relative addresses

```
[NO]ABSLIST
```

ASSERTION--Generates debugging aids in conjunction with ASSERT statement

```
ASSERTION [ = ] <assertion-level> , <procedure-name>
```

CODE--Lists instruction codes in octal if LIST is also enabled

```
[NO]CODE
```

COMPACT--Moves procedures into 32K gap in code area if they fit

```
[NO]COMPACT
```

CPU--Specifies system type on which object code is to run

```
CPU { TNS    }
    { TNS/II }
```

CROSSREF--Lists identifier cross references

```
[NO]CROSSREF [ <class>                       ]
             [ ( <class> [ , <class> ] ... ) ]
```

DATAPAGES--Sets size of data area for object program

```
DATAPAGES [ = ] <integer>
```

DECS--Decrements TAL S-register counter

```
DECS [ = ] <sdec-value>
```

DEFEXPAND--Lists invoked DEFINE text

```
[NO]DEFEXPAND
```

DUMPCONS--Dumps constant table in object code

```
DUMPCONS
```

ENDIF--Terminates conditional compilation

```
ENDIF { <toggle-number> }
      { <cpu-type>      }
```

ERRORS--Sets number of error messages at which to terminate TAL

```
ERRORS [ = ] <nnnnn>
```

EXTENDSTACK--Sets number of pages to add to existing stack size

```
EXTENDSTACK <value>
```

GMAP*--Lists global map if MAP is also enabled

```
[NO]GMAP
```

ICODE--Lists mnemonics after each procedure if LIST is enabled

```
[NO]ICODE
```

IF--Allows conditional compilation

```
IF[NOT] { <toggle-number> }
        { <cpu-type>      }
```

INHIBITXX--Inhibits emission of extended, indexed instruction

```
   [NO]INHIBITXX
```

INNERLIST--Lists mnemonics after each source statement if LIST is enabled

```
   [NO]INNERLIST
```

INSPECT--Selects INSPECT or DEBUG as default debugger

```
   [NO]INSPECT
```

LIBRARY--Specifies user library files for NonStop software

```
   LIBRARY <file-name>
```

LINES*--Sets maximum number of output lines per page

```
   LINES <value>
```

LIST--Lists source and enables other listings

```
   [NO]LIST
```

LMAP--Selects BINSERV load maps and cross references

```
           { <lmap-option>                                  }
   [NO]LMAP { ( <lmap-option> [ , <lmap-option> ] ... )     }
           { *                                              }
```

MAP--Lists identifier map if LIST is also enabled

```
   [NO]MAP
```

PAGE--Specifies header and causes page ejects if LIST is enabled

```
   PAGE [ " <heading-string> " ]
```

PEP--Sets word size of PEP table for BINSERV

```
   PEP [ = ] <pep-table-size>
```

PRINTSYM*--Selectively lists symbols

```
   [NO]PRINTSYM
```

RELOCATE--Emits BINSERV warnings if references to nonrelocatable global data occur

```
   RELOCATE
```

RESETTOG--Turns toggles off

```
   RESETTOG [ <toggle-number> [ , <toggle-number> ] ... ]
```

ROUND--Specifies scalar rounding for FIXED values

```
   [NO]ROUND
```

RP--Sets internal RP counter of TAL

```
   RP [ = ] <register-number>
```

SAVEABEND--Generates INSPECT save file if program ends abnormally

```
   [NO]SAVEABEND
```

SEARCH--Specifies object files from which to resolve external references

```
   SEARCH [ <object-file-name>                                    ]
          [ ( <object-file-name> [ , <object-file-name> ] ... ) ]
```

SECTION--Gives name to part of source file for use with SOURCE directive

```
   SECTION <text-name>
```

SETTOG--Turns toggles on

```
   SETTOG [ <toggle-number> [ , <toggle-number> ] ... ]
```

SOURCE--Specifies source to read from another input file

```
   SOURCE <file-name> [ ( <section-name> [ , <section-name> ] ... ) ]
```

STACK--Sets number of stack data pages

```
   STACK <value>
```

SUPPRESS--Suppresses all but header, diagnostics, and trailer

```
   [NO]SUPPRESS
```

SYMBOLS--Generates symbol table for use with INSPECT

```
   [NO]SYMBOLS
```

SYNTAX--Checks source code syntax; does not generate object file

```
   SYNTAX
```

WARN--Selectively turns on warnings; on NonStop 1+ system, turns on all warnings

```
   [NO]WARN [ <value> ]
```

APPENDIX E

ASCII CHARACTER SET

Char	Left	Right	Hex	Dec	Meaning
NUL	000000	000000	00	0	Null
SOH	000400	000001	01	1	Start of heading
STX	001000	000002	02	2	Start of text
ETX	001400	000003	03	3	End of text
EOT	002000	000004	04	4	End of transmission
ENQ	002400	000005	05	5	Enquiry
ACK	003000	000006	06	6	Acknowledge
BEL	003400	000007	07	7	Bell
BS	004000	000010	08	8	Backspace
HT	004400	000011	09	9	Horizontal tabulation
LF	005000	000012	A	10	Line feed
VT	005400	000013	B	11	Vertical tabulation
FF	006000	000014	C	12	Form feed
CR	006400	000015	D	13	Carriage return
SO	007000	000016	E	14	Shift out
SI	007400	000017	F	15	Shift in
DLE	010000	000020	10	16	Data link escape
DC1	010400	000021	11	17	Device control 1
DC2	011000	000022	12	18	Device control 2
DC3	011400	000023	13	19	Device control 3
DC4	012000	000024	14	20	Device control 4
NAK	012400	000025	15	21	Negative acknowledge
SYN	013000	000026	16	22	Synchronous idle
ETB	013400	000027	17	23	End of transmission block
CAN	014000	000030	18	24	Cancel
EM	014400	000031	19	25	End of medium
SUB	015000	000032	1A	26	Substitute
ESC	015400	000033	1B	27	Escape
FS	016000	000034	1C	28	File separator
GS	016400	000035	1D	29	Group separator
RS	017000	000036	1E	30	Record separator

ASCII CHARACTER SET

Char	Left	Right	Hex	Dec	Meaning
US	017400	000037	1F	31	Unit separator
SP	020000	000040	20	32	Space
!	020400	000041	21	33	Exclamation point
"	021000	000042	22	34	Quotation mark
#	021400	000043	23	35	Number sign
$	022000	000044	24	36	Dollar sign
%	022400	000045	25	37	Percent sign
&	023000	000046	26	38	Ampersand
'	023400	000047	27	39	Apostrophe
(	024000	000050	28	40	Opening parenthesis
)	024400	000051	29	41	Closing parenthesis
*	025000	000052	2A	42	Asterisk
+	025400	000053	2B	43	Plus
,	026000	000054	2C	44	Comma
-	026400	000055	2D	45	Hyphen (minus)
.	027000	000056	2E	46	Period (decimal point)
/	027400	000057	2F	47	Right slash
0	030000	000060	30	48	Zero
1	030400	000061	31	49	One
2	031000	000062	32	50	Two
3	031400	000063	33	51	Three
4	032000	000064	34	52	Four
5	032400	000065	35	53	Five
6	033000	000066	36	54	Six
7	033400	000067	37	55	Seven
8	034000	000070	38	56	Eight
9	034400	000071	39	57	Nine
:	035000	000072	3A	58	Colon
;	035400	000073	3B	59	Semicolon
<	036000	000074	3C	60	Less than
=	036400	000075	3D	61	Equals
>	037000	000076	3E	62	Greater than
?	037400	000077	3F	63	Question mark
@	040000	000100	40	64	Commercial at sign
A	040400	000101	41	65	Uppercase A
B	041000	000102	42	66	Uppercase B
C	041400	000103	43	67	Uppercase C
D	042000	000104	44	68	Uppercase D
E	042400	000105	45	69	Uppercase E
F	043000	000106	46	70	Uppercase F
G	043400	000107	47	71	Uppercase G
H	044000	000110	48	72	Uppercase H
I	044400	000111	49	73	Uppercase I
J	045000	000112	4A	74	Uppercase J

Char	Left	Right	Hex	Dec	Meaning
K	045400	000113	4B	75	Uppercase K
L	046000	000114	4C	76	Uppercase L
M	046400	000115	4D	77	Uppercase M
N	047000	000116	4E	78	Uppercase N
O	047400	000117	4F	79	Uppercase O
P	050000	000120	50	80	Uppercase P
Q	050400	000121	51	81	Uppercase Q
R	051000	000122	52	82	Uppercase R
S	051400	000123	53	83	Uppercase S
T	052000	000124	54	84	Uppercase T
U	052400	000125	55	85	Uppercase U
V	053000	000126	56	86	Uppercase V
W	053400	000127	57	87	Uppercase W
X	054000	000130	58	88	Uppercase X
Y	054400	000131	59	89	Uppercase Y
Z	055000	000132	5A	90	Uppercase Z
[	055400	000133	5B	91	Opening bracket
\	056000	000134	5C	92	Back slash
]	056400	000135	5D	93	Closing bracket
^	057000	000136	5E	94	Circumflex
_	057400	000137	5F	95	Underscore
`	060000	000140	60	96	Grave accent
a	060400	000141	61	97	Lowercase a
b	061000	000142	62	98	Lowercase b
c	061400	000143	63	99	Lowercase c
d	062000	000144	64	100	Lowercase d
e	062400	000145	65	101	Lowercase e
f	063000	000146	66	102	Lowercase f
g	063400	000147	67	103	Lowercase g
h	064000	000150	68	104	Lowercase h
i	064400	000151	69	105	Lowercase i
j	065000	000152	6A	106	Lowercase j
k	065400	000153	6B	107	Lowercase k
l	066000	000154	6C	108	Lowercase l
m	066400	000155	6D	109	Lowercase m
n	067000	000156	6E	110	Lowercase n
o	067400	000157	6F	111	Lowercase o
p	070000	000160	70	112	Lowercase p
q	070400	000161	71	113	Lowercase q
r	071000	000162	72	114	Lowercase r
s	071400	000163	73	115	Lowercase s
t	072000	000164	74	116	Lowercase t
u	072400	000165	75	117	Lowercase u
v	073000	000166	76	118	Lowercase v

Char	Left	Right	Hex	Dec	Meaning
w	073400	000167	77	119	Lowercase w
x	074000	000170	78	120	Lowercase x
y	074400	000171	79	121	Lowercase y
z	075000	000172	7A	122	Lowercase z
{	075400	000173	7B	123	Opening brace
|	076000	000174	7C	124	Vertical line
}	076400	000175	7D	125	Closing brace
~	077000	000176	7E	126	Tilde
DEL	077400	000177	7F	127	Delete

APPENDIX F

DATA TYPE CORRESPONDENCE

Table F-1 provides a table of corresponding data types for Tandem languages.

You will find this table useful if you are working with a particular language and want to:

1. Pass parameters to another language

2. Use data from a file that was created by a program in another language

If you are using the Data Definition Language (DDL) utility to describe your files, you do not need this table. You can ask DDL to produce equivalent data declarations in the language you specify.

All parameters being passed to a procedure written in a language other than TAL must be passed by reference.

Table F-1. Data Type Correspondence

Data	TAL	BASIC	COBOL	FORTRAN
8-Bit Integer	STRING	STRING	Alphabetic Numeric Display Alphanumeric-Edited Alphanumeric Numeric-Edited	N/A
16-Bit Integer	INT	INT INT(16)	COMP 9(1)-COMP 9(4) without P or V Index Data Item Index	INTEGER*2
32-Bit Integer	INT(32)	INT(32)	COMP 9(5)-COMP 9(9) without P or V	INTEGER*4

64-Bit Integer	FIXED(0)	INT(64)
		FIXED(0)
			COMP 9(10)-COMP 9(18) without P or V	INTEGER*8

64-Bit Fixed Point	FIXED(n)	FIXED(n)	COMP 9(10)-COMP 9(18) with appropriate V	N/A
32-Bit Floating Point	REAL	REAL	N/A	REAL
64-Bit Floating Point	REAL(64)	REAL(64)	N/A	DOUBLE PRECISION
64-Bit Complex	N/A	N/A	N/A	COMPLEX
Character	STRING	STRING	Alphabetic Numeric Display Alphanumeric-Edited Alphanumeric Numeric-Edited	CHARACTER
Character String	STRING (array)	STRING	Alphabetic Numeric Display Alphanumeric-Edited Alphanumeric Numeric-Edited	CHARACTER*n or CHARACTER array

Table F-1. Parameter Correspondence (continued)

Data	TAL	BASIC	COBOL	FORTRAN
Byte-Addressed Structure	16-bit STRING structure pointer	N/A	N/A	RECORD
Word-Addressed Structure	16-bit INT structure pointer or structure identifier	MAP buffer	01-level RECORD	N/A

INDEX

READER COMMENT CARD

Tandem welcomes your comments on the quality and usefulness of its software documentation. Does this manual serve your needs? If not, how could we improve it? Your comments will be forwarded to the writer for review and action, as appropriate.

If your answer to any of the questions below is "no," please supply detailed information, including page numbers, under Comments. Use additional sheets if necessary.

▶ Is this manual technically accurate?	Yes ☐	No ☐
▶ Is information missing?	Yes ☐	No ☐
▶ Are the organization and content clear?	Yes ☐	No ☐
▶ Are the format and packaging convenient?	Yes ☐	No ☐

Comments

Name Date

Company

Address

City/State Zip

Transaction Application Language (TAL™) Reference Manual

NonStop™ Systems
NonStop 1+™ System

82581 A00

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

B U S I N E S S R E P L Y M A I L

FIRST CLASS PERMIT NO. 482 CUPERTINO, CA, U.S.A.

POSTAGE WILL BE PAID BY ADDRESSEE

Tandem Computers Incorporated
Attn: Manager—Software Publications
Location 01, Department 6350
19333 Vallco Parkway
Cupertino CA 95014-9990

TAPE

TAPE

Tandem Computers Incorporated
19333 Vallco Parkway
Cupertino, CA 95014-2599