PERKIN-ELMER

# PASCAL
# USER GUIDE, LANGUAGE REFERENCE, AND RUN TIME SUPPORT

Reference Manual

48-021 R01

The Perkin-Elmer Corporation, Data Systems Group, 2 Crescent Place, Oceanport, New Jersey 07757

Printed in the United States of America

# TABLE OF CONTENTS

CHAPTERS

## PART I PASCAL USER GUIDE

1 PASCAL USER GUIDE

PART II LANGUAGE REFERENCE

2 LANGUAGE CONCEPTS AND SYNTAX GRAPHS

CHAPTERS (Continued)

CHAPTERS (Continued)

5 DATA CONSTANTS, TYPES, AND VARIABLE SELECTORS

6 EXPRESSIONS, TYPE COMPATIBILITY, AND TYPE CONVERSIONS

7 PASCAL EXECUTABLE STATEMENTS

CHAPTERS (Continued)

CHAPTERS (Continued)

## PART III RUN TIME SUPPORT INFORMATION

10 RUN TIME SUPPORT INFORMATION AND LANGUAGE EXTENSIONS

## PART IV APPENDIXES

## FIGURES

## FIGURES (Continued)

## TABLES

TABLES (Continued)

# PREFACE

This document defines the programming language Pascal as implemented for the Perkin-Elmer 3200 Series computers.

This document is divided into four main parts which further consist of Chapters or Appendices.

Part I, which comprises Chapter 1, is a Perkin-Elmer Pascal User's Guide, providing introductory overview information and detailed information on the Perkin-Elmer Pascal Compiler, compiling a user-written Pascal source program by executing the compiler with selected options, compiler operations, and establishing the user's compiled-program as an executable task under OS/32. Refer to Appendix M for RELIANCE environments.

Part II, which comprises Chapters 2 through 9, is a Perkin-Elmer Pascal Language Reference Manual, defining the Pascal Language as provided by this implementation.

> Chapter 2 presents a glossary of basic terms, detailing the heirarchy of identifier scopes, and introduces the use of syntax-graphs; which are used to define Pascal syntax.
>
> Chapter 3 presents the language elements, such as the Pascal Character Set as a subset of the ASCII character set, special symbols, special characters, comments, literal constants, rules on separating symbols and separators, reserved words of the language, predefined identifiers, and gives definitive summaries of available predefined/standard routines; with instructions for accessing some math routines from the Perkin-Elmer System Math Functions Library.
>
> Chapter 4 provides the introduction to program block structure, and the syntax of Declarations (LABEL, CONSTant, TYPE, and VARiable parts) and the Body of a block.
>
> Chapter 5 details not only syntax, but by extensive examples, constants, and the various Pascal data types, and contains the syntax of variable-selectors, as the means for selecting and/or specifying a reference to a variable.
>
> Chapter 6 details the syntax of expressions in Pascal, contains the rules concerning operator precedence, summarizes the operators and applicable data-types, and contains information on programming for "identity" of type and "assignment-compatibility" of type regarding type-compatibility in Pascal.
>
> Chapter 7 describes the various executable statements in the Pascal language, reserving the Pascal I/O statements to Chapter 8; with additional statements provided by the Prefix, and additional SVC call capability, given in Chapter 10.

Chapter 8 discusses Pascal files, and the file-type data-type, including Pascal TEXT files, and details the definitions of Pascal I/O procedure-call statements.

Chapter 9 details the syntax of PROGRAM, MODULE, PROCEDURE, and FUNCTION header statements; covers the program/module prefix syntax which is a Perkin-Elmer extension; and details several issues concerning the programming of routines.

Part III, which comprises Chapter 10, is the Perkin-Elmer Pascal Run Time Support Information and Language Extensions. It contains a description of the Pascal Runtime Library, the use of the Perkin-Elmer Pascal Prefix, the use of the Perkin-Elmer SVC capability in Perkin-Elmer Pascal, run time memory utilization, internal data storage mechanisms, and contains a description of this implementation's Pascal linkage conventions amongst internal routines, and between external routines declared EXTERN, either CAL routines or Pascal MODULEs, and those external routines declared with the directive FORTRAN.

Part IV contains various appendices which summarize key information contained in the body of this document, plus additional information on extensions and exceptions to standard Pascal listed in Appendix K; user information on preparing programs to run in a RELIANCE environment in Appendix M; and a summary of Functional Differences between Pascal R00 and Pascal R01 in Appendix P.

The reader is assumed familiar with Perkin-Elmer 32-bit Software Systems. The following manuals provide detailed information on related Perkin-Elmer Software.

| | |
|---|---|
| Common Assembler Language (CAL) Programming Reference Manual | S29-640 |
| OS/32 Library Loader Reference Manual | 48-020 |
| OS/32 Operator Reference Manual | S29-574 |
| OS/32 Application Level Programmer Reference Manual | 48-039 |
| OS/32 System Level Programmer Reference Manual | 48-040 |
| OS/32 Link Reference Manual | 48-005 |
| OS/32 Supervisor Call (SVC) Reference Manual | 48-038 |
| Perkin-Elmer System Mathematical Functions Reference Manual | 48-025 |
| FORTRAN VII Reference Manual | 48-017 |
| FORTRAN VII User Manual | 48-010 |

# PART I

# PASCAL USER GUIDE

# CHAPTER 1
# PASCAL USER GUIDE

## 1.1 INTRODUCTION

This chapter of the Pascal Reference Manual is a user operations guide with system overview information and details on how to operate the compiler, establish a compiled program as a task, and execute it under OS/32. Users preparing programs for a Reliance environment must refer to Appendix M and Section 1.5.3.14.

## 1.2 PASCAL PRODUCT OVERVIEW

The Perkin-Elmer Pascal software product consists of:

- A multi-pass optimizing compiler supplied in both overlaid (PASCAL.TSK) and resident task versions (PASCALR.TSK).
- A run time library (PASRTL.OBJ) of assembler written object routines which satisfy calls generated by the compiler in the compiled object program, to support the running user task. This library includes support routines for a basic set of SVC calls that the user may code in Pascal. Sample SVCs and prerequired source declarations are provided on the file, SMPLSVCS.PAS.
- CSS procedures to facilitate compilation and establishment of a user program as a task to run under OS/32.
- This product reference manual, providing a user guide, a Pascal language reference, and run time support information.
- A package information document with instructions for unpackaging the product. It contains an installation exercise to ensure installation of an operational package.
- A Perkin-Elmer Prefix of Pascal source declarations (PREFIX.PAS), which when prefixed to the user source program before compilation, provides Pascal language extensions for additional input/output (I/O) and other system services.
- The Perkin-Elmer System Mathematical Library (on PEMATH.OBJ) provides several math routines.

Figure 1-1 is a Pascal language system overview. It depicts the program development cycle and compiler operations.

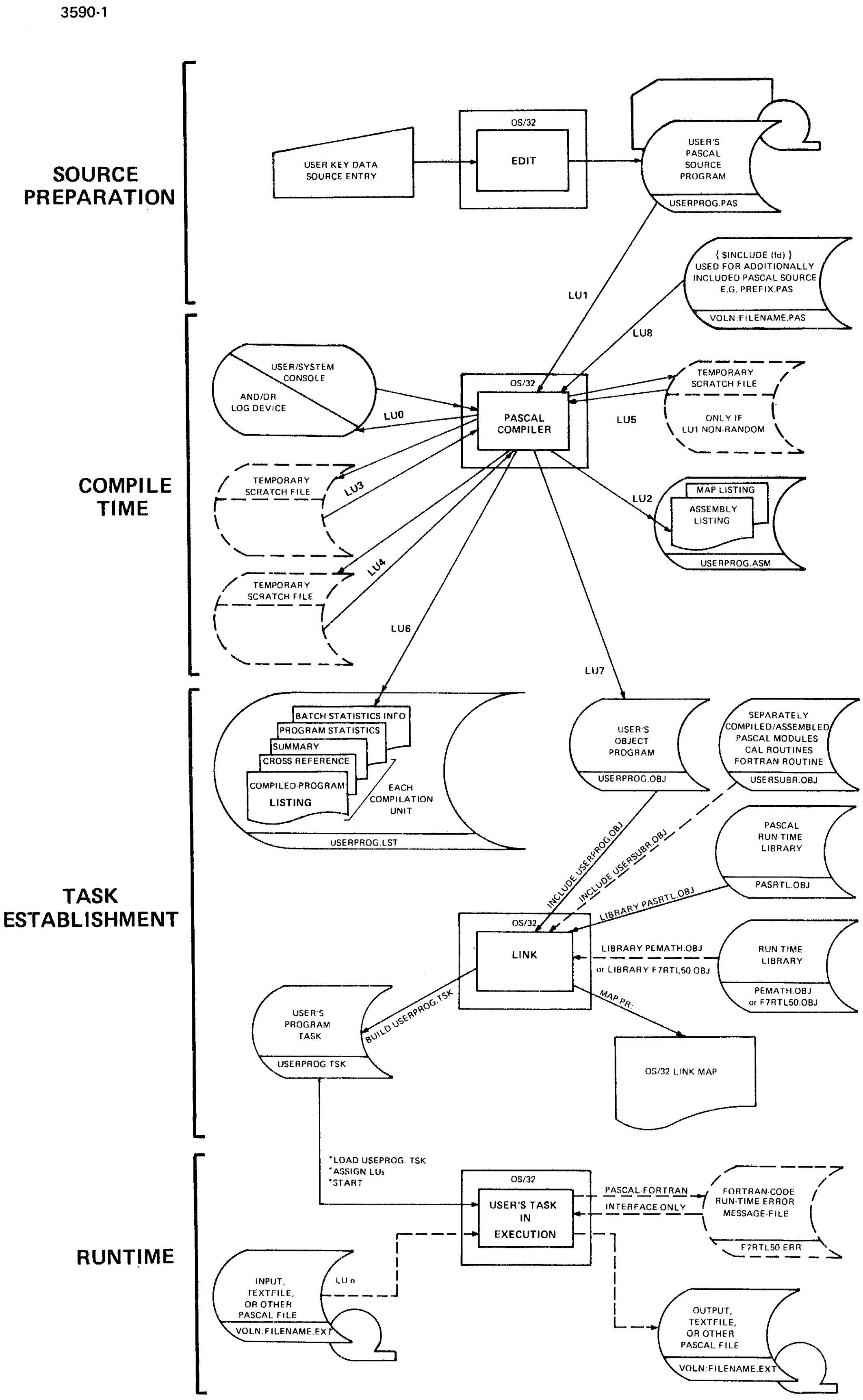


Figure 1-1 Pascal Language System Overview

## 1.3 PASCAL SYSTEM REQUIREMENTS

The minimum system configuration required to compile and run a Pascal program, in addition to this product, are:

- Perkin-Elmer 32-bit processor
- OS/32 R05.2 or higher
- OS/32 Link R00 or higher
- One direct access device
- A 168KB task memory partition under OS/32

The compiler is provided as two tasks; an overlaid version, PASCAL.TSK, and a resident version, PASCALR.TSK. Regardless of the compiler version used, its user operation is the same.

The overlaid version, not the resident version, of the compiler supports this minimum configuration. The overlaid version requires the least memory space at compile time with a slight penalty in compile speed because of the overhead of overlay loading. Overlay loading is otherwise transparent to the user. The root segment of this version is sharable and amounts to approximately 14KB of the compiler's code. The space required for the overlays themselves must be available in the user task partition. See Section 1.5.2.

The resident version can be shared by many users since all passes of the compiler are established, along with the root segment, as one pure task. The resident version provides an advantage in overall memory utilization when the number of concurrent users exceeds six.

The minimum configuration supports compilation of medium size Pascal source programs, in which the number of unique identifiers is about 1000 and in which routines are no more than about 500 lines.

The following additional hardware/software options enhance either compilation performance or the run time support environment.

- Distribution of source, object, listing and scratch files on more than one direct access device can significantly improve compilation speed.
- Increased partition size permits compilation of larger programs, or permits use of the resident compiler.
- The routines in the Perkin-Elmer System Mathematical Library can be called by the user program. See Section 3.5.9.
- Source programs may be prepared by OS/32 Edit or Text.

## 1.4 PASCAL SYSTEM FEATURES

The Pascal product provides several features to enact, enhance, and simplify program development. These are:

- Language features
- Compiler features
- CSS features
- Run time features

### 1.4.1 Language Features

The language in which programs are written is defined in Part II of this manual, Chapters 2 through 9. The language is a complete implementation of Pascal as described by Jensen and Wirth in the Pascal User Manual and Report, with several extensions. Also included are prominent stable features of the proposed ISO Pascal Language Standard published in SIGPLAN Notices, Volume 15, Number 4, April 1980.

Exceptions and extensions to this standard Pascal language provided by this implementation are described in Appendix K.

Additional extensions afforded the user by the Perkin-Elmer Prefix and SVC routine declarations have their use described in Sections 10.3 and 10.4, respectively.

The user is given the capability to code SVCs within his Pascal program, as Pascal procedure-call statements after declaring the necessary SVC source interfaces.

The Perkin-Elmer Prefix extends the language to allow the user to:

- Open a file or device
- Close a file or device
- Allocate a file
- Rename a file
- Reprotect a file
- Delete a file
- Change Access Privileges of a file or device
- Checkpoint a file or device
- Fetch Attributes of a file or device
- Rewind a file or device
- Write a file mark
- Backspace a Record
- Backspace to a file mark
- Forward space a record
- Forward space to a file mark
- Breakpoint a running Pascal program

- Obtain the Start Parameters
- Obtain the time
- Obtain the date
- Exit the task with a specified return code

### 1.4.2 Compiler Features

The compiler accepts Pascal source which can be prepared by OS/32 Edit or OS/32 Text. Pascal source, if on a non-random device, is automatically copied by the compiler to a scratch file for subsequent rereading by the compiler. Batch compilation of programs and modules is provided. It is also possible to merge additional source files into the source stream.

Several programming aids are generated by the compiler at the user's option, such as:

- Compiled Program Listing
- Cross reference listing
- Summary listing
- Assembly listing
- Map of object program listing
- Program statistics listing
- Batch statistics listing

Refer to Appendix A for a sample compiled-program listing.

Optimizations which may increase the efficiency of the user object program are performed by the compiler at the user's option. Object programs optionally can be compiled to contain additional data validity checking code; or not to contain the checking when space economy is of higher priority.

Object programs are generated as pure code, directly usable by Link.

The compiler can detect a variety of user coding errors and displays these diagnostic errors below the line in which the error was detected. Also, the compiler summarizes any diagnostic error messages at the end of each listing. The textual error messages indicate the nature of the error.

Refer to Appendix G for the Pascal diagnostic error messages displayed in compiled-program listings.

To expedite the identification of any unexpected compiler failures, there is a special compiler error message. See Section

1.5.4 on error handling. See Appendix B for instructions on reporting or requesting resolution via a Software Change Request (SCR).

### 1.4.3 CSS Features

Three Command Substitution System (CSS) Procedures are provided: a compile CSS, a link CSS, and a compile and link CSS, to establish the user program as an executable task from a user MTM terminal under OS/32. Three additional CSS procedures are similarly provided for operation from the system console under OS/32.

The Pascal CSS's are summarized in Appendix C, and detailed in Section 1.6.4. Also see some sample ease-of-use examples of the Pascal CSS's applying $BATCH and $INCLUDE in Section 1.10 at the end of this chapter.

### 1.4.4 Run Time Features

A variety of run time error messages, as listed in Appendix G, are provided to detect run time errors, in the running user task.

During execution of an established Pascal program task, the Pascal Run Time Library provides a variety of run time support routines. The Pascal Run Time Library is classified into six major parts, for the purpose of discussion, as follows:

- The Pascal Initializer and Common Error Message Routines
- The Pascal Task Pausing/EOT/Error Handler (PAS.ERR group)
- The RELIANCE-Pascal Interface/Error Handler (PAS.REL group)
- The Pascal Prefix Support Routines (PASPREF group)
- The Pascal SVC Support Routines (PASSVC group)
- The Pascal Library Routines (PASLIB group)

A Pascal R01 enhancement provides the entire Pascal Run Time Library one object file, PASRTL.OBJ. Some of its routines, and only those necessary to resolve a user program's external references, are linked to Pascal compiled program code during task establishment. Also added, is the flexibility to establish and run under either an OS/32 or RELIANCE system environment.

An established Pascal (R01 and up) program task always contains a version of the Pascal Initializer, always contains the common Error Message Routine, always contains either the PAS.ERR group or the PAS.REL group, but not both; and almost invariably contains several routines from the PASLIB group. If the original program source had referenced the standard Prefix definitions, and was compiled with the Prefix, then the established task contains several run time support routines from PASPREF. If the original program source declared and used the Pascal routines to issue SVC calls, then the established task contains several run time support routines from the PASSVC group.

The Pascal Initializer, P$INIT, initializes the memory management mechanisms for the task. If the Pascal program is interfacing with FORTRAN produced code or CAL routines using FORTRAN linkage conventions (as directed by the Pascal "FORTRAN" directive), then the alternate version of the Pascal Initializer is incorporated, and it also performs additional housekeeping for the FORTRAN interface.

Programs intended to run in a basic OS/32 environment (non-Reliance environment) are compiled into object form that users (references) PAS.ERR group for error handling, task pausing, and task termination at run time.

Programs intended to run in a RELIANCE environment are compiled with a user-specified compiler option, "RELIANCE", so that the compiled object program uses (references) the PAS.REL group for error handling, task pausing, and task termination; at run time.

The Pascal Prefix Support Routines, from the PASPREF group, provide the run time support for tasks using the standard Perkin-Elmer Prefix routine definitions.

The Pascal SVC Support Routines, the PASSVC group, are the run time support routines for a set of basic SVC calls, (callable from Pascal code), whose source interface declarations are provided with the product. See Sections 10.2.5 and 10.4.

The Pascal Library, the PASLIB group, is the run time support library of routines which enact certain language features of Pascal in the running task. The PASLIB group contains the routines which perform heap management, linkage to FORTRAN subprograms, copying and comparison of structured variables, set operations, Pascal file input and output, and formatted Pascal textfile input and output.

Chapter 10 provides in detail information on the above run time features and the use of the Prefix and SVC routines.

## 1.5 PASCAL COMPILER DESCRIPTION

The Pascal compiler performs 10 passes in compiling a Pascal source program, translating the program into coded intermediate language for inter-pass communication. Different compiler passes analyze, optimize, and convert that program into a linkable object form and a printed listing. Figure 1-2 is an overview of the operational phases performed in the 10 passes.

```
Start Compiler ---------->|
                          V
            -------------------------------
Source --->|          Pass 1               |
(lu 1)     |-------------------------------|
           |                               |
           |      Lexical analysis         |
            -------------------------------
                          |
                          V
                       (lu 3)
                          |
                          V
            -------------------------------
           |          Pass 2               |
           |-------------------------------|
           |                               |
           |      Syntax analysis          |
            -------------------------------
                          |
                          V
                       (lu 4)
                          |
                          V
            -------------------------------
           |          Pass 3               |
           |-------------------------------|
           |                               |
           |       Scope analysis          |
            -------------------------------
                          |
                          V
                       (lu 3)
                          |
                          V
            -------------------------------
           |          Pass 4               |
           |-------------------------------|
           |                               |
           |    Declaration analysis       |
            -------------------------------
                          |
                          V
                       (lu 4)
                          |
                          V
            -------------------------------
           |          Pass 5               |
           |-------------------------------|
           |                               | if error
           |       Body analysis           |--------->Pass10
            -------------------------------
                          |
                          V
```

Figure 1-2  Pascal Compiler Operations

```
            (lu 3)
              |
              V
 ------------------------------
|          Pass 6              |
|------------------------------|
|      Machine independent     |
|         optimization         |
 ------------------------------
              |
              V
            (lu 4)
              |
              V
 ------------------------------
|          Pass 7              |
|------------------------------|
|   Symbolic code generation   |
 ------------------------------
              |
              V
            (lu 3)
              |
              V
 ------------------------------
|          Pass 8              |
|------------------------------|
|Machine dependent optimization|
|Instruction/address selection |
 ------------------------------
              |
              V
            (lu 4)
              |
              V
 ------------------------------
|          Pass 9              |->Assembly Listing
|------------------------------|  Map Listing (lu 6)
|                              |
|    Object code generation    |->Object Program
 ------------------------------       (lu 7)
              |
              |     Pass 10
              |<---(lu 3)
              |     error-entry
              V
 ------------------------------      (lu 2)
|          Pass 10             |->Listings of:
|------------------------------|  Compiled Program
|                              |  Cross-reference
|   Listing/XREF generation    |  Summary
 ------------------------------   Program Statistics
                                  Batch Statistics
```

**Figure 1-2 Pascal Compiler Operations (Continued)**

Pass 1 performs a lexical analysis of the source, creating a symbol table index for each unique identifier. The source program is converted into a coded intermediate form which is written to a temporary disc scratch file on lu 3. Subsequent passes read, modify, and rewrite this representation of the program unit being compiled, interweaving inter-pass communication between lu 3 and lu 4.

Pass 2 performs a context-free syntax analysis of the source program inputing its intermediate form from lu 3 and outputing to lu 4 for Pass 3.

Pass 3 performs a scope analysis of the identifiers. It differentiates between local and global variables and replaces the earlier general indices with indices unique to each identifier at a given nesting level. It communicates to Pass 4 through lu 3.

Pass 4 performs a context-sensitive analysis of the CONST, TYPE, and VAR declarations and allocates storage on the stack for the variable declarations. It communicates to Pass 5 through lu 4.

Pass 5 performs the main body analysis; i.e., a context-sensitive analysis of the executable Pascal statements, and generates information on lu 3. If certain errors preclude the possibility of continuing to produce object code, Pass 5 abortively skips to Pass 10.

Pass 6 performs machine independent optimizations (see Section 1.5.5.1). This operation recognizes certain arrangements of intermediate code and reproduces it in a more efficient form independent of the machine architecture. It communicates to Pass 7 through lu 4. Pass 6 and 7 also detect some diagnostics.

Pass 7 generates symbolic machine code and selects specific register allocations. It communicates to Pass 8 through lu 3.

Pass 8 performs machine dependent optimizations on the symbolic intermediate code produced in Pass 7. Instruction formats are selected and addresses are assigned. Certain arrangements of this code are reproduced in a more efficient form (see Section 1.5.5.2) and communicated to Pass 9 through lu 4.

Pass 9 generates object code acceptable to Link. An object program map, giving relative locations of data and statements, is optionally listed on this pass. Also, a listing of the object program, as disassembled into Common Assembly Language (CAL/32) instruction formats, optionally can be produced on this pass.

Pass 10 generates a listing of the source program just compiled into an object program. A summary of internal compiler statistics pertaining to the user compiled program, and a cross reference of identifiers used in the program optionally can be produced on this pass. Batch statistics are produced if a batch compilation is in effect. Individual program statistics, such as file allocations, are also listed.

### 1.5.1 Pascal Compiler I/O Requirements

The Pascal compiler can use any of the following logical units (lu):

- logical unit 0, log device or file
- logical unit 1, source input device or file
- logical unit 2, listing device or file
- logical unit 3, scratch file
- logical unit 4, scratch file
- logical unit 5, scratch file
- logical unit 6, map or assembly listing device or file
- logical unit 7, object code output device or file
- logical unit 8, additional source input file

All necessary logical units, or those required for specified options, must be assigned to physical devices or files by the user. Logical unit 5, which is internally controlled by the compiler, and lu 8 which the compiler assigns to a user-specified $INCLUDE in-stream option are exceptions. If not using CSS procedures, the user must directly assign these logical units (lu 0, 1, 2, 3, 4, 6, and 7) after loading the compiler and before executing it. Refer to Section 1.6.4 for CSS information.

Table 1-1 is a listing of logical units, their use, when their assignment is required, data formats, logical record length requirements, and allowable media.

TABLE 1-1 PASCAL COMPILER I/O REQUIREMENTS

| LOGICAL UNIT (LU) | USE | ASSIGNMENT | DATA FORMAT | LOGICAL RECORD LENGTH | MEDIA |
|---|---|---|---|---|---|
| 0 | Output: Log information | Always required | ASCII | 64 to 256 | device/file |
| 1 | Input: User Pascal source program | Always required | ASCII | Up to 256 | device/file |

TABLE 1-1 PASCAL COMPILER I/O REQUIREMENTS (Continued)

| LOGICAL UNIT (LU) | USE | ASSIGNMENT | DATA FORMAT | LOGICAL RECORD LENGTH | MEDIA |
|---|---|---|---|---|---|
| 2 | Output: List, cross, summary, batch statistics and program statistics | Always required for program statistics | ASCII | 132 | device/ file |
| 3 | Input/output: Temporary compiler scratch information | Always required | BINARY | 512 | file only |
| 4 | Input/output: Temporary compiler scratch information | Always required | BINARY | 512 | file only |
| 5 | Input/output: Temporary compiler scratch information | Internally controlled by compiler | ASCII | 256 | file only |
| 6 | Output: Assembly listing and Map listing | Optional Required for map and assembly | ASCII | 132 | device/ file |
| 7 | Output: Compiled program object | Always required | BINARY | 126 | device/ file |
| 8 | Input: $INCLUDE file | Internally controlled by compiler | ASCII | Up to 256 | device/ file |

### 1.5.1.1 The Log Output Device

Assignment of the log output device is always required for the compiler to identify itself and to output other ASCII formatted compiler-operations messages. Refer to Appendix G for a list of compiler-operations messages that may occur during compilation.

The LOG option additionally enables the compiler to identify each Pass currently operating and the number of errors that occurred during the pass, if any; to the compiler's log device. Refer to Section 1.5.3.9 on the LOG option.

#### 1.5.1.2 The Source Input Device

The source input device or file on logical unit 1 contains the Pascal source program in ASCII format of logical record lengths of up to 256 bytes. If the source input records are larger than the print medium records, the listed line will wrap around to the next line.

When the source input unit is a device (non-random), the compiler automatically allocates and assigns lu 5 to the temporary file to copy lu 1 source onto lu 5 for subsequent reuse in Pass 10. Logical unit 5 is a temporary scratch file used only for this purpose.

#### 1.5.1.3 The Listing Device

Output to the listing device or file on logical unit 2 is in ASCII in logical record lengths of 132 bytes to produce:

- Compiled program listings
- Cross reference listings
- Summary option listings
- Program statistics listings
- Batch statistics listings

See Section 1.9 for descriptions of listing formats.

#### 1.5.1.4 Scratch Files

The compiler requires two temporary scratch files during compilation. The files are allocated by the user (when standard CSS's are not in use) to lu 3 and lu 4 as indexed files with a logical record length of 512 bytes, just as the Pascal CSS's do.

These file allocations are not controlled internally by the compiler. The compiler performs both input and output of binary data to these files for inter-pass communication. Some user sites may operate the compiler with lu 3 and lu 4 allocated to the same volume. Deferring these allocations to the user or CSS level gives the user the option, if possible, of placing them on different volumes for increased compilation speed.

It is possible to shorten compilation-time for large, nearly stable programs once the largest number of sectors used during compilation is known, by allocating temporary contiguous files for lu 3 and lu 4.

The Summary Listing for Pass 1 through Pass 8 displays this information (produced under the SUMMARY option). See Section 1.9.2.

Another scratch file, lu 5, is internally controlled by the compiler depending on whether lu 1 supports random-access as described in Section 1.5.1.2. If lu 1 is non-rewindable, lu 1 source is copied to lu 5 for subsequent rereading on Pass 10.

### 1.5.1.5 The Assembly Listing Device

Logical unit 6 (the assembly listing device or the map option listing device) can be a print device or file that can receive ASCII logical record lengths of 132 bytes. At the user's option, the compiler outputs the following listings to lu 6:

- Assembly listing
- Map listing

See Section 1.9 for descriptions of listing formats.

### 1.5.1.6 The Object Device

The object device or file, on lu 7, is required for every compilation although it may be assigned to the null device prior to starting the compiler. The compiler outputs to the object device binary object code in logical record lengths of 126 bytes; i.e., an object module of pure code which is directly usable by OS/32 Link.

### 1.5.1.7 Additional INCLUDE Source Files

The assignment of source files to lu 8 is internally controlled by the compiler for the $INCLUDE option. These source files are ASCII files of Pascal source that are merged with the main source stream from lu 1. The compiler assigns this lu 8 to the user-specified file descriptor (fd) given with each in-stream $INCLUDE option. Refer to Section 1.5.3.7 for details of the $INCLUDE option. The logical record length requirements of $INCLUDE files are identical to those specified for the main source input file on lu 1.

### 1.5.2 Compiler Memory Requirements and Speed

The size of the overlaid compiler is determined by the size of its root segment (14KB) and largest overlay (90KB) and is approximately 104KB taken together. The size of the resident compiler is determined by the size of its root segment (14KB) plus the sum of the sizes of all of the compiler passes and is approximately 560KB. Approximately 14KB of the overlaid version is the sharable root segment and the compiler passes as overlays themselves, occupy space (as they occur) in the user task partition at compile-time. The resident version is sharable.

With either compiler version, additional memory is required for table space at compile-time. The Pascal CSS's default this "memincr" to 64KB. The amount of required space is source program dependent. A memory segment-size increment of 300KB will usually suffice for compilation of 4000 line programs containing 500 unique identifiers with complex statement structures; e.g., a CASE statement with all 128 possible choices indicated.

The compiler has a compilation speed of approximately 1200 lines per minute without optimization, and 400 lines per minute with optimization. The exact compilation speed with optimization depends on the complexity of the source program. Programs that do not lend themselves to any of the available optimizations compile at a rate somewhat better than 400 lines per minute.

### 1.5.3 Pascal Compiler Options

The compiler processes the user-written Pascal source program into linkable object code. Compiling Pascal source code under the default option state allows generation of a compiled program listing, cross reference listing, and certain data validity bounds and range checks in the object code.

Additional compile time options provided by the Pascal compiler allow the user to tailor compilations and improve ease-of-use during the program development cycle.

The compiler defaults to an internally initialized state when it is started without any user-specified options. Certain options are considered off/on unless the user reverses them to on/off with either a start option or an in-stream option selection. A start option is passed to the compiler as an argument in the START (ST) directive. An in-stream option having a slightly different format is embedded in the source stream of the Pascal source program. Some options can be selected only as start options, others only as in-stream options. Some can be specified by either method. In-stream options over-ride start options.

The Pascal compiler options give the user flexibility in tailoring compilations that affect source, object code, listings, listing-format-control, memory allocation schemes, and ease-of-use. Each compiler option is detailed in the following sections and summarized in Table 1-2.

## TABLE 1-2 PASCAL COMPILE TIME OPTIONS

| OPTION-SPECIFIER | FUNCTION |
|---|---|
| ASSEMBLY | Assembly listing option prints out a listing on lu 6 a disassembly of the compiled object program in an assembler-level format. |
| BATCH | Batch compilation option compiles a batch or series of Pascal programs or modules from lu 1 until an end of file (EOF) or an end of medium (EOM) condition, or {$BEND} option is encountered on the source file or device. |
| BEND | Batch end option signals end of batch to the compiler. Required only if BATCH is in effect and lu 1 is a non-random access device. Must be placed on line after "END." |
| BOUNDSCHECK | Subrange bounds check option generates object code within the compiled program that will check for illegal out-of-bounds values assigned to variables of the subrange-type. |
| CROSS | Cross reference listing option prints out a listing on lu 2 which is a cross reference of labels and of identifiers used in the compiled program, relating place of definition and place of reference by source line number. |
| EJECT | Eject listing format control option, wherever encountered in the source stream, causes a top-of-form (or page ejection) within the compiled program listing being printed on lu 2 under the LIST option. |
| HEAPMARK | The Heapmark option allows the compiler to recognize MARK and RELEASE references in the user's source as predefined procedure identifiers. |
| INCLUDE (fd,arg1, arg2) | Include additional source option includes, wherever {$INCLUDE (fd)} is specified in-stream, additional source from the file indicated by fd. Arg1 or arg2 may be LIST or NLIST, or CROSS or NCROSS. |
| LIST | The List option outputs the compiled-program listing on lu 2. |
| LOG | Log option prints notifications (logs) on lu |

TABLE 1-2 PASCAL COMPILE TIME OPTIONS (Continued)

| OPTION-SPECIFIER | FUNCTION |
|---|---|
| | 0 notices of compiler operations, such as: current pass number and the number of errors encountered (if any), on lu 0. |
| MAP | Map option prints out a map of the compiled program object, giving relative address displacements of statements and data. |
| MEMLIMIT=xx | Memory allocation option defines a percentage of taskspace for Pascal system workspace so the remainder of the task partition can be available, for example, for get-storage requests from external CAL written routines. |
| OPTIMIZE | Optimization option generates optimized object code so object program space and execution time may be minimized. |
| RANGECHECK | Rangecheck option generates additional code within the compiled object program to check at run time for illegal out-of-range values used for subscripts, variant-tags, pointer values, and subrange value-parameters. |
| RELIANCE | The Reliance option causes the appropriate run time error, task pausing, and task termination mechanisms to be generated in in a RELIANCE environment, instead of the the compiled object code as is required in a Reliance environment, different from OS/32. |
| SUMMARY | Summary listing option prints out a listing on lu 2 of internal compiler statistics regarding table space, and file sizes used, for a particular compilation-unit and lists register usage and the number and kind of any optimizations that were performed in the object code. |

The compile-time option default states, and minimum abbreviated start or in-stream formats are listed in Table 1-3.

## TABLE 1-3 PASCAL COMPILER OPTION SETTINGS

| COMPILER OPTION | DEFAULT | PLACEMENT | ABBREVIATED START OPTION | | ABBREVIATED IN-STREAM OPTION FORMAT | |
|---|---|---|---|---|---|---|
| | | | ON | OFF | ON | OFF |
| ASSEMBLY | OFF | START or in-stream | AS | NAS | {$AS} | {$NAS} |
| BATCH | OFF | START or in-stream | BA | --- | {$BA} | ----- |
| BEND | NULL | In-stream only | | | {$BEND} | ----- |
| BOUNDSCHECK | ON | START or in-stream | BO | NBO | {$BO} | {$NBO} |
| CROSS | ON | START or in-stream | CR | NCR | {$CR} | {$NCR} |
| EJECT | NULL | In-stream only | | | {$EJ} | ----- |
| HEAPMARK | OFF | START or in-stream | HE | NHE | {$HE} | {$NHE} |
| INCLUDE | NULL | In-stream only | | | {$IN(fd)} | ----- |
| LIST | ON | START or in-stream | LI | NLI | {$LI} | {$NLI} |
| LOG | OFF | START only | LO | NLO | | |
| MAP | OFF | START or in-stream | MA | NMA | {$MA} | {$NMA} |
| MEMLIMIT | 100% | START or in-stream | ME=xx | | {$ME=xx} | |
| OPTIMIZE | OFF | START only | OP | NOP | | |
| RANGECHECK | ON | START or in-stream | RA | NRA | {$RA} | {$NRA} |
| RELIANCE | OFF | START or in-stream | RE | NRE | {$RE} | {$NRE} |
| SUMMARY | OFF | START or in-stream | SU | NSU | {$SU} | {$NSU} |

With the exception of LOG and OPTIMIZE, which are START-only options, the options can be specified in-stream by preceeding their names with a dollar sign and enclosing them within Pascal comment delimiters. Additionally, there are three options that can be specified in-stream only. They are:

- {$BEND}
- {$EJECT}
- {$INCLUDE (fd,arg1,arg2)}

Compiler option names indicate the positive condition of performing the function that they name. It is advantageous when invoking the options as start options to use an abbreviated mnemonic of the option-specifier: AS for turning on ASSEMBLY, or NCR for turning off the CROSS option, for example. The option-specifier names also can be abbreviated in-stream. In-stream option-specifier names must immediately be preceded by a dollar sign character ($) and must be enclosed within a pair of Pascal comment delimiters; either the pairing { and } or the pairing (* and *). Option-specifier mnemonics consist of at least the first two letters of their name and up to all the letters that make up their complete spelling. Preceding the option-specifier with the letter N negates those options which can be negated. Upper or lower case letters may be used to specify the options.

Pascal compile time options grouped by type are:

- Source options

    - BATCH
    - BEND
    - HEAPMARK
    - INCLUDE

- Listing options

    - ASSEMBLY
    - CROSS
    - LIST
    - MAP
    - SUMMARY

- Object options

    - BOUNDSCHECK
    - OPTIMIZE
    - RANGECHECK
    - RELIANCE

- Listing-format-control option

    - EJECT

- Memory allocation option

    - MEMLIMIT

- Pass Notification option

  - LOG

Options that can be specified as start options are:

- ASSEMBLY
- BATCH
- BOUNDSCHECK
- CROSS
- HEAPMARK
- LIST
- LOG
- MAP
- MEMLIMIT=xx
- OPTIMIZE
- RANGECHECK
- RELIANCE
- SUMMARY

### 1.5.3.1 ASSEMBLY Listing Option

The Assembly Listing option obtains a listing of the compiled-program machine instructions (generated for Pascal executable statements) disassembled into CAL/32 instructions and its EXTERN linkages, and its constants area code. The assembly listing is output to lu 6.

Because its default state is OFF, the user must specify either ASSEMBLY as a start-option, or {$ASSEMBLY} in-stream to obtain an assembly-listing.

Selecting ASSEMBLY as a start option in conjunction with the BATCH start option, causes the option state between individual jobs in the batch stream to be ON.

The assembly listing is printed prior to the map listing on lu 6. If lu 2 and lu 6 are assigned to the same file or device, the assembly listing and/or the map listing is printed on compiler PASS9 prior to those listings output to lu 2 on compiler PASS10.

### 1.5.3.2 BATCH Option

The Batch option allows the user to compile a series of Pascal programs and/or modules collectively located as a batch stream on a given file or device. This option is available both as a start option and as an in-stream option. All other start options specified in conjunction with the BATCH start option supersede the normal default state of those options, between compilation-units, i.e., creating a batch-start default state. That is, the compiler returns to the batch-start default state between individual compilations of programs and/or modules on the batch stream.

The format for specifying the Batch option in the start options is BATCH, minimally abbreviated BA. The format for specifying the Batch option in-stream is $BATCH enclosed in Pascal comment delimiters: {$BATCH}, (*$BATCH*). It should be on the first line of the source in-stream. Other options specified in conjunction with the "in-stream" {$BATCH} or (*$BATCH*) option apply only to the first job of the batch compilation. Only when the Batch option is selected as a compiler start option will its co-specified option settings determine a batch-start default state for all compilation-units of the batch.

Given the Batch start option, the compiler successively compiles the Pascal source batch stream from lu 1 (including any $INCLUDE options), until either an EOF/EOM is encountered or the {$BEND} in-stream option is encountered. The compiler succeedes the batch compilation with summarized information on the series of individual jobs processed during the batch run. Following all other listings output for individual compilation-units, a batch statistics listing is output to lu 2. Refer to Section 1.9.5 for information on the batch statistics listing.

### 1.5.3.3 BEND Option

Code the Batch-end option by specifing $BEND within the Pascal comment delimiters: {$BEND} or (*$BEND*). This option, if used, can be placed in-stream as the last line of source stream text of the batch-compilation to indicate an end of batch condition. However, it is not necessary for individual program or module compilations, nor is it necessary for source files with an EOF indicator. It is necessary on a sequential device like the card reader, which might not support an EOF condition.

The user is cautioned that the {$BEND} or (*$BEND*) option specifier must be placed after the end of a compilation-unit, which ends with the last line containing "END." as a program/module terminator. Placing the {$BEND} specifier in the middle of a compilation-unit may cause it to be ignored.

At the end of a batch compile, when the last compilation-unit is completed, the compiler automatically summarizes information relating to the series of completed jobs and prints out to lu 2 a listing of batch statistics. The batch statistics contains program/ module names, batch page number, batch line number, and end-of-task code for each compilation unit within the batch-stream. For each compilation unit that compiled without error, object size information is also given, i.e., object code size and literal constants code size. See Section 1.9.5 for details.

### 1.5.3.4 BOUNDSCHECK Option

The Bounds Check (BOUNDSCHECK) option, when selected by default or specification, causes the compiler to generate additional object code that checks for illegal values assigned to variables of the subrange type.
The user can specify whether or not the object program will incorporate this run time validity check.

A program compiled with the BOUNDSCHECK option on, and containing the source statement:

```
    VAR INDEX :  1..10;
```

contains additional object code that provides a run time error message if the execution of the program attempts the assignment:

```
    INDEX := expression;
```

where the evaluation of the expression results in a value less than 1, or greater than 10. The run time error generated contains the message VALUE RANGE ERROR.

The BOUNDSCHECK option can be selected as a start option or specified in-stream. The minimally abbreviated format for specifying BOUNDCHECK is to be ON as a start option is BO; to turn it off: NBO. It can be turned ON and/or OFF in-stream, so that only portions of the program will be affected. The in-stream formats are {$BOUNDSCHECK}, OR (*$BOUNDSCHECK*) to turn it on; and {$NBOUNDSCHECK} or (*$NBOUNDSCHECK*) to turn it off.

### 1.5.3.5 CROSS REFERENCE Listing Option

The Cross Reference Listing option is a program development aid that locates identifiers in the compiled program listing by source line numbers. The Cross Reference Listing option, when selected by default or specification, causes the compiler to generate a cross reference printout of all identifiers within the particular compiled program. It is output to lu 2 and follows the compiled program listing. It lists all user identifiers alphabetically, giving place of declaration or definition with a mnemonic code indicator for the general kind of identifier; i.e., constant, variable, type, etc. It lists where the user identifier was referenced within the compiled program listing and reflects where a change of value is programmed to occur. The place of declaration, definition, or reference is given by source line number making it easy to locate the identifier in the compiled program listing.

The Cross Reference option is ON by default for all compilations unless the user specifically turns it OFF. Restating that the

Cross Reference Listing is to be ON is specified by either starting the compiler with the start option CROSS (CR); or by invoking the option in-stream with either {$CROSS} or (*$CROSS*). Turning the Cross Reference Listing option OFF, is specified by either starting the compiler with the start option NCROSS (NCR); or by an in-stream specification of {$NCROSS}, or (*NCROSS*). When the Cross Reference Listing option is OFF, a cross reference listing is not generated.

### 1.5.3.6 EJECT Listing Format Control Option

The user may format the compiled-program listing with page ejections. The EJECT option encountered in the source stream causes a page ejection (form-feed) in the compiled program listing. This option can occur anywhere in-stream, but cannot be used as a compiler start option. This option only affects the listing produced when the LIST option is on.

To cause a page ejection encode:

```
    {$EJECT}

        or

    (*$EJECT*)
```

The in-stream format of the EJECT listing format control option is {$EJECT} or (*$EJECT*). The $EJECT must be enclosed within a pair of Pascal comment delimiters.

### 1.5.3.7 INCLUDE Option

The INCLUDE option enables the compiler to process and merge into the source stream an entirely separate file containing Pascal source. It can be specified as an in-stream option only and cannot be used as a start option specifier.

There is no limit on the number of files which may be included. One separate file will have its source included for each option specified. The included file may also contain other INCLUDE option specifiers.

To merge an additional Pascal source file, into the current source, use a Pascal option-specifier comment containing the $INCLUDE option. Specify the name of the additional file as an argument to the $INCLUDE option as a file-descriptor, within parentheses. Within either pair of acceptable Pascal comment-delimiters; the format of the INCLUDE option specifier is $INCLUDE followed by its arguments enclosed in parentheses. The arguments within the parentheses are the name of the fd, optionally followed by a comma, and another option specifier that can specify LIST, (LI), or NLIST (NLI), optionally followed by a comma and another option specifier, such as CROSS (CR) or NCROSS (NCR). The entire specification must be enclosed in a pair of Pascal comment delimiters and be self contained on one source line. The format of the $INCLUDE may be:

```
{ $INCLUDE ( fd ) }

     or

(* $INCLUDE ( fd ) *)
```

to include the source on file fd into the source stream being compiled. The listing and cross reference will or will not contain or pertain to that portion of the source from fd, depending on the option settings for the main stream.

Compiling under the default state with the LIST and CROSS listing options on, the listings would reflect the source of all $INCLUDE references that do not specify differently.

The user may disengage the LIST and CROSS reference listings for that portion of source which is included by the $INCLUDE option. The user may also turn the listings on, even when the main stream listing options are off. The complete format of the $INCLUDE option is:

Format:

```
{ $INCLUDE ( fd,arg1,arg2 ) }
```

Parameters:

| | |
|---|---|
| fd | is the file descriptor of a file containing Pascal source. |

Arg1 and/or arg2 are optional and can be any of the following:

| | |
|---|---|
| CROSS | CROSS turns the cross reference listing on |
| NCROSS | NCROSS turns the cross reference listing off |
| LIST | LIST turns the LIST listing option on |
| NLIST | NLIST turns the LIST listing option off |

Note that arg1 and arg2 contain no dollar sign.

Examples:

```
{ $INCLUDE ( M300:SEGMENT8.PAS ) }

(* $INCLUDE ( M300:EXCERPT.PAS,NCROSS,NLIST ) *)

{ $INCLUDE ( M400:PART34.PAS,NLI,CR ) }
```

### 1.5.3.8 LIST Listing Option

The LIST listing option specifies that a compiled program listing of the source compilation unit is to be produced. The compiled program listing is produced on lu 2, prior to any other listings being printed for that program on lu 2. However, if lu 2 and lu 6 were assigned to the same file, for example, and the MAP or ASSEMBLY options were on, those listings, as they are generated on Pass9 prior to Pass10 precede the compiled-program listing.

Because the default condition of the LIST option is on, the listing is normally produced without specifying any options. The LIST option is both a compiler start option and an in-stream option and can be turned on and off for different portions of the source program so that only the desired portions will be included in the compiled-program listing.

Specify NLIST as either a start option or an in-stream option to disengage the listing option. Specifying LIST or NLIST with the batch start option will engage or disengage the compiled-program listing for the entire batch compilation causing any in-stream $LIST or $NLIST settings to be ignored.

The format for setting the compiled program listing to on is LIST, minimally abbreviated LI; or to off, it is NLIST, abbreviated NLI; specified as a compiler start-option.

The format for setting the compiled program listing to on or off in the source stream, is $LIST, minimally abbreviated $LI; or $NLIST, minimally abbreviated $NLI; any one of them enclosed in a pair of Pascal comment delimiters. Examples of in-stream LIST option settings are:

```
{ $LIST }

(* $LI *)

{ $NLI }

(* $NLIST *)
```

or it can be embedded in a series of option settings:

```
{ $EJECT,$LIST,$MAP }
```

The compiled program listing is detailed in Section 1.9.1. Generally, the listing lists the user source program by line number and any diagnostic errors.

### 1.5.3.9 LOG Option

The LOG Option, specified as a compiler start option, enables the compiler to notify the user of its detailed PASS operations on lu 0. It is not available as an in-stream option and may only be selected as a compiler start option.

The LOG option enables the compiler to log the name of each currently operating Pass number and the number of any errors that occurred during the pass. For a correct compilation, with the LOG option set, the log might contain (in addition to normal log device information such as the compiler identification, end-of-task return code message, and other compiler-operations messages.

```
PASS1
PASS2
PASS3
PASS4
PASS5
PASS6
PASS7
PASS8
PASS9
PASS10
```

To reflect compilation errors encountered, the log might contain:

```
PASS1
PASS2
PASS3
PASS4
PASS5
145 ERRORS DETECTED IN PASS5
PASS10
```

and the compiler terminates with an abnormal termination return code after Pass 10 completes its listings. The number of errors in any pass is given in decimal.

The user can start the compiler with:

```
ST ,BA SU AS LO NLI NCR
```

setting the log option to monitor the compilation process through many unit compilations in a batch-stream, or start with:

```
ST ,LOG
```

to monitor the process of a single compilation with lu 0 previously assigned to the user terminal.

### 1.5.3.10 MAP Option

The MAP option, specified as a start option or within the source stream, causes the compiler to generate a Pascal map listing on lu 6. This listing is a map of the user object program, or object module, for each separately compiled compilation-unit, giving displacements which indicate the relative locations of the beginning of each source line containing an executable statement. The map also contains displacements (stack offsets) for the first datum on a given data-defining source line. See Section 1.9.7 for the map listing format of an individual Pascal compilation-unit object map. This is not the same kind of map as that generated by OS/32 LINK for a Link MAP, which displays the object locations of an entire established task. To relate the displacements given in a Pascal MAP, to object locations at run time, a Link MAP should be on hand to determine starting locations of any particular compilation-unit object.

The Pascal MAP option default state is off, and the user must specify MAP, abbreviated MA, as a start option to obtain a map listing. The in-stream format of the MAP option can be turned on and off throughout the source stream so only the portions of source code surrounded by {$MAP} and {$NMAP} are listed in the map listing, but it may not be advantageous to do so.

The format to specify the MAP listing option in-stream is:

```
{ $MAP }

    or

(* $MAP *)
```

or in conjunction with other option specifiers:

```
{ $LIST,$ASSEMBLY,$MAP,$SUMMARY }
```

Specifying either MAP or NMAP in conjunction with the batch start option determines the batch-start default state between jobs of the MAP option for each compilation-unit in the batch compile.

### 1.5.3.11 MEMLIMIT Memory Allocation Option

The Memory Allocation (MEMLIMIT) option may be specified as a compiler start-option or in-stream.

With this option, the user can specify that only a portion of task workspace be reserved for access by Pascal compiled-code for the heap and stack of the running Pascal program task. The remaining amount of workspace is available, for example, for any get storage SVCs issued within linked routines externally assembled from the program.

The default condition of the memory limit is 100 percent. That is, 100 percent of the task workspace available after linking and loading all object code, between CTOP and UTOP, remains in the task partition to be used by Pascal compiled-code for the heap and the stack, and other internal tablespace (Pascal SDA, RTL Scratchpad, etc.). Specifying 80 percent with the MEMLIMIT option, e.g., ME=80, reserves only 80 percent of the available workspace for access by Pascal compiled-code. Twenty percent of the workspace is reserved for other access, e.g., get-storage requests, from externally linked routines which are not Pascal compiled code. Pascal compiled-code does not enact access to the twenty percent set aside by the MEMLIMIT=80. See Figure 1-3 in Section 1.7.

The format of the Memory Allocation option is:

**Format:**

```
MEMLIMIT = xx
```

**Parameters:**

| | |
|---|---|
| xx | is a percentage, represented by a decimal number from 0 through 100. |

An illegal percentage, not in the range 0..100, causes an ILLEGAL OPTIONS compiler-operations message on the log device.

### 1.5.3.12 OPTIMIZE Option

The Optimization (OPTIMIZE) option instructs the compiler to attempt to make the user object code more efficient by analyzing both machine independent and specifically machine dependent code and recompiling them into optimized code on Pass 8. This option is best used when the user source program reaches a compile time error-free state. The specific optimizations performed by the compiler are detailed in Section 1.5.4. Specify the SUMMARY Listing option for an accounting of the optimizations performed during compilation (see Section 1.5.3.15).

The optimization option may be specified as a compiler start-option only. It cannot be turned on and off for only portions of source code in the source stream; nor can it be specified in-stream at all.

OPTIMIZE is the full format to specify that this additional optimization is to take place. It is minimally abbreviated OP; and although the default condition is OFF, the user can specify no optimization with NOPTIMIZE, minimally abbreviated NOP.

### 1.5.3.13 RANGECHECK Option

The Range Check Option, selected by default or specification, causes the compiler to generate additional object code that checks for illegal or out of range values being used for subscripts, variant tags, pointer values, and subrange value parameters. The user can specify whether or not his object program will incorporate these run time validity checks.

To compile a program with the RANGECHECK option on, if the program contains type definitions and variable declarations such as:

```
    TYPE COLOR = (RED,BLUE,BLACK);
    VAR A: ARRAY [1..99,COLOR] OF INTEGER;
```

then that program contains additional object code rangechecking that causes a run time error, if program execution attempts the array reference:

```
    A [INDEX1,INDEX2]
```

whenever the value of INDEX1 is not within the range 1 through 99, or the value of INDEX2 is not within the values RED, BLUE, or BLACK. This run time error contains the message INDEX RANGE ERROR. Other validity checks on variant tags, pointer values, or subrange value parameters could produce the run time errors with the messages VARIANT TAG ERROR, POINTER ERROR, or PARAM RANGE ERROR, respectively.

The RANGECHECK option can be selected as a start option or specified in-stream. As it is on by default, the user must specify NRANGECHECK (NRA) as a start-option or specify {$NRANGECHECK} or (*$NRANGECHECK*) in-stream, to turn it off.

### 1.5.3.14 RELIANCE Interface Option

Specification of the Reliance Interface Option (RELIANCE) is required as a compiler option for any compilations of compilation units which are intended to operate in a Reliance environment. It must not be specified for compilations not intended to operate in a Reliance environment. It is off by default.

The Reliance option, when specified, causes the compiled object program to contain run-time error handling, task pausing and task termination mechanisms compatible with a Reliance environment.

The Reliance option may be specified as either a start-option or in-stream option. The format for specifying this option as a start-option is RELIANCE, abbreviated RE. The format for specifying this option in-stream is {$RELIANCE}, abbreviated {$RE}; and must be on the first line of the source of the compilation unit.

Users preparing Pascal programs to run in a Reliance environment must also refer to Appendix M for the Pascal-Reliance information.

### 1.5.3.15 SUMMARY Listing Option

The SUMMARY Listing Option, specified as a start option or within the source stream, causes the compiler to generate a summary listing on lu 2. The summary listing contains information on internal compiler statistics that were accumulated during user program compilation. A paragraph is printed out for each of the first eight passes, giving file size information in use on lu 3 and lu 4 useful to users repeatedly compiling a large near-stable program who wish to pre-allocate lu 3 and lu 4 as contiguous scratch files with adequate space.

The summary listing also contains information on which optimizations were performed and how many times. When the user selects the OPTIMIZE option, the Pass 8 optimizations are recorded. The amount of memory space saved is also reflected. The machine independent optimizations performed during Pass 6, are given in the SUMMARY, Pass 6 paragraph. The machine dependent optimizations performed during Pass 8 are given in the SUMMARY, Pass 8 paragraph. The optimizations are listed by their abbreviated names and the number of times they were effected (refer to Section 1.5.5).

The default state of this option is off. To obtain a Summary Listing the user must specify SUMMARY, minimally abbreviated SU, as a compiler start option or use its in-stream format, which is:

```
    { $SUMMARY }

        or

    (* $SUMMARY *)
```

or in conjunction with other in-stream option specifiers:

```
    { $MAP,$SUMMARY,$LIST }
```

### 1.5.3.16 HEAPMARK Option

Specification of the HEAPMARK option is required only for those compilation units which contain references to the predefined procedures MARK and RELEASE. This option need not be specified if the compilation unit contains no such references. It is OFF by default.

In the default state, which is off, the compiler will not recognize MARK and RELEASE as predefined procedure identifiers.

The Heapmark option may be specified as either a start-option or in-stream option. The format for specifying this option as a start-option is HEAPMARK, abbreviated HE. The format for specifying this option in-stream is {SHEAPMARK}, abbreviated {SHE}. It should be on the first line of the source of the compilation unit, or prior to the first reference to either MARK or RELEASE.

### 1.5.4 Error Handling

The Pascal compiler provides extensive error diagnostics of the user source and documents any detected errors in the compiled program listing, which are discussed in Section 1.5.4.1 below.

The compiler also provides its own compiler operations messages to the user, as detailed in Section 1.6.1 and especially provides a warning message (when possible) prior to malfunctioning, see Section 1.5.4.2 below.

The compiler generates in the program object code certain runtime error checking depending upon its start-options. All of the user task run time error messages are presented in Section 1.5.4.3 and those run-time error checks, controlled by compiler-options are discussed.

Appendix G contains a complete list of PASCAL Diagnostic Messages, Compiler Operations Messages, and Run Time Error Messages.

### 1.5.4.1 Diagnostic Errors

User-written coding errors are detected by the compiler while processing the Pascal source and given diagnostic error messages in the compiled-program listing. The descriptive compiler-produced error messages contain the source line number of the offending construct and a 4-digit error code that indicates the pass number in which the error was detected and the error number in that pass and a brief text message describing the error.

Diagnostic error messages are presented in the compiled-program listing below the line of code in which the error was detected.

They are also presented collectively at the end of compilation as part of the program statistics listing. The format of a compile time diagnostic error message is:

Format:

```
****** LINE n, ERROR xyyy: message . . . . .
```

Where:

| | |
|---|---|
| n | is the offending source line number, |
| x | is the pass number that detected the error, |
| yyy | is the error code, and |
| message | is the error text, describing the error. |

The possible codes xyyy and messages are listed in Appendix G.

Some examples of the "messages" of the diagnostics detected are:

```
BAD NUMBER FORMAT: DIGIT REQUIRED
CASE STATEMENT SYNTAX
IDENTIFIER DECLARED TWICE
EXTERNAL ROUTINE CANNOT HAVE FORMAL ROUTINE PARAMETERS
OPERAND TYPE CONFLICT
MOD RELATIVE TO 0 OR NEGATIVE NUMBER
```

Refer to Appendix G for a complete list of the diagnostic errors displayed in listings.

### 1.5.4.2 Compiler Failure Errors

The Pascal compiler is subject to the same internal consistency checking as any other Pascal program. When the compiler is running as a task, a run time error can occur as compilations are performed. However, the normal run time error message is insufficient since the compiler is a multi-module program. If the compiler should malfunction, the user is given more information to write a software change request (SCR) and to identify where in compiling this program the compiler malfunctioned.

There is a run time error mechanism especially for the compiler. If the compiler code fails, or cannot continue due to insufficient accessible memory, the format of the run time error message which is sent to the log device is:

Format:

```
PASS n LINE xxxxx, ADDR yyyyyy message....
COMPILING LINE zzzzz OF PROGRAM name
```

Where:

| | |
|---|---|
| n | is the number of the compiler pass during which the error occurred. |
| xxxxx | is the source line number in the compiler pass where the error occurred. |
| yyyyyy | is the object address within the compiler where the error occurred. |
| message | is the textual run time error message, as follows:<br>- INDEX RANGE ERROR<br>- PARAM RANGE ERROR<br>- VALUE RANGE ERROR<br>- CASE LABEL ERROR<br>- TRUNC RANGE ERROR<br>- VARIANT TAG ERROR<br>- POINTER ERROR<br>- STACK OVERFLOW<br>- HEAP OVERFLOW |
| zzzzz | is the source line number in the user program, currently being compiled. |
| name | is the name of the user program, currently being compiled. |

The user can determine where in his program that the compiler faulted by examining the program's source line zzzzz and possibly change that source line as an avoidance procedure to continue.

If the HEAP or STACK OVERFLOW message occurs after the compiler is started or a CSS is invoked, reload the compiler with more task memory space (a greater memory segment-size-increment). Other abnormal unresolvable error messages occurring during compile time should be reported on an SCR (refer to Appendix B).

### 1.5.4.3 User Task Run Time Errors

Executing Pascal compiled-code enacts certain self-contained program logic and run-time data validation checks to detect exceptional circumstances that make it illogical or impossible for the program to continue executing. Subsequent to these run time error messages, the task is paused, and upon an attempt to continue with the OS CONTINUE command, the user task is terminated with END-OF-TASK, under OS/32. These run-time error messages may occur while executing Pascal compiled-code as follows, and are of the form:

Format:

```
LINE xxxxx, ADDR yyyyyy message...
```

Where:

| | |
|---|---|
| xxxxx | is indicating (when possible) the user's errant Pascal source line number in which the error was detectable, by the line's Pascal compiled-code; or xxxxx is zero, when the error was detectable by an RTL/support routine not having access to the user's line number. |
| yyyyyy | is the machine address in the compiled object code, of either the interupting ERR compiler generated instruction, near the detected error; or if line xxxxx is zero, yyyyy is the machine address in code which called the error detecting RTL routine; and |
| "message" | is one of the following:<br>INDEX RANGE ERROR<br>PARAM RANGE ERROR<br>VALUE RANGE ERROR<br>CASE LABEL ERROR<br>TRUNC RANGE ERROR<br>VARIANT TAG ERROR<br>POINTER ERROR<br>STACK OVERFLOW<br>HEAP OVERFLOW |

Each of these "messages" is described in detail in Appendix G under RUN TIME ERRORS. Some of them are generated by the compiler under certain compiler options. BOUNDSCHECK option, controls whether or not the VALUE RANGE ERROR checks will be generated in compiled-code for subrange-type range errors, and can be turned off. The RANGECHECK option, controls certain checks for illegal out-of-range run-time values for subscripts, variant-tags, pointer values, and subrange value parameters giving the messages, INDEX RANGE ERROR, VARIANT TAG ERROR, POINTER ERROR, and PARAM RANGE ERROR. The RANGECHECK option can be turned off.

Run time errors occuring during execution of user Pascal tasks are logged (via an SVC 2,log-message) to the console (user console in an MTM environment, system console in a stand alone OS/32 MT environment, or system journal in a RELIANCE environment).

Some errors may allow continuing execution, after pausing for correction by operator intervention, others may require reloading with more or differently arranged memory allocations, relinking the task, or reprogramming and recompile, to correct the problem.

NOT ENOUGH SPACE TO RUN PASCAL

> This message occurs immediately after starting a user Pascal task, when the memory allocations available to the task are not even large enough for the basic internal workspace needed for the Pascal SDA, FORTRAN SCA, or the RTL Scratch Pad area. The user task is then terminated.
>
> Reload or relink with more memory, and restart; or if MEMLIMIT was used, check the effect of the MEMLIMIT memory allocation option. If upon restarting, this message does not appear and the STACK OVERFLOW immediately does, or it occurs sometime thereafter, enough memory was added/arranged to accomodate the basic internal tables, but not enough for this particular user program's Global variables or stack data to be run. Reload or relink with greater memory, and restart.

When executing Pascal named file I/O (text file or non-text file), with RESET, REWRITE, READ, READLN, WRITE, WRITELN statements; the following runtime error messages may occur. Note that when the logical unit number, nnn, is an external Pascal named file; the position of the file-name in the PROGRAM header file-name-list determined its associated lu number. If the lu number, nnn, cannot possibly be an external file in the program concerned, an internal file-variable is of concern.

NO LU AVAILABLE TO ASSIGN INTERNAL FILE

READ ATTEMPTED ON A NON-RESET FILE, LU= nnn

READ ATTEMPTED PAST END-OF-FILE, LU= nnn

WRITE ATTEMPTED TO A NON-REWRITTEN FILE, LU= nnn

INVALID CHARACTER IN NUMERIC INPUT, LU= nnn

Additional system file error conditions may be detected while performing Pascal named file I/O, as follows.

```
I/O ERROR xxyy, LU= nnn
```

where xxyy is the non-zero hexidecimal OS/32 SVC 1 Error Status encountered on logical unit number, nnn, by an SVC 1 I/O being attempted.

After this message occurs the program is paused. Check the OS/32 SVC 1 status halfword as defined in the Operating System manual, and the file/device assigned to lu nnn; to determine the source of trouble.

In an OS/32 environment, the task is paused to allow operator intervention to correct the problem, and enter the OS/32 CONTINUE command to retry the SVC 1, and proceed.

```
ERROR IN INITIALIZING EXTERNAL FILE FOR READ/WRITE
SVC 7 ERROR, LU= nnn; FN= xxxxxxxxxxx, STATUS= yyyyyyyyyyyyyyyy
```

```
ERROR IN ASSIGNING INTERNAL FILE
SVC 7 ERROR, LU= nnn; FN= xxxxxxxxxxx, STATUS= yyyyyyyyyyyyyyyy
```

```
ERROR IN ATTEMPTING TO CLOSE INTERNAL FILE
SVC 7 ERROR, LU= nnn; FN= xxxxxxxxxxx, STATUS= yyyyyyyyyyyyyyyy
```

Each of the above messages are detailed in Appendix G under RUN TIME ERRORS.

The qualifier SVC 7 ERROR message of the above three messages identifies the logical unit number concerned, the function code attempted, and the error status encountered; and is of the form:

```
SVC 7 ERROR, LU= nnn; FN= xxxxxxxxxxx, STATUS= yyyyyyyyyyyyyyyy
```

where nnn is the logical unit upon which the SVC 7 was attempted, and the FN= xxxxxxxxxxx, is the function code attempted:

```
FN= ASSIGN
FN= CLOSE
FN= ALLOCATE
FN= RENAME
FN= REPROTECT
FN= DELETE
FN= CHANGE PRIV
FN= CHECK POINT
FN= FETCH ATTRB
```

and the STATUS= status-message encountered as an error may be:

```
STATUS= ILLEGAL FUNCTION
STATUS= ILLEGAL LU
STATUS= VOLUME
STATUS= NAME ERROR
STATUS= SIZE ERROR
STATUS= PROTECT ERROR
STATUS= PRIVILEGE ERROR
STATUS= BUFFER ERROR
STATUS= LU NOT ASSIGNED
STATUS= TYPE (DEVICE)
STATUS= FD SYNTAX ERROR
STATUS= SVC 6 DEVICE
STATUS= FILE IS /S OR /G
STATUS= I/O ERROR
```

### 1.5.5 Compiler Optimizations

The object code of a compiled program may be optimized to reduce memory space requirements and increase execution speed. Object code optimizations performed by the Pascal compiler are classified as machine independent and machine dependent optimizations.

The machine independent optimizations take place when the compiler recognizes where program logic level code efficiences are possible. The machine dependent optimizations take place when the compiler recognizes where machine architecture and instruction repertoire code efficiences are possible.

The compiler performs simple machine independent optimizations and numerous machine dependent optimizations. Some machine independent optimizations are performed in Pass 6; the remainder, as well as all machine dependent optimizations, are performed in Pass 8. The Pass 6 optimizations are always performed. Some of the Pass 8 optimizations, which might be more time consuming at compile-time, are under user control with the OPTIMIZE compiler start option (refer to Section 1.5.3.12).

Selecting the OPTIMIZE option as a compiler start option causes all possible pass 8 optimizations on the compiled object program to be performed. Starting the compiler without specifying the OPTIMIZE option is the default condition so that certain Pass 8 optimizations are not performed.

The user can identify which optimizations were performed on a particular compilation by also selecting the Summary listing option. Information on both Pass 6 and Pass 8 optimizations regarding which optimizations were performed and how many times are listed in the Summary listing. The amount of memory space required by the object code before and after optimization is given. See Section 1.5.3.15 for the Summary listing option, and Section 1.9.2 for the Summary listing format.

Pascal compiler optimizations are explained in the following paragraphs. Their abbreviated representation reflected in the Summary listing, follows each optimization name in parentheses, below.

### 1.5.5.1 Pass 6 Optimizations

Certain rudimentary optimizations are performed directly by the Pascal compiler at the program logic level, regardless of the machine architecture for the compiled object. In the following description, A is an array, B is a Boolean variable, e indicates a general expression, and V is a variable. When a constant is indicated, it can be a literal constant or a symbolic constant.

Constant computation (ARITHOPS) optimizes an expression consisting of literal or symbolic constants. They are evaluated at compile time and replaced with the result. For example, 2+3 is reduced to and replaced with 5.

Computations with zero (CMPRS) optimizes expressions involving operations with zero that are replaced with the value of expression. V+0 is replaced with V; and V*0 is replaced with 0.

Boolean constant computation (BOOLOPS) optimizes Boolean expressions involving constants replaced with their results. 5>7 is replaced with FALSE; and B OR TRUE is replaced with TRUE.

Index constant computation (INDEXEXPR) optimizes array references that involve constant indices and have their addresses computed at compile time.

Standard function evaluations (STDFUNCS) optimizes calls to standard functions that involve only type conversion that are eliminated. ORD(V) is reduced to a reference to the current value of V. Calls on standard functions with constant operands are replaced with the evaluated result.

Strength reductions (MULDIVS), where possible, replace multiply operations on integers with shifts.

Set Operations (SETOPS) involving complicated set constructors are reduced to simpler form, in preparation for later optimizations (see BUILDSETS). Given the variable ALPHANUMS, which is a variable set-type, SET OF CHAR, the statement:

```
    ALPHANUMS := ['0'..'9','A'..'Z']
```

is reduced to the assignment of a single precomputed constant to the set variable, ALPHANUMS.

For Statement Analysis (FORS) inspects FOR statements with constant initial and final values. Statements that will not execute because of the relation of the initial and final values are eliminated.

Conditional jumps (CONDJUMPS) occurs when in an IF or CASE statement, if the expression which selects an alternative is constant, then the conditional jump is replaced by an unconditional jump.

Index checks (INDXCHKS) occurs when an array index is constant, then the check of its value against the bounds of the index type is performed at compilation time.

Negations and unary minus (NEGNOTS) cancels double negatives, and double unary minus signs.

Range checks (RANGECHKS) is performed when a constant is assigned to a variable of subrange type, the check of its value against the bounds of the type.

MOD Operation (MODS) takes the expression I MOD c, where c is an integer constant power of two, and replaces it by a mask operation on I.

Set constructors (BUILDSETS) combines, in a set constructor, constant members and subranges.

Immediate operations (IMMEDS) changes assignments of the form A := A op e to immediate operations, if op is an integer addition, integer subtraction, Boolean "and", Boolean "or", set addition of a constant set, or set subtraction of a constant set.

Bit immediate operations (BITIMMEDS) changes assignments of the form A := A op e to immediate operations, if "op" is set addition or subtraction and the expression "e" is a set containing one element.

### 1.5.5.2 Pass 8 Optimizations

Pass 8 optimizations are performed on the intermediate, compiler produced, symbolic assembly language representing the source program being compiled. Most of these optimizations are dependent on compiler knowledge of the machine architecture and machine instruction repertoire. Those optimizations are commonly referred to as machine dependent optimizations. The user specifies to the compiler whether or not such optimizations are to be performed by the OPTIMIZE option. These optimizations may reduce the necessary memory space for the compiled program and may increase the execution speed. Some optimizations, such as COMBINE-LABELS, do not, by themselves improve code-efficiency. They may, however, cause other optimizations to be recognized.

Cross-Link Optimization (CROSS_LINK) examines two streams of code branching to a common label. Code duplicated in one stream is deleted and replaced with a branch to the appropriate entry point in the second stream.

Combine-Labels Optimization (COMBINE_LABELS) reduces multiple labels on a statement to a single label. All references to the deleted labels are adjusted.

Opt-Lab Optimization (OPT_LAB) removes unreferenced labels such as those generated from the combine-labels optimization.

Branch-Chain Optimizations (BRANCH_CHAIN) changes branches to yet another branch with a direct branch to the appropriate label. Branch chains may be reintroduced in resulting chains with short-form branch instructions.

Check-Cond1 Optimization (CHECK_COND1) deletes a conditional branch to the next instruction.

Check-Cond2 Optimization (CHECK_COND2) converts a conditional branch around an unconditional branch into a conditional branch with the opposite condition and deletes the unconditional branch.

Check-Cond3 Optimization (CHECK_COND3) deletes a conditional branch followed by an unconditional branch to the same label.

Opt-Branch1 Optimization (OPT_BRANCH1) deletes the unreachable code between an unconditional branch and the next label. An opt-branch1 optimization usually occurs for each cross-link optimization.

Opt-Branch2 Optimization (OPT_BRANCH2) deletes unconditional branches to the next instruction.

Opt-Branch3 Optimization (OPT_BRANCH3) converts a conditional branch, whose target is a conditional branch with the opposite condition, into a conditional branch to the instruction following the originally targeted conditional branch.

Opt-Codelab Optimization (OPT_CODELAB) converts certain nonbranch instructions that reference labels on branch instructions to reference the target of the branch.

Remove-No-Condition Optimization (REMOVE_NO_COND) deletes certain instructions whose purpose is to set the hardware condition code, when that condition code setting is altered prior to its use. This includes all compares, the load register instructions, and the AI, SI, CI, CLI, OI, and XI shift instructions with a constant operand of zero. This optimization generally occurs on a load register instruction following the After-RX or After-LI optimizations.

Null-Op1 Optimization (NULL_OP1) deletes certain instructions whose purpose is to set the hardware condition code, and is reached by a branch from a sequence that sets the condition code with the same value. Refer to the remove-no-condition optimization.

Null-Op2 Optimization (NULL_OP2) deletes certain instructions that set the hardware condition code to the same value as the previous instruction. Refer to the remove-no-condition optimization.

After-RR Optimization (AFTER_RR) reduces a pair of instructions such as: LR r1,r2 and LR r2,r1 to LR r1,r1. The resulting instruction, which merely sets the hardware condition code, can be inspected for removal by other optimizations.

NOTE

> The following three optimizations reduce an instruction that references a value or an address in memory to an RR instruction, if that value is available in another register.

After-RX Optimization (AFTER_RX) takes an RX instruction referencing an operand that was previously the source/target of a load/store instruction and changes the RX instruction to an RR instruction.

After-LI Optimization (AFTER_LI) takes an RI instruction referencing an operand previously loaded by an LI instruction and changes the RI instruction to an RR instruction.

After-LA Optimization (AFTER_LA) converts an LA instruction following an LA instruction for the same location into an LR instruction.

Before-Load Optimization (BEFORE_LOAD) deletes a load instruction, which precedes another load into the same register, without an intervening use of that register.

Before-Mult Optimization (BEFORE_MULT) deletes a load instruction, which precedes a load multiple instruction affecting the register of the former load instruction, without an intervening use of that register.

Opt-Br Optimization (OPT_BR) deletes the unreachable code between an unconditional BR and the next label.

NOTE

> The following two optimizations effectively change long branches to a label into short branches to another branch to the target label. These optimizations, together, can effect a significant reduction in the code of a CASE statement.

Try-Short-Branch Optimization (TRY_SHORT_BR) flags the number of intraprocedural branches that required the use of the 4-byte RX2 format rather than the 2-byte short format.

Chain-Branch Optimization (CHAIN_BRANCH) changes a long branch-to-label into a short branch to another branch-to-target-label.

Opt-Immediates Optimization (OPT_IMMEDS) optimizes immediate instructions by changing a Load Immediate to an Add Immediate wherever this change is valid and allows the instruction to be shortened.

## 1.6 PASCAL COMPILER OPERATING INSTRUCTIONS

The Pascal compiler is available as two established tasks to be run under Perkin-Elmer OS/32 R05.2 or higher. The user can select either version of the compiler task on file PASCAL.TSK, which is a root-segment and 10 overlays; or the compiler on file PASCALR.TSK, which is the resident version of the compiler.

The overlaid version requires less memory space and compiles slightly more slowly because of the overhead of overlay loading during compile time, although overlay loading is automatic and user transparent. This version is established with a sharable root segment, but the overlay space must be provided for in user task space. The resident version, with the passes and root segment established as one pure task, is sharable by many users. Use the larger resident version where many users will be simultaneously executing the compiler at one installation. Regardless of the version of the compiler task used, Pascal compiler operation is the same. To compile:

Either:

- load the compiler task;
- make the required file/device lu assignments;
- issue the operating system START command and options;

Or:

- invoke one of the CSS procedures, specifying arguments regarding fd names and any options, etc.

### 1.6.1 Executing the Compiler

The Pascal compiler requires several lu assignments for I/O transactions during operation. The user must assure, either by direct allocation and/or assignment commands or by CSS argument specifications, that all required or desired lu unit assignments are made for the compiler subsequent to loading it and prior to executing it. Refer to Table 1-1. Briefly, the compiler uses the following logical units:

- logical unit 0 is a log device or file
- logical unit 1 is a source input device or file
- logical unit 2 is a listing device or file
- logical unit 3 is a scratch file
- logical unit 4 is a scratch file
- logical unit 5 is a scratch file
- logical unit 6 is a map or assembly listing device or file
- logical unit 7 is an object code output device or file
- logical unit 8 is an additional source input file (for the $INCLUDE option).

The compiler controls assignments to lu 5 and 8. All other assignments must be made by the user either directly after loading the compiler, or through CSS procedures.

For example, OS/32 commands to load the compiler, assign lu's, and start the compiler are:

```
*LO PASCAL.TSK,100           load compiler with memory increment

*AS 0,CON:                   assign log device to console
*AS 1,USERPROG.PAS,SRO       assign source lu to existing file
*AS 2,PR:                    assign list to printer {or file}
*TE 3,IN,512                 assign temporary scratch file
*TE 4,IN,512                 assign temporary scratch file
*ALLO USERPROG.ASM,IN,132    allocate a file for ASSEMBLY/MAP
*AS 6,USERPROG.ASM,EWO       assign lu 6 to a file {or device}
*ALLO USERPROG.OBJ,IN        allocate a file to receive object
*AS 7,USERPROG.OBJ           assign lu 6 to file {or device}
*ST                          start compiler with default options
```

An option is a directive to the compiler to perform (or not perform) an operation during compile time. Each time the compiler is started, the user is implicitly (because of predefined default option states) selecting a compilation's options. The user may also explicitly select which options the compiler will perform, by specification, in the START command or through the CSS's.

The Pascal compiler minimally compiles Pascal source of one compilation unit into object, printing the compiled program listing and a cross reference listing and generating certain (bounds/range) validity checks in the object when it executes (without any user-specified options). The user can specify certain options either as start options or in the source stream and tailor the compilation process as required. Refer to Table 1-2 for a summary of available options. Refer to Section 1.5.3 for the definition of the Pascal compiler options.

The default states of the options are listed in Table 1-3.

Starting the compiler without specifying options is the same as entering the following OS/32 START command:

Format:

```
ST ,NAS,CR,LI,NMA,ME=100,NOP,RA,NSU
```

### 1.6.2 Compiler Operations Messages

The following compiler operations messages are listed to logical unit 0, the compiler's log device/file.

PERKIN-ELMER PASCAL Rnn-uu

> The Pascal Compiler logs this message to identify itself and notify the user that compilation has started; where Rnn identifies the Pascal compiler revision level and uu identifies the update level.

INVALID OPTION(S)

> Compiler was started with invalid Pascal compiler start-option(s), and compilation cannot begin. The compiler aborts with END-OF-TASK CODE 1. Correct the options given to the compiler prior to restarting.

PASSn

> When the LOG compiler start-option has been selected, this message is listed to lu 0 for each pass of compiler operations beginning, where n is from 1 to 10. This message does not appear, if the LOG option was not specified.

nn ERRORS DETECTED IN PASSn

> When the LOG compiler start-option has been selected, this message is additionally listed when a number (nn) of errors are detected in Pass n, where n is the pass number from 1 to 10. This message does not appear, if the LOG option was not specified; or pass n detects no errors.

UNABLE TO OPEN FILE filename.ext
EITHER FILE DOES NOT EXIST OR IS INACCESSIBLE

Compiler is attempting to perform a user-specified in-stream $INCLUDE (filename.ext) option, but cannot assign the file; and aborts the compiled unit with an END-OF-TASK CODE 5. Check the $INCLUDE specification, or why the intended filename.ext appears not to exist or is inaccessible. If under BATCH, a batch EOT 4 occurs.

INCLUDED FILE ATTEMPTED FROM NON RANDOM I/O DEVICE

A user-specified $INCLUDE in-stream option specified an argument file descriptor which is a non-random I/O device. Unit compilation aborts with EOT 5; batch concludes with EOT 4.

COMPILATION ERRORS

Compile-time diagnostic errors were detected in the user Pascal program or module source code just compiled, the diagnostic error messages are displayed in the listing on the lu 2 list device/file, and the compiler terminates a single compilation-unit with an END OF TASK CODE= 2 or 3.

The diagnostic error messages are displayed in the compiled-program listing on the lu 2 list device/file, if LIST is on, and/or listed in a group in the program statistics at the end of the listing. If LIST is off, the group of diagnostic errors message are still available on lu 2.

In a batch job with COMPILATION ERRORS, the end-of-task code 2 or 3 is listed in the batch statistics listing for each compilation-unit containing errors, and the entire batch terminates with an END OF TASK CODE= 4.

Check the listings; correct the source; and recompile.

xxxxxxxx-END OF TASK CODE= n

where xxxxxxxx is the identifier name of the system user/compiler-task ending, and n is the END OF TASK CODE. This system message occurs under OS/32 as the compiler terminates with an SVC 3, End of Task; giving an EOT code of n = 0,1,2,3,4, or 5 as detailed in Appendix F. An EOT of zero indicates a correct compile, a non-zero EOT indicates a problem. See Section 1.6.3 below.

The following message may occur when difficulty in performing I/O arises for the compiler.

I/O ERROR xxyy, LU= nnn

where xxyy is the non-zero hexidecimal OS/32 SVC 1 Error Status encountered on logical unit number, nnn. Examine the xxyy status, and lu 0 to 8 to determine the cause of I/O trouble, e.g., if lu = 1,5, or 8 the compiler cannot read source input. In an OS/32 environment, if operator intervention can correct the problem and continues with the OS CONTINUE command, the SVC 1 will be retried. Users not familiar with the xxyy SVC 1 Status halfword definitions may refer to the appropriate operating system manual on SVC 1.

As the compiler is executing as Pascal compiled-code itself, it too may encounter certain run time errors, described below or in Appendix G under RUN-TIME ERRORS. If an unrecoverable error occurs in its own code the compiler generates the message:

PASS n LINE xxxxx, ADDR yyyyyy message....
COMPILING LINE zzzzz OF PROGRAM name....

The compiler issues this message prior to pausing, when on pass n, at its own line number xxxxx, and object address yyyyyy, is encountering a run time error; while compiling user source at its source line number zzzzz of user program-name or module-name "name....". If the message is STACK OVERFLOW or HEAP OVERFLOW, it is possible that not enough memory has been allocated for the compiler to compile this program; and the user should reload with greater memory, and attempt to recompile. If the compiler ascertains, even prior to compiling lines of the user's source, it has not enough memory; the STACK/HEAP OVERFLOW message may occur as a Pascal compiled-code run time error message.

If the message is, during compile time:

INDEX RANGE ERROR
PARAM RANGE ERROR
VALUE RANGE ERROR
CASE LABEL ERROR
TRUNC RANGE ERROR
VARIANT TAG ERROR
POINTER ERROR

the compiler may be malfunctioning. Please report compiler malfunctions via an SCR as instructed in Appendix B.

### 1.6.3 Pascal Compiler Return Codes on Termination

The Pascal compiler task has both normal and abnormal termination conditions defined to register a successful or abortive attempt to compile the user program. Refer to Table 1-4 for a summary of termination codes, and paragraph 1.5.4.2 for instructions to resolve any malfunctioning compiler terminations.

TABLE 1-4 PASCAL COMPILER TERMINATION END OF TASK RETURN CODES

| CODE | MEANING |
|---|---|
| 0 | Normal termination. No compilation errors detected in either a single compilation or all of the compilations within a batch stream. |
| 1 | Illegal start options |
| 2 | Error detected in Passes 1 through 5 (syntax) |
| 3 | Error detected in Passes 6 through 9 (semantics) |
| 4 | Any error in processing one or more compilation units within a batch stream. |
| 5 | Error in locating or reading a {$INCLUDE (fd)} source file, fd. Abort. |

NOTE

Return codes 2 and 3 are applicable only when processing a single compilation unit; i.e., nonbatch operation. Return code 4 is applicable only when operating in batch mode. In batch mode, the appropriate return codes 0, 2, 3, or 5 for each individual compilation unit are found in the batch statistics listing.

### 1.6.4 Using the CSS Procedures

The Pascal product contains the following CSS procedures, for use under OS/32 MT under MTM, from a user terminal:

- PASCAL.CSS to compile and link a Pascal program
- PASCOMP.CSS to compile a Pascal source program
- PASLINK.CSS to link a compiled Pascal program

The formats to invoke the CSS procedures are:

**Format:**

```
PASCAL sourcename,list,options,assemlist,memincr,worksize

PASCOMP sourcename,list,options,assemlist,memincr

PASLINK objectname,list,worksize
```

**Arguments:**

| | |
|---|---|
| sourcename | is the name of the file containing the Pascal source program to be compiled. An extension of .PAS is assumed; i.e., no extension should be given as part of the CSS argument; but the source file must exist with an extension of ".PAS". |
| list | is the name of the file or device to which the compiled-program listing, cross reference, and linkage Link map are to be written. This argument is optional and defaults to PR:. |
| options | is a list of one or more compiler options separated by spaces. Refer to Section 1.5.2 to select the compiler options. Options do not need to be specified. |
| assemlist | is a file or device to which the assembly listing or map listing will be written if selected by appropriate options. This argument is optional and defaults to PR:. |
| memincr | is the memory segment size increment to be used to perform compilation, i.e, the additional memory space the compiler can use for stack and heap space to perform the |

|  |  |
|---|---|
|  | compilation. This argument is optional and defaults to 64KB. |
| worksize | is a 1- to 6-digit hexadecimal number indicating the number of bytes of main storage to be added to the end of the task objects for its maximum workspace. This argument is optional and defaults to that value provided by Link (256KB or X'40000'bytes). Pascal R01 CSS's default minimum workspace to approximately 1 1/2KB or X'624' bytes. |
| objectname | is the file containing the object of the main compiled program to be linked. An extension of .OBJ is assumed; i.e., no extension should be supplied as part of the CSS argument; but the file must exist with an extension of .OBJ for PASLINK. |

Any non-specified optional argument must have its position reserved by a comma if other arguments are to follow.

The Pascal product also contains three similar CSS procedures for use from the OS/32 system console:

- PASCAL.CON
- PASCOMP.CON
- PASLINK.CON

The functions of these ".CON" CSS procedures are identical to those CSS procedures previously described and their arguments are identically required. Their extension .CON must be used to invoke them, unlike the previous three that use .CSS, the assumed CSS file extensions.

These CSS procedures are designed for the simplest basic cases; i.e., only one disc device is assumed available. and the existing sourcename.PAS, the created sourcename.OBJ or sourcename.TSK, (or objectname.OBJ for PASLINK) are all assumed to be available or to be directed to that same disc volume, on which the CSS's reside. For task establishment it is assumed that no user-written routines other than the main program, which is linked to PASRTL.OBJ routines, are to be included.

Those CSS's which perform a compile, take the source from sourcename.PAS and produce the object on sourcename.OBJ; first deleting any file with that sourcename.OBJ. Those CSS's which perform a compile and link, also produce the established task on sourcename.TSK, first deleting any file previously existing with that sourcename.TSK. Those CSS's which perform just a link, take the object from objectname.OBJ and produce the established task on objectname.TSK; first deleting any file previously existing with that objectname.TSK. These deletions allow the CSS's to repeatedly recompile and relink a developing Pascal compilation-unit.

The CSS procedures are designed to provide a quick reference on what CSS arguments are expected. Invoking them without any arguments specified produces a brief description on the console. Enter one of the following:

```
*PASCAL

*PASCOMP

*PASLINK

*PASCAL.CON

*PASCOMP.CON

*PASLINK.CON
```

and a series of messages will be logged to help the user identify the necessary arguments.

The CSS procedures that result in an established user task are: PASCAL.CSS, PASCAL.CON, PASLINK.CSS and PASLINK.CON. See the OS/32 Link Reference Manual for details on task establishment.

If the user compiles a source program, for which the compiler requires greater than the default 64KB memincr of the CSS's or user-specified "memincr" given through the CSS's to the compiler, the compiler runtime error message LINE xxxxx, ADDR yyyyyy is displayed, stating:

```
HEAP OVERFLOW or
STACK OVERFLOW
```

Accidently entering a "memincr" of zero, 0, receives the error message: NOT ENOUGH SPACE TO RUN PASCAL, and task-termination.

The user must recall the CSS (or start the compiler) with a larger "memincr" than given previously.

## 1.7 ESTABLISHING A PASCAL PROGRAM AS A TASK

OS/32 Link must be used to link Pascal compiled-code to the Pascal run time library routines on PASRTL.OBJ and establish the objects as a task. The direct output onto lu 7 of the Pascal compiler is object program pure code that is directly acceptable as input to Link. Link INCLUDE commands may be used to include a main program, any separately compiled Pascal object external modules on the same file as the main program object, or on individual object files, or include other user objects, or an entire file containing a library of Pascal object modules.

The Link LIBRARY command, for programs compiled by Pascal R01 and higher, is used to LIBRARY PASRTL.OBJ so that (differing from Pascal R00) only those routines necessary to the external references in the object program are included in the task being established. Additional Link LIBRARY commands may be used to selectively edit the PEMATH.OBJ as the Perkin-Elmer Mathematical Sytem Library, the FORTRAN VII RTL and any user libraries.

When establishing a Pascal task, the Link options FLOAT, and DFLOAT must be specified for real and shortreal register use. The Link OPTION command WORK=(min,max) can be used to control the amount of main storage added to the task for workspace. In this command, "min" and "max" are <u>hexadecimal</u> numbers representing a number of <u>bytes</u>. <u>At load time</u>, the <u>default</u> size of the task workspace is the value of "min" established for the task; but this value can be changed to an amount up to but not greater than the "max" by specifying a "segment size increment" in the load command. Thus, in the load command "LOAD USERPROG.TSK,xxx", the "segment size increment", xxx, is a <u>decimal</u> number of <u>kilobytes</u> of task workspace to be added. If the Link OPTION command's WORK option is not used, or a "min" is not specified for task establishment, the Link supplies "min" to be 80 bytes (or X'50') and "max" to be 256 KB (or X'40000'). As the Link default for this workspace is 80 bytes, this is insufficient to run Pascal compiled-code, so the Pascal R01 CSS's set "min" to X'624', which is enough for Pascal's run time library's use. Likewise, the user must either specify an appropriate "min" at task establishment time, or be prepared to specify an appropriate "xxx" at load time. The "segment size increment" , xxx, used at load time for a Pascal task should include at least 1.50 KB (for the X'624' requirement) and enough storage beyond that for the global variables, stack, and heap, as required by the user task. When FORTRAN subprograms are included, additional memory must be allocated for the FORTRAN SCA, approximately X'5C' bytes plus X'10' bytes for each lu of MAXLU in the user task.

A sample command input for Link to establish a Pascal task is:

```
OPTION FLOAT,DFLOAT,WORK=(800,40000)
INCLUDE USERPROG.OBJ  {main Pascal program}
INCLUDE USERSUB.OBJ   {user external module(s)}{optional}
LIBRARY PASRTL.OBJ    {edit the Pascal Run Time Library}
LIBRARY PEMATH.OBJ    {optionally edit for math routines}
                      {or an RTL containing PEMATH}
MAP PR:               {Print a Link map}
BUILD USERPROG.TSK    {Build the user Pascal task}
END                   {signal end to Link}
```

Note that for programs that use <u>many</u> files an adjustment of MAXLU in a Link OPTION command and/or SYSSPACE Link option might be required. Pascal allows a maximum of 32 file names in its

PROGRAM header, which are external files, and allows any number of internal files. The default number of logical units available to a Pascal task by default is usually 15. Of this 15, or any user-specified LINK OPTION command MAXLU option, the user must assure that enough lu's are available for all external Pascal files listed in the PROGRAM header and any internal files which are used at the same time. Pascal compiled-code obtains temporary files for Pascal internal files from the next available lu below MAXLU, aside from the one lu reserved, when interfacing with FORTRAN produced code or the FORTRAN RTL, for the FORTRAN error message file. See the Link Manual for specifying other Link options.

Refer to Section 1.10 below for sample application BATCH/$INCLUDE Pascal compilations and other task-establishments under Link. Refer to Figure 1-3 for a memory map of the minimal initial state of a Pascal program task, under OS/32, and assuming no interfacing with the FORTRAN RTL or FORTRAN compiled-code. Refer to Chapter 10, Figure 10-4, for a more comprehensive memory map, involving FORTRAN interfacing and other options.

In the diagram of Figure 1-3 below, there are eight areas of memory. In order of increasing address, they are:

- the User-Dedicated Locations (UDL);
- object program code and constants;
- object RTL routines and optionally external modules; and

Task Workspace for:

- internal Pascal RTL Static Data Base Area (SDA);
- Pascal RTL Scrathpad;
- space for global variables;
- empty work space into which the stack and heap may expand;
- space between UTOP and CTOP, not used by Pascal compiled-code if MEMLIMIT=xx has specified xx <=99 percent.

The values of UPOT, UTOP and CTOP for a particular Pascal task, after loading, may be displayed by the user with a DISPLAY PARAM (D P) OS/32 command. CTOP is adjusted upon LOAD with any minimum increment specified during task establishment or at load time, as an appendage to the LOAD USERPROG.TSK,xxx; where xxx may specify an actual workspace value up to the maximum workspace, established in the task. UTOP is adjusted after entering the OS/32 START command, by the Pascal routine P$INIT called by Pascal compiled-code in order to initialize the task's memory management mechanisms.

Pointers to the global variable area (GB), the top of the stack (LB), and the bottom of the heap (SL) are in general registers; whose contents may be displayed by the user with a DISPLAY REGISTER (D R) OS/32 command.

| <u>Register</u> | <u>Use when executing Pascal compiled-code</u> |
|---|---|
| R0 | SL = Stack Limit (bottom of heap) |
| R1 | GB = Global Base |
| R2 | LB = Local Base (becomes the top of the stack starting after Global Variables) |

```
                       +-------------------+
                       |                   |<- CTOP {top of task}
                       |                   |
                       |                   |
             SL (R0) ->|-------------------|<- UTOP after START
                       |                   |
                       |   Remaining       |
                       |                   |   <-MEMLIMIT 7
                       |       Work        |     taskspace minus
                       |                   |     minimum memory
                       |         space     |     to start running
                       |                   |
                       |-------------------|<- Minimum memory
                       |                   |     needed if no
                       |    Global         |     routine-calls.
                       |       variables   |
                       |                   |
    LB (R2), GB (R1) ->|-------------------|<- Minimum memory to
                       |  RTL Scratchpad   |     start running.
                       |  X'600' Bytes     |
                       |-------------------|
                       |    Pascal SDA     |
                       |      36 Bytes     |
 See Link MAP for ---> |-------------------|<- UTOP after LOAD
   actual object sizes |                   |
                       |Pascal RTL objects |
                       |  and optionally   |
                       |  Pascal Modules   |
                       | . . . . . . . . . |
                       |                   |
                       |  Object Program   |
                       | Code & constants  |
    PROG label ENTRY ->|                   |
                       |                   |
          PROG label ->|-------------------|<- UBOT + X'100'
                       |                   |
                       |     UDL           |
                       |                   |
                       +-------------------+<- UBOT {Task Bottom}
```

**Figure 1-3 Minimal Initial Memory Map for Pascal Program**

If the Link OPTION command ABS=specifier is used at task establishment time, then an additional area may be reserved between the UDL and object program code; or as useful in a Reliance task the ABS=specifier may reserve a larger absolute area for a UDL of X'1D00'.

If interfacing with FORTRAN compiled-code or the FORTRAN RTL is involved for the task, an additional internal table for the FORTRAN Static Communications (SCA) resides between the RTL Scratchpad and the Pascal SDA.

If a program is linked so as to be shareable, then the object program code and constants are in a separate, shared segment.

## 1.8 EXECUTING A PASCAL PROGRAM TASK

An established Pascal program task is executed by loading it, making necessary lu assignments for Pascal external files, and starting the task, under OS/32. Users preparing programs for a Reliance environment must refer to Appendix M. Allocation and assignment of Pascal internal files is accomplished by code generated by the compiler. An established Pascal task is executed by:

- loading the task with the OS/32 LOAD command,
- assigning any logical units required for external files as referenced by the task, with the OS/32 ASSIGN command,
- starting the task with the OS/32 START command.

The task should then perform its intended purpose, unless there are run-time errors, such as may be caused by improper memory allocations of task workspace, I/O problems or other run time errors interupting task execution. Any run time exceptions fielded by compiler-generated code in the user object program are logged to the user console in an MTM environment, to the system console under standalone OS/32 or to the system journal under a Reliance environment. (see Section 1.5.4.3).

During execution of a Pascal program task, the memory map differs from its initial state as depicted in Figure 1-3 above in two respects. There may be local variables activated, and parameter data passed to activated routines. This information is housed in a stack within workspace and may grow up to, but not inclusively, the Stack Limit (SL).

Also, there may be dynamically allocated variables created on and/or then removed from the heap. As the heap within workspace grows downward to house dynamic variables the Stack Limit is reduced accordingly. Refer to Figure 10-5 for a comprehensive run time memory map of an executing Pascal program task; which depicts this activity.

Special run-time error messages, HEAP or STACK OVERFLOW will occur when there is not enough workspace in the user task for the heap and stack operation. If this occurs, the user may reload the task with the OS/32 LOAD command requesting additional minimum Workspace, e.g., "LOAD USERTASK.TSK,100" up to the maximum "worksize" established in the task (by default or specification), and restart; or reestablish the task with both a more adequate minimum and maximum workspace, to avoid having to respecify memory workspace minimums at load time.

## 1.9 LISTING FORMATS

The Pascal compiler generates several programming listing aids. On the final pass, Pass 10, the following listing aids are generated on lu 2:

- Compiled program listing, if LIST option is on
- Cross reference listing, if CROSS option is on
- Summary listing, if SUMMARY option is on
- Program statistics listing, on every compilation
- Batch statistics listing, if BATCH option is on

On the object generation pass, Pass 9, the following listings are generated on lu 6:

- Assembly Listing, if the ASSEMBLY option is on
- Map Listing, if the MAP option is on

If lu 2 and lu 6 are assigned to the same device or file, these listings appear before those generated on lu 6, as they are generated on PASS 9.

### 1.9.1 Compiled Program Listing

The compiler generates a listing of the input source program being compiled with heading information and an image of the input program with line numbers, routine nesting level, and statement nesting level information. The exclamation point character (!) indicates where column 1 of the source line begins.

Each source line is given a compilation-unit line number. If the compilation is part of a batch job, a second set of line numbers is provided to indicate the record number of the line in the batch stream.

If the source being listed was obtained from another file by the INCLUDE option, the compilation-unit line numbers are frozen at the current unit's line number of the {$INCLUDE (fd)} option-specifier line, the batch line numbers (if under BATCH) continue being incremented, and additional line numbers appear on the right hand margin, enumerating the INCLUDEd file's source line numbers, starting from 1.

A nested INCLUDE option occurring within a Pascal source file, already in the process of being included, has the line numbers in the right margin restarted from 1, and incremented by 1 for its duration; and upon completion, the right margin line numbers return to the previous including file's line numbering sequence.

The heading information contains page numbers and date and time of compilation.

Each page within a compilation-unit's listing is given a compilation-unit page number. A second set of page numbers is used when processing a batch job. This second set of batch page numbers increases sequentially as each compilation unit in the batch job is processed. The former set of compilation-unit page numbers is reset to 1 for each compilation unit within the batch. Heading information on each page also identifies the compiler along with the revision level and the customer software license number. Refer to Appendix A for a sample listing.

At the end of the compiled program listing, the compiler generates the following additional listings in the order indicated:

1. Cross reference listing, if CROSS option is on
2. Summary listing, if SUMMARY option is on
3. Program statistics listing, for every compilation
4. Batch statistics listing, if BATCH option is on. This listing appears at the end of all compilation units.

### 1.9.2 The Summary Listing

A summary of the Pascal compiler internal statistics accumulated for each compilation-unit during each of the first eight passes is produced if enabled by the SUMMARY compiler option. This summary reflects compile-time table memory space (number of unique user identifiers in use), scratch file size requirements, and optimizations performed on Pass 6. If the OPTIMIZE option is on, all optimizations performed on Pass 8 are reflected. For each of the eight passes summarized, file size information is listed in the form:

```
SUMMARY, PASSn, FILE LENGTH:  nn SECTORS
```

Users repeatedly compiling a near-stable program with long compile-times may wish to pre-allocate the scratch files on lu 3 and lu 4 as temporary contiguous files, given this information in the summary. The user is cautioned that when assigning lu 3 and lu 4 to temporary contiguous files, as in:

    "TE 3,CONTIGUOUS,fsize"    or    "TE 4,CONTIGUOUS,fsize"

that the "fsize" is a number of 256-byte sectors and must be at least the largest number of nn SECTORS (rounding any odd nn to at least the next even number because Pascal outputs 512 byte records to both lu 3 and lu 4) reflected for either the odd-numbered passes or the even-numbered passes. See Figure 1-2.

The odd-numbered passes output to lu 3, so lu 3 may be created with a "fsize" >= largest nn SECTORS shown for an odd-numbered Pass in the SUMMARY listing. If the largest nn is odd, it must be rounded to at least the next even number.

The even-numbered passed output to lu 4, so lu 4 may be created with a "fsize" >= largest nn SECTORS shown for an even-numbered Pass in the SUMMARY listing. If the largest nn is odd, it must rounded to at least the next even number.

Several summary paragraphs provide additional information. The PASS1 paragraph additionally lists the number of unique identifiers. The PASS3 paragraphs additionally lists the number of NOUNS USED and UPDATES USED. The PASS6 paragraph additionally lists the Pass 6 optimizations and how many times each optimization was performed, if any. The PASS7 paragraph additionally lists the REGISTERS ASSIGNED and the number of PROCEDURE/FUNCTION calls. The PASS8 paragraph additionally lists the Pass 8 optimizations and how many times each optimization was performed, if any. The amount of memory space saved by the optimizations is also reflected. Refer to Appendix A for an example.

### 1.9.3 The Cross Reference Listing

The cross reference listing provides a cross reference of all declared statement labels, predefined Pascal identifiers, and user-specified identifiers used in the compilation unit.

The cross-reference listing is produced if enabled by the CROSS option being on by default or specification.

An index to the mnemonics in the cross reference listing is available for quick reference on the first page under the title CROSS REFERENCE LISTING.

```
INDEX TO MNEMONICS

:# = CHANGE OF VALUE
:L = LABEL DECLARATION
:@ = LABEL REFERENCE
```

:C = CONSTANT DECLARATION
:T = TYPE DECLARATION
:V = VARIABLE DECLARATION
:P = PROCEDURE NAME
:F = FUNCTION NAME

Labels, if any are used, are first listed in the order of their integral value. Then the identifiers are listed alphabetically. Each predefined Pascal identifier, used in the unit, is listed with the compilation-unit line numbers in which they were referenced. Each user-specified identifier listed is first followed by its defining or declaration source line number, and then the line numbers in which they were referenced. A brief two-character mnemonic is attached to the first defining line number listed after each label or user-specified identifier, as:

:L = LABEL DECLARATION
:C = CONSTANT DECLARATION
:T = TYPE DECLARATION
:V = VAR DECLARATION
:P = PROCEDURE NAME
:F = FUNCTION NAME

The compilation-unit line numbers of all source lines which contain references to the label, predefined Pascal identifier, or user-specified identifier are listed.

Any of these source lines in which the user-specified identifier has had a change of value has that source line number followed by ":#".

For declared labels, a source line containing a label reference has that source line number followed by ":@".

The user may refer to the main body of the compiled-program listing to locate by compilation-unit line number all definitions and references listed. Refer to Appendix A for an example cross-reference listing.

### 1.9.4 Program Statistics Listing

The program statistics listing occurs for every compilation and is not a compiler option. It is output to lu 2 and contains:

- if no external Pascal file names were listed in the user's PROGRAM header statement, or when compiling MODULEs, the message: - NO EXTERNAL FILES USED or otherwise: a table associating external Pascal file names (as listed by the user in the file-name-list, of the PROGRAM header statement) with their respective logical unit numbers as associated by the

compiler, entitled: - EXTERNAL FILE TABLE.
For example, a Pascal PROGRAM header statement, such as:

```
PROGRAM(INPUT,OUTPUT,FILENAME,INFILE50,OUTFILE70);
```

has its external files associated to lu's in this table, such as:

- EXTERNAL FILE TABLE

```
INPUT     0
OUTPUT    1
FILENAME  2
INFILE50  3
OUTFILE70 4
```

The above listed logical units for the user task must be assigned after loading the user task, and prior to starting its execution.

- a table of the user file descriptors or device mnemonics used by the compiler to input source, and output listings, and object of the current compilation unit, entitled:

  - FILES USED FOR THIS COMPILATION

```
INPUT FILE:    name of source file/device compiled from.
LISTING FILE:  name of file/device listing was produced on.
OBJECT FILE:   name of file, object was produced on.
ASSEMBLY FILE: name of assembly/map-listing file/device.
```

- the message START OPTIONS: followed by the exact string of options passed to the compiler in its START command. If none were supplied, the message ** NONE ** is written.

- a table of all applicable compile time options along with their default state, their final state at completion of the compilations, and indications if they were passed in the START command.

- a summary of compiler detected error messages as required. If these diagnostic error messages appear, the user must correct the errant source causing the message, and recompile.

- the message COMPILATION COMPLETE, NO ERRORS; or the message COMPILATION ABORTED, nn ERRORS, if appropriate.

- if no compile time errors were detected, a statement of the object code size and literal constants table size (in decimal bytes) that was generated for the object program:

```
CODE SIZE = nnn BYTES CONST SIZE = nnn BYTES
```

where nnn is a decimal number of bytes. For a program, the summing of these two sizes, represents the size of the object program and constants, which resides at run-time beyond the X'100' bytes of the UDL (or other user-specified ABSOLUTE space) within the user task partition; and prior to the starting address of the first external module or RTL routine to be linked after the program.

Refer to Appendix A for a sample.

### 1.9.5 Batch Statistics Listing

The batch statistics listing is produced only when compilation is performed under the BATCH compiler option. See Figure 1-4.

If the compiler processed a batch stream, a final listing is printed with batch statistics information as follows:

- The program or module name of the compilation unit.
- The first line number of that compilation unit relative to the entire batch stream.
- The page number of the listing on which that compilation unit can be found.
- The end of task code associated with the processing of that compilation unit (see Table 1-4).
- For every compilation unit that was compiled without error diagnostics, or other error, a statement of the object code size and literal constant table size (in decimal bytes) generated for the object code of the compiled program.

```
                              BATCH STATISTICS INFORMATION

BATCH UNIT NAME          BATCH LINE NUMBER  PAGE NUMBER      EOT CODE            CODESIZE   CONSTLENGTH
MAIN                                     1            1             0                  40             0
MODULE1                                  8            4             0                  48             0
MODULE2                                 15            7             0                  32             0
MODULE3                                 23           10             0                  24             0
```

Figure 1-4 Sample Batch Statistics Listing

### 1.9.6 Assembly Listing

The machine instructions and literal constants generated in the object code of a compiled Pascal source program are available to the user in the Assembly Listing. The Assembly Listing is only produced when the user specifies the ASSEMBLY option to the compiler. For each Pascal source statement that causes object code to be generated; the Assembly Listing contains a hexadecimal representation of its generated machine instructions, their location (indicated by a displacement relative to their start address), and a mnemonic assembler format of each machine instruction. One or more lines of assembler-level code will follow each Pascal source line number listed.

Each source line listed is identified with its decimal number, xxxxx, from the main compiled-program listing, in the form:

* LINE xxxxx

For each machine instruction generated for that source line, the Assembly Listing contains the relative displacement of the machine instruction, followed by the actual object code of the machine instruction, and concluding with a mnemonic disassembly of that object code.

For a program task, established under default conditions, the run-time object addresses of this code within the user task partition, usually begin at X'100' bytes off UBOT after the UDL (or other user-specified ABSOLUTE space) in the Object Program Code and Constants area reflected in Figure 1-3.

The user must obtain a MAP from OS/32 LINK to ascertain the actual object locations of any particular program and/or its modules when the compilation-unit objects are linked to be established as a task.

In the LINK MAP, the Pascal PROGRAM name becomes a PROGRAM label (truncated to 8-characters) has its location reflected and this is where Pascal compiled object code begins. It is not necessarily where the main body begins. The LINK MAP also reflects the PROGRAM label as an ENTRY. The location reflected in the LINK MAP of the PROGRAM label as an ENTRY is where the main body of the Pascal program begins, and where its execution begins. One can easily locate the line number of the main body compound statement of a Pascal program from the compiled-program listing and associate it to the identical line number in a Pascal Assembly Listing.

Likewise, given the starting locations of the PROGRAM and ENTRY labels from an established task's LINK MAP, the program's object code listed in the Pascal Assembly Listing can be located. The displacement addresses in the Assembly Listing for MODULE object code displacements reflected in the Pascal Assembly Listing are relative to their beginning object address reflected in a LINK MAP for the particular name of the MODULE concerned. Pascal module names are also truncated to 8-character ENTRY labels with their object locations also reflected in the LINK MAP.

See Figure 1-5 for a portion of the Assembly Listing produced when compiling the sample program PRIMES of Appendix A.

```
                              PRIMES PROG
                              * LINE 29
000000   8880 001D            L38      ERR    R8,29
000004   50F2 0004            P101     ST     R15,4(R2)
000008   C502 0008                     CLHI   R0,12(R2)
00000C   2086                          BCS    L38
                              * LINE 30
00000E   5831 0160                     L      R3,352(R1)
000012   5031 0164                     ST     R3,356(R1)
                              * LINE 31
000016   5131 0160            L62      AM     R3,352(R1)
                              * LINE 32
00001A   5831 0160                     L      R3,352(R1)
00001E   F930 0001 86A0                CI     R3,100000
000024   4220 802E                     BP     L3
                              * LINE 33
000028   C930 0001                     CHI    R3,1
00002C   2115                          BMS    L39
00002E   F930 0001 86A1                CI     R3,100001
000034   2113                          BMS    L40
000036   8810 0021            L39      ERR    R1,33
00003A   C840 004E            L40      LHI    R4,78
00003E   C540 0080                     CLHI   R4,128
000042   2183                          BCS    L42
000044   8830 0021                     ERR    R3,33
000048   D241 4300 0173       L42      STB    R4,371(R1,R3)
                              * LINE 34
00004E   5831 0164                     L      R3,356(R1)
000052   4300 FFC0                     B      L62
                              * LINE 36
000056   58F2 0004            L3       L      R15,4(R2)
00005A   5822 0000                     L      R2,0(R2)
00005E   030F                          BR     R15
                              * LINE 38
000060   8880 0026            L43      ERR    R8,38
000064   C8F0 0064            P1       LHI    R14,100
000068   24D1                          LIS    R13,1
00006A   41F0 4000 0000                BAL    R15,P102
000070   41F0 4000 0000                BAL    R15,P103
000076   F502 0001 8820                CLI    R0,100384(R2)
00007C   208E                          BCS    L43
00007E   E6E2 000C                     LA     R14,12(R2)
000082   C8D0 0100                     LHI    R13,256
000086   24C0                          LIS    R12,0
000088   41F0 4000 0000                BAL    R15,P104
```

**Figure 1-5 Sample Assembly Listing Fragment**

Following the machine instructions listed, the Assembly Listing also contains the contents of the literal constants used by the compiled program. These literal constants also reside in the Object Program Code and Constants space reflected in Figure 1-3.

The Assembly Listing presents their relative displacements, and a representation of their actual object code; although they are only disassembled as DCF decimal values.

### 1.9.7 Map Listing

The map listing consists of the STATEMENT MAP and a DATA AREA MAP. The STATEMENT MAP lists, for each line of the compiled program's executable statements, the source line number and the object relative address of the first code-generated machine instruction for that source line. Some source lines with non-executable code; i.e., some ENDs, are not represented in the STATEMENT MAP listing. The displacements given for each line number listed in the STATEMENT MAP are identical in meaning to those described in the preceding section on the Assembly Listing (See Section 1.9.6 above).

The DATA AREA MAP lists a representation of the relative locations of the compiled program data. The source line number that defines a datum is called a data-defining line number, and the DATA AREA MAP lists each data-defining line number with a displacement (not including alignment padding) of the first datum in the line. The displacement shown in the DATA AREA MAP under "DISPL" is entirely different from those displacements listed in either the STATEMENT MAP or the Assembly Listing. The displacements listed in the DATA AREA MAP are displacements off the Global Base Register pointing into the Global Variables space reflected in Figure 1-3, for example, for global variables.

To use this portion of the MAP listing, the user must be familiar with in-depth run-time support information of Chapter 10. Specifically required is cognizance of the internal storage requirements of each different data-type and the padding mechanism utilized for object data alignment requirements.

The compiler reserves memory space for each datum defined in the user source program. Following the allocation of this space for all data defined on a data-defining line, the accumulated amount of reserved space dictated by the previous line is listed under "DISPL" for the line number listed under "LINE". This displacement points to where the first datum on that line number may begin.

If the value of the displacement does not meet the alignment requirements of the first datum because of its internal storage requirements, that first datum will be found at the displacement (shown under "DISPL") plus the number of bytes required by alignment padding (not shown in MAP). Refer to Chapter 10. Figure 1-6 is a sample map listing obtained from compiling the sample program PRIMES of Appendix A. Compilation was performed with the start option MAP.

```
STATEMENT MAP

 LINE  DISPL     LINE  DISPL     LINE  DISPL     LINE  DISPL     LINE  DISPL

   30 00000E       31 000016       32 00001A       33 000028       34 00004E
   36 000056       38 000060       39 00008E       40 000098       41 0000A2
   42 0000C2       44 0000CC       45 0000EC       46 0000F6       47 0000FC
   48 000132       49 000138       50 00014E       51 00015E       52 000166
   54 00016F       55 000174       56 0001A2       57 0001C0       58 0001E4
   59 0001E8       60 000200       61 000206       62 000226       63 000234
   64 00023E       65 000244       66 00024E       67 000258       69 00025E
   71 000276       72 000286       73 0002A2       74 0002B2       76 0002C6

DATA AREA MAP

 LINE  DISPL     LINE  DISPL     LINE  DISPL     LINE  DISPL     LINE  DISPL

    9 00000C       19 00015C       20 00015E       21 00016C       22 00016E
   23 000174       27 018814       37 018814
```

Figure 1-6  Sample Map Listing

## 1.10 SAMPLE PASCAL $BATCH/$INCLUDE APPLICATIONS

Examples of the system ease of use of batch compilation and the $INCLUDE compiler option are presented below. The first example is a straight-forward batch compilation of a main Pascal program and several external Pascal modules (procedures and functions). The second example shows the use of the $INCLUDE option to merge the source of external Pascal modules into a batch stream; a sample of the linkage process to PEMATH.OBJ math routines and/or a FORTRAN RTL for the linkage process to external FORTRAN compiled user-written routines. The third example concerns the inclusion of the standard Prefix for those systems using its extended language features for I/O and other operating system services. Pascal programs or modules referencing the Prefix routines must include the source of the Prefix as the foremost part of themselves as compilation-units. The fourth example reflects the use of batch compilation of a series of modules in order to create a library of a file of object modules.

For purposes of illustration, the file descriptors given in the examples have an extension which is indicative of their format and content. An extension of .PAS indicates that the file contains Pascal source; an extension of .CBJ indicates that a file contains standard linkable object code; and an extension of .FTN indicates that a file contains FORTRAN source. An extension of .LST indicates that a file contains ASCII formatted listing information, which could be printed to a device such as "PR1:" for a line printer. An extension of .TSK indicates that a file contains a loadable established task under OS/32. An extension of .CSS indicates a file containing a CSS procedure. The .CSS extension need not be specified when assumed by the operating system in a command entry to the operating system; it is presented in the CSS invocations below for clarity.

In all of the following examples the character string, voln:, stands for the volume on which the file named by the subsequent file descriptor resides. The voln: need not be specified by the user if all files exist on the same volume as the CSS's.

The asterisk(*) preceding a line of command entry indicates the prompt displayed by the OS/32. The right arrow (>) preceding a line of command entry indicates the prompt displayed by the command mode of OS/32 Link. Familiarity with OS/32 Link is assumed, such that the Link commands in the examples, may be presented in abbreviated form. The ellipsis (...) indicates a general continuance or assumed code unimportant to the main illustration at hand.

Example 1:

The first example depicts the use of batch compilation of a source stream of several compilation units, i.e., a main program and several external modules residing on the file, voln:SYSTEMA.PAS, as outlined below.

```
{$BATCH}
PROGRAM SYSTEMA(file-name-list);
  CONST ...
  TYPE ...
  VAR ...
  PROCEDURE P1(p1-paramlist);EXTERN;
  PROCEDURE P2(p2-paramlist);EXTERN;
  PROCEDURE P3(p3-paramlist);EXTERN;
  FUNCTION F1(f1-paramlist):type-id;EXTERN;
  FUNCTION F2(f2-paramlist):type-id;EXTERN;
  FUNCTION F3(f3-paramlist):type-id;EXTERN;
  FUNCTION LOCAL1(local1-paramlist):type-id;
  ...block;
  PROCEDURE LOCAL2(local2-paramlist);
  ...block;
  BEGIN {Body of main program SYSTEMA}
  ...
  END.
MODULE P1(p1-paramlist);
  ...
  BEGIN {Body of P1}
  ...
  END.
MODULE P2(p2-paramlist);
  ...
  BEGIN {Body of P2}
  ...
  END.
MODULE P3(p3-paramlist);
  ...
  BEGIN {Body of P3}
  ...
  END.
MODULE F1(f1-paramlist):type-id;
  ...
  BEGIN {Body of F1}
  ...
  END.
MODULE F2(f2-paramlist):type-id;
  ...
  BEGIN {Body of F2}
  ...
  END.
MODULE F3(f3-paramlist):type-id;
  ...
  BEGIN {Body of F3}
  ...
  END.
{$BEND}...or end-of-file...
```

The batch compilation of the batch stream on file voln:SYSTEMA.PAS above, through the use of PASCAL.CSS, could be performed with:

```
*PASCAL.CSS voln:SYSTEMA,PR1:,SU MA LI CR,PR1:,120
```

or if the {$BATCH} in-stream option specification (on the first line) were not available in the batch stream itself:

```
*PASCAL.CSS voln:SYSTEMA,PR1:,BA SU MA LI CR,PR1:,120
```

Note that this CSS invocation supplies BA as a compiler start option in the list of compiler options, as a member of the third argument to PASCAL.CSS.

With either method of specifying the batch option to the compiler, the batch-compilation through PASCAL.CSS produces the compiler-generated and linkable object code of the main program and all the subsequent modules: P1, P2, P3, F1, F2, and F3 on one file, voln:SYSTEMA.OBJ. The program name in the main program header need not be identical to the name of the file containing the batch-stream. The standard Pascal CSS's use the file descriptor filename of the first CSS argument (the extensions are assumed) as the basis for generating the filenames of the object file and task file.

On the file, voln:SYSTEMA.TSK, the established user program task is generated.

On device PR1: are the listings requested by the options SU MA LI CR. The LI and CR listing and cross-reference options are the default conditions. Also, if the default device, PR:, of the standard Pascal CSS's were available on the user system; the former of the previous CSS invocations would be:

```
*PASCAL.CSS voln:SYSTEMA,,SU MA,,120
```

which would produce the listings on PR:. If the user wished to save the listings generated during compilation and print them later, the CSS printer arguments could be specified as files, such that the previous CSS invocation would be:

```
*PASCAL.CSS voln:SYSTEMA,voln:SYSTEMA.LST,SU MA,voln:AMAP.LST,120
```

from which the main listings would be on file, voln:SYSTEMA.LST; and the map listing (also ASSEMBLY listing if the AS option were specified) would be on file, voln:AMAP.LST. A memory increment argument of 120 is given to the compiler through the CSS invocations in these examples assuming they contain large compilation units requiring greater compilation space to process. If compilations of an actual user batch stream are compilable with the default value of the compiler no memory increment need be specified.

Implicit in the standard Pascal CSS, is an INCLUDE command to OS/32 Link to INCLUDE voln:SYSTEMA.OBJ, in this example, and all required linkages are resolved between the main program and its external modules, followed by a LIBRARY PASRTL.OBJ to link to the RTL.

To illustrate, the above batch stream on voln:SYSTEMA.PAS could have been compiled and not linked, through the use of the PASCOMP.CSS, for example with:

```
*PASCOMP.CSS voln:SYSTEMA,PR1:,SU MA LI CR,PR1:,120
```

which performs the batch compilation, generating the file, voln:SYSTEMA.OBJ, which contains the linkable object code for the main program and its external modules. The file, voln:SYSTEMA.OBJ, is available for a Link sample input command stream, where the character ">" indicates the Link prompt, such as in:

```
>OPTION FLOAT,DFLOAT,WORK=(800,40000)
>INCLUDE voln:SYSTEMA.OBJ  {include main/module objects}
>LIBRARY voln:PASRTL.OBJ   {link/lib Pascal Run Time Library}
>MAP PR1:,ADDRESS,XREF     {obtain a Link map}
>BUILD voln:SYSTEMA.TSK    {build user task}
>END
```

The INCLUDE voln:SYSTEMA.OBJ command directs Link to include the objects on that file; and because the batch-compilation generated the objects for both the main program and its external modules onto one file, voln:SYSTEMA.OBJ, several required linkages are established at this point. With the LIBRARY voln:PASRTL.OBJ command Link prepares to attempt to resolve all remaining run time support routine linkages. The user program task is available on file, voln:SYSTEMA.TSK, if all linkages were resolved. The user must refer to the LINK MAP produced on PR1: to identify any unresolved linkage condition.

Once the the batch-compilation provides SYSTEMA.OBJ, the user could have also performed task-establishment through the use of the standard PASLINK.CSS by invoking it as:

```
*PASLINK.CSS voln:SYSTEMA,PR1:
```

to also obtain the established task on file, voln:SYSTEMA.TSK.

To execute the user task, a short series of OS/32 directives is required to load the task, assign logical units, if required by the task, and issue the START command.

For example, where xxx = a minimum memory increment,

```
*LOAD voln:SYSTEMA.TSK,xxx
*ASSIGN 0,CON:
*ASSIGN 1,MAG1:
*START
```

Example 2:

Another sample of assembling a system for batch compilation involves a main Pascal program which requires external Pascal modules already residing on separate source files, linkage to the Perkin-Elmer System Mathematical Library math routines, PEMATH.OBJ (without argument checking), and/or a FORTRAN VII RTL for external user-written FORTRAN routines. In this sample, the $INCLUDE Pascal compiler option merges the separate source modules into the batch stream. A batch stream may be arranged as follows:

```
{$BATCH}
PROGRAM SYSTEMB(file-name-list);
  CONST ...
  TYPE ...
  VAR ...
  FUNCTION DSIN(X:REAL):REAL;FORTRAN;
  PROCEDURE P1(p1-paramlist);EXTERN;
  PROCEDURE P2(p2-paramlist);EXTERN;
  PROCEDURE PP3(pp3-paramlist);FORTRAN;
  FUNCTION F1(f1-paramlist):type-id;EXTERN;
  FUNCTION F2(f2-paramlist):type-id;EXTERN;
  FUNCTION FF3(ff3-paramlist):type-id;FORTRAN;
  FUNCTION LOCAL1(local1-paramlist):type-id;
    ...block;
  PROCEDURE LOCAL2(local2-paramlist);
    VAR L1,L2,L3:REAL;
    BEGIN {Body of LOCAL2}
      ...
      L1:=L2 + DSIN(L3);
      ...
    END;  {End of LOCAL2}
  BEGIN {main body of program SYSTEMB}
  ...
  END.
{$INCLUDE (voln:P1.PAS)}
{$INCLUDE (voln:P2.PAS)}
{$INCLUDE (voln:F1.PAS)}
{$INCLUDE (voln:F2.PAS)}
{$BEND} ...or end-of-file...
```

In this case, the batch stream on file, voln:SYSTEMB.PAS, could be compiled with PASCOMP.CSS generating a linkable object file on voln:SYSTEMB.OBJ with the CSS invocation:

```
*PASCOMP.CSS voln:SYSTEMB,PR1:,SU MA LI CR,PR1:,120
```

The above sample, SYSTEMB, assumes separately available files, whose source would be merged into the batch stream by the Pascal $INCLUDE options specified in the batch stream source enclosed within Pascal comment brace delimiters, { and }. Sample SYSTEMB assumes the external P1, P2, F1, and F2 routines source are separately available, i.e.,

```
on voln:P1.PAS resides:           on voln:P2.PAS resides:

   MODULE P1(p1-paramlist);         MODULE P2(p2-paramlist);
     ...                              ...
     BEGIN {Body of P1}               BEGIN {Body of P2}
     ...                              ...
     END.                             END.
```

and separately,

```
on voln:F1.PAS resides:          on voln:F2.PAS resides:

   MODULE F1(f1-paramlist):type-id; MODULE F2(f2-paramlist):type-id;
     ...                              ...
     BEGIN {Body of F1}               BEGIN {Body of F2}
     ...                              ...
     END.                             END.
```

It is further assumed that the external FORTRAN user-written routines, such as:

```
on voln:PP3.FTN resides:          on voln:FF3.FTN resides:

     SUBROUTINE PP3(pp3-params)     FUNCTION FF3(ff3-params)
     ...param-types                 ...param-types
     ...                            ...
     ...                            ...
     END                            END
```

where the external compilation of PP3 and FF3 by the FORTRAN compiler produced their respective object code on files: voln:PP3.OBJ and voln:FF3.OBJ.

Note that the sample SYSTEMB requires linkage in addition to PASRTL.OBJ, to external user-written FORTRAN-compiled routines (PP3 and FF3) and thereby linkage to a FORTRAN RTL; while also requiring linkage to PEMATH.OBJ (if the FORTRAN RTL does not include the sine routines of PEMATH.OBJ) due to the sine function call in PROCEDURE LOCAL2. A sample command input to OS/32 LINK, where the character ">" indicates a prompt for input commands, must then include additional Link commands to establish the user task, on voln:SYSTEMB.TSK, as follows:

```
>OPTION FLOAT,DFLOAT,WORK=(800,40000)
>INCL voln:SYSTEMB.OBJ       {include main/module objects}
>LIB voln:PASRTL.OBJ         {link/lib the Pascal RTL}
>INCL voln:PP3.OBJ           {include external FORTRAN object}
>INCL voln:FF3.OBJ           {include external FORTRAN object}
>LIB PEMATH.OBJ              {link/lib PE System Math library}
>LIB F7RTL50.OBJ         {link/lib FORTRAN RTL w/o argchecks}
>MAP PR1:
>BUILD voln:SYSTEMB.TSK      {build user task}
>END
```

The user task is then available on file, voln:SYSTEMB.TSK, ready for loading, lu assignments, and execution.

Example 3:

A third example is presented, SYSTEMC, to illustrate for users using the I/O language extensions, provided by the standard Prefix (Refer to Chapter 10 or Appendix N, for information on the standard Prefix). A batch stream is coded as follows, to reflect the inclusion of the prefix for systems where both the main program and two of the external procedures, P1 and P2, reference the Prefix routines.

In this example, SYSTEMC, the standard Prefix source is assumed available on the file, voln:PREFIX.PAS. Also assumed are that both the main program and two external procedures, P1 and P2, reference prefix (language extension) routines; which requires that both the main program and the P1 and P2 modules be compiled with the inclusion of the prefix source. Note the boldface comments {$INCLUDE (voln:PREFIX.PAS)}, preceding the compilation units requiring a prefix. Batch compilation and task establishment could be performed by the CSS invocation:

```
*PASCAL.CSS voln:SYSTEMC,PR1:,SU MA,PR1:,120
```

which would compile and link and establish the user program task on the file, voln:SYSTEMC.TSK, and the user task is then available for loading, lu assignments and execution.

Note that if the MODULE paramlists below for P1 and P2 contain non-predefined user-specified type-identifiers to type their parameters, a TYPE declarations part of a prefix must first declare those type-identifiers prior to the MODULE header and also occur prior to the $INCLUDE of the PREFIX.PAS because the

TYPE declarations cannot occur after the routine-declarations in the prefix of PREFIX.PAS, syntactically.

```
{$BATCH}
{$INCLUDE (voln:PREFIX.PAS)}
PROGRAM SYSTEMC(file-name-list);
  CONST ...
  TYPE ...
  VAR ...
  PROCEDURE P1(p1-paramlist);EXTERN;
  PROCEDURE P2(p2-paramlist);EXTERN;
  PROCEDURE P3(p3-paramlist);EXTERN;
  FUNCTION F1(f1-paramlist):type-id;EXTERN;
  FUNCTION F2(f2-paramlist):type-id;EXTERN;
  FUNCTION F3(f3-paramlist):type-id;EXTERN;
  FUNCTION LOCAL1(local1-paramlist):type-id;
    ...block;
  PROCEDURE LOCAL2(local2-paramlist);
    ...block;
  BEGIN {Body of main program SYSTEMC}
  ... EXIT(0);
  END.
{$INCLUDE (voln:PREFIX.PAS)}
MODULE P1(p1-paramlist);
  ...
  BEGIN {Body of P1}
  ... TIME(T_BUF); DATE(D_BUF);
  END.
{$INCLUDE (voln:PREFIX.PAS)}
MODULE P2(p2-paramlist);
  ...
  BEGIN {Body of P2}
  ... WRITE_FILE_MARK(2,STATUS1); CLOSE(2,STATUS2);
  END.
MODULE P3(p3-paramlist);
  ...
  BEGIN {Body of P3}
  ...
  END.
MODULE F1(f1-paramlist):type-id;
  ...
  BEGIN {Body of F1}
  ...
  END.
MODULE F2(f2-paramlist):type-id;
  ...
  BEGIN {Body of F2}
  ...
  END.
MODULE F3(f3-paramlist):type-id;
  ...
  BEGIN {Body of F3}
  ...
  END.
{$BEND}...or end-of-file...
```

Example 4:

In a fourth example, a Pascal system library could be generated by performing a batch compilation of a series of modules, with the batch stream residing on the following file, voln:SYSLIB4.PAS, such as:

```
{$BATCH}
MODULE P1(p1-paramlist);
  ...
  BEGIN {Body of P1}
  ...
  END.
MODULE P2(p2-paramlist);
  ...
  BEGIN {Body of P2}
  ...
  END.
MODULE P3(p3-paramlist);
  ...
  BEGIN {Body of P3}
  ...
  END.
MODULE F1(f1-paramlist):type-id;
  ...
  BEGIN {Body of F1}
  ...
  END.
MODULE F2(f2-paramlist):type-id;
  ...
  BEGIN {Body of F2}
  ...
  END.
MODULE F3(f3-paramlist):type-id;
  ...
  BEGIN {Body of F3}
  ...
  END.
{$BEND}...or end-of-file...
```

If the MODULE header "paramlists" or "type-id's" contain any user-specified type-identifiers, the declaration of the type-identifiers may precede each MODULE in a TYPE declarations part of a user-specified prefix. See Example 3 above for placement cautions when PREFIX.PAS is also being included.

In the sample, SYSLIB4.PAS, is a batch stream of source Pascal modules external to the main program, which separately resides on the file, voln:SYSTEMD.PAS, as outlined below:

```
PROGRAM SYSTEMD(file-name-list);
  CONST ...
  TYPE ...
  VAR ...
  PROCEDURE P1(p1-paramlist);EXTERN;
  PROCEDURE P2(p2-paramlist);EXTERN;
  PROCEDURE P3(p3-paramlist);EXTERN;
  FUNCTION F1(f1-paramlist):type-id;EXTERN;
  FUNCTION F2(f2-paramlist):type-id;EXTERN;
  FUNCTION F3(f3-paramlist):type-id;EXTERN;
  FUNCTION LOCAL1(local1-paramlist):type-id;
    ...block;
  PROCEDURE LOCAL2(local2-paramlist);
    ...block;
  BEGIN {Body of main program SYSTEMD}
  ...
  END.
```

Then, compiling the batch stream of the library of modules, on file, voln:SYSLIB4.PAS, with the CSS invocation:

```
*PASCOMP.CSS voln:SYSLIB4,PR1:,SU MA LI CR,PR1:,120
```

produces the linkable object code of the external modules, P1, P2, P3, F1, F2, and F3 on the file, voln:SYSLIB4.OBJ.

Compiling the main program on file, voln:SYSTEMD.PAS, with the CSS invocation:

```
*PASCOMP.CSS voln:SYSTEMD,PR1:,SU MA LI CR,PR1:,120
```

produces the compiled object code of the main program on file, voln:SYSTEMD.OBJ.

Then, the following command input to Link includes the main program on the file, voln:SYSTEMD.OBJ, with the library of external modules on file, voln:SYSLIB4.OBJ, and establishes the user task on file, voln:SYSTEMD.TSK.

```
>OPTION FLOAT,DFLOAT,WORK=(800,40000)
>INCL voln:SYSTEMD.OBJ       {include main program object}
>LIB voln:SYSLIB4.OBJ        {edit module object library}
>LIB voln:PASRTL.OBJ         {edit the Pascal RTL}
>MAP PR1:
>BUILD voln:SYSTEMD.TSK      {build user task}
>END
```

Then, the user task is available on file, voln:SYSTEMD.TSK, and ready for loading, lu assignments, and execution.

# PART II
# LANGUAGE REFERENCE

# CHAPTER 2
# LANGUAGE CONCEPTS AND SYNTAX GRAPHS

## 2.1 LANGUAGE CONCEPTS

This chapter defines several basic concepts in the language of Pascal that will be used throughout the remainder of the document. Chapters 2 through 9 compose the language reference section of the Pascal manual.

### 2.1.1 Blocks

A Pascal program or separately compilable external module is organized into a heading, a logical block, and a terminator (the period character). A procedure or function is organized into a heading, a logical block and is terminated by a semicolon. Each block may contain:

- declarations section (contains declarations of names) including routine declarations creating other blocks
- body (executable statements)

The declared names and routines and the executable statements within a body "belong" to the block in which they are contained.

Chapter 4 details an introduction to program structure, and the syntax of blocks, declarations, and the body of a block. Chapter 9 covers the program, module, procedure and function headers and associated concepts in detail. A brief overview follows.

The outermost block of a compilable program unit must be headed by a PROGRAM header, and its block declarations are visible over the entire program unit, even within the blocks of its routines. However, the main program declarations are not visible to the code of an external Pascal MODULE, and the declarations of a MODULE are not visible to the program. Entities are transportable in and out of a module only as value or variable parameters through its module parameter list. Often user-specified type-identifiers are required in a module parameter list, for its parameters, so type-identifier declarations may precede a module header in a type-definition part of a prefix.

This implementation of Pascal additionally allows a prefix of constant, types, and object supported routines to be declared prior to a PROGRAM or MODULE header. See Chapter 9.

This implementation of Pascal allows not only internal routines to be declared in a program, but also external procedures or functions, with the directive EXTERN or FORTRAN replacing their block in the declaration. Separately compileable Pascal MODULEs, or CAL written routines using Pascal calling conventions, may have their object linked to the main program for those declared external routines having had their blocks replaced with the directive EXTERN. FORTRAN produced code, or CAL routines using FORTRAN calling conventions, may be linked to the main program for those declared external routines having had their blocks replaced with the directive FORTRAN.

Within a Pascal program, all user written procedures and functions are blocks (or directives) headed by a PROCEDURE or FUNCTION header. Once named by declaration or defined with their own block, routines may be called into execution from the body of their own block or from a block on the same or lower level of declarations.

When procedures or functions are declared within other procedures or functions, such definitions are said to be "nested". See Section 9.6.7 on nesting routines. As they belong to the block in which they are declared, they can be called from the body of their block, or from the blocks of other routines on the same or lower level of nesting but inner nested routines cannot be called from outer blocks.

Pascal also allows routines to recursively call themselves. See Section 9.6.8 on recursion.

### 2.1.2 Identifiers

A noteworthy characteristic of Pascal is that every name that a programmer is going to use in the program must first be declared unless it is a predefined Pascal identifier. The reserved word symbols that constitute the Pascal language cannot be used as identifiers. Also, several predefined identifiers are available as the names of constants, types, file-variables, and routines (see Section 3.3.3).

A declaration introduces a user-specified name as an identifier and a definition (or its meaning). The definition determines how the name can be used in the program, and where, with the same meaning. This name is called an identifier. The definition is applicable to the name over a certain part of the program called the scope. To say that the identifier is "visible" in "scope" means that its reference in subsequent declarations and executable statements of a block, use it with the intended meaning established at its point of definition by declaration.

User-specified identifiers can be redefined within more tightly binding scopes. The predefined identifiers can also be redefined but then they are no longer available with their assumed standard meanings. See "scope" below.

### 2.1.3 Scope

A scope is a region of a program code in which an identifier is used with a single meaning. A user-specified identifier must be introduced before it is used. (The only exception to this rule is a pointer-type declaration, as that declaration may refer to a user-specified target type-identifier that has not yet been defined.)

Once an identifier is introduced by declaration, it has a scope established by whichever block in which it became introduced. An identifier can only be introduced once within the outermost declarations of a block with one meaning or else it receives a diagnostic error message: IDENTIFIER DECLARED TWICE. However, the same identifier can be re-introduced with another meaning in another inner block, or an inner scope of a record-type definition introducing field names.

A scope, meaning area of VISIBILITY, spans either a program, module, routine block, record-type definition, or a WITH statement. A PROGRAM, MODULE, or routine PROCEDURE or FUNCTION header establishes a block in which identifiers are introduced by declaration, thereby giving each identifier its scope; the scope of the entire block. A record-type definition creates another scope within the declarations part of a block, and closes upon its completion. A WITH statement creates another scope within the body of a block and closes upon its completion. A WITH statement, by specifying a record variable-selector, allows the fields of that record's type to be referenced by their names established in a record-type definition, without preceding each field-reference with the name of the particular record variable-selector. However, the identifiers used as the record-variable selector of a WITH statement must be visible in scope in order to be specified in the WITH statement as record variable selectors.

In general, if the declaration of an identifier is visible at all in a block, and the same identifier has subsequent re-declarations also visible in scope, at the point of reference, the identifier will be taken to mean that declaration carrying the more tightly binding scope. The names of fields introduced in a record-type definition in a type definition part can be made visible by the WITH statement; and a reference to an identifier (of a field) within a WITH statement means that field even if there were global or local variable identifiers (visible in scope to the WITH statement) of the same name. The expansion of scope introduced by a particular WITH statement ends (or closes) at the end of that WITH statement. A nested WITH statement within another WITH statement creates another scope within a scope. Also, within one WITH statement, reference to more than one record variable selector is possible, and the hierarchy of nested scopes introduced is the order in which the sequence of record variable selectors are listed.

The scope of a declaration of an identifier is determined by the following rules:

1. A declaration is visible to the entire block to which it belongs, from the point of its introduction.

2. The block created by the declaration of a routine also defines and makes visible the name of that routine within the immediately enclosing block.

   That is, the name of a routine declared in the outermost block of a program is visible to itself, subsequent declarations, and the body of the program, but the identifiers introduced inside the routine are not visible to the program. Likewise, the name of a nested routine is visible to the enclosing block in which it is declared, but not the identifiers introduced within the nested routine. The name of a nested routine is not visible to any block outside its immediately enclosing block.

3. If a declaration is visible within a block, then the identifier is visible with that declared meaning within any routine that belongs to that block, unless it is overridden by another declaration of the same identifier within that routine (a more tightly binding scope).

   Stated informally, declarations are freely imported into a routine, but the reverse is not true.

4. On the other hand, declarations are never exported from a routine block. That is, a declaration within an inner block is never visible outside the block. The scope of such a declaration is entirely contained within that block.

When a scope is defined within another scope, we have nested outer scope and inner scope. As identifiers can be redeclared in different scopes, a reference to such an identifier from a point which can see several scopes poses the problem of which one is meant, at the place of reference. In this case, the inner meaning applies following the declaration in the inner scope, and the outer meaning applies in the outer scope.

Ambiguity exists when, within a scope, an identifier declared in an outer scope is first referenced and then redeclared. In the current implementation such a reference will use the outer declaration without generating a diagnostic error. If the reference was intended, however, as a forward reference (as in a pointer type declaration), the incorrect result will be obtained. References preceding intended redeclarations, particularly using an identifier which is already visible in scope as an intended target-type forward reference, are considered illegal and should be avoided.

Following is the hierarchy of scopes:

```
prefix                                prefix
(program (record-types)               (module (record-types)
 (nested routines(record-types)        (nested routines(record-types)
  (WITH statement                       (WITH statement
   (nested WITH statements))))           (nested WITH statements))))
```

Therefore, within each scope, only certain identifiers are or become visible in scope.

A program or module can use as a reference:

(1) any predefined Pascal identifier;

(2) constant, type, and routine identifiers introduced within either its respective program or module prefix;

(3) constant, type, variable, or routine, identifiers declared in the outermost declarations section of either their respective program or module outermost block, i.e., those identifiers global in scope, to the compilation-unit.

A program cannot use the external file names in its header, as file-variables until they are declared and re-introduced as file-variables. As a program and module establish mutually exclusive global scopes for themselves as compilation-units, a program does not have access to the declarations of a module. Linkage from the program to the module must be directed by an EXTERN directive and communication between the two can only be transacted through corresponding parameter list declarations and appropriate invocations of the module.

From the program's point of view the module is an external routine and only has access to the name of the module from its EXTERN declaration, not the parameter identifiers of that EXTERN declaration.

From the module's point of view, the scope of the program is not visible. A module does not have access to the global declarations of the program, nor its prefix.

A module can additionally use the value or variable parameter identifiers introduced in its module parameter list. A module cannot use the external file-variables of a program unless they are passed to the module as variable-parameters.

Although the outermost declarations of a module are global in scope to the module as a compilation-unit, note that a variable declaration in a module is not a global variable, but a local variable.

A routine can use as a reference:

(1), (2), (3) defined above, respective to whether contained in a program or a module compilation-unit; and

(4) all identifiers introduced within the routine itself and by its outer (enclosing) routines. This includes its own name and all identifiers introduced as value and variable parameters and the local constant, type, variable or routine declarations in the declarations part of the routine block.

A record-type definition can use as a reference:

any constant-identifier, or type-identifier from:

(1), (2), or (3) defined above, and

those constant-identifiers or type-identifiers from:

(4) if the record-type definition is within a routine block.

A WITH statement can use in its references:

(1), (2), (3) defined above, and

(4) defined above, if within a routine body, and

(5) all identifiers made visible by the WITH statement itself and those made visible by its enclosing WITH statements, if it is a nested WITH statement. This includes all field identifiers introduced in the record definition of the type of those record-variable selector(s) listed in the WITH statement. The WITH statement opens a scope, in which the field identifiers can be referenced, without tediously repeating the record variable selector. When more than one record-variable selector is listed in one WITH statement, the hierarchy of newly opened scopes procedes from left to right, opening the scope of the first listed record-variable's type, then the second, and so on. See Section 7.3.7 for an example on the WITH statement.

### 2.1.4 Constants

Some of the data used in a Pascal program are fixed values and cannot change value during execution of the program. These unchangeable values are called "constants". Constant values can be literally represented, while coding, and these values are called literal-constants (see Section 3.3.4). Constant values may also be given names that are identifiers defined with the value of the constant in the constant declarations part of any block (see Section 4.2.2). The syntax of the language construct, "constant", is given in Section 5.1.

Usually, the constants to be used in a program can be collectively defined in the outermost block of the main program. If an identifier is defined to be a constant in an inner nested routine, that identifier may not be available to an outer block.

### 2.1.5 Variables

A variable is a datum that can have its value changed during the execution of a program.

A variable may be declared with a name in a variable declarations part of a block (see Section 4.2.4), or may be dynamically created with the predefined procedure NEW, (see Section 5.3.11). The programmer declares a variable to be of certain type, and that variable can assume values of its defining type, or of an assignment-compatible type. The basic operations on a variable are assignment of a new value to it and reference to its current value. Variable data is of four kinds: global, local, parameter, and dynamically allocated. When a global variable comes into existence by declaration, or when a declared local variable comes into existence by a routine invocation, or when a dynamic variable is created, it has no defined value until a value is assigned to it in an executable statement. Within the block of a routine-definition, the value and variable parameter identifiers can be assumed to come into existence with the actual argument data passed to them in the routine invocation.

### 2.1.6 Global and Local Variables

If the declaration of a variable is in the outermost block of the main program and not within a routine, nor within a module, is said to be global. A global variable exists during the entire execution of a program, and the scope of its identifier spans the entire program.

If the declaration of a variable name is inside a routine, that variable is not directly visible outside the routine, but is visible not only to the body of the routine itself, but to also any nested routine contained in the declarations of that routine.

If the declaration of a variable name is inside the outermost block of a separately compileable module, the identifier is global in scope to the entire module, but the variable is actually a local variable which comes into existence only upon invocation of the module from its calling code.

All declared local variables come into existence when the routine is invoked, as do the routine's dummy parameters, as place holders for the actual argument data which will be passed upon invocation of the routine. Parameter identifiers can be value parameters, variable parameters, or formal routine parameters.

As routines can be invoked recursively, there can be several copies of the same set of local variables and parameters in existence at one time. When there are no indirect invocations of routines, any use of the name of a local variable refers to the most recently created copy of that variable.

To describe what happens when a routine is invoked indirectly, that is, as a result of being passed to a formal routine, it is useful to define the concept of the environment of an invocation. This is the set of data, referred to by nonlocal names, that the routine can use, and is called the environment of the routine at the time of its invocation. Briefly, when a routine name is passed to a formal routine parameter, it takes its environment along with it.

When a routine finishes execution and returns control to the place where it was invoked, the most recently created set of local variables is destroyed.

### 2.1.7 Dynamically Allocated Variables

A program can also create dynamically allocated variables. These variables are not automatically created and destroyed, as an effect of the flow of control among procedures, but by explicit creation and disposal commands within the program. Such variable data are not referred to directly with their own identifiers, but as the targets of pointer variables. See Section 5.3.11 on the pointer-type.

### 2.1.8 Data Types

Every datum that is manipulated by a Pascal program is of a particular type. The type determines the possible values for the datum, the legal operations on it, and the meaning of the datum to the programmer. Pascal has several predefined data types and allows the programmer much freedom to create new user-specified types (see Section 5.3).

User-specified data type definitions are declared in a type-definitions part of a block (see Section 4.2.3).

Data type declarations and the rules of the language are designed to make it easy for the programmer to guarantee that only meaningful operations are performed on any datum. However, this requires careful application of the rules of type-compatibility in Pascal.

### 2.1.9 Data Type-Compatibility and Conversion

There are two degrees of compatibility in Pascal: "identity", and "assignment compatibility".

In Pascal, data of user-specified types usually require "identity" of type to be compatible for most operations such as relational structured comparisons, or assignment. The scalar literal-constants, data of the predefined scalar type-identifiers, and scalar components of user structured-types can usually be compatible for applicable operations amongst their respective scalar types in the broader sense of "assignment compatibility". Literal-strings of only the same fixed length can be assigned to string-arrays, but literal-strings of any length can be passed to value parameters of string-type.

Specifying actual argument data in a routine call also requires Pascal type-compatibility rules to be adhered to. Arguments passed to VAR variable parameters, require "identity" of type. However, arguments passed to value parameters, only require "assignment compatibility" of type.

The strongest degree is "identity", where the two types of two different datum are identical. Two data are "identical" in type if the fact that the types are equal can be ascertained without looking at the internal structures of the declarations.

Identity of type can be established with a declaration such as: VAR A,B,C:T; within a single group variable declaration whether or not T is a user-specified type or type-identifier, making A, B, and C of identical compatible types. However, if T were not a type-identifier, no other data could be made "identically" compatible to A,B, or C.

Identity of type can also be established declaring type-identifiers such as TYPE T1 = type; and separately declaring VAR AA:T1 and VAR BB:T1; making AA identically compatible to BB.

Also, it follows from type declarations like TYPE "T2 =T1"; where T2 and T1 are both type-identifiers, that VAR CC:T1 makes CC identically compatible to AA and BB. If T1 were not a type-identifier, CC, would not be compatible.

A broader concept is "assignment compatibility". This definition is more inclusive than that of "identical".

A datum of one date type is "assignable compatible" to another if, with appropriate conversion of machine representations when necessary, the assignment of an expression of the former type to a variable of the latter type is meaningful and allowed by the language.

For example, integers and reals do not have similar internal machine representations, nor similar storage sizes, so they are not of "compatible" types in the strictest sense of "identity". However, in Pascal, integers are "assignment compatible" to reals; but reals are not "assignment compatible" to integers. An integer value may be assigned to a real variable or passed to real value parameter, but a real value may not be assigned to an integer variable; nor passed to an integer value parameter. However, as VAR parameters always require "identity" of type of their arguments: VAR parameters of BYTE, SHORTINTEGER, INTEGER, REAL, SHORTREAL type can only be passed "identically" typed BYTE, SHORTINTEGER, INTEGER, REAL, SHORTREAL arguments; respectively. Integers and reals may be mixed in expressions yielding real results, so such an expression is not assignable to an integer variable; nor passable to an integer value parameter.

See Section 6.2 on type compatibility.

See Section 6.3 on data conversions.

### 2.1.10 Selector

A selector is a means of specifying one variable or a component part of a variable. A variable identifier is a selector, specifying the entire global or local variable to which it refers.

A dynamically allocated variable can be selected as the target of a pointer. An element of an array, can be selected with an array-component selector. The field of a record can be selected with a record-field selector.

Note that access to a particular datum imbeded in a highly complex structure, e.g., a dynamic targeted record containing arrays of records containing other records, can require the successive application of several kinds of selectors. See Section 5.4.

### 2.1.11 Expressions

An expression is, informally, a sequence of one or more operand factors and/or operators that can be interpreted to yield a defined value.

Literal constants or constant identifiers, variable-selectors of defined variables, set-constructors, as well as the values returned by function calls can be operands which evaluate to yield defined values, even without operators, and are expressions. From these factors of expressions, more complex expressions can be formed, using the operators that are part of the language.

Expressions can be parenthetically nested. Only certain operations are legal with certain types of data, so various operators require certain types of operand factors. Amongst the various operators, certain rules of precedence are established differing from FORTRAN, whereby parenthetic differentiation is explicitly required, to establish correctness and meaning. The operators that apply to a given data type, and the results of such applications, are summarized in Table 6-6.

There are several kinds of expressions, which yield defined values of a particular type. They are arithmetic expressions which yield a numeric value, relational expressions which yield a Boolean value, logical (or Boolean) expressions which yield a Boolean value, set expressions which yield a set value, and set test membership expressions which yield a Boolean value. Descriptions are given in Chapter 6.

### 2.1.12 Statements

The instructions that manipulate the data of a program are written and expressible by executable statements. Executable statements can include such actions as computation of expressions, assignment of their values to variables, transfer of execution control, routine-invocations, logical decision making, I/O, and the testing of conditions. A statement can be simple or structured; i.e., it can contain other statements. Pascal includes a rich set of constructs for statements with complex patterns of structuring conditional and repetitive action.

A compound statement is a sequence of statements, separated by semicolons, bracketed with the keywords BEGIN and END. A compound statement is a mechanism for collecting a group of simple statements or other compound statements into a single entity. Most importantly, the compound statement serves to form the body of a block, and can also be used anywhere in the structured statements syntactically where they require one statement.

See Chapter 7, for the syntax of executable statements of Pascal; Chapter 8, for Pascal I/O procedure-call statements; and Sections 10.3 and 10.4 for additional extensions afforded by the Perkin-Elmer Prefix and SVC routines. Chapter 9 details the program, module, procedure, or function header statements; which are not executable but rather the delineators of Pascal blocks.

## 2.2 UNDERSTANDING SYNTAX GRAPHS

The syntax of a programming language determines what is a correctly written program. It consists of an orderly arrangement of the language elements (see Chapter 3) and other language constructs (depicted by syntax graphs throughout the language reference part of this manual). Adherence to the syntax determines the way in which words or symbols are put together to form constructs to communicate meaning. Programming in Pascal begins with an understanding of its syntax.

There are two levels of syntax -- context-free syntax and context-sensitive syntax. The former can be likened to the vocabulary and grammar of a language, and the latter can be likened to the semantics of a language.

Context-free syntax, graphically representable, specifies the rules by which one construct can be viewed as a combination of smaller constructs.

Context-sensitive syntax specifies the rules by which one construct's meaning in one place of the program is dependent upon other relations of another construct in another place of the program. For example, the rule that every name must be defined before it is used is a context-sensitive syntax rule. These rules are defined in the text and not necessarily by the syntax graphs.

A syntax graph is a representation of fixed symbols of the language and language constructs connected by arrows that indicate the correct sequences of symbols and constructs. For example, one construct in Pascal is the compound statement as defined by the following syntax graph:

**Compound-Statement**

```
---> BEGIN ---> statement ---> END --->
            ^               |
            |               |
            <----   ; <-----v
```

This graph is interpreted as follows: The underlined header, **Compound-Statement**, defines the name of the language construct being presented by the graph. This context-free syntax graph defines the basic word symbols, special-characters, other elemental constructs, and the sequence in which they are to be correctly written to form the larger construct, the compound statement.

In the syntax graphs of this manual, the basic word symbols of the Pascal language are represented by capitals, such as: BEGIN and END; and by special characters, such as: the semicolon (;). Other language constructs are represented inside the graphs by their names written in lowercase letters, such as: statement. The sequence of correct arrangement is represented by directional arrows. Sequence normally flows from left to right, except where repetition of a construct and basic symbol is expressed by looping back from right to left and rejoining the original direction. That is, the construct: statement, and the following semicolon is a repeatable sequence.

Alternative selections at any one serial position in the flow of sequence is represented by vertically parallel constructs. For example, the construct identifier is defined by:

### Identifier

```
---> letter ----------------------------->
                  ^                 |
                  |<-- letter <-----|
                  |                 |
                  |<--- digit <-----|
                  |                 |
                  |<--underscore<---V
```

This means that the construct, identifier, consists of a letter, followed by any of four alternatives. The alternatives, after the first letter, are the empty condition or nothing, another letter, or another digit or the underscore character. Once the empty condition is taken, the construct is ended. However, the sequence of taking the path of either letter, digit, or underscore is repeatable. The limit of repetition, not represented in the graph, is an implementation-defined quantity; and the repetition may continue until a total number of 140 letters, digits, or underscores constitute the identifier.

If a construct that is being graphically defined includes itself as one of its elements, syntactical recursion is being expressed. For example, the construct, factor, includes the construct, factor, as shown in an excerpt from a syntax graph below:

### Factor

```
    .                       .
    .                       .
    .                       .
    |                       |
    |--> ( expression ) --->|
--->|                       |--->
    |-->NOT---->factor----->|
    |                       |
    .                       .
    .                       .
```

That is, one of the alternatives in forming a factor of an expression is the option of using the keyword NOT (for logical negation), followed by a recursive use of the construct, factor. For example, the factor NOT TRUE contains the factor TRUE, a Boolean constant. The factor NOT (A>=B), contains the parenthesized relational expression A>=B as a factor which yields a Boolean result.

Although "factor" is a subconstruct of an "expression", "factor" also contains the construct "expression" but surrounded by parentheses. That is, complete expressions contain simple expressions, which contain terms, which contain factors, which again in turn can contain parenthesized expressions. The syntactical recursion expressed here is that expressions may be nested within other expressions by enclosing them in parentheses; and in some cases, parentheses are required to communicate the explicit meaning of an expression, especially when the normal Pascal operator precedences make an expression ambiguous in meaning or illegal due to the type of operand factors required by certain operators.

A syntax graph signifies the possible components of writing program text to form a certain language construct. The graph looks like a flowchart and can be read as such. A path through a syntax graph can indicate a possible expansion of the construct being defined, and reference to the graphs of other constructs might be required.

# CHAPTER 3
# LANGUAGE ELEMENTS

## 3.1 INTRODUCTION

A Pascal program is written, or programmed, requiring only certain characters, called the character set of PASCAL. The character set of Pascal is a subset of the American Standard Code for Information Interchange (ASCII). However, Pascal can manipulate, as data, the full untagged ASCII character set.

The source code of a computer program written in Pascal also consists of a certain lexical combination of basic symbols and separators which enables Pascal source code to be written rather freely, without stringent horizontal or vertical restrictions.

The Pascal character set, basic symbols, and separators (including Pascal comments) are presented in this chapter as language elements.

## 3.2 THE CHARACTER SET OF PASCAL

Computer programs written in Pascal are encoded using a subset of ASCII.

The character set required by the language of Pascal, in which Pascal programs are written, is as follows:

| | |
|---|---|
| Letters: | A B C D E F G H I J K L M<br>N O P Q R S T U V W X Y Z<br>a b c d e f g h i j k l m<br>n o p q r s t u v w x y z |
| Digits: | 0 1 2 3 4 5 6 7 8 9 |
| Hexadecimal digits: (only used after #) | 0 1 2 3 4 5 6 7 8 9 A B C D E F<br>a b c d e f |
| Special characters: | & ' ( ) * + , - . / :<br>; < = > [ ] ^ # @ { } _ |
| Space character | |

It is possible to write programs with equipment that does not support lowercase letters.

The character set that Pascal can manipulate, distinct from that which is needed to write programs, is the full set of 128 untagged ASCII characters. The ASCII character set can be represented in three ways: graphically or by name, by the character's ordinal decimal value, or in its hexadecimal coded form. Table 3-1 shows the full untagged ASCII character set of 128 characters.

TABLE 3-1 THE ASCII CHARACTER SET

| | MSD 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|
| LSD 0 | nul | dle | | 0 | @ | P | ` | p |
| 1 | soh | dc1 | ! | 1 | A | Q | a | q |
| 2 | stx | dc2 | " | 2 | B | R | b | r |
| 3 | etx | dc3 | # | 3 | C | S | c | s |
| 4 | eot | dc4 | $ | 4 | D | T | d | t |
| 5 | enq | nak | % | 5 | E | U | e | u |
| 6 | ack | syn | & | 6 | F | V | f | v |
| 7 | bel | etb | ' | 7 | G | W | g | w |
| 8 | bs | can | ( | 8 | H | X | h | x |
| 9 | ht | em | ) | 9 | I | Y | i | y |
| (A) | lf | sub | * | : | J | Z | j | z |
| (B) | vt | esc | + | ; | K | [ | k | { |
| (C) | ff | fs | , | < | L | / | l | \| |
| (D) | cr | gs | - | = | M | ] | m | } |
| (E) | so | rs | . | > | N | ^ | n | ~ |
| (F) | si | us | / | ? | O | _ | o | del |

In Table 3-1, the graphic representation of the uppercase character M is simply M in the chart. The uppercase character M is represented in hexadecimal as 4D. That is, the most significant digit (MSD) is 4 and the least significant digit is (LSD) hexadecimal D. The ordinal value or ordinal number of a character (ch) is determined with the formula:

ORD(ch) = 16 * MSD + LSD

Hexadecimal MSD and LSD are obtained from Table 3-1. For the character M, the ORD('M') = (16 * 4)+ 13, which is decimal 77. The characters with ordinal numbers 0 through 31 and 127 are unprintable ASCII control characters. Because they are unprintable, these characters are listed by abbreviated names. Note that the character with ordinal number 32 (20 hexadecimal) is the blank or space, and the character with ordinal number 95 (5F hexadecimal) is the underscore.

A complete list of the ASCII character set, showing all three representations: (graphic, hexadecimal and ordinal) will be found in Appendix H.

This implementation of Pascal recognizes uppercase and lowercase letters interchangeably in program text; e.g., in reserved word symbols, identifiers, real numbers or hex integer constants. That is, Begin is equivalent to BEGIN or begin; and the identifier "TRUE" is equivalent to "true". Also, in real numbers the exponential letter "e" is equivalent to "E"; and the integer hex constant "#12abcdef" is equivalent to "#12ABCDEF".

However, in comments and string literals of character data, uppercase letters are distinct from lowercase letters.

A character is defined by the graph:

### Character

```
    |---> graphic character --->|
    |                           |
--->|                           |--->
    |---> control character --->|
```

A graphic character is further defined by:

### Graphic-Character

```
---------> special character --------->
     |                          |
     |---------> letter --------->|
     |---------> digit  --------->|
     |---------> space  --------->|
```

A character can be either a graphic character or a control character. There are 95 graphic characters and 33 control characters in the 128 characters of the ASCII set.

The special characters of Pascal (required to write Pascal code) have fixed meanings to the language (when not within a string or comment); as distinct from the special characters of ASCII, which are taken to represent themselves as graphic characters within strings and comments.

### 3.2.1 Graphic Characters

A graphic character is a printable character and can be classified as a special character, letter, digit, or the space character.

- The special characters of ASCII are:

  ! " # $ % & ' ( ) * + { } [ ] , - . / : ; < = > ? ^ @ _ | ~ \ `

- The letters of both ASCII and the Pascal character set are:

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

  a b c d e f g h i j k l m n o p q r s t u v w x y z

- The decimal digits are:

  0 1 2 3 4 5 6 7 8 9

The space character, although printed as a blank, is also considered a member of the graphic characters.

### 3.2.2 Control Characters

In this implementation of Pascal, an unprintable control-character in the ASCII character set can be represented within strings by its ordinal value enclosed within delimiting symbols. The ordinal values of the control characters can be computed from Table 3-1 or found in Appendix H. A control-character construct in Pascal is represented by an unsigned integer of from one to three decimal digits enclosed in the delimiting symbols (: and :) as follows:

<u>Control-Character</u>

```
---> (: ---> digit ---> :) --->
         ^          |
         |          |
         |<---------V
```

The valid range of one or more digits is from 0 to 127, decimal inclusive, but this representation is only necessary for control characters whose ordinal values are 0 to 31 and 127.

For example, the control character "ff" listed in Table 3-1, with hexadecimal value X'0C', is the formfeed and can be defined in the constant declarations part as:

```
    CONST       FF='(:12:)';
```

Note that any sequence of characters within a string which forms the construct control-character is taken as such and not as individual graphic-characters. See examples in Section 3.3.4.

## 3.3 BASIC SYMBOLS

A program written in Pascal can be viewed as a combination of symbols and separators, where a symbol is defined by the syntax graph:

Symbol

```
-------> special symbol -------->
    |                      ^
    |---> word symbol ---->|
    |                      |
    |----> identifier ---->|
    |                      |
    |      literal         |
    V----> constant ------>|
```

and a separator is defined by the syntax graph:

Separator

```
--------------------> space ------------------------>
    |                                          ^
    |                                          |
    |--------------> new line ---------------->|
    |                                          |
    |----> (* --> comment-sequence --> *) ---->|
    |             not including *)             |
    |                                          |
    V----> { ---> comment-sequence ---> } ---->|
                  not including }
```

where the syntax of comment-sequence is:

Comment-sequence

```
 ------------------------------------>|
 ^                                    |
 |    |---> graphic-character --->|   V
----->|                           |----->
   ^  |------>  new line  ------->| |
   |                                |
   |<-------------------------------V
```

A symbol may be a special symbol (see Section 3.3.1) or a "basic symbol". The basic symbols are the word symbols (see Section 3.3.2), identifiers (see Section 3.3.3), or literal constants (see Section 3.3.4).

A word symbol is terminated by any character other than a letter. An identifier is terminated by any character other than a letter, digit, or the underscore of an identifier. A literal constant is ended upon the first character not part of its syntax. Therefore, if two word symbols, identifiers or constants appear in succession, they must be separated by a separator or special symbol; in order to distinguish them as separate entities.

A separator may be a space, a new line, or a Pascal comment which is a comment-sequence enclosed in delimiters, as shown by the previous two syntax graphs.

At least one separator or special symbol (excluding the delimiters for comments, control-characters, hex constants, or strings; or underscore) must separate any two consecutive basic symbols such as the word symbols, identifiers, or literal constants.

That is, we can write A+B but not BANDC, which is taken as an identifier.

There may be an arbitrary number of separators between any two of the basic symbols. That is, an arbitrary number of spaces, new lines, or comments are allowed between any two of the basic symbols.

These are the ground rules that allows Pascal code to be written freely horizontally without columnar restrictions and vertically continuable without restrictions imposed by concern for continuation line formats.

However, separators may not occur within any symbols, and all symbols may not be split across source line record boundaries.

This means that although comment-sequences may extend across source line record boundaries the constants, particularly notable, the literal string constants may not be split across source lines.

Note that a comment which begins with { must end with }. A comment which begins with (* must end with *). The intermingled pairs (* and } or { and *) are not allowed and not recognized to delimit a comment-sequence. This means that any symbol occuring after either beginning comment delimiter is taken as part of a comment-sequence as graphic characters.

Also note, that different from symbols, lower case letters in a comment-sequence are not mapped into upper case letters.

Any number of comments may occur prior to the first symbol of a compilation-unit, i.e., either a PROGRAM or a MODULE.

Example:

```
    (* THIS IS A COMMENT AHEAD OF THE PROGRAM *)

    {THIS IS ANOTHER COMMENT AHEAD OF THE PROGRAM}

    (*Ditto, but this comment was written with small letters*)

{}(**){This line has 2 comments first with no comment-sequences}

    PROGRAM STARTER;
      VAR a,b:INTEGER;  {Symbols a and b also mean A and B below}

          {   This comment contains

                    blank new lines

                          and extends across

                                  several lines   }

    BEGIN

      A := 3;     {CAUTION: This comment mistakenly attempts to
      B := 5;       cross lines causing B := 5; to be consumed}

      WHILE A<>B DO{Comment between symbols DO and A}A:=A+1;

    END.{RESTRICTION: Comment after termination must end on line}
```

Note that no compile-time diagnostics can detect such unintended mistaken comments consuming code {B:=5; above} causing this program to execute ad infinitum at run-time or until A becomes B's undefined value; quite divergent from the anticipated two iterations expected of the WHILE statement at first glance.

Comments are further detailed as separators, with their additional use for specifying compiler in-stream options described, in Section 3.4.

### 3.3.1 Special Symbols

The special symbols of Pascal are a group of special characters and combinations of special characters. They have fixed meanings in the language, except where they are taken as graphic-characters when they appear within string constants (except for the control-character delimiters) and within comment-sequences.

Refer to Table 3-2 for the Pascal special symbols, their names, and their functions.

## TABLE 3-2 SPECIAL SYMBOLS

| SPECIAL SYMBOL | SYMPOL NAME | SYMBOL FUNCTION |
|---|---|---|
| + | Plus | Arithmetic operator for addition and unary plus; or set operator for union |
| - | Minus | Arithmetic operator for subtraction and unary minus; or set difference operator |
| * | Multiplier | Arithmetic operator for multiplication or set intersection |
| / | Division (real) | Arithmetic operator for real division |
| & | Ampersand | Alternative for the Boolean operator AND |
| = | Equal | Relational operator for comparison of equality |
| <> | Not equal | Relational operator for comparison of inequality |
| < | Less than | Relational operator for comparison of less than |
| > | Greater than | Relational operator for comparison of greater than |
| <= | Less than or equal | Relational operator for Less Than or Equal comparison or set Containment |
| >= | Greater than or equal | Relational operator for Greater Than or Equal compare or set operator for "Contains" |
| ( | Left parenthesis | Delimiter in many constructs e.g., to nest expressions, or to delimit a list of: files, parameters, arguments, field-lists in record-variants, or user-defined enumeration list |
| ) | Right parenthesis | Delimiter to end that which began with a left parenthesis |
| := | Becomes | Assignment operator for the Assignment and FOR statements |

| SPECIAL SYMBOL | SYMBOL NAME | SYMBOL FUNCTION |
|---|---|---|
| . | Period or decimal point | Program/Module Terminator or Field-selectors; real numbers |
| , | Comma | Delimiter in many constructs |
| ; | Semicolon | Delimiter in many constructs and is a statement separator |
| : | Colon | Delimiter in many constructs |
| .. | Range symbol | Subrange; Set-member-interval |
| [ | Left square bracket | Delimiter beginning set-constructor or for array index/subscripting |
| ] | Right square bracket | Delimiter ending a set-constructor or for array index/subscripting |
| ^ | Up arrow | Pointer-type beginning or target dereferencing symbol |
| @ | At sign | Alternative for up arrow |

| OTHERS | CHARACTERS NAME | SYMBOL FUNCTION |
|---|---|---|
| # | Pound sign | Hexadecimal constant beginning delimiter |
| ' | Single quote | Character string literal beginning/ending delimiter |
| (* | Comment begin | Alternative delimiter for comment beginning |
| *) | Comment end | Alternative delimiter for comment ending |
| { | Left brace | Delimiter for comment beginning |
| } | Right brace | Delimiter for comment ending |
| (: | Left parenthesis and colon | Delimiter for control character beginning |
| :) | Colon and right parenthesis | Delimiter for control character ending |
| _ | Underscore | May be part of an identifier |

### 3.3.2 Word Symbols

The word symbols are the keywords of the language and, as such, are reserved words. That is, the word symbols have fixed significances in the language, and their meaning cannot be overridden. The programmer cannot use these words for anything else except for what they mean, and where they are syntactically correct in Pascal language constructs. That is, even if no CASE statements were being used in a program, a datum could not be given the identifier "CASE". Table 3-3 lists the reserved word symbols.

TABLE 3-3 PASCAL WORD SYMBOLS

| RESERVED WORD SYMBOLS | |
|---|---|
| AND | MOD |
| ARRAY | MODULE |
| BEGIN | NIL |
| CASE | NOT |
| CONST | OF |
| DIV | OR |
| DO | OTHERWISE |
| DOWNTO | PACKED |
| ELSE | PROCEDURE |
| END | PROGRAM |
| EXTERN * | RECORD |
| FILE | REPEAT |
| FOR | SET |
| FORTRAN * | THEN |
| FORWARD * | TO |
| FUNCTION | TYPE |
| GOTO | UNIV |
| IF | UNTIL |
| IN | VAR |
| LABEL | WHILE |
| | WITH |

* Three of the reserved word symbols are further classified as directives, in that their use serves as a replacement of a block in a procedure or function declaration. They are EXTERN, FORTRAN, and FORWARD.

### 3.3.3 Identifiers

Identifiers are the names used by the programmer to denote particular entities, such as constants, types, variables, procedures, functions, modules, the program name and file names. An identifier consists of from 1 to 140 characters, where each character must be a letter, digit, or the underscore character, and the first character must be a letter. In an identifier, lowercase letters are equivalent to uppercase letters. An identifier may not cross the break from one source line to another. Note that no spaces are allowed within an identifier. However, the special character symbol, the underscore, "_", is allowed in Perkin-Elmer Pascal within identifiers for creating meaningful and easily readable complex names.

The syntax of an identifier is:

Identifier

```
-----> letter ------------------------->
                ^                  |
                |                  |
                |<----- letter <----|
                |                  |
                |<----- digit  <----|
                |                  |
                |<-- underscore <---V
```

Some identifiers are predefined, and others are defined by the programmer.

The context-sensitive restriction on the use of an identifier to reference an entity is that every identifier must be defined or declared as the name of the entity before it is used to reference that entity. Only one exception to this rule in Pascal exists, and that is that a pointer-type identifier may be bound to a target-type identifier before the target-type identifier has been defined (see pointer-types).

Several constants, types, procedures, functions, and two files have predefined definitions that are available to the program writer as predefined identifiers.

These predefined identifiers may be given new definitions overriding the predefined meaning by explicit user-written declarations of the identifiers, although the user is cautioned that the predefined meanings are then no longer available in any program redefining them. Table 3-4 lists the predefined identifiers in this implementation of Pascal.

Note that the pointer constant, NIL, is a reserved word symbol and not included here as a predefined identifier, i.e., its meaning cannot be overridden.

## TABLE 3-4 PREDEFINED IDENTIFIERS

| PREDEFINED CONSTANT IDENTIFIERS | PREDEFINED TYPE IDENTIFIERS | PREDEFINED ROUTINE IDENTIFIERS | PREDEFINED TEXT FILE IDENTIFIERS |
|---|---|---|---|
| FALSE | BOOLEAN | ABS | INPUT |
| MAXINT | BYTE | ADDRESS | OUTPUT |
| MAXSHORTINT | CHAR | CHR | |
| TRUE | INTEGER | CONV | |
| | REAL | DISPOSE | |
| | SHORTINTEGER | EOF | |
| | SHORTREAL | EOLN | |
| | TEXT | GET | |
| | | LENG | |
| | | LINENUMBER | |
| | | MARK * | |
| | | NEW | |
| | | ODD | |
| | | ORD | |
| | | PAGE | |
| | | PRED | |
| | | PUT | |
| | | READ | |
| | | READLN | |
| | | RELEASE* | |
| | | RESET | |
| | | REWRITE | |
| | | ROUND | |
| | | SHORTCONV | |
| | | SHORTEN | |
| | | SIZE | |
| | | SQR | |
| | | STACKSPACE | |
| | | SUCC | |
| | | TRUNC | |
| | | WRITE | |
| | | WRITELN | |

**NOTE**

* MARK and RELEASE are only available under the compile-time option HEAPMARK.

For example, logical data have the predefined type-identifier BOOLEAN and the predefined Boolean constant identifiers, TRUE or FALSE, available for its values.

Numerical data have predefined type-identifiers available such as: BYTE, INTEGER, SHORTINTEGER, REAL, and SHORTREAL and the constant identifiers MAXINT for the integer value +2147483647 and MAXSHORTINT for the shortinteger value +32767.

Character data has the predefined type-identifier, CHAR.

Files consisting of lines of formatted textual character data have the predefined type-identifier, TEXT; and two predefined files, INPUT and OUPUT, which are textfiles of type TEXT.

Several routines are available with predefined identifiers, such as the procedures for manipulating dynamic data structures, NEW and DISPOSE, or MARK and RELEASE (under HEAPMARK only); and the arithmetic functions for absolute, ABS, and square, SQR. These predefined routines are briefly summarized in Section 3.5.3.

Additional standard math functions, not having predefined identifiers, but available with external FORTRAN declarations, are sine, cosine, arctangent, exponential, square root, and natural logarithm (see Section 3.5.3 below).

For example, several legal user-specified identifiers are:

A
R5
B2_
D34AB
ITEM56
PROCESS_NEXT_ENTRY
CONDITION_RED
Sum_The_Elements
ORDER_ENTRY
THIS_IS_A_LEGAL_IDENTIFIER
MASTERFILE1
UPDATE_FILE
DELETIONS
EMPLOYEE_NUMBER
employee_number

Several illegal identifiers are:

| | |
|---|---|
| LABEL | {reserved word symbol, not an identifier} |
| NAME! | {special characters not allowed} |
| ILLEGAL-SEPARATOR | {hyphen not allowed, only the underscore} |
| SLOG | {special characters not allowed} |
| FOR(*COMMENT*)TRAN | {comments not allowed within identifiers} |
| ITEM ONE | {illegal space, taken as two identifiers} |
| 565798 | {numbers alone do not form an identifier} |
| 3APPLES | {first character must be a letter} |
| _DATUM | {first character must be a letter} |

### 3.3.4 Literal Constants

Another basic symbol is the literal constant, which literally represents constant values of the different data-types of Pascal (described in detail in Chapter 5). Briefly, they are the Boolean constant identifiers, TRUE and FALSE; the pointer constant, NIL; the unsigned integers; the unsigned real numbers; the character literal constants; and the literal string constants. Unsigned integers and real numbers may be optionally preceded by a special symbol, the plus or minus sign to form a "constant" or (signed) number.

An unsigned-integer literal constant consists of one or more consecutive decimal digits. An integer literal constant is of the type INTEGER, not SHORTINTEGER or BYTE, but these three types are all assignment compatible. An unsigned integer preceded by an optional sign forms a (signed) integer literal constant. Integer literal constants (implicitly signed) may also be represented in their hexadecimal form using the prefix symbol # and the hexdigits of their values. However, the hexadecimal constant cannot be output in a WRITELN statement, to be displayed as a hexadecimal; but rather its internal value will be output in a decimal representation. The value of a literal integer is the integral value of the string of digits and the optional sign in the usual decimal representation. A limitation on the value of an integer is imposed by any implementation; for Perkin-Elmer 32-bit machines, the extreme values are -(2**31) and (2**31)-1, for integers of type INTEGER.

A real literal constant designates a real number. An unsigned real number consists of one or more consecutive digits followed by either a decimal point and consecutive digits (optionally followed by an exponential representation) or by the letter E (denoting exponent) preceding an optional plus sign or the minus sign followed by one or more digits expressing an exponent of the power of 10. The lowercase letter "e" is equivalent to "E" in real numbers. The unsigned-real-number literal constant is of type REAL, not SHORTREAL. The distinction is that SHORTREAL numbers provide less precision but take up less memory space. The limits on the magnitude of real literal constants are those of the hardware implementation of floating-point numbers (see Section 5.3.7).

A character-literal constant consists of a single character enclosed in single quote characters with one exception, the single quote itself. The single quote is represented as a character literal by four consecutive single quotes, ''''. A character literal is of type CHAR.

A character-string literal constant consists of one or more consecutive graphic-characters or control-characters enclosed in single quotes. It may not cross a source line record boundary. The single quote may be represented within a character-string literal by two successive single quotes. A character-string literal with N internal characters is of a structured array type:

```
    ARRAY [1..N] OF CHAR {if N>1}
```

If N=1 then it is of type CHAR. A character-string literal of N internal characters is assignment-compatible to any string-type array of length N.

The literal constants are graphically defined by:

### Literal-Constant

```
     |-----------> TRUE ------------>|
     |                               |
     |-----------> FALSE ----------->|
     |                               |
     |-----------> NIL ------------->|
     |                               |
---->|------> unsigned integer ----->|---->
     |                               |
     |------> unsigned real number-->|
     |                               |
     |------> character literal ---->|
     |                               |
     |------> character string ----->|
```

### Boolean Constant Identifiers

```
    |--->TRUE--->|
--->|            |--->
    |--->FALSE-->|
```

### Pointer Constant

```
----->NIL----->
```

### Unsigned-Integer or Digits

```
----->digit---->
  ^         |
  |<--------v
```

### Integer Constant (Signed)

```
            --> + -->
            ^       |
            |       v
--------------------------> unsigned-integer ---------->
   |        |       ^                               ^
   |        v       |                               |
   |        --> - -->                               |
   |                                                |
   |                                                |
   V------------> # ------->   hexdigit  ---------->|
                       ^                       |
                       |                       v
                       |<-----------------------
```

Unsigned-Real-Number

To represent the single character, quote, in a character-literal, we use ''''. Within a character-string constant, the single quote is obtainable by sequentially encoding two single quotation characters.

All characters within a literal string are taken as graphic characters, unless they form a control-character construct. This includes the comment-delimiter characters, as demonstrated by the last example, below.

Examples of character-string constants are:

```
'THIS IS A STRING'    {<--string with 16 internal characters}
'DON''T'              {<--string with 5 internal characters:
                          DON'T}
'(:7:)'               {<--ASCII control-character for bell;
                          ordvalue is 7}
'MESSAGE(:13:)'       {<--string with 8 internal characters,
                          last one is carriage-return}
'(*OOPS*){!}'         {<--string with 11 internal characters}
```

A literal string constant of length n, may be assigned to any string-type variable array of characters which is also of length n.

Also note, that as the control-character construct has a special meaning within strings, to explicitly write out the sequence of characters "(:13:)", one would have to break up the sequence in into multiple consecutive strings to avoid its interpertation as a control-character, e.g. instead of WRITE('(:13:)'); which produces a carriage-return one could write WRITE('('); WRITE(':13:)'); to produce the printable sequence: (:13:).

## 3.4 SEPARATORS AND COMMENTS

A separator is an indicator that one basic symbol is ending and another is beginning, as detailed above in Section 3.3. A separator can be a space character, a new line, or a comment.

A comment is any sequence of graphic characters, called a comment-sequence, enclosed within Pascal comment delimiting symbols. The comment-sequence itself can contain a new-line, i.e., cross source line record boundaries. The graphic-characters of a comment-sequence maintain the distinguishment of small letters from capital letters, i.e., small letters are not mapped into capital letters.

Refer briefly back to Section 3.3 for the syntax graphs of separators and comment-sequences where they were discussed in relationship to symbols.

The Pascal comment delimiting symbols can be the pair (* and *) or the pair { and }. The preferred delimiters are the paired braces, { and }. The matching closing delimiting symbol cannot appear as part of the comment-sequence (or the comment will end).

```
(* THIS IS A COMMENT*)
{THIS IS A COMMENT}
```

A comment may be of either of two forms as shown in the preceeding example. Note that a comment which begins with { must end with }. A comment which begins with (* must end with *).

The intermingled pair of { and *) or (* and } are not allowed nor recognized to distinguish a comment-sequence.

```
{comment with no or improper ending *) consumes remaining
 text or code as comment-sequence text, including END.
 unless/until a match to the beginning delimiter is found}
```

Therefore, comments may be "nested", but only within another comment, which uses the other pair of Pascal comment delimiters. For instance:

```
{  I := I + 1;   (*COMMENT*)
   J := J - 1;                }
```

is all one comment, removing the statements from compilation as code. When the braces are removed the assignment statements become statements and the (*COMMENT*) remains as a comment.

The user is cautioned against unintentional errors such as depicted below, for which the compiler generates a diagnostic syntax error depending on which syntax evaluation is in progress prior to encountering the beginning of the comment.

```
{This comment mistakenly attempts to {nest comments} }
(*Illegal(*nested comment*)may cause diagnostic errors*)
```

As at least one or more separators may occur between any two basic symbols, a comment may appear anywhere a space may appear, and many comments may appear in succession.

Separators can occur arbitrarily between symbols and identifiers, but they can not occur within symbols. For instance, applying these rules to comments:

```
FOR(*...*)TRAN
```

is the same as:

```
                    FOR       TRAN
```

and consists of the keyword FOR followed by the identifier TRAN.

The comment delimiters are not recognized as such when contained in a character string. Therefore, the character-string literal 'AB(*111*)BC' contains 11 internal graphic characters; and the string 'MORNING{STAR}' contains 13 internal graphic characters.

A comment-sequence may run on from one source line to the next, but note that the double-character (* or *) symbols may not contain any separator between the asterisk and parenthesis. Also, comments may extend across several source lines; with one exception. Any comments begun after the program/module terminator line containing " END." must end on that last line.

Example:

```
PROGRAM NAME(INPUT,OUTPUT);
  BEGIN
    ...
  END. {After terminator "." this comment mustn't trail on}
```

In this implementation, a special use of the comment constructors is defined for specifying in-stream options, which serve as instructions to control the compilation process. This special form of comment constructor is called an option-specifier comment and is of the form:

<u>Option-specifier comment</u>

```
---> { ---> option-string ---> } --->
```

or

```
---> (* ---> option-string ---> *) --->
```

where option-string is of the form:

<u>Option-string</u>

```
-------> $ ---> option-specifier ------->
   ^                                |
   |                                |
   |<------------   ,  <------------v
```

An option-specifier comment is a Pascal comment. The first nonblank character that follows the beginning comment delimiter is a dollar sign ($). Any comment so distinguished must only contain an option string; no intermixing of comment text and $option-specifiers are allowed.

The $option-specifiers are fully detailed in Chapter 1. Briefly, a representative sample of an option-string is:

```
     $BATCH,$ASSEMBLY,$MAP,$LIST,$CROSS,$SUMMARY,$NRANGECHECK
```

Enclosed within the comment delimiters, the option-specifier comment would be:

```
     {$BATCH,$ASSEMBLY,$MAP,$LIST,$CROSS,$SUMMARY,$NRANGECHECK}
```

Other examples of option-specifier comments are:

```
     { $INCLUDE (VOLN:FILENAME.EXT,NLIST,NCROSS)}
     { $EJECT }
     (* $NBOUNDSCHECK *)
     { $BEND }
```

## 3.5 PREDEFINED/STANDARD ROUTINES

The predefined routines available in Pascal have their identifiers listed in Table 3-4. These identifiers are also listed and further classified as either procedures or functions in Table 3-5.

TABLE 3-5 PREDEFINED ROUTINE IDENTIFIERS

| PROCEDURES | FUNCTIONS |
|---|---|
| DISPOSE | ABS |
| GET | ADDRESS |
| MARK | CHR |
| NEW | CONV |
| PAGE | EOF |
| PUT | EOLN |
| READ | LENG |
| READLN | LINENUMBER |
| RELEASE | ODD |
| RESET | ORD |
| REWRITE | PRED |
| WRITE | ROUND |
| WRITELN | SHORTCONV |
| | SHORTEN |
| | SIZE |
| | SQR |
| | STACKSPACE |
| | SUCC |
| | TRUNC |

The predefined routines are further detailed throughout the manual. Chapter 8 explains file handling routines, and Section 5.3 explains routines whose arguments or function values are related to particular data-types.

Briefly described and classified by area of applicability, the predefined routines are listed below. The procedures are presented by the procedure-call statements (with or without arguments) which invoke them. The functions are presented by the function-references (with or without arguments), which invoke them. Any function-reference may be used in an expression. Additional standard math functions, not having predefined identifiers, but available with external FORTRAN declarations, are sine, cosine, arctangent, exponential, square root, and natural logarithm (see Section 3.5.3 below).

### 3.5.1 File Handling Procedures

There are two genre of file-types in Pascal; either of which may be permanent external files outliving program-execution, or temporary internal files transacted with only during program-execution. A Pascal file is treated in Pascal I/O as a sequential open-ended collection of components, each of some component-type. A Pascal text file of characters allows ASCII formatted transactions, line-structured textual input/output, to be programmed. A text file is any file that has been declared as a file-variable of the standard type TEXT (i.e., VAR FILENAME:TEXT;), or the standard text files, INPUT and OUTPUT. Other file-types are detailed in Chapter 8.

| | |
|---|---|
| GET(f) | inputs a component from file f to file-buffer variable f^. |
| PUT(f) | outputs the value of file-buffer variable f^ to file f. |
| RESET(f) | initializes the file f to a read-only state; prepares for any queries on EOF(f) in case the file is empty; and if not empty, buffers the first component for the first READ/READLN. |
| REWRITE(f) | initializes the file f to a write-only state; positioning the file at its beginning. |
| PAGE | causes page ejection to standard text file, OUTPUT. |
| PAGE(f); | causes page ejection to text file, f. |

READ inputs data from the next component of a file. The file is left positioned at either the next component, or in the case of text files, at the first character not part of the formatted data-type being read into v. READ skips over any leading spaces or line-markers before numeric data on text files. When end of file is reached on the standard text file, INPUT, EOF becomes TRUE. When end of file is reached on any other file, EOF(f) becomes TRUE. READ procedure-call statement formats are:

| | |
|---|---|
| READ(v); | reads ASCII formatted data (with conversion for numerics) into variable selector v, from text file INPUT. |
| READ(v1,...,vn); | reads ASCII formatted data (with conversion for numerics) into variable selectors v1,...,vn from text file INPUT. |

| | |
|---|---|
| READ(f,v); | reads data into variable selector v, from file f. |
| READ (f,v1,...,vn); | reads data into variable selectors v1,...,vn from file f. |

In the previous two READs, if f is a text file, ASCII formatted data (with conversion for numerics) is read from text file f into each variable v, as described below for READLN. In the previous two READs, if f is a Pascal file which is not a text file, then the variables must be of identical-type to the component-type of the file for data to be read into them.

READLN inputs from a text file advancing that text file to the next line. When end of file is reached on the standard text file INPUT, EOF becomes TRUE. When end of file is reached on any other text file, f, EOF(f) becomes TRUE. READLN procedure-call statement formats are:

| | |
|---|---|
| READLN; | skips to next line on text file INPUT. |
| READLN(f); | skips to next line, as above, but on text file f. |
| READLN(f,v); | reads ASCII formatted data (with conversion for numerics) into variable selector v, from text file f. |
| READLN(f,v1,...,vn); | reads ASCII formatted data (with conversion for numerics) into variable selectors v1,...,vn from text file f. |
| READLN(v); | reads ASCII formatted data (with conversion for numerics) into variable selector v from text file INPUT. |
| READLN(v1,...,vn); | reads ASCII formatted data (with conversion for numerics) into variable selectors v1,...vn from text file INPUT. |

For text files, in the preceding READ and READLN examples, v denotes a variable selector of type CHAR, BYTE, INTEGER, SHORTINTEGER, or subrange thereof, or REAL or SHORTREAL. The n characters of a string may be read into the elements of a character array; i.e., at a character at a time into an array variable of type ARRAY[1..n] of CHAR. Although text file input is restricted to these simple types, automatic conversion of the textual input to internal form is performed. More complex data types in addition to these simple types may be read from nontext files; e.g., arrays or records.

WRITE outputs to the current component of a file. WRITE procedure-call statement formats are:

| | |
|---|---|
| WRITE(f,e); | outputs the expression value e to non-textfile f. The expression must be assignment-compatible to the component-type of the file f. |
| WRITE(f,e1,...,en); | outputs the expression values e1,...,en to non-textfile f. The expressions must be assignment-compatible to the component-type of the file f. |
| WRITE(p); | outputs the formatted write-parameter p to text file OUTPUT. |
| WRITE(p1,...,pn); | outputs the formatted write-parameters p1,...pn to text file OUTPUT. |
| WRITE(f,p); | outputs the formatted write-parameter p to text file f. |
| WRITE(f,p1,...,pn); | outputs the formatted write-parameters p1,...,pn to text file f. |

WRITELN outputs to a text file, advancing that text file to next line. If any previous WRITEs have had their output text buffered for the text file, the buffered text is output prior to that of the current WRITELN. WRITELN procedure-call statement formats are:

| | |
|---|---|
| WRITELN; | outputs any buffered data (by previous WRITEs) and skips to next line on text file OUTPUT. |
| WRITELN(f); | outputs any buffered data (by previous WRITEs) and skips to next line on text file f. |
| WRITELN(f,p); | outputs the formatted write-parameter p to text file f. |
| WRITELN(f,p1,...,pn); | outputs the formatted write-parameter p1,...pn to text file f. |

| | |
|---|---|
| WRITELN(p); | outputs the formatted write-parameter p to text file OUTPUT. |

WRITELN(p1,...,pn);

outputs the formatted write-parameters p1,...,pn to text file OUTPUT.

In the preceding WRITE and WRITELN examples p denotes a write-parameter which may be a literal character-string or a variable selector or expression whose value is of type BOOLEAN, CHAR, BYTE, INTEGER, SHORTINTEGER, REAL, or SHORTREAL. Although textfile output is restricted to these simple types, automatic conversion of the internal form to the textual output is provided. More complex data types in addition to these simple types may be written to non-textfiles; e.g., arrays or records. To text files, Boolean values are written as either TRUE or FALSE; integer values are written in a decimal character representation; and characters or literal strings are written as characters. Minimum field width may be specified; and default formats are given in Chapter 8. Reals/Shortreals may be written in either of their respective (appropriate to the differences in their displayable precisions) floating-point forms, within a specified field-width; or both may be written in a fixed-point form with user-specified field-width and fractional-digits length.

### 3.5.2 Dynamic Memory Allocation Procedures

In the following example formats for the dynamic memory allocation procedures, (which mark, individually create new or dispose of dynamic variables, or release storage to the mark), p and m denote pointer-variables.

NEW creates an individual dynamic variable.

| | |
|---|---|
| NEW(p); | creates a target-variable of the data-type that the pointer-variable p has been bound to, and assigns the location of the newly created variable to p; (NEW points p to the newly created variable, but does not put a value into the variable). The target-variable is then referenceable by p^. |

DISPOSE releases the storage occupied by an individual dynamically created variable (called a target-variable).

| | |
|---|---|
| DISPOSE(p); | releases the storage occupied by a single target-variable that pointer p is pointing to. |

MARK obtains the value of an address at the current frontier of the heap, the dynamically allocated memory area used for storage of dynamic variables. The procedure MARK is only available to compilation-units compiled under the compiler-option HEAPMARK (see Chapter 1).

| | |
|---|---|
| MARK(m); | with m restricted to be a pointer-variable of type ^INTEGER, sets m to the value of an address at the frontier of the heap so that RELEASE(m) may be called later to relinquish any storage used, beyond this frontier, by NEW(p) calls made subsequent to MARK(m). If DISPOSE, which frees up space by disposing of individual targets, is being used in combination with MARK, some NEW calls may use available storage behind the frontier. If DISPOSE is not being used in combination with MARK and RELEASE, a MARK(m), followed by several NEW(p)'s, and associated processing of the p^ targets, followed by a RELEASE(m); relinquishes all storage used since the MARK(m). |

RELEASE relinquishes the storage occupied by one or more dynamic target-variables created since a previous call on MARK, and exactly which storage is relinquished depends on whether DISPOSE has been also used. The procedure RELEASE is only available to compilation-units compiled under the compiler-option HEAPMARK (see Chapter 1).

| | |
|---|---|
| RELEASE(m); | with m restricted to be a pointer-variable of type, ^INTEGER, and m previously set by a call on MARK(m), relinquishes all the storage used beyond m for targets created by NEW(p); in one of two ways.<br>1. If DISPOSE has not been used in combination with MARK and RELEASE, the NEW calls subsequent to MARK use storage only beyond the frontier at increasingly lower addresses (the heap grows downward). Therefore, the effect of RELEASE(m) is identical to Pascal R00, in that all storage is relinquished, and all target-variables created by NEW(p) are destroyed, since the last MARK(m).<br>2. If DISPOSE is in use in combination with MARK and RELEASE, NEW calls, made subsequent to a MARK, may use either storage behind or beyond the frontier of the heap obtained by a MARK. In this case, the effect of RELEASE(m) is that only the heap storage, used beyond the frontier at lower addresses than m since the last MARK(m), is relinquished. |

Note that, as DISPOSE was not supported in Pascal R00, this advanced mechanism which provides greater flexibility in effectively using available memory, causes MARK and RELEASE to be redefined, as above. Pascal R01 and up, not only uses memory more efficiently, but also requires that the arguments to MARK and RELEASE be pointer-variables of type ^INTEGER. As dynamic data structures are mostly applicable to target-variables of the record-type, the user is referred for details to Section 5.3.10 on the record-type, Section 5.3.11 on pointer-types, and Chapter 10 for run-time information on the heap memory allocation scheme in use.

### 3.5.3 Arithmetic Functions

Also refer to Section 3.5.9 for a summary of how to access the mathematical functions, other than ABS or SQR, from either the Perkin-Elmer Sytem Mathematical Library (on PEMATH.OBJ), or a FORTRAN VII Run Time Library (e.g.: F7RTL50.OBJ) which is built to contain those math routines on PEMATH.OBJ. The FORTRAN VII Run-Time Library Error Message file (e.g., F7RTL50.ERR) should also be available in the user system, when such routines are in use, at run time. For information on the PEMATH routines, themselves, refer to the Perkin-Elmer System Mathematical Functions Reference Manual, PN 48-025. Refer also to the FORTRAN VII User Manual, PN 48-010. Be mindful in coding declarations for Pascal-FORTRAN interfaces that the Pascal type REAL and the FORTRAN type DOUBLE PRECISION both pertain to double-precision floating-point machine real-number representations; and that the Pascal type SHORTREAL and the FORTRAN type REAL both pertain to single-precision floating-point machine real-number representations.

| | |
|---|---|
| ABS(x) | computes the absolute value of the argument expression x. The data-type of the function result is the same type as that of the argument expression x, which may be of type BYTE, INTEGER, SHORTINTEGER, REAL, or SHORTREAL. |
| SQR(x) | computes the square of the value of the argument expression x. The data-type of the function result is the same type as that of the argument expression x, which may be of type BYTE, INTEGER, SHORTINTEGER, REAL, or SHORTREAL. |
| SIN(x) | computes the SHORTREAL sine of the argument expression x, which may be of type BYTE, INTEGER, SHORTINTEGER, REAL, or SHORTREAL, whose value is passed as a Pascal SHORTREAL. Usage requires linkage to PEMATH or an F7RTL containing same, and an external FORTRAN declaration:<br>FUNCTION SIN(Z:SHORTREAL):SHORTREAL;FORTRAN;<br>The function result is always of type SHORTREAL. |
| DSIN(x) | computes the REAL sine of the argument expression x, which may be of type BYTE, INTEGER, |

| | |
|---|---|
| | SHORTINTEGER, REAL or SHORTREAL, whose value is passed as a Pascal REAL. Usage requires linkage to PEMATH or an F7RTL containing same, and an external FORTRAN declaration:<br>FUNCTION DSIN(Z:REAL):REAL;FORTRAN;<br>The function result is always of Pascal type REAL. |
| COS(x) | computes the SHORTREAL cosine of the argument expression x, which may be of type BYTE, INTEGER, SHORTINTEGER, REAL, or SHORTREAL, whose value is passed as a Pascal SHORTREAL. Usage requires linkage to PEMATH or an F7RTL containing same, and an external FORTRAN declaration:<br>FUNCTION COS(Z:SHORTREAL):SHORTREAL;FORTRAN;<br>The function result is always of type SHORTREAL. |
| DCOS(x) | computes the REAL cosine of the argument expression x, which may be of type BYTE, INTEGER, SHORTINTEGER, REAL or SHORTREAL, whose value is passed as a Pascal REAL. Usage requires linkage to PEMATH or an F7RTL containing same, and an external FORTRAN declaration:<br>FUNCTION DCOS(Z:REAL):REAL;FORTRAN;<br>The function result is always of Pascal type REAL. |
| ATAN(x) | computes the SHORTREAL arctangent of the argument expression x, which may be of type BYTE, INTEGER, SHORTINTEGER, REAL, or SHORTREAL, whose value is passed as a Pascal SHORTREAL. Usage requires linkage to PEMATH or an F7RTL containing same, and an external FORTRAN declaration:<br>FUNCTION ATAN(Z:SHORTREAL):SHORTREAL;FORTRAN;<br>The function result is always of type SHORTREAL. |
| DATAN(x) | computes the REAL arctangent of the argument expression x, which may be of type BYTE, INTEGER, SHORTINTEGER, REAL or SHORTREAL, whose value is passed as a Pascal REAL. Usage requires linkage to PEMATH or an F7RTL containing same, and an external FORTRAN declaration:<br>FUNCTION DATAN(Z:REAL):REAL;FORTRAN;<br>The function result is always of Pascal type REAL. |
| EXP(x) | computes the SHORTREAL exponential of the argument expression x, which may be of type BYTE, INTEGER, SHORTINTEGER, REAL, or SHORTREAL, whose value is passed as a Pascal SHORTREAL. Usage requires linkage to PEMATH or an F7RTL containing same, and an external FORTRAN declaration:<br>FUNCTION EXP(Z:SHORTREAL):SHORTREAL;FORTRAN;<br>The function result is always of type SHORTREAL. |
| DEXP(x) | computes the REAL exponential of the argument expression x, which may be of type BYTE, INTEGER, SHORTINTEGER, REAL, or SHORTREAL, whose value is passed as a Pascal REAL. Usage requires linkage to |

| | |
|---|---|
| | PEMATH or an F7RTL containing same, and an external FORTRAN declaration:<br>FUNCTION DEXP(Z:REAL):REAL;FORTRAN;<br>The function result is always of Pascal type REAL. |
| SQRT(x) | computes the SHORTREAL square root of the argument expression x, may be of type BYTE, INTEGER, SHORTINTEGER, REAL, or SHORTREAL, whose value is passed as a Pascal SHORTREAL. Usage requires linkage to PEMATH or an F7RTL containing same, and an external FORTRAN declaration:<br>FUNCTION SQRT(Z:SHORTREAL):SHORTREAL;FORTRAN;<br>The function result is always of type SHORTREAL. |
| DSQRT(x) | computes the REAL square root of the argument expression x, which may be of type BYTE, INTEGER, SHORTINTEGER, REAL or SHORTREAL, whose value is passed as a Pascal REAL. Usage requires linkage to PEMATH or an F7RTL containing same, and an external FORTRAN declaration:<br>FUNCTION DSQRT(Z:REAL):REAL;FORTRAN;<br>The function result is always of Pascal type REAL. |
| ALOG(x) | computes the SHORTREAL natural logarithm of the argument expression x, which may be of type BYTE, INTEGER, SHORTINTEGER, REAL, or SHORTREAL, whose value is passed as a Pascal SHORTREAL. Usage requires linkage to PEMATH or an F7RTL containing same, and an external FORTRAN declaration:<br>FUNCTION ALOG(Z:SHORTREAL):SHORTREAL;FORTRAN;<br>The function result is always of type SHORTREAL. |
| DLOG(x) | computes the REAL natural logarithm of the argument expression x, which may be of type BYTE, INTEGER, SHORTINTEGER, REAL or SHORTREAL, whose value is passed as a Pascal REAL. Usage requires linkage to PEMATH or an F7RTL containing same, and an external FORTRAN declaration:<br>FUNCTION DLOG(Z:REAL):REAL;FORTRAN;<br>The function result is always of Pascal type REAL. |

### 3.5.4 Boolean Functions

| | |
|---|---|
| ODD(x) | returns the Boolean value of TRUE if the argument expression x is odd; if x is even returns the Boolean result FALSE. The argument expression x may be of type BYTE, INTEGER, or SHORTINTEGER. |
| EOF(f) | returns the Boolean value of TRUE when, while reading the file f, the end of file occurs; otherwise returns the Boolean value FALSE. |
| EOF | returns the Boolean value of TRUE when, while reading the standard text file INPUT, the end of |

| | |
|---|---|
| | file occurs; otherwise returns the Boolean value FALSE. |
| EOLN(f) | returns the Boolean value of TRUE when, while reading the text file f, the end of current line is reached; otherwise returns the Boolean value of FALSE. |
| EOLN | returns the Boolean value of TRUE when, while reading the standard text file INPUT, the end of current line is reached; otherwise returns the Boolean value of FALSE. |

### 3.5.5 Real/Integer Data Conversion Functions

| | |
|---|---|
| TRUNC(r) | truncates the fractional part of the real argument expression r, truncating the value toward zero, and returns that remaining integer value. That is, TRUNC(r) returns the greatest integer less than or equal to r for r>=0, or the least integer greater than or equal to r for r<=0. The argument expression r may be of type REAL or SHORTREAL and the resultant function value, TRUNC(r), is of type INTEGER. If the value of r exceeds representation in an INTEGER, a run time error message "TRUNC RANGE ERROR" occurs. |
| ROUND(r) | converts the real argument expression r, rounding away from zero, into its nearest corresponding integer value. The argument expression r may be of type REAL or SHORTREAL and the resultant function value, ROUND(r), is of type INTEGER. If the value of r exceeds representation in an INTEGER, a run time error message "TRUNC RANGE ERROR" occurs. |
| LENG(s) | lengthens the SHORTREAL argument expression s into a corresponding resultant function value of type REAL. |
| SHORTEN(r) | shortens the REAL argument expression r into a corresponding resultant function value of type SHORTREAL. |
| CONV(i) | converts the integer argument expression i into a corresponding resultant function value of type REAL. The argument expression i may be of type BYTE, INTEGER, or SHORTINTEGER. |
| SHORTCONV(i) | converts the integer argument expression i into a corresponding resultant function value of type SHORTREAL. The argument expression i may be of type BYTE, INTEGER, or SHORTINTEGER. |

### 3.5.6 Ordered Data Transfer Functions

| | |
|---|---|
| ORD(x) | returns the ordinal number of the argument expression x, which may be of type BOOLEAN, CHAR, a user-defined enumeration type, or in this implementation of Pascal, a pointer-type. When x is Boolean, the ORD(FALSE) = 0 and the ORD(TRUE) = 1. When x is of type CHAR, the ORD(x) is the ordinal number of the character x in the ASCII character set. When x is a member of a user-defined enumeration type, ORD(x) is the natural number indicating the order of appearance of the value of x in the user-defined enumeration type list of identifiers(counting starts at zero). When x is a pointer expression, then ORD(x) is an integer which is the machine address of the target of pointer x. This function, when applied to character data, is the inverse of CHR. |
| CHR(x) | returns the character in the ASCII set which corresponds to the ordinal number x. The argument expression x may be of the integer types BYTE, INTEGER, or SHORTINTEGER and must be in the range 0..127. This function is the inverse of ORD, when ORD is applied to character data. |

### 3.5.7 Discrete Data Transfer Functions

| | |
|---|---|
| PRED(x) | returns the predecessor of the argument expression x, which may be of type , CHAR, or the integer types BYTE, INTEGER, and SHORTINTEGER; or of a user-defined enumeration type. When x is of type CHAR, the PRED(x) returns the character preceding x in the ASCII character set but is undefined for the first character, e.g., when ORD(x)=0. When x is of the integer types, PRED(x) returns the preceding integer (x-1). When x is a member of a user-defined enumeration type, PRED(x) returns the member immediately preceding x in the user-defined enumeration type list; and PRED(x) is undefined if x is the first member in that list. |
| SUCC(x) | returns the successor of the argument expression x, which may be of type CHAR, or the integer types BYTE, INTEGER, and SHORTINTEGER; or of a user-defined enumeration type. When x is of type CHAR, the SUCC(x) returns the character following x in the ASCII character set but is undefined for the last character, e.g., if ORD(x)=127. When x is of the integer types, SUCC(x) returns the succeeding integer (x+1). When x is a member within a user-defined enumeration type, SUCC(x) returns the member following x in the user-defined enumeration type list; but SUCC(x) is undefined if x is the last element in that list. |

### 3.5.8 Miscellaneous Functions

| | |
|---|---|
| ADDRESS(v) | returns a 32-bit INTEGER value which is the machine address of the argument variable or selector v, which may be of any type but may not be a literal constant, nor a constant-identifier. This address is defined to be the location from which the value of v would be found for any use of v at the place in program where ADDRESS(v) is referenced. If v were a pointer variable, p, the ADDRESS(p) returns the location of pointer p; the ORD(p) returns the location of the target-variable p is pointing to; the ADDRESS(p^) returns the location of the target-variable p is pointing to. In this implementation, ORD(p) = ADDRESS(p^) -8; and the ADDRESS (p^)=ORD(p)+8. |
| SIZE(vt) | returns an INTEGER value which is the number of bytes required to represent the argument datum or type vt in internal storage. The argument may be the name of a variable, selector, or a type-identifier. The argument vt may not be a literal constant or a constant-identifier. |
| LINENUMBER | returns the current source line number of that source statement containing the function reference LINENUMBER. The resultant function value of LINENUMBER is of type INTEGER. No arguments are required or accepted by this function. That is, if the 35th line of a program were: WRITELN(LINENUMBER); upon execution, the integer 35 would be written on standard text file OUTPUT. |
| STACKSPACE | returns an INTEGER value which is the number of bytes currently available at the time of executing the statement containing the function reference between the "stack" run time storage of local data of routines and the "heap" run time storage of dynamically allocated variables. No arguments are required nor accepted by this function. |

### 3.5.9 Accessing Additional Math Routines from PEMATH or FORTRAN RTLs

Mathematical functions available via linkage to the Perkin-Elmer System Mathematical Library on the file PEMATH.OBJ or to a FORTRAN VII Run Time Library containing those math routines, (such as F7RTL50.OBJ), are the functions: arctangent, sine, cosine, exponential, square root, and natural logarithm. These basic external math routines may be accessed for use as function references by declaring them in the routine declarations part of the user's main program as external FORTRAN routines, with the Pascal FORTRAN directive. The routines themselves are documented in Perkin-Elmer Systems Mathematical Functions Reference Manual, PN 48-025. The FORTRAN VII Run-Time Library Error Message file (e.g., F7RTL50.ERR) should also be available in the user system, when such routines are in use, at run time. Refer to Tables 3-6 and 3-7 for declaration details. Linkage to the external PEMATH (or if they are incorporated into a FORTRAN VII RTL) math routines must then be performed at task establishment time (see Chapter 1). This linkage is performed by either linking to PEMATH.OBJ or a FORTRAN VII RTL that has been built to, or already includes, those routines on PEMATH.OBJ. To interface with Pascal compiled-code the versions of any PEMATH.OBJ or F7RTL containing same, must be those versions of the routines without argument checking.

Note the the value parameter identifier, Z , given in the example external FORTRAN declarations, may be any appropriate user-specified identifier.

### TABLE 3-6 BASIC EXTERNAL FORTRAN MATH ROUTINE ACCESS (SHORTREAL FUNCTIONS)

| TO OBTAIN SHORTREAL FUNCTION | ENTER IN ROUTINE DECLARATIONS | USE FUNCTION REFERENCE |
|---|---|---|
| arctan(X) | FUNCTION ATAN(Z:SHORTREAL):SHORTREAL;FORTRAN; | ATAN(X) |
| sin(X) | FUNCTION SIN(Z:SHORTREAL):SHORTREAL;FORTRAN; | SIN(X) |
| cos(X) | FUNCTION COS(Z:SHORTREAL):SHORTREAL;FORTRAN; | COS(X) |
| exp(X) | FUNCTION EXP(Z:SHORTREAL):SHORTREAL;FORTRAN; | EXP(X) |
| sqrt(X) | FUNCTION SQRT(Z:SHORTREAL):SHORTREAL;FORTRAN; | SQRT(X) |
| ln(X) | FUNCTION ALOG(Z:SHORTREAL):SHORTREAL;FORTRAN; | ALOG (X) |

## TABLE 3-7 BASIC EXTERNAL FORTRAN MATH ROUTINE ACCESS (REAL FUNCTIONS)

| TO OBTAIN REAL FUNCTION | ENTER IN ROUTINE DECLARATIONS | USE FUNCTION REFERENCE |
|---|---|---|
| arctan(X) | FUNCTION DATAN(Z:REAL):REAL;FORTRAN; | DATAN(X) |
| sin(X) | FUNCTION DSIN(Z:REAL):REAL;FORTRAN; | DSIN(X) |
| cos(X) | FUNCTION DCOS(Z:REAL):REAL;FORTRAN; | DCOS(X) |
| exp(X) | FUNCTION DEXP(Z:REAL):REAL;FORTRAN; | DEXP(X) |
| sqrt(X) | FUNCTION DSQRT(Z:REAL):REAL;FORTRAN; | DSQRT(X) |
| ln(X) | FUNCTION DLOG(Z:REAL):REAL;FORTRAN; | DLOG(X) |

Users not specifying these external declarations with the Pascal FORTRAN directive and associative REAL/SHORTREAL type-identifiers pertinent to each function name exactly as depicted above will receive unpredictable results.

# CHAPTER 4
# PROGRAM STRUCTURE, BLOCKS, AND DECLARATIONS

## 4.1 INTRODUCTION TO PROGRAM STRUCTURE

A Pascal program syntactically consists of an optional prefix of declarations, a program-heading, and a block followed by an end of program indicator, the period character.

A block may follow not only a program-heading, but a procedure, or function heading or an external module heading as well.

The prefix syntax and the program heading, as well as module, procedure, and function headings, are detailed in Chapter 9.

This chapter is merely introducing an overview of program structure, and the definition of a block and its components, a declarations part and a body. In Pascal, a block is of the general form:

```
LABEL    label declarations, if any
CONST    constant declarations, if any
TYPE     type declarations, if any
VAR      variable declarations, if any

PROCEDURE or FUNCTION routine declarations, if any

BEGIN

         Body of the Block
END
```

A block is comprised of an optional declaration part and always contains a body. The declaration part defines or declares such entities as labels, or types, constants, variables, and routine identifiers, that are referenced in other subsequent declarations or in the body.

When the declarations contain routine definitions, then other blocks are created within the outer block, creating nested scopes (see Section 2.1.2 and 2.1.3).

The body is a compound statement containing one or more executable statements that specify the actions to be performed by the program.

The syntax of a Pascal program is:

## Program

```
 ---> prefix --->|
 ^               |
 |               V
-------------------> program-heading ---> block ---> .
```

where the syntax of program-heading is:

## Program heading

```
                              -------------------->|
                               ^                   |
                               |                   V
---> PROGRAM ---> identifier ---> file-name-list ---> ; --->
```

where the identifier is the name of the program, and the file-name-list is a list of user-specified identifier(s) which are the names of external files, (separated by commas when more than one); and which must also be declared as file-variables. The predefined files INPUT or OUTPUT may also be listed.

After a program-heading has been specified, with or without a file-name-list, and required semicolon; a block may be defined.

The syntax of block is:

## Block

```
---> declarations ---> body --->
```

Although not evident from the "block" syntax graph, the construct declarations, may be empty as defined by its syntax graph in Section 4.2 below. The body is always required to constitute a block. Declarations are detailed in Section 4.2, and the body is detailed in 4.3.

The general form of the layout of a program is depicted as follows. Note that within a block the words LABEL, CONST, TYPE and VAR need not be present if no definitions follow them; however, the words BEGIN and END must always be present in a block to form the body of a program, module, procedure, or function.

> { Optionally precede the program-header with either a user written prefix or the predefined Pascal Prefix, which is required only for those programs utilizing prefix routine calls or prefix identifiers in their code. }

```
PROGRAM FORM ( list of filenames separated by commas ) ;

  { Block of entire program follows: }
  { Declare/Define labels/Data prior program Body, for example:}

  LABEL              {   optionally define any statement-labels}
        1000,2000 ;
  CONST              {   optionally define any global constants}
        LIMIT = 10 ;
  TYPE               {optionally define global type-identifiers}

        SAMPLE_ARRAY_TYPE = ARRAY [1..LIMIT] OF CHAR ;

  VAR                { declare global variables with their type}

        STRING : SAMPLE_ARRAY_TYPE ; R : REAL; S : SHORTREAL;
        I,J,K : INTEGER ; H : SHORTINTEGER ; P,Q : BOOLEAN;

        {Declare external user-named files from file-name-list }

        FILE45 : TEXT ;

  { Optionally define any number of external/internal routines:}
  PROCEDURE MOD_NAME1 (module-parameter-list); EXTERN;
  FUNCTION DCOS(z:REAL):REAL;FORTRAN;

  PROCEDURE A1(A1-parameter-list);
    {Block of A1 follows:}
    LABEL {... optionally define any statement-labels in A1} ;
    CONST {... optionally define any local-to-A1 constants} ;
    TYPE  {... optionally define any local-to-A1 types} ;
    VAR   {... optionally declare any local-to-A1 variables} ;
    {optionally define any nested routines local-to-A1} ;
    BEGIN  { Body of procedure A1}
      statement-sequence
    END;   { End of procedure A1}

  FUNCTION F1(F1-parameter-list): function-type-identifier ;
    {Block of F1 follows:}
    LABEL {... optionally define any statement-labels in F1} ;
    CONST {... optionally define any local-to-F1 constants};
    TYPE  {... optionally define any local-to-F1 types};
    VAR   {... optionally declare any local-to-F1 variables};
    {optionally define any nested routines local-to-F1};
    BEGIN  { Body of function F1}
      ...
      F1 := expression ;
      ...
    END;   { End of function F1}

  BEGIN  { Body of main program FORM }
    1000 : statement;
    A1(A1-argument-list);              {sample invocation of A1}
    R:=F1(F1-argument-list)+DCOS(R);  {invocation of F1 & DCOS}
    2000 : statement;
  END.   { End of main program FORM }
```

For a sample program, the following program, CUBE, inputs an integer length from standard textfile INPUT and outputs its dimensions on standard textfile, OUTPUT.

```
PROGRAM CUBE (INPUT,OUTPUT);
  VAR LENGTH,SIDE_AREA,SURFACE,VOLUME : INTEGER ;
  BEGIN
    READ (LENGTH) ;
    SIDE_AREA := SQR (LENGTH) ;
    SURFACE := 6 * SIDE_AREA ;
    VOLUME := SIDE_AREA * LENGTH ;
    WRITELN ('CUBE DIMENSIONS');
    WRITELN ('LENGTH        IS', LENGTH, ' LINEAR FEET');
    WRITELN ('SIDE AREA     IS', SIDE_AREA, ' SQUARE FEET');
    WRITELN ('SURFACE       IS', SURFACE, ' SQUARE FEET');
    WRITELN ('VOLUME        IS', VOLUME, ' CUBIC FEET');
  END. {CUBE}
```

Upon compilation of this source program into object and linkage of the object into an established task, the task can be loaded with task workspace, have its files assigned, and executed under the operating system.

Upon execution of CUBE, if the input data on the physical device/file assigned to logical unit 0 which has been associated with the standard textfile INPUT (by virtue of its position in the file-name-list) is:

```
     5
```

The output on standard textfile OUTPUT (on LU 1) is:

```
CUBE DIMENSIONS
LENGTH        IS          5 LINEAR FEET
SIDE AREA     IS         25 SQUARE FEET
SURFACE       IS        150 SQUARE FEET
VOLUME        IS        125 CUBIC FEET
```

Program CUBE uses two predefined external files, INPUT and OUTPUT; has four global variables, LENGTH, SIDE_AREA, SURFACE, and VOLUME; all of data-type INTEGER; and has no other items defined in its declarations. A function reference on a predefined Pascal function, SQR, to obtain the square is used in the expression in the assignment statement: SIDE_AREA := SQR(LENGTH).

The body of the program CUBE begins with the word BEGIN and ends with the word END. The program is terminated with a period. The READ statement reads the input data 5 into the variable LENGTH, from the file, INPUT. The assignment statements obtain values for SIDE_AREA, SURFACE, and VOLUME. The first WRITELN statement outputs a literal-string, CUBE DIMENSIONS, to the file OUTPUT.

The subsequent WRITELN's each output a line consisting of a literal-string, an integer, and another literal-string.

Note that the arguments in the WRITELN statements are simply integer variables, so that the default textfile field width is in effect to produce the integer values: 5, 25, 125, and 150 right justified in a field width of ten positions. The literal strings following the integer variables have an imbedded leading space, so that the output provides a space between the integers and following strings. Another method of formatting text output would be to specify field widths directly in the write-parameter arguments. See Chapter 8 on write-parameters.

More complex program structuring, including routine definitions, their invocations, and nesting or recursive use, is delayed to Chapter 9 until after the declarations part and body of a block are defined in this chapter, and the Pascal data-types, expressions, and statements are defined in subsequent chapters.

## 4.2 DECLARATIONS

The declarations part of a block declare labels to identify that certain statements in the body of the same block are labeled with a statement-label; and declare identifiers as the names of constants, types, variables, or routines which become visible in scope to the block. The syntax of declarations is defined by:

### Declarations

```
------>label declarations----->|
   |                       ^   |
   |                       |   |
   V----------------------->|   |
                               |
                               |
   |<--------------------------V
   |
   |
   |  |<--const definitions<--^   ^-------------------->|
   |  |                       |   |                     |
   |  V                       |   |                     V
   V--------------------------------->var declarations----->|
      ^                       |                             |
      |                       |                             |
      |<--type definitions<---V                             |
                                                            |
                                                            |
                                |<-----------------------^  |
                                |                        |  |
                                V                        |  |
                              <---routine declarations<-----V
```

Note the order of the five kinds of declaration parts. Label Declarations precede Constants and Type Definitions, all of which are followed by Variable Declarations, and lastly declared are routines.

Note that the declarations part of a block may be completely empty, or not contain a label declarations part, a constants definitions part, a type definitions part, a variable declarations part, or a routine declarations part.

Many implementations of Pascal allow only one of each kind of declaration part in the declarations section of a block.

In this implementation of Pascal, in each block, there may be only one Label Declarations part, more than one Constants Definitions part and more than one Type Definitions part, only one Variable Declarations part, and only one Routine Declarations part (although it may define any number of routines).

Also, in this implementation of Pascal, the constant definitions parts and the type definitions parts may be intermixed, prior to the beginning of any variable declarations part in a block. See example code of mixed constant and type definitions parts in Section 4.2.3.

Each of the declaration parts are described in the following sections.

### 4.2.1 Label Declaration Part

The Label Declaration Part introduces user-specified unsigned integers as labels.

Any executable statement in a block's body can be identified by prefixing it with a label (see syntax chart of "statement" in Section 7.1), and a colon. A label is a defining reference point within the body of a program's block of a particular statement. Transfer of execution control to a labeled statement is provided by the GOTO statement, described in Section 7.2.4.

All labeled statements must have their labels declared in the label declaration at the beginning of the block in which their embodying body is contained. The compound statement serving as the body of a block is never labeled for the purpose of going to it. The label declarations part of a block's declarations is defined by:

Label-Declarations

```
---> LABEL ---------> label -----> ; --->
                    ^            |
                    |            |
                    |<--- ,  <--V
```

and label is defined by:

Label

```
--------> digits -------->
```

The syntax of the label declarations part consists of the word LABEL, followed by one or more "labels", each separated from each other by a comma, when there is more than one label; and the entire label declarations part ends with a semicolon.

In this implementation of Pascal, the number of decimal digits in a label is expanded to 10, the number of digits required to express the value of MAXINT (Section 5.3.4). Many Pascal implementations allow only four digits. A label is considered to be an unsigned integer. That is, the value of the unsigned integer which represents a label must be in the range 1 to MAXINT. Leading zeros in a label are not significant.

Examples of valid label declarations parts are:

```
LABEL  01, 10, 999999;

LABEL  454,1359;          {label 454 and label 1359}

LABEL  2147483647;        {largest label allowed}

LABEL  10, 20, 30,
       50, 60, 70;        {entire label part ends with ";"}
```

Examples of invalid label declarations parts:

```
LABEL #2D0;               (*label has hexadecimal digits*)

LABEL 3000                (* no ending semicolon *)

LABEL 01, 1;              (*duplicate declaration*)

LABEL 4000000000;         (*label has value>MAXINT*)

LABEL 10; 20; 30;         (* commas must separate labels *)

LABEL 10, 20; 30, 40;     (* only one semicolon is allowed *)
      50, 60, 70; 90;     (* to end the Label Declarations *)
```

Only one label declarations part is allowed in any one declarations section of a block, and it must precede any constant, type, variable or routine declarations parts inside the block.

### 4.2.2 Constant Definition Part

The Constant Definition Part introduces user-specified identifiers as the names of constant values, and defines their specific values.

Data that do not change during the execution of a program are called constants. Constants represent specific values which may be used as operand factors within expressions. Constants can be referred to by either a literal representation of their value or by the names, established as identifiers of their values, in a constant definitions part of a block.

Those constants, directly representable in source code without identifiers, are called literal constants, as presented in Section 3.3.4.

Literal-constants may be used to define the values of constant-identifiers introduced in the constant definitions part.

The syntax of the constant definition part is:

Constant-Definitions

```
---> CONST ---> identifier ---> = ---> constant ---> ; --->
            ^                                            |
            |                                            |
            |<-------------------------------------------V
```

The constant definitions part consists of the word, CONST, followed by one or more individual constant definitions, each separated from each other by a semicolon, when there is more than one; and the entire constant definitions part ends with a semicolon. Each individual constant definition introduces a user-specified identifier, followed by an equals sign, followed by a "constant".

The construct "constant" is syntactically presented in Section 5.1. It may be a (signed, if numeric) literal constant, or a predefined or previously declared constant identifier. It cannot be a constant expression such as: 16-5+3, or -1+MAXINT, or an expression such as (FALSE AND TRUE).

The constant definition part need not be present in any or every program, module, procedure, or function block. However, if the word CONST is present, at least one constant definition must be present.

The constant definition part, when present, comes after label declarations and before the variable declaration part. This implementation of Pascal permits more than one constant definitions part and permits constant definitions parts and type definitions parts to be intermixed. See example code at the end of Section 4.2.3.

Example of a constant definitions part is:

```
    CONST
       DATE    =   '03/18/80';
       PI      =   3.141529;
       NEGPI   =  -3.141529;
       CHARA   =  'A';
       TWO      =  2 ; THREE = +3;
       NEGTWO   = -2 ;  NEGTHREE = -THREE;
       TWOTOO   = TWO ;
       INDENT2 = +TWOTOO;
       BACKSPACE = -TWO;
```

The individual constant definition defines the specific value to be substituted for the constant identifier wherever it occurs within the scope of the declaration, unless a redefinition of the constant identifier occurs in a more tightly binding scope prior to the place of reference within the inner scope. See Section 2.1.3 on scope.

The compiler applies typing to the constant identifier by analyzing the constant value. In the example above, the type of DATE is a string-type of length 8; the type of PI is REAL; the type of CHARA is CHAR; the type of TWO and TWOTOO is integer.

In this implementation of Pascal, a constant definitions part may also be declared in a "prefix" of declarations placed prior to a PROGRAM or MODULE header, giving those constant-identifiers a scope global to the entire compilation-unit; respective to whether the prefix is preceding a program or module. In a prefix the constant definitions parts may be intermixed with type definitions parts, prior to routine declaration headings, but may not follow the routine declarations of the prefix. See the syntax of prefix in Chapter 9.

### 4.2.3 Type Definition Part

The Type Definition Part introduces user-specified identifiers as the names of types, and defines their types.

Different data have certain descriptive attributes, such as size, structure, and internal representation, and thereby can assume only some finite range of possible values. The set of possible values that a datum can assume is called its data type.

There are several different kinds of data types in Pascal, and programmers can also define new data types. Only certain operations can be performed on each data type. The language of Pascal, to assure that only meaningful operations are performed on appropriate types of data, has strict rules regarding type-compatibility.

Certain operations, such as relational structured comparisons, or passing arguments to VAR variable parameters; require "identity" of type between the two entities. Other operations on scalar data, or such as passing arguments to value parameters, or assigning expression values to variables, only require "assignment compatibility" of type between the two entities. See a brief definition in Section 2.1.9 or details in Section 6.2 on type-compatibility.

Therefore, the programmer must plan the typing of the program data adhering to the rules of Pascal type-compatibility, respective to the intended use of the data.

The syntax of the type definitions part is:

### Type-Definitions

```
---> TYPE ---> identifier ---> = ---> type ---> ; --->
            ^                                       |
            |                                       |
            |<--------------------------------------v
```

The syntax of the type definitions part of a block consists of the word TYPE, followed by one or more individual type definitions, each separated from the other by a semicolon, when there is more than one, and the entire type definitions part ends with a semicolon.

Only one new user-specified type-identifier may be associated to its defining "type" in each individual type-definition between semicolons. Each individual type definition consists of a new identifier, an equal sign, and a defining type.

An individual type definition introduces the identifier as the name of a type. This identifier becomes visible as a type identifier.

The type identifier becomes visible in scope to the block in which it is declared. It is possible to redeclare the same identifier with a different meaning inside an inner scope. See Section 2.1.3 on scope.

The type identifier can then be referenced in subsequent type declarations or associated to other data to define their type, such as in variable declarations. In general, a data type cannot refer to its own type identifier.

The syntax of "type" is given in Section 5.3. The specific kinds of data types available are described in Sections 5.3.1 to 5.3.11.

User-specified type-identifiers in order to be associated with data in subsequent variable declarations in the block must be first defined in the type declarations of a block which makes them visible in scope for reference. The type definitions part, when present, come after label declarations constant definitions parts, if any are present, and before variable declarations in the declarations portion of a block.

This implementation of Pascal permits constant and type definition parts to be interlaced. Often, interspersing constant and type definitions parts will be an aid to readability and maintainability. For example:

```
CONST  ID_LENGTH = 12;
TYPE   IDENTIFIER = ARRAY [1..ID_LENGTH] OF CHAR;
       .
       .
       .
CONST  OUTPUT_LENGTH = 132;
TYPE   OUTPUT_BUFFER = ARRAY [1..OUTPUT_LENGTH] OF CHAR;
```

may be clearly preferable to:

```
CONST  ID_LENGTH = 12;
       OUTPUT_LENGTH = 132;
       .
       .
       .
TYPE   IDENTIFIER = ARRAY [1..ID_LENGTH] OF CHAR;
       OUTPUT_BUFFER = ARRAY [1..OUTPUT_LENGTH] OF CHAR;
```

where the vertical ellipses may represent several hundred lines of possibly unrelated type or constant definition code.

In this implementation of Pascal, a type definitions part may also be declared in a "prefix" of declarations placed prior to a PROGRAM or MODULE header, giving those type-identifiers a scope global to the entire compilation-unit; respective to whether the prefix is preceding a program or module.

In a prefix the constant definitions parts may be intermixed with type definitions parts, prior to routine declaration headings, but may not follow the routine declarations of the prefix. See the syntax of prefix in Chapter 9.

Particularly notable, as user-specified type-identifiers are often used for typing structured parameters in a module-parameter-list, those type-identifiers must be declared ahead of the MODULE header, but also must not conflict in position to the routine declarations of a prefix.

### 4.2.4 Variable Declaration Part

The Variable Declaration Part of a block introduces user-specified identifiers as the names of variables and associates a data-type to each variable identifier, so introduced. The value of the variable at its declaration point is undefined.

One of the most important characteristics of Pascal is that all named data to be referenced must be previously declared in a declarations part and be visible in scope. This includes variables, which are data that can have their values changed during execution. The kind and range of values that a variable can assume is determined by its type. The basic operations on a variable are assignment of a new value to it and a reference to its current value.

The variable declarations part introduces variable names and associates them with a specific data type. The syntax of the variable declaration part is defined by:

Variable-Declarations

```
---> VAR ---> identifiers ---> : ---> type ---> ; --->
          ^                                        |
          |                                        |
          |<---------------------------------------V
```

where identifiers is defined by:

Identifiers

```
---> identifier --->
 ^                |
 |                |
 |<----   ,  <----V
```

The variable declarations part consists of the word VAR, followed by one or more individual groups of variable declarations, separated by and ending with a semicolon. Each individual group of variable declarations consists of one or more new identifiers separated by a comma, followed by a colon and a specified type. A detailed syntax description of "type" is given in Section 5.3.

The type must be either a predefined Pascal type-identifier, or a previously defined user-specified type-identifier from a type definition part, visible in scope. It can also be a user-specified defining type, but caution is advised in using this form of defining the type of a variable within the variable declaration, as few operations can be legally performed on structured variables without "identity" of type established for them. However, "identity" of type is established for all variables associated to one "type" in one and the same group variable declaration such as A, B, C : type; within a VAR declarations part.

An individual group of variable declarations introduces one or more new identifiers, separated by commas, as variables of the type following the colon.

A variable declared in the variable declarations part of the outermost block of the main program is called a global variable, and is visible in scope over the entire program, unless the same identifier is redefined in a more tightly binding scope in an inner block prior to its reference in the inner block.

A variable declared in the variable declarations part of a routine is called a local variable, and comes into existence only upon invocation of the routine. The local variable identifier is visible in scope only to the block of the routine in which it is declared, and to any inner blocks declared within the routine.

The variable declarations part, when present, occurs after the label, constant, and type definitions parts, and before the routine declarations part of the declarations section of a block.

An example of a variable declarations part is:

```
VAR
  CH     :  CHAR;
  FLAG   :  BOOLEAN;  B : BYTE;
  I,J,K  :  INTEGER;
  DELTA  :  REAL;
  CFILE  :  TEXT;
  SUBSCRIPT, INDEX : 1..20;

  {Using the type-identifiers established above in 4.2.3}

  STRING12  : IDENTIFIER;
  PRINTLINE : OUTPUT_BUFFER;
```

Note that no variable declarations part is allowed in a prefix, unlike the constant, type, and routine declarations which are allowed in a prefix.

### 4.2.5 Routine Declaration Part

The Routine Declaration Part introduces user-specified procedure or function identifiers and either a complete routine definition containing their defining block, or a directive indicative that the routine's block is located elsewhere.

There are two different kinds of routines. A procedure performs a process, and a function computes a value. A procedure is called into execution with a procedure-call statement (see Chapter 7), whereas a function is called into execution when referenced within an expression.

A procedure declaration begins with the Pascal word symbol: PROCEDURE. A function declaration begins with the Pascal word symbol: FUNCTION.

As a language construct, a routine declaration declares its name, a list of parameters (if any) to define its interface, local declarations visible only to itself and its nested routines, and a body of code that operates on the foregoing declarations. When the routine is called into execution, the specific data values of the parameters that the routine is to operate on are passed to the routine in the form of arguments. The routine is written to reference the parameter identifiers, but the actual data passed by arguments become associated with the appropriate routine parameters at execution time.

All user written routines must be identified in a routine declaration part before being referenced except for those predefined routine identifiers of Pascal. See Table 3-4.

The syntax of the routine declarations part is defined by:

### Routine-Declarations

```
   |<--;<--procedure<----^
   v                     |
------------------------------->
   ^                     |
   |<--;<--function<-----v
```

Each procedure or function declaration is separated from the other and terminated by a semicolon.

Any number of procedures or functions may be declared in the routine declarations part of a block, and there is no order restriction on which precedes or follows. That is, a procedure may be defined, and then a function, or vice versa.

The routine declarations follow the label declarations, constant and type definitions, and variable declarations of a block. The routine declarations are the last group in the declarations portion of a block prior to the body.

The routine declaration part introduces the names of these routines as identifiers, defines their parameters, and either defines the block of code or declares how the routine block is defined elsewhere.

The routine block can have its definition delayed, but not its declaration, if the routine is to be referenceable. Delaying a block definition is accomplished by replacing the block with the word FORWARD. This routine block can then be completed later by repeating its heading (without its parameter list and/or function type) followed by its block. FORWARD declaration is only required to be used when two routines must call each other (mutual recursion); but it can also be used whenever desired. The block of a forwardly declared routine must occur somewhere in the same routine declarations part, in which it was declared FORWARD.

Some routines may be externally and separately compiled. A separately compiled routine, written in Pascal, may have its block replaced by the directive, EXTERN. A separately compiled routine written in FORTRAN must have its block definition replaced by the directive, FORTRAN. The declaration of an assembler language routine has its block replaced by either symbol, EXTERN or FORTRAN, depending on whether it uses Pascal or FORTRAN calling conventions, respectively. Any of these routines, to be callable, must first have their names introduced, and parameter lists and if functions, their function types, specified in a routine declarations part.

Procedures and functions will be defined in detail in a later chapter, Chapter 9, with associative concepts dealing with routines. At this point, we are merely introducing the overview and placement of the routine declarations part within the declarations section of a block.

The reader may also refer to related concepts on blocks and scopes given in previous Section 2.1.2 and Section 2.1.3.

At this point, an example of a routine declarations part is:

```
PROCEDURE  ABC (A,B,C:BYTE); EXTERN;
PROCEDURE  SECTOR (RADIAN:REAL); FORTRAN;
PROCEDURE  ADD (X,Y:INTEGER; VAR Z:REAL); FORWARD;
PROCEDURE  READINT (VAR NN:INTEGER);
  BEGIN
    WHILE NOT EOF DO
      READ (NN);
  END;
PROCEDURE  ADD;
  BEGIN
    Z := X + Y;
  END;
```

```
FUNCTION  SEQUENCE(CH:CHAR):INTEGER; EXTERN;
FUNCTION  COSINE(RADIAN:REAL):REAL; FORTRAN;
FUNCTION  NEXTODD(NUMB:INTEGER):INTEGER; FORWARD;
FUNCTION  EVEN(NUMBER:INTEGER):BOOLEAN;
  BEGIN
    IF NUMBER MOD 2 = 0
      THEN
        EVEN := TRUE
      ELSE
        EVEN := FALSE;
  END;
FUNCTION  NEXTODD;
  BEGIN
    IF EVEN(NUMB)
      THEN
        NEXTODD := NUMB + 1
      ELSE
        NEXTODD := NUMB + 2
  END;
```

These example routine declarations establish the procedures as callable units, which may be called into execution by procedure call statements, such as (assuming VAR R:REAL; and VAR I:INTEGER;) in the following:

```
BEGIN
  ABC(5,3,8);
  SECTOR(5.34267);
  ADD(9,4,R);
  READINT(I);
END;
```

The example routine declarations also establish the functions which can be referenced within expressions, to obtain their function values, such as in the following (assuming VAR R:REAL; and VAR I:INTEGER;):

```
BEGIN
  I  := SEQUENCE('A');
  R  := COSINE(6.54732);
  I  := NEXTODD(I);
  IF NOT EVEN(I) THEN WRITELN(I);
END;
```

Both procedure and function headings, routine and parameter declarations, routine invocations and argument specifications, argument to parameter compatibility, and the programming of routines are discussed in detail in Chapter 9.

## 4.3 THE BODY

The body of a program, module, function, or procedure block is a compound statement which usually contains a sequence of executable statements embodied within it.

The body begins after the declarations part of a block, if there are any, and once a body begins no further new identifiers may be introduced.

The syntax of a body is:

Body

```
---> BEGIN ------> statement ------> END --->
            ^                    |
            |                    |
            |<------ ; <------v
```

The body is a compound statement which consists of the word BEGIN, one or more statements (separated by a statement separator, the semicolon) and the body is terminated with the word END.

The word END will be followed by a period when ending the block of a program or module. (See syntax graph of program in Section 4.1 above for program and Section 9.2 for module.) The word END will be followed by a semicolon when ending the block of a procedure or function definition. See syntax graph of Routine-Declarations in Section 4.2.5 above. The end of a body is also the end of a block.

Execution of the statements between BEGIN and its associated END occurs one statement at a time, sequentially.

Execution of the program, begins with execution of the body of the program block. Execution of the body of a procedure block occurs upon invocation of the routine name in a procedure-call statement. Execution of the body of a function block occurs upon invocation of the function name with a function reference in an expression.

The repertoire of executable statements available in Pascal is described in Chapter 7. Also, various procedure-call statements for programming dynamic data structures is given in Section 5.3.11 using NEW, DISPOSE, MARK, or RELEASE calls; or the Pascal I/O procedure-call statements on RESET, REWRITE, GET, PUT, PAGE, READ, WRITE, READLN, and WRITELN are described in Chapter 8. Additional language extensions, described in Section 10.3 and 10.4, are provided as procedure-calls on the Perkin-Elmer Prefix routines and SVC support routines.

The syntax for a compound statement is the same as that for a body. A compound statement is also one or more statements, each separated from the other with a semicolon, and the entire statement-sequence is bracketed with the words BEGIN and END.

A compound statement is a mechanism for collecting a group of other statements or other compound statements into a single entity. Most Pascal structured statement constructs operate on a single statement, but a compound statement is always acceptable wherever any single statement construct is required inside a Pascal structured statement.

See Section 7.3.1 for further details on the compound statement, and its uses other than serving as the body of a block.

# CHAPTER 5
# DATA CONSTANTS, TYPES, AND VARIABLE SELECTORS

## 5.1 INTRODUCTION

Data used within a program can be either constant or variable. The process of identifying attributes of the data is known as giving it a type. Variable data may be globally, or locally, declared or the user may create dynamic variables by programmed commands. The means by which a declared or dynamic variable, is specified or accessed is called a variable-selector, in this Pascal.

Data may exist either internally or externally. Internal data is that which exists in memory during program execution. This chapter deals exclusively with Pascal constants, types and variables which are used to represent data within memory during program execution.

External data may exist before and after execution of the program. External data has a structure, called a file, and a data-type to describe that structure, known as a Pascal file-type. A Pascal program may communicate with its external environment by operations on files such as input, output, and some auxiliary positioning and initialization functions.

The file-type and I/O available in Pascal with the RESET, REWRITE, PUT, GET, READ, WRITE, READLN, WRITELN, PAGE procedure-call statements and predefined EOF, or EOLN functions are syntactically detailed in Chapter 8. Brief definitions of these predefined routines are summarized in Chapter 3, Section 3.5; the examples in this chapter make reference to them.

## 5.2 CONSTANTS

A constant is a specific value that does not change during the execution of a program. A constant may be represented literally or by name; i.e., by a literal-constant or by a constant-identifier. The several types of data, described fully in Section 5.3, may have their values represented by constants. Therefore, whether represented literally or by an identifier, the syntax of a constant in Pascal is:

## Constant

```
--------> constant-identifier --------->
    |                                ^
    |                                |
    |---> enumeration-constant --->|
    |                                |
    |---> real-constant  --------->|
    |                                |
    |---> string-constant  ------->|
    |                                |
    V---> pointer-constant ------->|
```

where enumeration constant has the syntax:

## Enumeration-Constant

```
--------> user-defined-enumeration -------->
    |         constant-identifier        ^
    |                                    |
    |-----> constant-identifier ----->|
    |                                    |
    |------> character-constant ----->|
    |                                    |
    |------> boolean-constant  ------>|
    |                                    |
    V------> integer-constant  ------>|
```

A constant definition introduces an identifier as the name of a constant value, and that name becomes a constant-identifier. A user-defined enumeration type-definition also introduces identifiers as the names of its values. Within the scope of that type definition, those identifiers become constant-identifiers.

A constant, then, may be a constant-identifier, an enumeration constant, a real constant, a string constant, or pointer-constant. The enumeration constants are those constants of ordinal types, including a Boolean constant, a character constant, or an integer constant, or a constant-identifier of the foregoing, or a constant-identifier listed in a user-defined enumeration type-definition.

Boolean, character, and integer constants are defined to be enumeration constants since their possible values are discrete and ordered scalar values. Real constants are continuous, non-discrete scalar values. If numeric (integers or reals), the constants or constant-identifiers may be signed. String constants, either literal or declared string constant-identifiers are considered structured arrays of characters, which may be assigned to string variables of the same fixed length, or written out, in their entirety. Two predefined literal Boolean constant-identifiers, TRUE and FALSE, are available. There exists only one predefined literal pointer constant namely, NIL, which is a reserved word symbol (not an identifier) i.e., whose meaning that a pointer is pointing to nothing at all, cannot be overridden by redeclaring the word NIL, as can be done with identifiers.

The syntax for coding literal values of each of these several types of data as literal constants is presented in Section 3.3.4 on Literal Constants.

The syntax for programming a constant definition to introduce an identifier as the name of a constant value to become a constant-identifier in the CONST constant declarations part is presented in Section 4.2.2.

A constant may be used as a factor within an expression or anywhere an "expression" is used (expressions are detailed in Chapter 6). Enumeration constants are used as case-labels, in a CASE statement. The value established for a constant-identifier is substituted for the constant-identifier, wherever it is used (referenced in a syntactically correct construct).

Examples of constant-identifiers and constant values of the various Pascal data types are given in each of the following Sections 5.3.1 through 5.3.11 respective to each data type described.

## 5.3 DATA TYPES

A data type determines the kind of possible values that data may assume. Only certain operations are permissible between data of particular data types. The collection of values that a data type defines are those values which a variable of that type may assume. A data type also determines certain attributes of data; i.e., size or structure.

There are several predefined data types in Pascal, and, in addition, the user may create his own data type definitions. Types may be defined in the type definition part of a prefix, program, module, or routine. An individual type definition within the TYPF part has the syntax:

### Type-Definition

```
--->identifier---> = --->type---> ; --->
```

A type-identifier type-definition must reside in a Type Definition part as given in Section 4.2.3, headed by the keyword "TYPE". The format of a type-definition is an identifier, followed by an equal sign, followed by a type, and ending with a semicolon. When a type definition introduces an identifier as the name of a data type, that identifier becomes known as a type-identifier, and is visible in scope from its point of declaration and to the block containing that declaration. It may then be used as a type, to attribute its nature to data variables or to define other type-identifiers. In general, a data type definition cannot refer to its own type-identifier. However, a pointer-type may refer to a data type before that data type has been defined.

The several kinds of data types in Pascal are: type-identifiers (either predefined or as defined by a previous type definition), the several enumeration or ordinal types, REAL and SHORTREAL data types, an array-type, record-type, set-type, pointer-type, or a file-type.

The enumerations are classified as those types of data which are either the predefined enumeration types or non-predefined enumeration types. The predefined enumeration types are character, Boolean, byte, integer and shortinteger data which by their nature implicitly have the attribute of enumeration, and they have predefined type-identifiers. The non-predefined user-specified enumeration types require the user to explicitly define the type's specific values, i.e.; and they may be given a type-identifier. The user specifies enumerations by listing their named values in increasing order or as a subrange determined by two specific values, a minimum and maximum value. The syntax of the language construct, type, is therefore defined by:

### Type

```
----------> type-identifier --------->
      |                         ^
      |--> enumeration-type -->|
      |     (ordinal-types)    |
      |                        |
      |--------> REAL -------->|
      |                        |
      |-----> SHORTREAL ------>|
      |                        |
      |-----> array-type ----->|
      |                        |
      |-----> record-type ---->|
      |                        |
      |-------> set-type ----->|
      |                        |
      |-----> pointer-type --->|
      |                        |
      V------> file-type ----->|
```

The file-type is discussed in Chapter 8.

Data have certain attributes according to their type. A simple type does not contain component parts and is one which can be operated upon as a whole, i.e., without selecting a component part. The simple types have scalar values. The simple scalar types are the predefined enumeration (character, Boolean, byte, integer, and shortinteger) types; the non-predefined enumeration types (user-defined enumerations and subranges); the REAL and SHORTREAL; and the pointer-type. All of these simple types can be operated on only as a whole, as they do not contain component parts.

Some data are defined in terms of other types and may be operated upon either as a whole or by selecting one of their component parts. These are considered structured types. The structured types of data are the arrays (including strings), records, and set types. These structured types contain component parts. An array contains array-elements, a record contains record-fields, a set contains members, and a string-variable contains characters as its array-elements. Although a literal or named string-constant may be assigned to string-variables of the same fixed length, in its entirety, the literal-string or string constant-identifier can only be operated upon in its entirety, not by component-parts. Although a pointer-type is defined in terms of other types, a pointer-variable can only be operated upon as a whole. When a pointer-variable is pointing to a structured target-type, the target-variable may be operated on either as a whole, or by selecting one of its component parts.

A data type is ordered if there is a meaningful relationship "greater than" or "less than" among its values. All simple types, except for pointer-types, are ordered. String-types and set types are also ordered but with limitations. Only strings of the same fixed length may be compared. Comparisons of sets, using the relational operators, become set-operations in the usual mathmatical sense.

A data type is discrete if there are meaningful relations of predecessor or successor among its values. All predefined and user-defined enumeration types and their subranges are discrete types. The discrete types are BOOLEAN, CHAR, BYTE, SHORTINTEGER, INTEGER, user-defined enumeration types, and subranges of any of these discrete types. All other types are considered nondiscrete types. REAL and SHORTREAL types are continuous types, i.e., nondiscrete scalars. The functions PRED or SUCC are defined for discrete types, but not for the REAL or SHORTREAL non-discrete types.

The following sections describe in detail the data types: enumeration or ordinal types, array-type, REAL and SHORTREAL type, record-type, set-type, and the pointer-type.

### 5.3.1 Enumeration Types (or Ordinal Types)

Data having the attributes of a finite collection of values which are discrete, ordered, and enumerable in value, are called enumeration or ordinal types.

Pascal identifies several predefined enumeration types of data: character data, Boolean data, byte, shortinteger and integer data. Data is declared as having these types by using these predefined type-identifiers: CHAR, BOOLEAN, BYTE, INTEGER and SHORTINTEGER, respectively, in a variable declaration.

Non-predefined enumeration types are those whose finite collection of values is defined by the user. The user may either list the values in increasing order or specify the minimum and maximum of a group of values to define a user-specified enumeration or subrange type. Refer to Section 5.3.5 for user-defined enumeration types. Refer to Section 5.3.6 for subrange type. Once a type-identifier is defined, that type-identifier is attributable to data as a type.

The syntax of several predefined enumeration types is defined by the language, and is not variable from one implementation to another. The syntax of enumeration type is defined by:

### Enumeration-Type (ordinal-types)

```
------------------> CHAR ------------------>
   |                                   ^
   |------------> BOOLEAN ------------>|
   |                                   |
   |--------------> BYTE ------------->|
   |                                   |
   |----------> SHORTINTEGER --------->|
   |                                   |
   |------------> INTEGER ------------>|
   |                                   |
   |-> user-defined-enumeration-type ->|
   |                                   |
   V----------> subrange-type -------->|
```

and where the user-defined enumeration type is defined by:

### User-Defined Enumeration Type

```
---> ( ---> identifier ---> ) --->
        ^                  |
        |                  |
        |<----   ,   <----V
```

and where the subrange-type is defined by:

### Subrange-Type

```
---> constant --->  ..  ---> constant --->
```

The following predefined Pascal functions apply to enumerations or ordinal type data:

| | |
|---|---|
| SUCC (x) | The result is the successor value of x (if it exists). |
| PRED (x) | The result is the predecessor value of x (if it exists). |
| ORD (x) | The result is the integer value which is the ordinal value of x within its enumeration type. For user-defined enumerations, the first identifier has ordinal value zero. |

### 5.3.2 Character Type

A predefined type-identifier, CHAR, is available in Pascal to type and identify character data. The finite set of values of the character type are the 128 ASCII characters listed in Appendix H. The character set required to write Pascal programs and characters as a basic language element are detailed in Section 3.2.

Values of type CHAR may be denoted by character literals as previously defined in Section 3.2. For example, the syntax of a character constant is:

**Character-Literal-Constant**

```
-------> ' -------> character -------> ' ------->
```

such as: 'A', 'a', 'B', 'b', '1', '!'

The predefined functions ORD, CHR, PRED, SUCC can be used to manipulate CHAR data. The ordering of characters is defined by their ordinal values.

**ORD**

If ch is an expression (or variable) of type CHAR, then ORD(ch) is of integer type and is the ordinal value of the character of ch. The value of ORD(ch) for any particular character can be found in Appendix H, or from the hex MSD/LSD coordinates of Table 3-1, the ASCII Character Set, as

```
    ORD(ch) = 16 * MSD + LSD.
```

For example, the ORD('M') is hex 4D, or decimal 77.

### CHR

If v is an expression of any integer type: BYTE, SHORTINTEGER, or INTEGER, and has a value between 0 and 127 inclusive, then CHR(v) is of type CHAR. The function CHR is the inverse of ORD as applied to the type CHAR: that is,

```
CHR(ORD(ch)) = ch; and ORD(CHR(v)) = v.
```

CHR(v) is not defined if v is outside the range from 0 to 127. For example, the CHR(77) is M, or the CHR(#4D) is M.

### PRED

If ch is an expression of type CHAR, then PRED(ch) is the predecessor of ch and also of type CHAR, and satisfies

```
ORD(PRED(ch)) = ORD(ch) - 1
```

PRED(ch) is not defined if the ordinal value of ch is zero. For example, the PRED('M') is L, and the PRED('m') is l.

### SUCC

If ch is an expression of type CHAR, then SUCC(ch) is the successor of ch and also of type CHAR, and satisfies

```
ORD(SUCC(ch)) = ORD(ch) + 1
```

For example, the SUCC('M') is N, and the SUCC('m') is n. SUCC(ch) is not defined if the ordinal value of ch is 127; i.e., when ch := '(:127:)'. All comparisons using the relational operators, are valid on quantities of type CHAR. The order of values of type CHAR is determined by their ordinal numbers.

To convert the digit characters to the number of the digit, it is true that the expression:

```
ORD('5') - ORD('0') = 5
```

To use the predefined type-identifier CHAR, variables of CHAR type may be introduced in a VAR variable declarations:

```
VAR  CH : CHAR;
     LETTER, ITEM : CHAR;   M,N : CHAR;
```

Then the character typed variables may be assigned literal character constant values in assignment statements, such as occur in the following compound statement:

```
BEGIN
  CH := 'B';
  LETTER := 'u';
  ITEM := 'z';
  M := 'z';
  N := '!';
END
```

Character type variables may be assigned the values of other character type variables in assignment statements, such as:

```
LETTER := ITEM;
CH := LETTER;
M := N;
```

The relational operators: <, >, <>, <=, >=, or = may used for comparisons of character data, in relational expressions yielding Boolean results.

For example, the following reads a single character from the INPUT file into CH; and compares the character in CH to several literal character constants in the relational expressions yielding Boolean values in the IF statements:

```
PROGRAM TEXTIO(INPUT,OUTPUT); VAR CH:CHAR;
    BEGIN
      READ(CH);
      IF CH = 'T'
        THEN WRITELN(CH)
        ELSE IF CH ='F'
          THEN WRITELN(CH)
          ELSE IF (CH >= 'A') AND (CH <= 'Z')
            THEN WRITELN(CH)
            ELSE WRITELN('INPUT WAS NOT T,F, OR CAP LETTER');
    END.
```

The above sample code suggests a method for initially reading at least the initial T or F of the Boolean values written out to textfiles as TRUE or FALSE, as there is no automatic reading of variable-length strings, e.g., such as TRUE, true, FALSE, false, etc., from textfiles into Boolean variables provided by Pascal.

There are no other operators, other than assignment or relational, which manipulate character data directly, e.g., they cannot be used in arithmetic expressions, i.e., 1 + 'A' is illegal; nor can characters be used as the direct operand factors of the Boolean operators, i.e., 'A' AND 'B' is illegal. However, sets of characters may be formed for set operations, and set-test-membership expressions, or arrays of characters (one-dimension) may be formed for processing strings.

Characters may be read from an input textfile one at a time using the predefined procedure READ, which does not skip automatically over blanks (or EOLN's, as READLN does) for character type data as it does for integer or real data. The following example code demonstrates one method of skipping over blanks (including any EOLN blank in textfile buffer from carriage-returns typed on terminals) and that the literal character constants can be used as case-statement labels.

```
CONST
  SPACE = ' ';
VAR
  CH : CHAR;

BEGIN
  REPEAT        {bypasses blanks including EOLN space}
    READ(CH)
  UNTIL CH <> SPACE;
   {CH contains first non-blank character occuring on INPUT}
  CASE CH OF
    'A' : statement;
    'B' : statement;
    'C' : statement;
     .
     .
     .
    'Z' : statement;
    '0','1','2','3','4','5','6','7','8','9' : statement;
    OTHERWISE  statement
    END;
END
```

To write a single character, the statement:

```
WRITE(CH)
```

writes the character value in the variable CH to the OUTPUT file.

The statement:

```
WRITE(CH :  total_width)
```

where total_width is an integer value, writes the character in CH, preceded by total_width - 1 spaces, to the OUTPUT file. Refer to Chapter 8 on I/O definitions and write-parameters.

Outputting several character data variables, such as those from the first assignment statements in this section above, is accomplished by:

```
WRITE(CH,LETTER,ITEM,M,N)
```

which would produce on the OUTPUT file:

```
|column 1 of textfile field
|
V
Buzz!
```

Blanks may be explicitly output to indent or position character output, on textfiles. For example, using a total_width of four, to indent:

```
WRITE(' ':4,CH,LETTER,ITEM,M,N)
```

would then produce on the OUTPUT file:

```
|column 1 of textfile field
|
V
    Buzz!
```

whereas the following use of total_widths:

```
WRITE(' ':3,CH:2,LETTER:2,ITEM:2,M:2,N:2)
```

would produce:

```
|column 1 of textfile field
|
V
    B u z z !
```

Note that although the Pascal compiler maps lower-case letters of the source program into upper-case letters, within a literal string or character constant the lower-case letters are not mapped, affording the user the ability to output either lower or upper case letters.

Strings of characters are considered structured types, i.e., an array of characters; such that literal strings cannot be used as case-statement labels. However, literal strings are easily output as write-parameters within WRITE or WRITELN statements. For example,

```
CH := '1';
WRITE('MESSAGE',CH,':');
WRITELN('OUTPUTTING LITERAL STRINGS IS EASY AS 1,2,3');
```

produces on the OUTPUT file:

```
|column 1 of first textfile field
|
V
MESSAGE1:OUTPUTTING LITERAL STRINGS IS EASY AS 1,2,3
```

Note that the output of the WRITE statement is buffered until a WRITELN statement is executed in Pascal; or until the entire file-variable control block (for textfiles, 256 character bytes) is full.

The user may use the CHAR predefined type-identifier to identify character component types of structured types, such as array components or fields of records; or anywhere a type-identifier is syntactically acceptable, such as in an index-type, or parameter type-identifier declaration, etc. The user is cautioned that as the type CHAR has 128 values, the entire type CHAR is not allowed to define the type of a record-variant discriminating tag-field, in this implementation of Pascal, as the tag-field type, used to differentiate variants, is limited to a type whose ordinal values lie in the range 0..31. However, as a CASE statement allows the case-selector to be of a type of 128 values, a case-selector can be of CHAR type. Also, as a set-type requires that its base-member-type be of ordinal type with its values in the range 0..127, sets of characters may be formed from the base-member-type CHAR. Refer to Section 5.3.9.1 for further details on string-types as an application of arrays to manipulate strings of character data. For example, a string-type may be introduced in the TYPE declarations, as an array-type of characters:

```
TYPE
    STREAM = ARRAY[1..256] OF CHAR;
```

The following example, declares an array variable of 256 characters of the above string-type; and its integer index:

```
VAR  LINE : STREAM;        {string-array variable declaration}
     INDEX : INTEGER;      {integer variable declaration}
```

The user may define functions whose resultant value is of type CHAR. Functions are defined in the routine declarations part of a block (see Chapter 4 or 9). An example of a user-written function whose function-value type-identifier is CHAR, follows.

```
FUNCTION NEXT (PARSER:INTEGER;STRING256:STREAM):  CHAR;
BEGIN
  IF PARSER < 257
    THEN
       NEXT := STRING256[PARSER]
    ELSE
       NEXT := '(:13:)' {control carriage-return character}
END;
```

When the string-array LINE is initialized with character values, after the non-textfile I/O, reading fixed or variable-length strings, such as with a GET(F) or READ(F,LINE) where VAR F:FILE OF STREAM; [and note that the carriage-return is visible with ordinal value decimal 13 in non-textfile terminal input] then the character-typed function NEXT may be called, passing to it INDEX for parameter PARSER and LINE for parameter STRING256, and receiving a character as the value of the function NEXT.

```
PROGRAM NONTEXTIN(F,OUTPUT);
TYPE STREAM=ARRAY[1..256] OF CHAR;
VAR F:FILE OF STREAM;
    LINE:STREAM; INDEX:INTEGER; CH:CHAR;
{assume the above function NEXT is declared here}
BEGIN
RESET(F);      {RESET automatically performs first read of F}
LINE := F^;                 {See Chapter 8 on file buffer F^}
INDEX := 0;
REPEAT
    INDEX := INDEX + 1;
    CH := NEXT(INDEX,LINE);
    {CH contains next character from LINE or last cr}
    {CH processable here, for example: it can be written out}
    WRITE(CH);
    UNTIL (INDEX = 256) OR (CH = '(:13:)');
WRITELN;
WRITELN('ENTERED LINE''S LENGTH=',INDEX-1,' CHARS');
WRITELN('AGAIN, FIRST LINE ENTERED WAS:');WRITELN(LINE);
 ...
GET(F);
 ...
END.
```

When the file F is declared as VAR F:FILE OF CHAR, only a single character will be requested/allowed entered per line.

Note that even if a constant were defined to be of the string-type such as:

```
CONST    STRINGC =  'DONE';
```

and the type of STRINGC is a one-dimensional array of characters, of length four; it can only be assigned to a string variable of the same fixed length of 4; and can only be assigned in its entirety, or passed to a value parameter in its entirety; i.e., a constant string cannot be indexed into one character at a time. That is, we can write:

```
CONST  STRINGC = 'DONE';
TYPE   STRING4 = ARRAY[1..4] OF CHAR;
VAR    STRINGV : STRING4;  CH1,CH2,CH3,CH4 : CHAR;
BEGIN
  STRINGV := STRINGC;
{Then the variable selector can be indexed into, to get at
individual characters within the variable, not the constant}
  CH1 := STRINGV[1];
  CH2 := STRINGV[2];
  CH3 := STRINGV[3];
  CH4 := STRINGV[4];
  WRITELN(CH1,CH2,CH3,CH4);
  WRITELN(STRINGV);
END;
```

Both WRITELNs together produce on the OUTPUT file:

```
|column 1 of textfile field
|
V
DONE
DONE
```

In this implementation of Pascal, strings of any length are permitted to be passed to value-parameters of any string-type. As required of all arguments passed to variable-parameters, strings passed to variable-parameters must be of "identical" type. Refer to Section 5.3.9.1 on strings. See Section 6.2 on establishing Pascal type-compatibility. See Section 9.6.5 on argument to parameter type-compatibility rules.

### 5.3.3 Boolean Type

Data of the Boolean type has two logical values, true and false. The name of this predefined type is the predefined type-identifier, BOOLEAN.

Its two values are literally represented by the predefined Boolean constant-identifiers, TRUE and FALSE.

Variables may be introduced to be of the Boolean type in the VAR declarations part, such as:

```
VAR
      SWITCH : BOOLEAN;
      FLAG, EMPTY, FOUND : BOOLEAN;
      A,B : BOOLEAN;
```

Then the Boolean typed variables may be assigned values in assignment statements, such as in the following compound statement:

```
BEGIN
  SWITCH := TRUE;
  FLAG := FALSE;
  EMPTY := TRUE;
  FOUND := FALSE;
  A := TRUE;
  B := TRUE;
END
```

Boolean type variables may be assigned the values of other Boolean type variables in assignment statements, such as:

```
FLAG := EMPTY;
SWITCH := FOUND;
A := B;
```

Logical operations can be performed on Boolean variables or other Boolean valued expressions such as parenthesized relational expressions or set-test-membership expressions, by using the Boolean operators: AND, OR, and NOT.

The Boolean operator NOT performs the logical negation of its following single operand factor, of Boolean type.

The Boolean operator AND performs the logical product of its two operand factors.

The Boolean operator OR performs the logical sum of its two operand factors.

The result of a Boolean expression is a Boolean value, either TRUE or FALSE.

Table 5-1 defines the truth table values of Boolean operations performed on two Boolean variables, A and B, having the assumed values shown in the table under A and B.

## TABLE 5-1 BOOLEAN OPERATIONS

| A | B | NOT A | A AND B | A OR B |
|---|---|---|---|---|
| TRUE | TRUE | FALSE | TRUE | TRUE |
| TRUE | FALSE | FALSE | FALSE | TRUE |
| FALSE | TRUE | TRUE | FALSE | TRUE |
| FALSE | FALSE | TRUE | FALSE | FALSE |

Note that NOT is a prefix operator, and the other operators are infix operators. Operands of a Boolean expression are evaluated from left to right. In a Boolean expression using all three Boolean operators, NOT is applied first, then AND, and then OR.

Boolean values can be produced from operations on other values by tests for set membership, by comparisons in relational expressions, and by certain predefined or user-defined functions. Comparisons are tests for equality, inequality, and ordering. Using the relational operators (=, <>, <, <=, >, or >=) to perform comparisons within a relational expression yields a resultant expression value of type Boolean. Tests for equality and inequality may be performed on data of all types; tests for ordering may be performed only on same ordered types. If A and B are of identical types, or one is assignment-compatible to the type of the other (see Section 6.2), then the comparisons:

| | |
|---|---|
| A = B | has the Boolean value true if A and B have the same value, and is false; otherwise, |
| A <> B | has the Boolean value (A = B). |

The predefined function ORD is applicable to Boolean expressions;

```
ORD(FALSE) = 0          ORD(TRUE) = 1
```

The ordering of Boolean values is the same as that of their ordinals, i.e.,

FALSE < TRUE.

Boolean expressions (involving the Boolean operators: NOT, AND, or OR), and expressions which yield a Boolean value are very useful in decision making and repetitive execution control statements, such as the IF statement, the WHILE statement, and the REPEAT statement. However, cautioned is advised in forming a FOR statement, such as:

```
FOR V := A TO B DO statement;
```

when A and B are Boolean values because:

```
 FOR V := FALSE TO TRUE DO statement;
FOR V := TRUE DOWNTO FALSE DO statement;
```

only execute the for-controlled statement twice; and

```
 FOR V := TRUE TO FALSE DO statement;
FOR V := FALSE DOWNTO TRUE DO statement;
```

define an empty progression for the for-control variable V, and the for-controlled statement would not be executed at all.

For examples of expressions yielding Boolean values, these declarations are assumed:

```
VAR
    COUNT,SUM : SHORTINTEGER;
    SIZE, WEIGHT : REAL;
    FINISHED : BOOLEAN;
    ITEM : CHAR;
    LETTERS : SET OF 'A'..'Z';
    CODE : INTEGER;
```

then the following are expressions which yield a Boolean value:

```
COUNT < SUM
(COUNT = SUM) AND (SIZE >= WEIGHT) AND FINISHED
```

When relational expressions are separated by Boolean operators, they must be bracketed by parentheses.

An expression yielding a Boolean value is also one which involves the test set membership operator, IN. For example, consider the statement-sequence:

```
LETTERS := ['A','E','I','O','U'];
READLN(ITEM);
IF ITEM IN LETTERS        {tests set membership}
  THEN statement
  ELSE statement;
```

The predefined function ODD, given an integer argument, yields a Boolean value. If the integer argument x is odd, the ODD(x) returns the value TRUE; if the integer argument x is even, the ODD(x) returns the value FALSE. For example, given the declarations:

```
    VAR
        SWITCH : BOOLEAN;     TOTALS : INTEGER;
        AMOUNT : SHORTINTEGER;    INDEXB : BYTE;
```

the following statement-sequence reflects uses of ODD:

```
    AMOUNT := 6;
    INDEXB := 127;
    TOTALS := AMOUNT + INDEXB + 1;
    SWITCH := ODD(AMOUNT);       {SWITCH becomes FALSE}
    SWITCH := ODD(INDEXB);       {SWITCH becomes TRUE}
    SWITCH := ODD(TOTALS);       {SWITCH becomes FALSE}
```

Function references to the predefined functions, EOF and EOLN, yield Boolean values. The argument of either function is a file-identifier, which would have its EOF or EOLN condition indicated, but when omitted returns the EOF or EOLN status on the predefined text file INPUT. Refer to the summary descriptions of predefined functions in Section 3.5. For an example of the use of the Boolean value returned for EOF, as the EOF function is an indicator that the end of a previously written textfile, e.g., the INPUT file, has been reached, the end of file condition may be checked for before reading. For example,

```
    IF  EOF             {send message to OUTPUT file}
      THEN WRITELN('END OF FILE OCCURRED ON INPUT FILE')
      ELSE WHILE NOT EOF DO
             BEGIN READ(SIZE); {process SIZE} ; END
```

or

```
    WHILE NOT EOF DO      {NOT EOF is a Boolean expression}
        BEGIN
        READLN;READ(ITEM);
        {...process ITEM...}
        END;
```

If FLAG is a Boolean variable, then FLAG may be referenced as a variable-selector in a write-parameter expression in WRITE or WRITELN statements. For example,

```
    WRITELN(FLAG);
```

outputs either "TRUE " or "FALSE" on the OUTPUT file.

Also, the user may define functions which return a Boolean value, with a FUNCTION declaration that specifies a function value type-identifier of BOOLEAN. Programming function definitions are detailed in Chapter 9. For an example,

```
    FUNCTION JACKPOT(MINIMUM,MAXIMUM,VALUE:INTEGER):  BOOLEAN;
      BEGIN
        IF (MINIMUM <= VALUE) AND (VALUE <= MAXIMUM)
          THEN
            JACKPOT := TRUE
          ELSE
            JACKPOT := FALSE;
      END;
```

This user-defined function, when referenced within an expression, yields a Boolean value. For example:

```
    READLN(CODE);
      IF JACKPOT(11,14,CODE)
        THEN COUNT := COUNT + 1;
```

adjusts the COUNT by one, for each integer CODE read in, that hits the jackpot of 11 <= CODE <= 14; i.e., where the function reference: JACKPOT(11,14,CODE), returns the value TRUE.

### 5.3.4 Integer, Shortinteger, and Byte

Data, whose values may be a finite set of successive whole numbers; i.e., integers, may be defined to have the types of integer data, shortinteger data, or byte data. These types are represented by the predefined type-identifiers INTEGER, SHORTINTEGER, or BYTE.

Literal representations of integer numbers are compiled to be of type: INTEGER. Subranges expressed with literal integer constants have an enclosing-type of INTEGER. Literal integer constants are defined by their syntax graphs in Section 3.3.4.

The type INTEGER represents 32-bit signed binary integers in the range -2147483648..+2147483647. That is, the type INTEGER includes all integer values representable in a processor fullword of memory (in two's complement notation). The minimum value is therefore defined as -(2**31), and the maximum value is (2**31)-1. A predefined identifier, available as an implementation-defined constant, is MAXINT, the maximum positive integer value of (2**31)-1 or 2147483647.

The type SHORTINTEGER represents 16-bit signed binary integers in the range -32768 .. +32767. That is, the type SHORTINTEGER includes all integer values representable in a processor halfword of memory (in two's complement notation). The minimum value is therefore -(2**15), and the maximum positive integer value of (2**15)-1. The type SHORTINTEGER is conceptually a subrange of INTEGER. A predefined identifier, MAXSHORTINT, is available as an implementation defined constant, whose value is 32,767.

The type BYTE represents 8-bit unsigned binary integers in the range 0 .. 255. BYTE type data are integer values representable in a processor byte of memory but not in two's complement notation. BYTE type data are only positive integers, where the minimum value is 0 and the maximum value is (2**8)-1 or 255. The type BYTE, therefore, is conceptually a subrange of SHORTINTEGER and INTEGER. The advantage of SHORTINTEGER over INTEGER is that the variables of type SHORTINTEGER require less storage. On the current 32-bit series, a SHORTINTEGER requires a halfword (2 bytes) while an INTEGER requires a fullword (4 bytes).

The ordering of INTEGER, SHORTINTEGER, and BYTE data is the usual mathematical ordering of integer values. All comparisons are legal. All three types are compatible across arithmetic expressions. The arithmetic operators which apply to these types are given in Table 5-2. These operators are all infixes; and the operators + and - are also prefixes. The resultant values are of type INTEGER, except for the division sign operator, (/), which produces a result of type REAL. Note that the second operand of /, DIV, or MOD must not have a value of zero. The second operand of MOD should not be negative.

TABLE 5-2 INTEGER, SHORTINTEGER, AND BYTE ARITHMETIC OPERATORS

| OPERATOR SYMBOL | OPERATION |
|---|---|
| + | Addition |
| - | Subtraction |
| * | Multiplication |
| / | Division, producing REAL result |
| DIV | Division, producing INTEGER result (truncated toward 0) |
| MOD | Modulo, where A MOD B = (A - (B * K)); for integral K such that 0 <= A MOD B < B. |

All three integer types are assignment-compatible and they may be mixed freely in expressions. However, if an expression of INTEGER type is assigned to a variable, or passed to a value-parameter, of SHORTINTEGER or BYTE type, or if an expression of SHORTINTEGER or INTEGER type is assigned to a variable, or passed to a value parameter, of BYTE type, and the integral value of the integer is out-of-range of the shorter type, then a significant truncation error may occur, as the shorter integers are implemented as subranges of the INTEGER type, and occupy smaller storage units. If the program was compiled under option RANGECHECK, and BOUNDSCHECK, a run time error, containing the message VALUE RANGE ERROR, or PARAM RANGE ERROR is issued.

Also, if the integers are mixed with reals or shortreals, or across the / real division operator, the real-valued expression becomes non-assignable to integer variables.

The following predefined functions apply to integer type data. They can accept one argument of type INTEGER, SHORTINTEGER, or BYTE.

| | |
|---|---|
| ABS(x) | The result is of the same type as the argument x and is the absolute value of the integer x. |
| CHR(x) | The result is of type CHAR and is the character with the ordinal value of the integer x. The value of the argument x should be in the range of 0 to 127. |
| CONV(x) | The result is of type REAL and is the real value corresponding to the integer x. |
| SHORTCONV(x) | The result is of type SHORTREAL and is the shortreal value corresponding to the integer x. |
| ODD(x) | The result is of type Boolean and is true if the integer x is odd, and the result is false if the integer x is even. |
| SQR(x) | The result is of the same type as the argument x, and is the square of the value of x. |

The following predefined functions produce an integer-valued result. They accept one argument of any type.

| | |
|---|---|
| ADDRESS(x) | The result is of type INTEGER and is an integer which is the 32-bit representation of the address of the argument x. The argument x cannot be a literal constant or a constant-identifier. |
| SIZE(x) | The result is of type INTEGER and is an integer which is the size in bytes of the argument x. The |

argument x cannot be a literal constant or a constant-identifier.

The following predefined functions produce an integer-valued result and require no arguments.

| | |
|---|---|
| LINENUMBER | The result is of type INTEGER and is an integer which is the current source line number containing the function reference to LINENUMBER. |
| STACKSPACE | The result is of type INTEGER and is the number of bytes remaining between the heap and the stack at the run time of executing the function references to STACKSPACE. |

Some examples of declaring INTEGER values as constants in the constant declarations part are:

```
    CONST


        ONE = 1;    TWO = 2;

        NEGONE  = -1;
        NEGTWO  = -TWO;

        HUNDRED = +100;
        ONETHOU =  1000;
        TWOTHOU =  2000;

        HEXADDR1 = #01234DEF;     {example hexadecimal integer}
        HEXADDR2 = #867B9A5C;     {example hexadecimal integer}

        MOSTNEG   = #80000000;    {decimal value -2147483648}
        LEASTNEG  = #FFFFFFFF;    {decimal value -1}
        ZERO      = #0;           {decimal value  0}
        LEASTPOS  = #0C000001;    {same as #1; decimal value 1}
        MOSTPOS   = #7FFFFFFF;    {same as MAXINT; +2147283647}
```

Integer variables may be declared in the variable declaration part of a block:

```
    VAR
        B1,B2 : BYTE;
        I,J,K,L : SHORTINTEGER;
        DIMENSION, ANSWER1, ANSWER2 : INTEGER;
```

User-defined functions may be written whose function value is BYTE, INTEGER or SHORTINTEGER; and integer-valued parameters (value or variable) may be declared as BYTE, INTEGER, or SHORTINTEGER in parameter-lists. Note the use of integers in the expressions of the assignment statements within the functions.

For example:

```
  FUNCTION DIFFERENCE_OF_SUMS(A,B,C,D:  SHORTINTEGER):INTEGER;
    BEGIN
      DIFFERENCE_OF_SUMS := (A+B) - (C+D);
    END;

  FUNCTION MULTIPLY_THE_QUOTIENTS(A,B,C,D :SHORTINTEGER):INTEGER;
    BEGIN
      MULTIPLY_THE_QUOTIENTS := (A DIV B) * (C DIV D);
    END;

  FUNCTION PERIMETER(SIDE1,SIDE2:  BYTE):SHORTINTEGER;
    BEGIN
      PERIMETER := (2*SIDE1) + (2*SIDE2);
    END;
```

This program body also reflects the assignment of values to integer variables, and references to the functions defined above; and outputting expressions of integers.

```
    BEGIN
      B1 := 255;    {largest value (magnitude) for a byte}
      B2 := 2;
      I := 1;       {a integer may be assigned an integer value}
      J := TWO;     {TWO declared as constant-identifier of 2}
      K := 9 MOD 6; {K becomes 3; remainder of 9 divided by 6}
      L := I + K;   {L becomes 4; the value of expression I+K}
      ANSWER1 := DIFFERENCE_OF_SUMS(I,J,K,L);
      ANSWER2 := MULTIPLY_THE_QUOTIENTS(I,J,K,L);
      DIMENSION := PERIMETER(B1,B2);
      WRITELN(ANSWER1,ANSWER2,DIMENSION);
      WRITELN(ANSWER1:5);
      WRITELN(ANSWER2:5);
      WRITELN(DIMENSION:5);
    END.
```

and the output would be:

```
|column 1 of first textfile field
|
V
        -4         0       514
   -4
    0
  514
```

Note the difference in the output format of these integer values. In the first WRITELN statement the integer-valued expressions are output with the assumed default total_widths of integer fields of ten positions. In the subsequent WRITELN statements, the integer-valued expressions are output with only total_widths of five, as specified in the write-parameters. See Chapter 8 for write-parameter details.

### 5.3.5 User-Defined Enumeration Type

In addition to the predefined enumeration types of character, Boolean, and byte, integer, and shortinteger data; Pascal provides a powerful scalar data-type that the programmer himself defines, to represent and program abstract data. This is the user-defined enumeration data-type, which is simply a list of the values, by name, that may be assumed by a variable of the type.

To create the name of a user-defined enumeration type, introduce a type-identifier in the TYPE definition part, such as:

```
    TYPE user-defined-enumeration-type-identifier = type;
```

where the identifier before the equal sign becomes the name of, or the user-defined enumeration type-identifier of, the user-defined enumeration "type" being defined after the equal sign.

A user-defined enumeration type has the context-free syntax:

**User-Defined Enumeration Type**

```
---> ( ---> identifier ---> ) --->
        ^               |
        |               |
        |<----   ,  <----/
```

The values of this data-type are denoted by the identifier(s) that the programmer encodes in the list. These identifiers, once introduced in such a list, become enumeration constant values.

A maximum of 128 identifiers may be defined in any one list. For example, several user-defined enumeration type-identifiers may be established in the TYPE declarations, as follows:

```
TYPE
    PROTEIN = (VEAL,BEEF,LAMB,PORK,CHICKEN,TURKEY,DUCK,
                BLUE,SOLE,SHRIMP,LOBSTER,FLOUNDER) ;
    REDWINES = (BURGUNDY,SHERRY,PORT);
    WHITEWINES = (CHENINBLANC,CHABLIS_BLANC,GRAPE);
    ROSEWINES = (ROSE,CHABLIS_ROSE,CHIANTI);
    WINE = (RED,PINK,WHITE);
    GREEN_VEGETABLE = (ARTICHOKE,ASPARAGUS,SPROUTS,PEAS);
    COLOR_VEGETABLE = (BEETS,CARROTS,CORN);
    CARBOHYDRATE = (POTATOE,RICE,PASTA);
```

which establish the identifiers listed within parentheses as enumeration constant-identifier values.

Also, the above declarations establish PROTEIN, REDWINES, WHITEWINES, ROSEWINES, WINE, GREEN_VEGETABLE, COLOR_VEGETABLE, and CARBOHYDRATE as user-defined enumeration type-identifiers. Variables may then be declared to be of these user-defined enumeration type-identifiers; and these variables may only assume the values defined in the lists of the above type-definitions. For example,

```
VAR
    ENTRE : PROTEIN ;
    SIDEDISH : GREEN_VEGETABLE;
    SIDE_ORDER2 : COLOR_VEGETABLE;
    STARCH : CARBOHYDRATE;
    WINECHOICE : WINE;
    REDWINE : REDWINES;
    WHITEWINE : WHITEWINES;
    ROSEWINE : ROSEWINES;
```

We can also declare variables to be of these types directly, although it is preferable to keep the type definition separate from the variable declaration, so the type-identifier is available to establish Pascal "identity" of type in separate variable, local variable, or variable parameter declarations. For example:

```
VAR
    ENTRE : (VEAL,BEEF,LAMB,PORK,CHICKEN,TURKEY,DUCK,
              BLUE,SOLE,SHRIMP,LOBSTER,FLOUNDER) ;
    REDWINE : (BURGUNDY,SHERRY,PORT);
    WHITEWINE : (CHENINBLANC,CHABLIS_BLANC,GRAPE);
    ROSEWINE : (ROSE,CHABLIS_ROSE,CHIANTI);
    WINECHOICE : (RED,PINK,WHITE);
    SIDEDISH : (ARTICHOKE,ASPARAGUS,SPROUTS,PEAS);
    SIDE_ORDER2 : (BEETS,CARROTS,CORN);
    STARCH : (POTATOE,RICE,PASTA);
```

An enumeration constant-identifier may not belong to more than one type in one scope. Distinct user-defined enumeration types are disjoint; and an identifier cannot simultaneously denote members of two different user-defined enumeration types (in the same scope). For example, we could not write:

```
TYPE
    SALAD = (LETTUCE,TOMATO,CARROT,CUKE,OLIVE);

    and

    COLORED_VEGETABLE = (BEETS,CARROT,CORN);
```

in a declarations section of a block that establishes one scope, because the enumeration constant-identifier CARROT may only be a member of one type. Established constant-identifiers cannot be re-introduced in the same scope by declaring variables such as:

```
VAR
    REDMEAT : (VEAL,BEEF,LAMB,PORK);
    FOWL    : (CHICKEN,TURKEY,DUCK);
    FISH    : (BLUE,SOLE,SHRIMP,LOBSTER,FLOUNDER) ;
```

Since we have the type PROTEIN already using these enumeration constant value identifiers within one scope, or area of visibility, it is not permissible to declare type subclassifications of PROTEIN such as:

```
TYPE
    REDMEATS = (VEAL,BEEF,LAMB,PORK);
    FOWLS    = (CHICKEN,TURKEY,DUCK);
    FISHES   = (BLUE,SOLE,SHRIMP,LOBSTER,FLOUNDER);
```

It is permissible to declare type subclassifications of PROTEIN as subranges (see Section 5.3.6), for example:

```
TYPE
    REDMEATS = VEAL..PORK;
    FOWLS = CHICKEN..DUCK;
    FISHES = BLUE..FLOUNDER;
```

and then declare variables to be of the subrange types:

```
VAR
    REDMEAT : REDMEATS; {or directly  REDMEAT:VEAL..PORK;}
    FOWL : FOWLS;       {or directly  FOWL:CHICKEN..DUCK;}
    FISH : FISHES;      {or directly  FISH:BLUE..FLOUNDER;}
```

The variables of an enumeration type may be assigned values by assigning the variables to enumeration constant-identifiers, as follows:

```
BEGIN
  ENTRE := BEEF;
  SIDEDISH := PEAS;
  SIDE_ORDER2 := BEETS;
  STARCH := RICE;
  WINECHOICE := WHITE;
  REDWINE := BURGUNDY;
  ...
END;
```

Mixed assignments are not allowed, as two variables of two different user-defined enumeration types are not compatible. We could not write:

```
BEGIN
  ENTRE := POTATOE;

  WINECHOICE := LOBSTER;

  FISH := CHABLIS_BLANC;
END;
```

because the above would result in a compilation error.

Pascal compiled-code automatically monitors variables of any user-defined enumeration type to ensure that they can only be assigned one of the legitimate values (i.e. proper enumeration constants). Run-time errors are generated during execution if illegal assignments are attempted. For example, sequentially executing the two statements:

```
ENTRE := SHRIMP;        {legal assignment of ENTRE}

FOWL := ENTRE;          {illegal at run-time for FOWL:=SHRIMP}
```

This data-type capability gives the user the ability to define his own data-types specifically gearing his problem solution to be manageable and meaningful abstractions of logical entities.

A variable of a user-defined enumeration type and its enumeration constant-identifiers can be used in much of the same ways as integers. For example, the FOR statement is often useful with user-defined enumerations defining the control-variable progression:

```
WRITELN(' COMBINATION MENUS ');
FOR ENTRE := VEAL TO FLOUNDER DO
    FOR SIDEDISH := ARTICHOKE TO PEAS DO
      FOR STARCH := POTATOE TO RICE DO
        FOR WINECHOICE := RED TO WHITE DO
          BEGIN
          COMPUTE_COMBO_PLATTERS(ENTRE,SIDEDISH,STARCH);
          SELECT_WINE(WINECHOICE);
          END;
```

Variables of the user-defined enumeration types may be passed to routines as variable or value parameters; and the enumeration constant-identifiers may be passed to routines as value parameters. If the above routines were defined to receive their arguments as value parameters, the sample invocations could be:

```
COMPUTE_COMBO_PLATTERS(VEAL,ASPARAGUS,RICE);
SELECT_WINE(RED);
```

A variable of the enumeration types may be a case-selector in a CASE statement and the enumeration constants used as case labels, (see Section 7.3.2). For example,

```
CASE ENTRE OF

  VEAL,BEEF,LAMB,PORK : WINECHOICE := RED;

  CHICKEN,TURKEY,DUCK : WINECHOICE := PINK;

  BLUE,SOLE,SHRIMP,LOBSTER,FLOUNDER : WINECHOICE := WHITE;

END;
```

User-defined enumeration types are effectively used in combination with set-type data (see Section 5.3.7), for example suppose we had previously declared the variables:

```
VAR
     REDSTOCK : SET OF REDWINES;
     ROSESTOCK : SET OF ROSEWINES;
     WHITESTOCK : SET OF WHITEWINES;
```

and these variables were assigned to the specific values of set-constructors:

```
REDSTOCK := [SHERRY,PORT];
ROSESTOCK := [CHABLIS_ROSE]
WHITESTOCK := [CHABLIS_BLANC,CHENINBLANC];
```

then another CASE statement could be:

```
    CASE WINECHOICE OF
      RED : IF PORT IN REDSTOCK THEN REDWINE := PORT
                 ELSE IF SHERRY IN REDSTOCK
                         THEN REDWINE := SHERRY
                 ELSE IF BURGUNDY IN REDSTOCK
                         THEN REDWINE := BURGUNDY
                 ELSE WRITELN('OUT OF STOCK');
      WHITE : IF CHENINBLANC IN WHITESTOCK
                 THEN WHITEWINE := CHENINBLANC
                 ELSE IF CHABLIS_BLANC IN WHITESTOCK
                         THEN WHITEWINE := CHABLIS_BLANC
                 ELSE IF GRAPE IN WHITESTOCK
                         THEN WHITEWINE := GRAPE
                 ELSE WRITELN ('OUT OF STOCK');
      PINK  : IF ROSE IN ROSESTOCK THEN ROSEWINE := ROSE
                 ELSE IF CHIANTI IN ROSESTOCK
                         THEN ROSEWINE := CHIANTI
                 ELSE IF CHABLIS_ROSE IN ROSESTOCK
                         THEN ROSEWINE := CHABLIS_ROSE
                 ELSE WRITELN('OUT OF STOCK');
    END;  {end of CASE statement}
```

The variables of a user-defined enumeration type can be used as array indices or subscripts, once the array is suitably defined, see Section 5.3.9. For example, if the following are defined as array-type-identifiers:

```
    TYPE
        PRICES = ARRAY[VEAL..FLOUNDER,ARTICHOKE..PEAS,
                       POTATOE..RICE, RED..WHITE] OF REAL;
        ENTRECOSTS = ARRAY[VEAL..FLOUNDER] OF REAL;
        SIDECOSTS = ARRAY[ARTICHOKE..PEAS] OF REAL;
        STCOSTS = ARRAY[POTATOE..RICE] OF REAL;
        WINECOSTS = ARRAY[RED..WHITE] OF REAL;
```

and the following array variables are declared to be of the above array-types (and assuming VAR PROFIT:REAL);

```
    VAR
        PRICE     :  PRICES;
        ENTRECOST : ENTRECOSTS;
        SIDECOST :  SIDECOSTS;
        STCOST  :   STCOSTS;
        WINECOST :  WINECOSTS;
```

then array-references using either enumeration constants or user-defined enumeration type variables, as indices, could be:

```
    ENTRECOST[VEAL] := 10.50;
    SIDECOST[ARTICHOKE] := 0.75;
    STCOST[POTATOE] := 0.25;
    ENTRE:=VEAL;SIDEDISH:=ARTICHOKE;STARCH:=POTATOE;
    WINECHOICE:=RED;
    WINECOST[WINECHOICE] := 6.00;
    PRICE[ENTRE,SIDEDISH,STARCH,WINECHOICE] :=  PROFIT +
        ENTRECOST[ENTRE] + SIDECOST[SIDEDISH] + STCOST[STARCH] +
        WINECOST[WINECHOICE];
```

Now we shall discuss some key points on the use of the user-defined enumeration typed data.

The order in which the identifiers are listed in an enumeration type definition is significant and establishes an inherent ordering of the values; which may be used to advantage in certain applications. For example, if we defined the enumeration type:

```
    TYPE
         DAY = (SUNDAY,MONDAY,TUESDAY,WEDNESDAY,
                THURSDAY,FRIDAY,SATURDAY);
```

and we declared the variables:

```
    VAR
         WEEKDAY,WEEKEND : DAY;
```

we should not write, (as the "statements" would not execute):

```
    FOR WEEKDAY := SATURDAY TO MONDAY DO
        statement;
    FOR WEEKEND := SATURDAY TO SUNDAY DO
        statement;
```

but we could write, and the "statements" will be executed :

```
    FOR WEEKDAY := MONDAY TO FRIDAY DO
        statement;
    FOR WEEKEND := SUNDAY TO SATURDAY DO
        statement;
```

Each of the enumeration constant-identifiers within a user-defined type has an ordinal number value, with counting starting from zero, and ascending by one, to 127; as there may be a maximum of 128 enumeration constant-identifiers listed in the defining type.

Three predefined functions are applicable to user-defined enumeration types. They are ORD, PRED and SUCC.

### ORD

The value of ORD(x) is the ordinal number of the argument expression x in the list of identifiers composing the user-defined enumeration type. If x is a variable of an enumeration type, the value of the ORD(x) is the ordinal number of the value of the variable x. If x is an enumeration constant identifier itself, the value of the ORD(x) is the ordinal number of x in the defining list of which x is a member.

### PRED

The value of the PRED(x) is the predecessor value, relative to the argument x, whether x is a variable of an enumeration type or an enumeration constant-identifier itself. That is, the result of PRED(x) returns the predecessor of x in the defining list of the user-defined enumeration type.

### SUCC

The value of SUCC(x) is the successor value, relative to the argument x, whether x is a variable of an enumeration type or an enumeration constant-identifier itself. That is, the result of SUCC(x) returns the successor of x in the defining list of the user-defined enumeration type.

For example, if we define the type:

```
TYPE
    BASIC_COLORS =(WHITE,BLACK,RED,GREEN,YELLOW,PURPLE,BLUE);
```

the ORD(WHITE) is zero, the ORD(BLACK) is 1, the ORD(BLUE) is 6.

Defining two variables to be of type BASIC_COLORS:

```
VAR   CARCOLOR, UPHOLSTERY : BASIC_COLORS;
```

then if CARCOLOR := BLACK; a subsequent ORD(CARCOLOR) produces the value 1; if CARCOLOR := RED; a subsequent ORD(CARCOLOR) produces the value 2; and so on.

User-defined types whose use readily improves the solubility and readability of programs are not transportable outside the program as strings, themselves. That is, their identifier names cannot be displayed by WRITE and WRITELN statements, nor assigned directly from input data with the READ and READLN statements.

Therefore, we could not write:

```
    WRITE(WHITE); WRITELN(CARCOLOR);
```

However, we could write:

```
    WRITELN(ORD(WHITE),ORD(BLACK),ORD(BLUE));
```

and the output would be:

```
    |column 1 of first textfile field
    |
    V
            0          1          6
```

or we could write:

```
    FOR CARCOLOR := WHITE TO BLUE DO WRITELN(ORD(CARCOLOR));
```

and the output would be:

```
    |column 1 of textfile field
    |
    V
             0
             1
             2
             3
             4
             5
             6
```

Using a FOR statement with a CASE statement:

```
    FOR CARCOLOR := WHITE TO BLUE DO
        CASE CARCOLOR OF
          WHITE : WRITELN('White');
          BLACK : WRITELN('Black');
          RED   : WRITELN('Red');
          GREEN : WRITELN('Green');
          YELLOW: WRITELN('Yellow');
          PURPLE: WRITELN('Purple');
          BLUE  : WRITELN('Blue');
        END;
```

would output:

```
White
Black
Red
Green
Yellow
Purple
Blue
```

Operations permissible on user-defined enumeration type data are assignment and the relational operators ( <, >, =, <>, >=, <=). The resulting expressions using the relational operators on user-defined enumerations have a Boolean result. For example, some assignments of the above typed variables are:

| <u>Assignments</u> | <u>Value of CARCOLOR</u> |
|---|---|
| CARCOLOR:=RED; | RED |
| CARCOLOR:=SUCC(CARCOLOR); | GREEN |
| CARCOLOR:=PRED(BLACK); | WHITE |

Some examples of Boolean expressions comparing user-defined enumeration typed data are:

| <u>Comparisons</u> | <u>Value of expression</u> |
|---|---|
| TUESDAY < WEDNESDAY | TRUE |
| BLACK > WHITE | TRUE |
| GREEN > YELLOW | FALSE |

Following the assignment statements:

```
CARCOLOR := RED;
UPHOLSTERY := GREEN;
```

a comparison of the variables may be made, for example:

```
IF CARCOLOR <> UPHOLSTERY
               THEN ...statement... ELSE ...statement...
```

As the first member of a user-defined enumeration list has no predecessor, and the last member of a list has no successor it is hazardous to program loops containing PRED and SUCC which would produce undefined values at these extremes. For example, instead of writing, which would fail upon executing SUCC(BLUE):

```
    CARCOLOR := WHITE;
    WHILE CARCOLOR <= BLUE DO
          BEGIN
            UPHOLSTERY := CARCOLOR;
            CARCOLOR := SUCC(CARCOLOR);
            ...{process}...
          END;
```

we write:

```
    FOR CARCOLOR := WHITE TO BLUE DO
          BEGIN
            UPHOLSTERY := CARCOLOR;
            ...{process}...
          END;
```

User-written functions may declare their function value type to be a user-defined enumeration type-identifier.

### 5.3.6 Subrange Type

Data of any enumeration or ordinal type, which is therefore discrete and ordered, may have a consecutive portion of that data-type represented by the subrange type. The subrange type has the syntax:

**Subrange-Type**

```
--->constant---> .. --->constant--->
```

A subrange type is represented by two constants separated by a range symbol (double period). The two constants must denote values of the same type, called the "enclosing type" of the subrange. The type of the constants may be integer, character, or a user-defined enumeration type. The constants may not be real or strings, i.e., subranges of reals or strings are not allowed. The constants may be literal constants or predefined or declared constant-identifiers. The first constant must be less than or equal to the second in the order defined for the enclosing type. The values of the subrange type are then all those values of the enclosing type that are greater than or equal to the first constant, and less than or equal to the second constant. That is, the first constant is the minimum value of the subrange, and the second constant is the maximum value of the subrange.

All operations and comparisons which can be performed on data of the "enclosing type" can be performed on data of the subrange type. A variable of the subrange type may be assigned to a variable of the enclosing type. In particular, an expression whose value is of the enclosing type can be assigned to a variable of the subrange type. However, if the expression's value does not lie within the subrange, a run time error occurs; and if the program were compiled under the RANGECHECK and BOUNDSCHECK options, the run time error message VALUE RANGE ERROR occurs.

Examples of subrange type declarations are:

```
    TYPE
       LETTERS = 'A'..'Z';    {capitals subrange of characters}
       DIGITS  = '0'..'9';    {digits subrange of characters}
       SIZE    = 1..256;      {integer subrange}
```

Note that subranges must represent consecutive values of its ennclosing type; it is not plausible to declare the hexadecimal digits thusly:

```
    TYPE
       HEX_DIGITS = '0'..'F';   {incorrect for intended subrange}
```

Although the enclosing range includes '0'..'9' and 'A'..'F', other characters are also present between '9' and 'A' in the enclosing type. They are the special characters defined in the ASCII set with ordinal values greater than that of '9' and less than that of 'A'. It is possible to declare the abstraction HEX_DIGITS as a set type whose set-constructor members are defined by mutually exclusive subranges. See Section 5.3.8 for an example.

Subranges of user-defined enumerations are also permissible. If the following user-defined enumeration type is declared:

```
     TYPE   COLOR=(RED,PINK,ROSE,BLUE,NAVY,ROYAL,AQUA);
```

then we may also have:

```
    TYPE
         RED_HUE  = RED..ROSE;
         BLUE_HUE = NAVY..AQUA;
```

Subrange types are most useful for representing the index-types of an array-type definition, (see Section 5.3.9).

Variables of the subrange-type may be declared in the variable declarations part of a block.

```
    VAR
        LEADING_CHAR : LETTERS;
        NUMERIC_CHAR : DIGITS;
        SUBSCRIPT    : SIZE;
        SHADE        : RED_HUE;
        INDEXER      : 1..10;
        BLUES        : BLUE..AQUA;
```

Note that LETTERS, DIGITS, SIZE, and RED_HUE are subrange-type-identifiers defining LEADING_CHAR, NUMERIC_CHAR, SUBSCRIPT and SHADE to be subrange variables; whereas INDEXER is being defined as a subrange of integer values 1 to 10 by a literal representation of the subrange-type: constant..constant. Also BLUES is a subrange variable being defined by a literal representation of the subrange-type BLUE..AQUA which determines the enclosing type of BLUES to be COLOR.

LEADING_CHAR is a subrange variable of enclosing type CHAR with its range restricted to 'A' to 'Z'. NUMERIC_CHAR is a subrange variable of enclosing type CHAR with its range restricted to '0' to '9'. SUBSCRIPT is a subrange variable of enclosing type INTEGER with its range restricted to 1 to 256. INDEXER is a subrange variable of enclosing type INTEGER with its range restricted to 1 to 10. SHADE is a subrange variable of enclosing type COLOR with its range restricted to RED, PINK, or ROSE; i.e., the subrange RED_HUE. BLUES is a subrange variable of enclosing type COLOR with its range of values restricted to BLUE, NAVY, ROYAL, or AQUA.

The advantage here is that subranges take the burden of programming range-checking or bounds-checking procedures off of the programmer and pass it onto the compiled code. The compiler can generate range/bounds checking code in the object code, such that during execution, when the value of a subrange variable is changed, it can checked to be within its intended limitations.

Note that subrange variables may be used where a variable of the enclosing type may be used. For example, subrange variables of INTEGER or CHAR enclosing types, may be used as arguments to the predefined READ, READLN, WRITE, WRITELN procedures; but subrange variables of a user-defined enumeration enclosing type may not. That is, we may write:

```
    READ(LEADING_CHAR);
    READLN(NUMERIC_CHAR);
    WRITE(SUBSCRIPT);
    WRITELN(INDEXER);
```

but we may not write:

```
    READ(SHADE);
    WRITELN(BLUES);
```

Any operator defined for variables of an enclosing type may be used with a subrange variable of the enclosing type. Subrange variables may be used in expressions and different subranges of the same enclosing type may be mixed in expressions. Subrange variables may be used on both sides of the assignment operator. For example, given the additional variable declarations:

```
    VAR
        ANSWER : INTEGER;
        TALLIES : ARRAY[1..10,SIZE] OF REAL;
```

we may write statements, such as:

```
    LEADING_CHAR := 'C';
    NUMERIC_CHAR := '3';
    SHADE := PINK;
    BLUES := ROYAL;
    FOR INDEXER := 1 TO 10 DO
      FOR SUBSCRIPT := 1 TO 256 DO
        TALLIES[INDEXER,SUBSCRIPT] := SUBSCRIPT / INDEXER;
    ANSWER := SUBSCRIPT + INDEXER;
    ANSWER := ANSWER MOD INDEXER;
    ANSWER := ANSWER + SUBSCRIPT DIV INDEXER;
    SUBSCRIPT := ANSWER;
```

However, a run time error results if an attempt is made to assign a value out of range to a subrange variable. For example, in the last statement above, when ANSWER is a value not in the range 1 to 256, a run time error results; if the program was compiled with the compiler option BOUNDSCHECK on. If the program was compiled with the compiler RANGECHECK on, then the index subrange variables indexing into the array are also checked for proper values to be in range of their index types; which are expressed as subrange types. The following illegal assignments result in a run-time error message: VALUE RANGE ERROR,

```
    LEADING_CHAR := '8';
    NUMERIC_CHAR := 'M';
    SUBSCRIPT := 1000;
    SHADE := NAVY;
    BLUES := ROSE;
    INDEXER := 256;
```

even though the values are constants of their enclosing types.

Subrange variables may also be passed to functions as arguments, if the function is defined to receive values as value-parameters of the subrange type itself, of the enclosing type of the subrange variable, or of a type assignment compatible to either the subrange type or the enclosing type. However, the resultant value of the function need not necessarily return a value within the subrange. For example,

```
    INDEXER := 10;
    ANSWER:= SQR(INDEXER);
```

Here the answer returned is 100 which is not in the range 1..10 of INDEXER; but is allowable for assignment of the function-value to the integer variable ANSWER.

Passing any argument variable to a routine's variable parameters requires identity of type to the parameter's type-identifier.

### 5.3.7 REAL and SHORTREAL Data Types

The data type for real-valued data, as approximations of real numbers, is defined by the predefined type-identifiers REAL or SHORTREAL.

The types REAL and SHORTREAL are models of floating point numbers, with implementation-defined sets of possible values. For this implementation, the values include zero and certain nonzero numbers with absolute values between 16.0**(-65) and approximately 16**63. The differences between the two types are that REAL takes more storage space and yields greater precision.

The predefined type REAL consists of a finite subset of real numbers represented by real constants in 8 bytes (48-bit mantissa, 7-bit signed base-16 exponent). The short type real, SHORTREAL, consists of a finite subset of real numbers represented by real constants in 4 bytes (24-bit mantissa, 7-bit signed base-16 exponent). The decimal range magnitudes representable are approximately 5.4E-79 to 7.2E+75, where E means the exponent that follows is a power of 10.

Values of type REAL can be designated by real literal constants. Their syntax is presented with the literal constants in Section 3.3.4.

A real literal constant designates a real number. Informally, it is an integer followed by a designation for a power of 10; or an integer followed by a decimal fraction, which may optionally be followed by a designation for a power of 10.

The syntax requires that a real number must include either a decimal point or a "power-of-10" to differentiate it from an integer. If there is a decimal point, it must be followed by at least one digit. The letter E (or e) represents the scale factor 10. Note that in Pascal real constants must have a digit preceding the decimal point and either a digit or the letter E (or e) following the decimal point.

In this implementation, all literal real constants are compiled to be of type REAL.

The ordering of REAL and SHORTREAL values is the usual mathematical ordering of real values. All comparisons are legal. The result of using a relational operator (<, =, >, <=, <>, >=), between two real valued operands, is a Boolean value.

The types REAL and SHORTREAL are assignment-compatible; operands of these types may be freely mixed in expressions and assignments. Either of them may be passed to value-parameters of either of their type-identifiers. However, only a SHORTREAL variable may be passed to a SHORTREAL variable-parameter, and only a REAL variable may be passed to a REAL variable-parameter. A literal real constant may be passed to either a REAL or SHORTREAL value parameter, but never to a variable-parameter.

Integers (type BYTE and SHORTINTEGER included) may be mixed with real numbers in expressions; the integers are automatically converted to type REAL or SHORTREAL; yielding real-valued expressions. However, real-valued expressions may not be assigned to integers, nor may real-valued expressions be passed to integer value-parameter parameters. REAL and SHORTREAL are not assignment-compatible to an INTEGER, BYTE, or SHORTINTEGER type. Conversion from real to integer is not provided automatically. REAL or SHORTREAL data may be converted to INTEGER data by the predefined functions TRUNC and ROUND.

The operations on REAL and SHORTREAL expressions are given in Table 5-3. The result of an arithmetic operation is either the REAL or SHORTREAL value depending on the operand types. If an operator has operands of both of these types, then the SHORTREAL is converted to REAL. These operators are infixes; and + and - can also be used as prefixes.

**TABLE 5-3 REAL AND SHORTREAL OPERATIONS**

| SYMBOL | OPERATION |
|---|---|
| + | Add |
| - | Subtract |
| * | Multiply |
| / | Divide |

The second operand of / must not have the value zero.

Six predefined functions accept one argument of REAL or SHORTREAL type:

1. ABS(x) The result is the absolute value of the REAL or SHORTREAL x.

   ABS(x) is defined by:

   ```
   ABS(X) = |X|
   ```

2. TRUNC(x) The result is the (truncated) integer value corresponding to the REAL or SHORTREAL x.

   TRUNC ( X ) is defined by these conditions:

   ```
   TRUNC(X) is an integer
   ABS( TRUNC(X) ) <=ABS(X),
   ABS( X - TRUNC(X) ) < 1.
   ```

3. ROUND(x) The result is the (rounded) integer value corresponding to the REAL or SHORTREAL x.

   ROUND ( X ) is defined by:

   ```
   If X >= 0 THEN ROUND(X) = TRUNC(X + 0.5)
             ELSE ROUND(X) = TRUNC(X - 0.5);
   ```

4. LENG(x) The result is the value of type REAL corresponding to SHORTREAL x.

   LENG(X) is defined by:

   ```
   x is of SHORTREAL type
   LENG(X)  is of REAL type
   ```

5. SHORTEN(x) The result is the (truncated) value of type SHORTREAL corresponding to REAL x.

   SHORTEN(x) is defined by:

   ```
   x is of REAL type
   SHORTEN(x) is X truncated of SHORTREAL type.
   ```

6. SQR(x) The result is the square of x of either type REAL or

SHORTREAL, depending on the type of x. That is, SQR(x) returns x*x.

Six more real-valued mathematical functions are available for both REAL and SHORTREAL data. They are arctangent, sine, cosine, exponential, square root and natural logarithm. Standard Pascal identifies these functions as ARCTAN, SIN, COS, EXP, SQRT, and LN. Perkin-Elmer Pascal implements these functions with slightly different names and requires explicit external FORTRAN function declarations. Full details are given in Chapter 3 in Sections 3.5.3 and 3.5.9.

Some examples of declaring REAL values as constants in the constant declarations part are:

```
CONST


     PI             = 3.14159;
     PI_PRECISE     = 3.141592653589793;
     NEG_PI         = -PI;

     ONE = 1.0;
     TWO = 2.0E+00;

     NEGONE  = -1.0E+00;
     NEGTWO  = -2.0;

     HUNDRED = +1.0E2;
     ONETHOU =  1.0E3;
     TWOTHOU =  2.0E3;

     SMALLPOSREAL = 5.4E-74;
     SMALLNEGREAL = -5.4E-74;
     LARGEPOSREAL = +7.2E75;
     LARGENEGREAL = -7.2E+75;
```

Real variables may be declared in the variable declaration part of a block:

```
VAR
     P,Q,R,S : SHORTREAL;
     DIMENSION, ANSWER1, ANSWER2 : REAL;
```

User-defined functions may be written whose function value is REAL or SHORTREAL; and real-valued parameters (value or variable) may be declared as REAL or SHORTREAL in parameter-lists. Note the use of reals in the expressions of the assignment statements within the functions. For example:

```
FUNCTION DIFFERENCE_OF_SUMS(A,B,C,D:  SHORTREAL):REAL;
BEGIN
  DIFFERENCE_OF_SUMS := (A+B) - (C+D);
END;

FUNCTION MULTIPLY_THE_QUOTIENTS(A,B,C,D:  SHORTREAL):REAL;
BEGIN
  MULTIPLY_THE_QUOTIENTS := (A/B) * (C/D);
END;

FUNCTION PERIMETER(SIDE1,SIDE2:  REAL):SHORTREAL;
BEGIN
  PERIMETER := (2*SIDE1) + (2*SIDE2);
END;
```

This program body also reflects the assignment of values to real variables, and references to the functions defined above; and outputting expressions of reals.

```
BEGIN
  P := 1;        {a real may be assigned an integer value}
  Q := TWO;      {TWO declared constant-identifier of 2.0E+00}
  R := 3.0;
  S := 4.0;
  ANSWER1 := DIFFERENCE_OF_SUMS(P,Q,R,S);
  ANSWER2 := MULTIPLY_THE_QUOTIENTS(P,Q,R,S);
  DIMENSION := PERIMETER(3.0,4.0);
  WRITELN(ANSWER1,ANSWER2);
  WRITELN(DIMENSION,PERIMETER(1.0,2.0));
  WRITELN(ANSWER1:10:7);
  WRITELN(ANSWER2:10:7);
  WRITELN(DIMENSION:4:1);
  WRITELN(PERIMETER(1.0,2.0):4:1);
END.
```

and the output would be:

```
|column 1 of first textfile field
|
V
-4.0000000000000022E+00 3.7500000000000000E-01
 1.4000000000000008E+01 6.0000000E+00
-4.0000000
 0.3750000
14.0
 6.0
```

Note the difference in the output format of these real values. In the first two WRITELN statements the real-valued expressions are output in floating-point representation (similar to scientific notation of real numbers). Also, the field-width defaults used in outputting these real numbers are different for REAL (24 positions) and for SHORTREAL (14 positions). As ANSWER1, ANSWER2, and DIMENSION are of type REAL, their output formats show more digits of precision. As PERIMETER yields a function-value of type SHORTREAL, fewer digits of precision are reflected in 14 positions: " 6.0000000E+00". Leading plus signs are represented by a space. In the subsequent WRITELN statements, the real-valued expressions are output in fixed-point representation as directed by user-specified field-widths and fractional-digits specifications in the write-parameters. See Chapter 8 for write-parameter details on outputting reals and shortreals.

### 5.3.8 Set Types

A set is a certain collection of member elements of the same data type, called the base-member-type. Data of any enumeration or ordinal type, which is thereby discrete, ordered, and finite may be the base-member-type, if its ordinal values lie in the range 0..127. A particular set may be designed to contain up to 128 values, or up to the number of values in the base-member-type, where each of its values can be a member element of the set, and each of the member elements must have an ordinal value within the range established for the base-member-type.

A set-type is a Pascal structured data type. Variables, which are declared to be of set-types, are known as set-variables. A set-variable may then have its value assigned in an assignment statement by assigning to it a set-expression value. These set-expressions must have expression values whose types are set-types. Set-expressions may consist of a set-variable, a set-constructor, or one or more of them operated on with those set-operators, which yield set-values.

When a set-expression is defined with a set-constructor, the set-constructor contains a list of all the members of the set expression value. The members are represented in the list with expressions that yield values of the base-member-type.

The members listed in a set-constructor must all belong to some discrete, ordinal data-type, whose ordinal values are in the range 0..127. This ordinal type is the base-member-type of the set-constructor.

Set variable assignment, set operations, or set comparisons can be performed with set-expressions and set-variables whose base-member-type is assignment-compatible.

Expressions containing set comparisons are known as relational set-expressions and yield Boolean valued results.

To create a set-type introduce a type-identifier in the TYPE definition part, such as:

```
TYPE set-type-identifier = set-type;
```

where identifier becomes the name of the set-type, and "set-type" is defined with the syntax:

### Set-Type

```
---> SET OF ----> base-member-type ---->
```

In the type-definition, the identifier on the left of the equal sign, is then available as a set-type identifier which may be used to declare the type of set-variables. The set-type on the right of the equal sign is of the form "SET OF base-member-type".

For example, given the availability of the following type-declarations as base-member-types:

```
TYPE

    UNITS          = 0..127;
    QUALIFICATIONS = (HS,BA,BS,MA,MS,MSCS,MBA,PHD,
                      YR1,YR5,YR10,YR15,YR20,YR25,YR30);
    DEGREES        = HS..PHD;
    YEARS          = YR1..YR30;
```

We may define set-type-identifiers, thusly:

```
TYPE

    CHARSETS = SET OF CHAR;    {CHAR may be base-member-type}
    UNITSETS = SET OF UNITS;
    QUALIFICATIONSETS = SET OF QUALIFICATIONS;
    DEGREESETS = SET OF DEGREES;
    YEARSETS   = SET OF YEARS;
```

Set-types of a discrete scalar type, or subranges thereof, can consist of all of the subsets that can be formed from the member values in that base-member-type. If the base member type has n values, then the set-type may have 2**n set values. However, the current implementation restricts the number of values allowable in the base-member-type. The base-member-type may be a discrete scalar ordinal or enumeration type with no more than 128 values. That is, the ordinal values of the base-member-type must be in the range 0..127, This allows the base-member-type to be the predefined type-identifier CHAR, or a user-defined enumeration type, or subranges thereof, or a subrange of type INTEGER 0..127 inclusively.

Once a set-type is defined, a variable in the VAR declarations part may have its type declared to be of that set-type, such as:

```
    VAR set-variable-identifier :  set-type-identifier;
```

or

```
    VAR set-variable-identifier :  set-type;
```

but this second form, depending on the use of set-variables may prohibit establishing Pascal type-compatibility "identity" for other separately declared variables, local variables, or variable-parameters requiring identity of type through type-identifiers.

This set-variable-identifier is the name of a set-variable whose members, when defined in an assignment statement, will be chosen from the base-member-type. The values of the members of any set are defined by specifying or listing its members in a set-constructor. Note that all members listed must be of the same type. The syntax of the set-constructor is:

Set-Constructor

```
          ---------------------------------->|
          ^                                  |
          |                                  V
---> [   -------> expression..expression -------> ] --->
            ^ |                           ^ |
            | |                           | |
            | V-------> expression ------>| |
            |                               |
            |<------------- , <-------------V
```

where each expression listed, or subrange of expressions, defines the members of the set-value represented by this set-constructor. The expressions must all have values belonging to some discrete, ordinal data-type, which has ordinal values in the range 0..127. When the set-constructor is assigning values to a set-variable, the members must also be of the same base-member-type as noted earlier. The empty or null set is represented by the brackets alone. The set-constructor [ ] denotes the empty set. The format of the set-constructor is a left bracket, followed by none, one, or more members, denoted by an expression or a subrange of expressions, either of which is separated from its preceding member by a comma, and ending with a right bracket. An error will occur if a set-constructor contains a member with an ordinal value outside the range 0..127, or it is assigned to a set-variable whose base-member-type does not include the member.

For example we may define the following set-variables:

```
    VAR
        VOWELS, DIGITS, HEXDIGITS: SET OF CHAR;
        NUMBERSET  : UNITSETS;
        SKILLS,JOB : QUALIFICATIONSETS;
        EDUCATION  : DEGREESETS;
        EXPERIENCE : YEARSETS;
```

and then use set-constructors, thusly, to assign set-values to the set-variables:

```
    BEGIN
        NUMBERSET := [12, 25..29, 13, 15, 1..5];
        DIGITS := ['0'..'9'];
        HEXDIGITS := ['0'..'9', 'A'..'F'];
        VOWELS := ['A', 'E', 'I', 'O', 'U'];
        JOB := [MBA, YR5];
        EDUCATION := [HS, BA, MBA];
        EXPERIENCE := [YR1, YR5];
    END;
```

Note that where the set members are consecutive values of a base-member-type, they may be advantageously represented as subranges of expressions as in the set-constructors defining values for DIGITS and HEX_DIGITS, whereas the nonconsecutive members required to represent the set of VOWELS are presented as individual literal constants of the base-member-type. The members or subranges may be in any order, as represented in the example NUMBERSET.

The base-member-type of NUMBERSET and its assigned set-constructor is the integer subrange 0..127. The base-member-type of DIGITS, HEXDIGITS, and VOWELS and their assigned set-contructors is the type CHAR. The base-member-type of JOB is the user-defined enumeration type QUALIFICATIONS.

Set operations may then be performed between set-variables, or between set-constructors whose base-member-types are the same. There are three binary operators that specifically operate on sets to produce new sets. They are "+" for set union, "-" for set difference, and "*" for set intersection. The operators must be used as infixes and be between set operands whose base member types are the same. The result is another set of the same base-member-type as the operands. These operations are defined in Table 5-4 below.

There are four relational operators which apply to the set data type. The comparison operator "=" for set equality, "<>" for set inequality, "<=" for set inclusion, and ">=" for set containment, can be performed on sets if their base-member-types are compatible. They produce a Boolean result. For example, the operator "<=" when applied to sets, as in the example S1<=S2, has the Boolean value TRUE if every member of S1 is also a member of S2, otherwise the result is false. The Boolean expression S1>=S2 is TRUE if and only if every member of set S2 is also a member of set S1.

For example, following the above assignment statements, we may also write:

```
BEGIN
  NUMBERSET := [1..5, 6..9] - [2, 4, 6, 8];
  VOWELS := VOWELS + ['Y'];
  SKILLS := EDUCATION + EXPERIENCE;
  IF JOB <= SKILLS {compares two set-variables}
            THEN WRITELN ('MEETS REQUIREMENTS')
            ELSE WRITELN ('SKILLS LACKING');
END;
```

In the above examples, only constants or constant-identifiers are used to denote set members within the set-constructors. As the set-constructor syntax allows an expression to denote members of the set, any expression which yields a value of the base-member-type may also be used in a set-constructor, such as the use of the variable CH, and function-references depicts below.

```
BEGIN
  CH := 'Y';
  VOWELS := ['A','E','I','O','U',CH];
  DIGITS := ['0',SUCC('0')..PRED('9'),'9'];
END
```

Another useful decision-making set operator is the test for set membership. The set membership operation, designated by the keyword IN, can be a quick way to perform a series of disjoint tests that would otherwise be cumbersome. It is also an infix operation between the element being tested and the set being tested. To test set membership, the first operand must be a member of the base-member-type and the second operand must be a set-variable or a set-constructor of the same base-member-type. The result is a Boolean value. If the first operand is a member of the second operand set value, the test set membership expression's value is true, otherwise it is false. For example, given the additional declarations:

```
TYPE ALPHABET = CHARSETS;
VAR   ALPHAS:ALPHABET;   CH : CHAR;
BEGIN
  ALPHAS:=['A'..'Z', 'a'..'b'];    {set of letters}
  READLN (CH);              (* Reads a character into CH *)
  IF CH IN ALPHAS             (* Tests CH for set membership *)
  THEN WRITELN (CH)
  ELSE IF CH IN DIGITS THEN BEGIN statement-sequence END;
END
```

## TABLE 5-4 SET OPERATIONS

| PASCAL SET OPERATOR | CONVEN-TIONAL SYMBOL | OPERATION NAME | OPERATION FUNCTION |
|---|---|---|---|
| [...] | {...} | Define Set | Set-value member definition |
| + | $\cup$ | Union | An element is contained in a union of two sets, denoted A+B, if and only if it is an element of set A or set B or both. |
| - | - | Difference | An element is contained in the difference of two sets, denoted A-B, if and only if it is an element of set A but not an element of set B. |
| * | $\cap$ | Intersection | An element is contained in the intersection of two sets, denoted A*B, if and only if it is an element of both set A and set B. |
| = | = | Equality | If A and B are sets, the relation A=B is TRUE if and only if every member of each set is a member of the other set. |
| <> | $\neq$ | Inequality | The relation A<>B is TRUE if and only if A=B is FALSE. |
| <= | $\subseteq$ | Contained in | The relation A<=B is TRUE if and only if every member of set A is also a member of set B. In effect, this relationship says that the set A is included in or contained by the set B. |
| >= | $\supseteq$ | Contains | The relation A>=B is TRUE if and only if every member of set B is also a member of A. |
| IN | $\in$ | Membership | If A is an element of type T, and B is a set over the base member type T, the relation A IN B is TRUE if and only if the element A is contained in the set B. |

### 5.3.9 Array Types

One of the structured data types built up from simpler data types is the array-type. A structured data type differs from a simple scalar variable, in that the structured variable, such as an array, has more than one component. An array is a collection of components of identical data type. A significant feature of a structured type is the way in which its components are accessed. The component (or element) of an array may be directly accessed. An entire array is referenced by its variable name, as introduced in the VAR declarations. Any element in that array can be referenced by means of the array variable name and expressions of the array indices. For example, the indices into an array representing a matrix may have one index to indicate row, and another index to indicate column. Entire arrays can also be referenced for assignment to, and tests for equality or inequality with, other arrays of compatible types.

To establish an array-type-identifier, first define the array-type in the TYPE definitions part:

```
    TYPE array-type-identifier = array-type;
```

where the syntax of the construct array-type is:

**Array-Type**

```
  -PACKED>
  ^      |
  |      v
------------>ARRAY-->[-->index-type-->]-->OF-->component-type--->
                       ^            |
                       |            |
                       |<---- , <---V
```

The syntax of the construct index-type is any discrete scalar type (not reals), and the syntax of the construct component-type is any scalar or structured type (other than a FILE), even another array-type or record-type.

In the type definition, the identifier on the left side of the equal sign is the name of the array-type being introduced. The format of the array-type on the right side of the equal sign is then the optional keyword, PACKED, followed by the keyword, ARRAY, followed by one or more index-types, separated by commas and enclosed in square brackets, followed by the keyword, OF; and ending with the component-type.

The optional keyword, PACKED, causes the compiler to pack the internal representations of the elements so that no extraneous filler space occupies memory. However, caution must be used in applying this feature to array-types of components requiring alignment, or of records or arrays dues to their alignment requirements for comparisons. Refer to Chapter 10 on storage mapping for PACKED arrays.

The meaning of an array-type is simplest when there is only one index type. Then, a variable of the array-type consists of a collection of variables of the component-type, one corresponding to each possible value of the index type. For example, a variable of type:

```
    ARRAY [3..12] OF REAL
```

contains 10 variables of type REAL.

To define the meaning of the more general array type, we can say that:

```
    ARRAY [index_type1,index_type2] OF component_type
```

is equivalent to:

```
    ARRAY [index_type1] OF ARRAY [index_type2] OF component-type
```

For example:

```
    ARRAY [1..4] OF ARRAY [CHAR] OF SHORTINTEGER
```

is equivalent to:

```
    ARRAY [1..4,CHAR] OF SHORTINTEGER
```

A variable of this type contains 4 * 128 variables of type SHORTINTEGER; it is organized as four variables, each of which consists of 128 SHORTINTEGERs. There is no limit on the depth of nesting or declaration of inner arrays, i.e., the number of index type specifications is not bound.

The number of index-types in the definition of the array type is called the dimension of the array; i.e., if three index types are defined in the array-type, then the array of this type is a three-dimensional array. The index-types may be any enumeration type, such as a user-defined enumeration type or subranges. They may not be the keywords INTEGER or SHORTINTEGER, but may be a subrange of them. For example, if there were a user-defined enumeration type such as:

```
TYPE SECTION = (LEFT,CENTER,RIGHT);
```

an example of an array-type definition for a collection of elements might be:

```
TYPE COLLECTION = ARRAY[SECTION,1..50,-15..15] OF REAL;
```

The array-type identifier, COLLECTION, is then available as an array-type of 4,650 elements of type REAL. It has three index-types: SECTICN, the subrange 1..50, and the subrange -15..15.

Once an array-type is defined and identified by name, an array name variable may be declared to be of that array-type-identifier. The array-type-identifier is then available to be used as a type for deciding array variables in either the VAR declarations part, or the VAR variable parameter declaractions of a parameter-list; or as the "component-type" in an array of arrays or in a file of arrays; or as a type for a field-identifier in a record of arrays. An array variable (file component, or record field) may be declared to be of an array-type directly but this second method of "typing" runs the risk of violating Pascal rules of type-compatibility (see Section 6.2) for the variables, when used with other separately declared variables, local variables, or variable parameters. In the VAR variable declarations part an array variable declaration has the format:

```
VAR array-variable-identifier :  array-type-identifier;
```

or

```
VAR array-variable-identifier :  array-type;
```

where the identifier on the left of the colon is a new variable identifier, the array name; and the array-type on the right side of the colon is either a previously defined type-identifier of an array-type, or an array-type itself.

Either

```
VAR REALDATA,REALDATA2 :  COLLECTION;
```

or

```
VAR REALDATA3 :  ARRAY[SECTION, 1..50,-15..15] OF REAL;
```

which establishes a 4,650 element array, REALDATA, of the array-type, COLLECTION; and an identically compatible array, REALDATA2. REALDATA3 is not considered a compatible array (see Section 6.2 on type-compatibility).

Then in the main body of the program, the entire array may be referenced by using the array name. For example, the basic operation of assignment may be applied to whole arrays or their individual elements. For example, assignment of a whole array, REALDATA, to another compatible array, REALDATA2, follows the initialization of REALDATA in the example below. To initialize the entire array, each element must be initialized individually, for example:

```
VAR I : SECTION;          {declare variables for the indices}
    J, K: INTEGER;
BEGIN
 FOR I := LEFT TO RIGHT DO
     FOR J := 1 TO 50 DO
         FOR K := -15 TO 15 DO
             REALDATA [I,J,K] := 0.0;
     .                           {Each element is initialized}
     .
     .
  REALDATA2 := REALDATA;        {whole array is assignable}
                  {if both arrays have "identity" of type}
END
```

To reference an individual element in the array, use the construct, array-component, whose syntax is:

### Array-Component (Selector)

```
--->variable-selector---> [ --->index-expression---> ] --->
                            ^                      |
                            |                      |
                            |<------- , <-------v
```

where the variable-selector is the array name, and the index-expression is a specific value of the associated index-type declared in the definition of the array-type. The array-component is accessed by a selector, as it selects an element of the array for some purpose. The format of the array-component selector is the array name identifier, followed by one or more index-expressions, separated by commas, and enclosed in square brackets. Refer to Section 5.4 on variable-selectors to access array-components of an array which is part of another structure, such as a record.

A component of an n-dimensional array is selected by means of its array name identifier followed by n index-expressions (enclosed in brackets and separated by commas). The number of index-expressions in the selection must equal the number of index-types in the array-type definition, and the index-expressions used must have values of the corresponding index-types in the array-type definition.

For example, let the array variable, REALDATA, hold the ticket prices by seat for a theatre. Suppose this theatre has two aisles creating the three sections: left, center, and right; 50 rows per section; and 31 seats per row. The prices may be represented by the REAL data elements of the array, i.e., the component-type of the array is type REAL. The subclassifications of seat positions can be represented by the user-defined enumeration type: SECTION=(LEFT,CENTER,RIGHT). Then the subrange, 1..50, represents the rows, and the subrange, -15..15, represents the seats within a row. To access the elements (price for one seat), could then be:

```
REALDATA [LEFT,1,-15]  accesses the first seat price;
REALDATA [RIGHT,50,15] accesses the last seat price;
REALDATA [CENTER,24,0] accesses front row center;
```

Indices, also called subscripts, may be represented by variables. It is often convenient to use a variable as a subscript when selecting an array element, particularly within repetitive statements performing a general operation on all, successive, or several elements in the array. For example, we declare the variables AREA, ROW, and SEAT, to be variables of types compatible with the example's index-types:

```
VAR AREA: SECTION;
    ROW: 1..50;
    SEAT: -15..15;
```

Then each of the elements of the example array variable, REALDATA, may be individually, by a generalized reference, addressed by an array-component selector, after defining indices:

```
AREA := CENTER;
ROW  := 24;
SEAT := 0;
REALDATA [AREA,ROW,SEAT] := 36.50;
```

Arrays (non-strings) as whole units have only the basic operations of assignment (:=), and the comparison operators of equality (=) and inequality (<>) defined. The arrays must be of identically compatible types. Individual array elements may be operated upon just as any variable of the component-type.

### 5.3.9.1 String Array Types

One specific application of one-dimensional arrays is to represent strings. A string is a series of characters. A one-dimensional array of component-type CHAR with m character elements is called a string-type array (or simply string) of length m. Its values may be defined by a literal string-constant of length m, whose definition has been seen to be:

Character-String-Literal-Constant

```
-----> ' --------------> character -------------> ' ----->
          ^  |                            ^  |
          |  |                            |  |
          |  |                            |  |
          |  |-> (: -->  digits  --> :) ->|  |
          |                                  |
          |                                  |
          |<---------------------------------V
```

and is considered to be of type:

```
    ARRAY [1..m] OF CHAR
```

which is assignment-compatible to another string variable, if and only if they are of the same fixed length, not necessarily having the same index-types. That is, a literal string or named string constant of length m, is assignment-compatible to a variable of the type:

```
    ARRAY [i..j] OF CHAR
```

where ORD(j)-ORD(i)+1 = m.

Literal strings or named constant strings of the same fixed length may be assigned to, or compared in relational expressions to, any string variable of the same fixed length as the literal or constant string. This is is a Perkin-Elmer Pascal extension of Pascal assignment-compatibility.

However, string variables must be of identically compatible type to be assignment-compatible to each other, not just of the same length.

For comparisons, the ordering of characters defines the ordering of strings. Note that "digits" denotes an integer constant in the range 0..127 and denotes that the character with that specified ordinal value replaces the (:digits:) sequence in the string constant. All of the relational operators, in addition to the equality or inequality operators applicable for whole arrays, are defined for comparing strings of equal length. The relational operators for comparing strings are:

```
=   Equality
<>  Inequality
<   Less than
>   Greater than
<=  Less than or equal
>=  Greater than or equal
```

The result of the relational comparison is a Boolean value.

For example, we may declare string constants, such as:

```
CONST
    MESSAGE = 'HAVE A HAPPY DAY';
```

we may also declare string array-types, such as:

```
TYPE
    STRING16 = ARRAY[1..16] OF CHAR;     {string length = 16}
    SIXTEEN2 = ARRAY[21..36] OF CHAR;    {string length = 16}
    STRING80 = ARRAY[1..80] OF CHAR;     {string length = 80}
```

and we may declare string-type variables to be of those types, and another string-type variable MISFIT, to demonstrate incompatibility, such as:

```
VAR
    LIVE,WIRE        : STRING16;
    LINE             : STRING16;
    ECHOSTRING       : SIXTEEN2;
    CARD : STRING80;
    MISFIT, MISFITS_PARTNER : ARRAY[1..16] OF CHAR;
```

Then we may initialize the string variables, in program text, with either literal string constants, a declared constant string constant-identifier such as MESSAGE, or another string variable of identically compatible type.

The length of a string array is fixed and defined by its type. In Pascal, although variable-length strings cannot be read into a fixed-length string array except by a character at a time, they can be written out as strings, in their entirety.

For example, the following assignments can be made:

```
    LIVE := 'HAVE A HAPPY DAY';     {literal string assignment}
    WIRE := MESSAGE;      {assigns defined constant to variable}
    LINE := WIRE;         {assignment of string array variables}
    ECHOSTRING := 'HAVE A HAPPY DAY'; {assign to 16-char array}
    MISFIT := 'HAVE A HAPPY DAY'; {assign to any 16-char array}
    MISFIT := MESSAGE;    {assigns defined constant to variable}
    MISFITS_PARTNER := MISFIT; {identically typed variables}


{ Then the following WRITELN statements are legal and all have }
{ the same effect on the textfile OUTPUT: }


  IF (LIVE=WIRE) AND NOT (MESSAGE<>'HAVE A HAPPY DAY')
    THEN
      BEGIN
        WRITELN(LIVE);
        WRITELN(MESSAGE);
        WRITELN(LINE);
        WRITELN('HAVE A HAPPY DAY');
        WRITELN(ECHOSTRING);
        WRITELN(MISFIT);
      END;
```

and the output would be:

```
|column 1 of textfile field
|
V
HAVE A HAPPY DAY
HAVE A HAPPY DAY
HAVE A HAPPY DAY
HAVE A HAPPY DAY
HAVE A HAPPY DAY
HAVE A HAPPY DAY
```

However, as string variables only of identical type are assignment compatible to each other, it is illegal and will generate a diagnostic error if the following attempts are made to assign incompatibly typed strings or otherwise assign strings mismatched in length.

```
MISFIT := LIVE;           {illegal due to incompatibility}
ECHOSTRING := LIVE;       {illegal due to incompatibility}
MISFIT := WIRE;           {illegal due to incompatibility}
LIVE   := MISFIT;         {illegal due to incompatibility}
WIRE   := MISFIT;         {illegal due to incompatibility}
CARD   := LIVE;           {illegal & mismatched lengths}
CARD   := WIRE;           {illegal & mismatched lengths}
CARD   := MESSAGE;        {mismatched lengths}
CARD   := 'HAVE A HAPPY DAY';  {mismatched lengths}
CARD   := MISFIT;         {illegal & mismatched lengths}
```

Also to initialize string variables, characters may be read in one at a time to each array element, from a textfile, a file of type TEXT, such as the predefined file INPUT. However, character data read from a textfile with READ does not automatically skip over blanks or EOLN characters as it does for reading integers and reals, so these conditions must be programmed for as desired, with READLN or querying EOLN. Strings may also be read from a non-textfile FILE OF "string-type" or single-characters from a FILE OF CHAR. Some examples of character I/O are given in Section 5.3.2 on the CHAR type. Also see Chapter 8 on Pascal I/O and the file-type definitions.

We can read fixed-length strings into string-arrays from non-textfiles, where the file's component type are string arrays. For example, consider the following example program which simply copies card image to another file.

```
PROGRAM COPYCARD(CARDFILE,TAPEFILE);

TYPE
    STRING80 = ARRAY[1..80] OF CHAR;
VAR
    CARD : STRING80;
    CARDFILE, TAPEFILE : FILE OF STRING80;

BEGIN
  RESET(CARDFILE);
  REWRITE(TAPEFILE);
  WHILE NOT EOF(CARDFILE) DO
    BEGIN
      READ(CARDFILE,CARD);
      {card data may be processed here}
      WRITE(TAPEFILE,CARD);
    END;
END.
```

See Chapter 8 on file-type, file-variable declarations, and I/O routine definitions. See Chapter 9 on the file-name-list in the program header and also some restrictions on Pascal I/O in MODULEs; e.g. implicit READ/READLNs from INPUT and implicit WRITE/WRITELNs to OUTPUT must be replaced with explicit calls on file-variables passed as VAR variable parameters.

In this implementation of Pascal, the rule requiring assignment-compatible arguments passed to value parameters is relaxed to permit an argument string of any length to to be passed to a value parameter of any string-type.

If the argument string is shorter than the value parameter string, then the value parameter contains undefined values beyond the length of the passed argument string. If the argument string is longer than the value parameter string, then the extra characters attempted to be passed, simply do not get passed.

Although argument strings of any length can be passed to string parameters, the "assignment-compatibility" rules across the assignment-operator are still in effect within the routine receiving strings. That is, the programmer must design a method to find the end of a variable length string so passed to a routine. One method is to design the routine to define a parameter to which an argument length can be passed. Another method is to arrange all strings to have an imbedded end-of-string character, that the routine can search for and query.

The Pascal rule requiring identity of type to pass an argument variable to a variable parameter, even regarding string variables, is not relaxed. Only a string-variable of identical type to the variable parameter's type-identifier may be passed to that variable parameter.

### 5.3.10 Record Types

Another kind of structured data type is the record-type. The record is a structure used for a collection of data with fairly complex relationships, where the individual elements do not have to be all of the same data type. Whereas the array is also a structured data type, its elements must be of the identical data type, even though arrays of records (where the records may have mixed field types) are allowable. The record's components may be any of a variety of data types, including arrays and other records, and pointer-types. The components of a record are accessed record-variable-selectors using field names, not by the subscripts (index-expressions) as used to access array components. We do not index into a record.

Introducing a record-type, or establishing a record type-identifier in the TYPE definitions part, is a process of outlining all of the possibilities of a record's data structure. By first giving the record type-definition a name in the TYPE definition declarations, that type-identifier is available to attribute that record-type to variable data, making them identically compatible in type, in Pascal. Declaring a record variable in the VAR variable declarations group, creates the data storage for one record variable of its type and introduces an identifier as the name of that specific record variable. The record variable has no defined values at the point of its declaration as a variable. The record variable is then available to have values read into it or assigned to its components. An entire record variable may be assigned the value(s) of another defined record variable if their record-types are identical.

A record has a fixed or variable number of components or fields for data. The record-type structure allows a record to have either or both a fixed part and variant-part. If it has both, the fixed part must come before the variant-part. In addition to the data fields of the fixed part, record variants in the variant-part allow us to set up a record-type whose precise structure will vary for different record variables declared to be of that record-type. Given the record variable name, each field in any record may be uniquely identified and thereby easily referenced. The name of each field, called the field-identifier, must be unique within the record-type definition, but only within that record-type definition. It may identify another field in another record-type. This is because referencing a field usually requires the record variable name and the field-identifier, unless the With-statement is used to open up the scope of the record-type-definition. Therefore, abstractions of data with fairly complex relationships without homogenous types can be readily structured, accessed, and processed by use of the record-type.

To establish a record type-identifier, the record-type may be first defined in a TYPE definitions part.

```
    TYPE record-type-identifier = record-type;
```

where the syntax of "record-type", is:

**Record-Type**

```
  --->PACKED--->|              --------------->|
  ^             |              ^               |
  |             V              |               V
--------------------> RECORD ---> field-list ---> END --->
```

In this type definition, the identifier on the left of the equal sign becomes the type-identifier of the record-type. The format of the record-type on the right side of the equal sign is then the optional keyword, PACKED; followed by the keyword, RECORD; followed by the construct, field-list; followed by the keyword, END.

The optional keyword, PACKED, causes the compiler to eliminate extraneous memory space normally required for alignment of certain types of data. This space saving is done at the expense of component access time. A component of a PACKED record variable cannot be passed to a variable parameter of a procedure or function, whereas it can be passed to a value parameter. The same cautions for PACKED array-types apply to PACKED record-types.

For general information on the use of this PACKED feature, refer to Section 5.3.9 on Array Types. For detailed information on the packing process refer to Section 10.6.1 on internal data storage requirements for arrays and records.

The field-list is optional. When not present, the record-type is empty. The syntax of the construct, field-list, is:

Field-List

```
 ---------------------->|                    ---> ; --->|
 ^                      |                    ^          |
 |                      V                    |          V
---> fixed-part ---> ; ---> variant-part ------------------->
                 |                        ^
                 |                        |
                 V----------------------->|
```

The field-list contains either a fixed-part, or a variant-part, or both. The field-list lists the names of the record fields known as field-identifiers.

The fixed-part of the field-list lists the fields of the record that are always available, i.e., fields that are always present, in every record-variable of the record-type with a fixed-part defined. The syntax of the fixed-part is:

Fixed-Part

```
--------->identifier---------> : ---->type---------->
    ^  ^              |                       |
    |  |              |                       |
    |  |<----- , <----v                       |
    |                                         |
    |                                         |
    |                                         v
    |<----------------------- ; <--------------
```

Each of these identifiers are user specified names of the fields that occur in every record declared to be of this record-type; i.e., they are fixed fields in the structure of records of this record-type. The identifier(s) on the left of the colon is(are) called the field-identifier(s), and the type on the right side of the colon is the data type of these field-identifier(s). When more than one field of the same type are to occupy consecutive positions in the record they may be defined in a group. This is coded by specifying each of their names, separated by commas, where the entire list of fields is followed by a colon and their mutual type.

The type of the field may be any predefined Pascal type-identifier, previously declared type-identifier, or a type which is ordinal, real, structured type, or pointer-type. However, in this implementation of Pascal, it cannot be the introduction of a user-defined enumeration type. To type a field as a user-defined enumeration, the user-defined enumeration type, must have been first established with a type-identifier, visible in scope, to the record-type definition. Then a field may be given its type by the user-defined enumeration type-identifier.

The declaration of each field with its type, or list of field identifiers (separated by commas) with their mutual type, is separated from the next with a semicolon.

Within a record-type definition a variant-part can be delineated by naming a tag-field and/or declaring a tag-field data type in a CASE clause of the record-type definition. If the tag-field is named with a field name, space for it as a variable will be reserved for it in each variant. If the tag-field is not named, but only a tag-field type is used, no space is allocated for a tag-field in each variant. For each specific value possible for the tag-field type, a different variant structure of subcomponent fields may be outlined.

In this implementation of Pascal, the maximum number of variants which may be specified in a record type-definition, at any one level of nesting of variants, is limited to 32. This is because the tag-field type is limited to be a type whose ordinal values lie in the range 0..31.

Each variant within the variant-part of a field-list has associated with it another field-list, listing each field in the variant and declaring each field's data-type. The syntax of the variant-part of a field-list is:

<u>Variant-Part</u>

```
            tag-field               type
--->CASE--->identifier--> : --->identifier-->OF--->variant--->
          |                   ^                  ^          |
          |                   |                  |          |
          V------------------>|                  |<--- ; <--V
```

The identifier after the keyword CASE is called the tag-field, if two identifiers, separated by a colon, precede the keyword OF. The tag-field is a special field-identifier that names the general circumstance for which several sets of fields, called the variants, will be available. The tag-field-identifier, when present, is followed by a colon. The tag-field identifier and its subsequent colon is optional but not its type-identifier.

The identifier following the colon is a type-identifier, i.e., the type of data that the tag-field represents. The tag-field's type must be an ordinal type, with a finite limitation of 32 values, such as Boolean, user-defined enumeration types, the integer subrange 0..31, or subranges thereof whose ordinal values lie in the range 0..31. Following the tag-field's type, is the keyword, OF, followed by a listing of one or more variants. The variants are specified sets of fields, represented in a variant's field-list, of the possible variant circumstances of the tag-field's type. That is, the value of the tag-field determines the further contents of the record of this record-type. This means that although all of the possible variants are outlined in the record-type definition, only one variant's information is actually occupying the structured data area of the variant-part of an actual record variable. Which variant's structure is in the record is determined by the value of the tag-field. Note that each variant's fieldlist may contain either or both fixed and variant-parts.

The syntax of each variant is:

<u>Variant</u>

```
---> constant ---> : ---> ( ---> field-list ---> ) --->
 ^             |             |               ^
 |             |             |               |
 |<---  ,  <---V             V-------------->|
```

After the keyword OF, one or more variants are listed, each separated from the other by a semicolon. Each variant is of the format of one or more constants, each separated from the other by a comma, followed by a colon, followed by a left parenthesis, followed by a field-list, and terminating with a right parenthesis. The constants are to be values of the tag-field type-identifier, which is limited to have values whose ordinal values are in the range 0..31. A variant may associate a constant with an empty field-list.

Variants may be nested up to a limit of 16. That is, within the field-list of a variant, that field-list may contain a fixed-part, and/or a variant-part, or both. The depth of this nesting is 16 levels.

Once a record-type is defined and identified by name, that record-type-identifier is available to be used as a type for declaring record variables in either the VAR variable declarations part or in the VAR variable parameter declarations of a parameter-list, as a "component-type" of an array of records or a file of records, or as a type for field-identifiers within a record-type, itself. If the record-type-identifier has been previously bound to a pointer-type-identifier (see the following Section 5.3.11 on pointer-types) and pointer-variables are established, dynamic record variables may be created.

An array or file component-type, or field type may be declared to be of a record-type directly (without a record-type-identifier) but this second method of typing runs the risk of violating Pascal rules for type-compatibility (see Section 6.2) for the variables, when used with other separately declared variables, local variables, or variable parameters, requiring identity of type to be established through type-identifiers. In the VAR variable declarations part, a record variable declaration has the format:

```
    VAR record-variable-identifier : record-type-identifier;
```

or

```
    VAR record-variable-identifier : record-type;
```

where "record-variable-identifier" becomes the name of a record-variable. For example, to establish a record-type-identifier, RECTYPE1:

```
    TYPE

     TAGTYPE = 0..31;

     RECTYPE1=RECORD
                FIXEDFIELDNAME1:INTEGER;
                FIXEDFIELDNAME2:SHORTINTEGER;
                FIXEDFIELDNAME3:SHORTREAL;
                FIXEDFIELDNAME4:REAL;
                FIXEDFIELDNAME5, FIXEDFIELDNAME6:BOOLEAN;
                FIXEDFIELDNAME7:REAL;
                CASE TAGFIELDNAME:TAGTYPE OF
                  1:(VARIANT1FIELDNAME1:REAL;
                     VARIANT1FIELDNAME2,VARIANT1FIELDNAME3:CHAR;
                     VARIANT1FIELDNAME4:SHORTREAL);
                  2:(VARIANT2FIELDNAME1, VARIANT2FIELDNAME2:BYTE;
                     VARIANT2FIELDNAME3:SHORTINTEGER;
                     VARIANT2FIELDNAME4, VARIANT2FIELDNAME5:REAL;
                     VARIANT2FIELDNAME6:INTEGER);
                  3, 4:(VARIANT34FIELDNAME1:CHAR;
                        CASE NESTEDVARIANTTAG:TAGTYPE OF
                          1,2 : (FIXFIELD1:REAL);
                          3,4 : (FIXFIELDB:BYTE;
                                 FIXFIELDC:CHAR;
                                 FIXFIELDR:RECORD
                                    CASE TAG:TAGTYPE OF
                                    1,2,3:(INNEREC1FIELD:CHAR;
                                           INNEREC2FIELD:BYTE);
                                     4,5: (INNERECAFIELD:BYTE;
                                           INNERECBFIELD:CHAR);
                                           END ));
              END; {end of entire RECTYPE1 record-type}
```

we may declare variables to be of that record-type or to contain components of that record-type.

```
VAR

       {The following variable declaration }
       {declares record variable RECNAME of type RECTYPE1:}

   RECNAME : RECTYPE1;

   {This declares f as file-variable of RECTYPE1 records:}

        f : FILE OF RECTYPE1;

   {Declares array (of 100 records) component-type RECTYPE1}

   ANAME   : ARRAY[1..100] OF RECTYPE1;
```

A record variable of a particular record-type contains fields corresponding to the field-identifiers declared in the fixed-part, if present.

If there is a variant-part, the value of the tag field at execution time determines the further contents of the record. If that value of the tag field has not been associated as a constant to a fieldlist, there are no contents to access. When the value of the tag field is a constant whose value has been associated to a fieldlist, the contents of that fieldlist are accessible in the record. This means when the value of the tagfield (during execution) matches a variant constant, only the fields listed for that constant (in the record-type definition) are present, referenceable, or accessible in the record-variable.

An entire record variable can be referred to or accessed by a record variable selector (its separately declared record-variable identifier name when not part of another structure).

When a record variable is part of another structure, e.g., an array of records, it is accessed by an array-component-selector; when a record variable is a field within a record itself, it is accessed by a record-field-selector.

Any field of a RECORD can be referred to by a record-field-selector. The field of a RECORD can be referred to, or accessed, by its field-identifier alone, when the scope of the record-type-definition has been opened up by a With-statement. See Section 5.4 for details on selectors. For example, in simplest form:

```
            {Reads component from file f into entire RECNAME}
    READ(f,RECNAME);

            {accesses a fixed-field in a record variable}
    RECNAME.FIXEDFIELDNAME1 :=expression;

            {assigns a value to a record's tagfield}
    RECNAME.TAGFIELDNAME := 1; {ORD(1) in ORD(0)..ORD(31)}

            {accesses a record variant's fixed-field}
    RECNAME.VARIANT1FIELDNAME1 := expression;

            {chooses a variant having nested variants}
    RECNAME.TAGFIELDNAME := 4;    {assigns tagfield value 4}

           {accesses a fixed-field inside a chosen variant}
    RECNAME.VARIANT34FIELDNAME1 := expression;

            {assigns value to tagfield in a nested variant}
    RECNAME.NESTEDVARIANTTAG := 3;

            {accesses a fixed-field inside a nested variant}
    RECNAME.FIXFIELDB := 255;

      {accesses and assigns tagfield in nested variant's record}
    RECNAME.FIXFIELDR.TAG := 2;

    {accesses field in a nested record inside a nested variant}
    RECNAME.FIXFIELDR.INNEREC1FIELD := 'A';
```

The syntax for selecting a field within a record is:

### Record-Field Selector

```
 ^---------------------------->|
 |                             |
 |                             v
--->variable-selector---> . ----->field-identifier--->
                        ^                          |
                        |                          |
                        <---------------------------v
```

where "variable-selector" is naming a datum of a record type: either a declared record-variable, a targeted record variable, or a record field or an array-element which is a record; and "field-identifier" is the name of the field. An abbreviated form of such field references is provided by the WITH statement, which allows the optional path in the above graph skipping the variable-selector and period to be taken (see Section 7.3.7.).

It would be tedious to repeatedly have to specify the record variable selector when accessing each of many fields in a record.

When the WITH statement initially specifies a record variable-selector, within the scope of that WITH statement, a field may be accessed without repeating the record variable selector. (See the example below).

An assignment of one record to another record variable of the "identical" type, is legal when the type has no variant-part. When the type has a variant part, then the assignment is legal only if the tag-fields of the two record variables on the two sides of the assignment operator have the same values, so that their variant-parts will have the same structures. Records of the same identical type can be compared for equality or inequality. Fxamples of RECORD type declarations are the type-identifier DATF, PERSON, and COORDINATE below:

```
TYPE   COORDS = (CARTESIAN, POLAR);
       GENDFR = (MALE, FEMALE);
       MONTH  = (JAN, FEB, MAR, APR, MAY, JUN,
                 JUL, AUG, SEP, OCT, NOV, DEC);
       ALPHA = ARRAY [1 .. 16] OF CHAR;
       STATUS = (MAPRIED, WIDOWED, DIVORCED, SINGLE);

       DATE = RECORD
                MO: MONTH;
                DAY: 1 .. 31;
                YEAR: SHORTINTEGER;
              END;

       PERSON = RECORD
                  NAME: RECORD
                          FIRST, LAST: ALPHA;
                        END;
                  SS: INTEGER (*SOCIAL SECURITY NO. *);
                  SEX: GENDER;
                  BIRTH: DATE;
                  CASE MS: STATUS OF
                    MARRIED,
                    WIDOWED: (MDATE: DATE);
                    DIVORCED: (DDATE: DATE;
                               FIRST: BOOLEAN);
                    SINGLE: (INDEPENDENT: BOOLEAN)
                END (*PERSON*) ;

       COORDINATE = RECORD
                      CASE KIND: COORDS OF
                        CARTESIAN: (X, Y: REAL);
                        POLAR: (R: REAL; A: REAL);
                    END;
```

Note that in the above record-type definitions, that DATE is a record-type with only a fixed-part; PERSON is a record-type with both a fixed-part and a variant-part; and COORDINATE is a record-type with only a variant-part.

Then in the variable declarations part of a block, we may declare record variables to be of the above record-types:

```
VAR
    LOCATION : COORDINATE;    EMPLOYEE_RECORD : PERSON;
```

Then we may initialize the record variable LOCATION as follows:

```
BEGIN
  LOCATION.KIND := CARTESIAN;  {assigns constant to tagfield}
  LOCATION.X := 5.78634;
  LOCATION.Y := 6.32418;
END;
```

or

```
BEGIN
  LOCATION.KIND := POLAR;      {assigns constant to tagfield}
  LOCATION.R := 4.876302;
  LOCATION.A := 5.384756;
END;
```

or using the WITH statement, the record EMPLOYEE_RECORD, may be initialized:

```
BEGIN
  WITH EMPLOYEE_RECORD DO
    BEGIN
      NAME.FIRST := 'JOHN           ';
      NAME.LAST  := 'DOE            ';
      SS := 2123456738;
      SEX := MALE;

      WITH BIRTH DO
        BEGIN
          MO := SEP;
          DAY := 28;
          YEAR := 1960;
        END;

      MS := MARRIED;    {assigns constant value to tagfield}

      WITH MDATE DO
        BEGIN
          MO := MAY;
          DAY := 23;
          YEAR := 1982;
        END;
    END;
END;
```

A CASE statement is often used in parallel with the CASE clause used to outline a record-type with a variant-part. For example, assume the following procedure is being passed different record variables, of the "identical" record-type PERSON, to the variable-parameter EMPLOYEE_STATUS, also of record-type PERSON.

```
PROCEDURE PROCESS_RECS ( VAR EMPLOYEE_STATUS : PERSON );

BEGIN
  WITH EMPLOYEE_STATUS DO
    BEGIN
      WRITELN(NAME.FIRST);
      WRITELN(NAME.LAST);
      WRITELN(SS);
      WRITELN('BORN ',
          ORD(BIRTH.MO)+1,'-',BIRTH.DAY:2,', ',BIRTH.YEAR:4);

      CASE MS OF
        MARRIED :
            WITH MDATE DO
            WRITELN('MARRIED ',ORD(MO)+1,'-',DAY:2,', ',YEAR:4);
        WIDOWED :
            WITH MDATE DO
            WRITELN('WIDOWED ',ORD(MO)+1,'-',DAY:2,', ',YEAR:4);
        DIVORCED :
            WITH DDATE DO
            WRITELN('DIVORCED ',ORD(MO)+1,'-',DAY:2,', ',YEAR:4);
        SINGLE :
            WRITELN('SINGLE');
      END; {end of CASE statement}
    END; {end of compound-statement within WITH statement}
END; {end of procedure PROCESS_RECS}
```

### 5.3.11 Pointer Types

The data types covered thus far, both simple scalars and structured, are static data types. That is, a fixed amount of memory is allocated for each static variable declared. Accessing a declared variable has a fixed access method, see Section 5.4 on variable selectors. The global variables remain in existence during the entire execution of the program; the local variables remain in existance during the activation of a routine by invocation. Even a file data type, though variable in size, has a predetermined form and access method. Many programming situations call for dynamic data structures, where components vary in form and size, in length of time of existence, and in their means of access. Different kinds of programming problems necessitate a variety of linked data structures where the individual elements are linked by pointers to other related elements of the structure. Dynamic data structures, such as stacks, directed graphs, binary trees, or linked lists, may expand or contract dynamically as the program executes and as elements are created, linked, manipulated, and removed from the overall structure.

Pascal provides a generalized dynamic data type, called the pointer-type, by which the programmer may create these data structures and program their application. The variable components of such a structure are called dynamic variables. The data storage area used for dynamic variables is called the heap. Dynamic variables may be of any data type, except the file data type. As they are created, manipulated, or destroyed by specific programmed commands; their existence is not connected to nor dependent on any fixed section of the program.

Dynamic variables are not referred to directly by a user-declared identifier, but rather indirectly by pointer-variables, declared to be of the pointer-type data type. The pointer-variables point to the dynamic variables, which are called the targets of the pointer-variables. The type of data being represented by the dynamic variables, is called the target-type, i.e., the data type of the dynamic variables. To create a dynamic data structure, both the pointer-type(s) and target-type(s) are defined in a TYPE definition part, and pointer-variables are declared in a VAR variable declaration part. The target variables are not declared, but created and destroyed by programming procedure-calls on certain standard routines.

The dynamic variables are created by use of the predefined procedure, NEW, which associates a pointer variable to its target dynamic variable and allocates storage for it on the heap. Any dynamic variable created by NEW can be destroyed (releasing its occupied storage) by use of the predefined procedure DISPOSE. Two other predefined procedures, MARK and RELEASE, also provide a method of recovering the storage allocated to dynamic variables; but these routines are only available to program units compiled under the compiler option HEAPMARK (see Chapter 1). These four procedures are described further on in this section. The target variable is accessed through its pointer-variable.

To establish a dynamic data variable, first define a pointer-type-identifier in the TYPE definitions part:

```
    TYPE pointer-type-identifier = pointer-type;
```

where the identifier on the left of the equal sign is the introduction of a pointer-type-identifier, and the syntax pointer-type is:

__Pointer-Type__

```
----> ^ ----> target-type----->
```

The format of a pointer-type is then the up arrow (^) followed by the target-type, an identifier of the data type of the dynamic variables to be created. Once established, the pointer-type-identifier is available to be used, for example, within a forward definition of the target-type. The target-type is any previously defined type identifier (other than a file-type) or a type identifier which is going to be defined elsewhere in the same TYPE definitions part where the pointer-type is being defined. Note that in all other cases, the declaration of an identifier comes before any use of it. Use of a target-type identifier here, before it has been defined, is the only case in Pascal where an identifier may be specified before it is defined. In this case, target-type identifiers must be defined following the pointer-type declaration. Upon declaration of a pointer-type data type in the TYPE definition part, the pointer-type is said to be "bound" to the target-type identifier. See the following example:

```
TYPE P = ^T;

     T =  RECORD

            ELEMENT:CHAR;
            NEXT:P
          END;
```

In this example, P, is the pointer-type identifier of the pointer-type bounded to the target-type identifier, T, which is the data type of the dynamic variables to be created. Then T is defined to be a record-type, i.e., the dynamic variables (that pointers of type P will point to) will be records of type T. The record-type of the example contains two fields, one named ELEMENT, which is of the character type; and another field, NEXT, which is of the pointer-type, P. The ability for a dynamic record target variable to contain fields which may contain pointers to other targets is the key to creating a variety of dynamic data structures.

To establish pointer-variables, i.e., variables of a defined pointer-type and which may be used to point to dynamic variables, they must be declared in the VAR variable declaration part:

```
VAR POINTER1:P;

    POINTER2:P;

        M2:^INTEGER;
```

The declared pointer-variables themselves are considered static variables, but the variables to which they will point are dynamic variables. The declaration of a pointer-variable in the VAR (variable declarations) part does not yet create any variable to which it points, only the capability to do so. Refer to the procedure descriptions of MARK and RELEASE below to see how the pointer-variable M of pointer-type: ^INTEGER is used. Creation of dynamic variables is then possible, having established the pointer-type bound to a target-type and a pointer variable of the pointer-type. This is achieved by calling on the standard procedure NEW, thusly,

```
    NEW(POINTER1);
```

The result of this procedure call creates a new variable of the target-type and sets the pointer-variable, POINTER1, to point to the new variable.

The dynamic variable that was created has undefined contents at this point; however, it may be referenced by use of the variable selector construct, pointer-target. The syntax of the pointer-target selector is:

**Pointer-Target Selector**

```
---> variable-selector ---> ^ ---->
```

The format for referencing a pointer-target, in simplest form, is the pointer-variable identifier followed by an up arrow (^), for example:

```
    POINTER^ references the entire dynamic variable created.
```

In the example, the dynamic variable created by the call:

```
    NEW(POINTER1);
```

is a variable of the record-type, such that each of the fields in the dynamic record would be referencable by:

```
    POINTER1^.ELEMENT  {selects the first field in a record}
    POINTER1^.NEXT     {selects the second field in a record}
```

That is, we may assign values to those fields, thusly:

```
    POINTER1^.ELEMENT := 'A';
    POINTER1^.NEXT := NIL;
```

and we may create a second dynamic variable , link it to the first one, and assign values to the second dynamic variable's fields:

```
    NEW(POINTER2);
    POINTER1^.NEXT := POINTER2;
    POINTER2^.ELEMENT := 'B';
    POINTER2^.NEXT := NIL;
```

creating a list of two nodes at this point; with POINTER1 pointing to the head of the list.

As shown above, a special pointer-type constant, NIL, is available. When assigned to a pointer-variable it indicates that the pointer points to nothing at all. The pointer constant ,NIL, may be used with any pointer variable regardless of its target-type. That is, the value NIL is compatible to all pointer-type variables. Otherwise, distinct pointer-types are disjoint and are not compatible to each other. A pointer-variable is given the value NIL in the following assignment statement:

```
    POINTER1 := NIL;
```

Attempting to dereference (access a target through) a pointer-variable whose value is NIL or otherwise undefined results in a run time error message: POINTER ERROR.

Pointer variables may be tested for the occurrence of the value, NIL, which explicitly indicates that a pointer does not point to anything as in the following examples:

```
    IF POINTER1 = NIL THEN....

                or

    IF POINTER2 <> NIL THEN....
```

The basic operations available for pointer-variables, then, are assignment or comparison of two pointers to test for equality or inequality. Both pointer-variable operands must be pointers to compatibly typed targets, i.e., their targets must have identical target-types.

An assignment from one pointer-variable to another pointer-variable of the same pointer-type assigns the address of the target in that pointer to the other pointer-variable. That is, the result of such an assignment leaves both pointers pointing to the same target. For example, suppose POINTER1 and POINTER2 are pointer variables that point to two different target nodes of a list; or to two different lists with nodes of the same target-type:

Example List1:

```
                 --------------------------    -------------
POINTER1---->|ELEMENTA   |  pointer-to-b|--->|ELEMENTB
                 --------------------------    -------------

                 --------------------------    -------------
POINTER2---->|ELEMENTD   |  pointer-to-e|--->|ELEMENTE
                 --------------------------    -------------
```

Execution of the following assignment statement changes POINTER1 to point to the same node POINTER2 was pointing to, and POINTER2 remains the same:

```
POINTER1 := POINTER2
```

produces:

Example List2:

```
                 --------------------------    -------------
POINTER1--|   |ELEMENTA   |  pointer-to-b|--->|ELEMENTB
          |      --------------------------    -------------
          |
          |->--------------------------        -------------
POINTER2---->|ELEMENTD   |  pointer-to-e|--->|ELEMENTE
                 --------------------------    -------------
```

Note that this is not the same as executing an assignment statement between target variables. For example, execution of the following assignment statement between the target variables that POINTER1 and POINTER2 are pointing to, as first depicted in Example List1, would produce change in the contents of the dynamic variables. The pointer variables, POINTER1 and POINTER2, remain pointing to the space of their original dynamic variables; but the contents of the target POINTER1^ changes so that its node contents containing ELEMENTA and its pointer-to-b is lost, and the node's contents is changed. That is,

```
POINTER1^ := POINTER2^
```

produces:

Example List3:

```
                 -------------------------
POINTER1---->|ELEMENTD  |  pointer-to-e|->
                 -------------------------  |
                                            |
                                            |
                                            |
                                            |
                 -------------------------  V  -------------
POINTER2---->|ELEMENTD  |  pointer-to-e|--->|ELEMENTE
                 -------------------------     -------------
```

The node of B still exists but no pointer to it. To have inserted nodes D and E, between List1 nodes A and B, from the first example, Example List1, we could have written:

```
POINTER2^.NEXT^.NEXT := POINTER1^.NEXT;
POINTER1.NEXT        := POINTER2;
```

In the previous example, two pointer variables may be pointing to the same dynamic target variable of a structure. Therefore, two pointers may be compared to test for this condition. The result of a pointer-variable comparison is a Boolean value, TRUE or FALSE. However, only the equality or inequality comparisons can be made between pointer-variables. Operations allowable between target variables are dependent upon their target-type data types.

The memory used by either an individual dynamic variable or an entire dynamic data structure can be released and made available for other uses. When a dynamic variable is created by the NEW procedure, it stays in existence even after the procedure which called NEW ceases execution. The programmer may destroy a dynamic variable and release the space it occupies by using the procedure DISPOSE. Procedures MARK and RELEASE, available only to compilation units compiled under the compiler option, HEAPMARK. The procedures MARK and RELEASE may be used to mark where an entire dynamic structure begins, and after using NEW to create dynamic variables, release the additional storage obtained, with a call on RELEASE. With these procedures, data can easily be manipulated as a stack.

The four predefined procedures available for pointer-type variables are as follows: Each of these procedures accept one argument of the pointer-type, i.e., p and m below denote pointer-variables.

| | |
|---|---|
| NEW(p) | This procedure call creates a target-variable of the target-type that pointer-variable, p, has its pointer-type bounded to; NEW(p) points the pointer-variable, p, to the location of the newly created target-variable but does not |

| | |
|---|---|
| | put a value into it. The target-variable is referenceable by P. Components of a target-variable whose type is structured is referenceable by a variable selector (see avove or Section 5.4). If the target-type is a RECORD-type with a variant part, the enough space is allocated for the largest variant defined for the tag-field. |
| DISPOSE(p) | This procedure call, if p is a pointer variable which currently has a target, destroys that target (makes the target p^ inaccessible) and makes the space which it occupied, available for other dynamic variables. It does not change the value in p, and the user is cautioned not to reference a now non-existent target p^ nor access the value in p directly after DISPOSE(p), until p has been adjusted, e.g. set to NIL, or used again in NEW(p); or set to point to another dynamic variable, for example for another call on DISPOSE(p) or for other processing uses. Once a DISPOSE(p) has been invoked, any other pointer-variable cannot reference the now non-existent target, e.g. the sequence q := p; DISPOSE(p); also disallows referencing q^ because the target has been disposed of. Attempting to reference a non-existent target results in a run-time error message: POINTER ERROR. The argument pointer-variable p in a DISPOSE(p) call, may not be undefined or have the value NIL, and doing so results in a run-time error message: POINTER ERROR. The user is cautioned not to dispose of a dynamic record target variable p^, within a routine where the target (or component thereof) has been passed to a routine's VAR variable parameter; nor within a WITH statement, whose record-variable-selector is an element of the target; or both. |
| MARK(m) | This procedure call, if m is a pointer-variable of type ^INTEGER, sets m to the value of an address at the current frontier of the heap, the dynamically allocated memory area used for storage of dynamic variables. If no NEW calls have yet been made, MARK(m) obtains in m the very beginning of the heap storage; the first frontier. As dynamic variables are created by NEW, the heap expands and the frontier moves downward, i.e., unless DISPOSE makes space available behind the frontier, and NEW calls create targets which can fit in that freed space, the heap grows downward. The argument pointer-variable m in a MARK(m) call may be |

undefined or have the value NIL; but note that if m is currently pointing to a target datum, that value of m will be lost; e.g., a second MARK(m) using the same pointer m as set by a previous MARK(m) will lose the value of the first m. The purpose of obtaining a mark is so that RELEASE(m) may be called later to relinquish any storage used, beyond this frontier mark, by NEW(p) calls made subsequent to MARK(m). If DISPOSE, which frees up space by disposing of individual targets, is being used in combination with MARK, some NEW calls may use available storage behind the frontier. If DISPOSE is not being used in combination with MARK and RELEASE, a MARK(m), followed by several NEW(p)'s, and associated processing of the p^ targets, followed by a RELEASE(m); relinquishes all storage used since the MARK(m). The procedure MARK is only available to compilation-units compiled under the compiler-option HEAPMARK (see Chapter 1).

RELEASE(m) This procedure call, if m is a pointer-variable of type ^INTEGER, and has been previously set by MARK(m) causes all additional heap storage allocated at lower addresses than m to be relinquished. The argument pointer-variable m in a RELEASE(m) call may not be undefined, have the value NIL, or be of any other pointer-type other than ^INTEGER (i.e., it must not be pointing to a target-variable, for example as the argument p to NEW and DISPOSE does). RELEASE relinquishes the storage occupied by one or more dynamic target-variables created since a previous call on MARK, and exactly which storage is relinquished depends on whether DISPOSE has been also used:

1. If DISPOSE has not been used in combination with MARK and RELEASE, the NEW calls subsequent to MARK use storage only beyond the frontier at increasingly lower addresses (the heap grows downward). Therefore, the effect of RELEASE(m) is identical to Pascal R00, in that all storage is relinquished, and all target-variables created by NEW(p) are destroyed, since the last MARK(m).

2. If DISPOSE is in use in combination with MARK and RELEASE, NEW calls, made subsequent to a MARK, may use either storage behind or beyond the frontier of the heap obtained by a MARK. In this

case, the effect of RELEASE(m) is that only the additional heap storage, used beyond the frontier at lower addresses than m since the last MARK(m), is relinquished.

The procedure RELEASE is only available to compilation-units compiled under the compiler-option HEAPMARK (see Chapter 1).

Note that, as DISPOSE was not supported in Pascal R00, this advanced heap memory management mechanism in Pascal R01 and up causes MARK and RELEASE to be redefined, as above. Pascal R01 and up, not only uses memory more efficiently, but also requires that the arguments to MARK and RELEASE be pointer-variables of type ^INTEGER. As dynamic data structures are widely applicable to target-variables of the record-type, the user is referred for details to Section 5.3.10 on the record-type, this section on pointer-types, and Chapter 10 for run-time information on the heap memory allocation scheme in use internally.

If not enough memory has been generated in the user task workspace to accomodate either the user program heap or stack memory requirements a run-time error message: HEAP/STACK OVERFLOW occurs (see the user-guide information in Chapter1).

Pascal R01 and up also provides a new predefined function, STACKSPACE.

| | |
|---|---|
| STACKSPACE | This function-call returns a value of type INTEGER and is an integer which is the number of bytes remaining between the user heap and stack, at the run-time of executing the function-call to STACKSPACE. This function accepts no arguments. |

The predefined functions ADDRESS and ORD apply to pointer-variables.

| | |
|---|---|
| ADDRESS(p) | The result is the machine address of pointer-variable p. If p is a pointer-variable, then ADDRESS (p^) is an integer, which is the machine address of the target variable that p is pointing to. In this implementation, ADDRESS(p^) = ORD(p) + 8. |
| ORD(p) | If P is a pointer variable, then ORD(p) is the integer value of the machine address of the target of p. In this implementation, ORD(p) = ADDRESS(p^) - 8. |

To illustrate the basic uses of the pointer-type, see the following examples. They are the procedures: MAKELIST, on how to create a forward-sorted, singly-linked list of nodes; REMOVENODE, on how to remove one or more nodes; and ADDTOLIST, on how to insert a node. The overall program is called POINTEREXERCISER.

```
PROGRAM POINTEREXERCISER(OUTPUT);         (*Program-heading*)

TYPE P=^T;                        (*Declare a pointer-type*)
     T=RECORD                     (*Define a target-type*)
         DATA:CHAR;
         NEXT:P
       END;

VAR POINTER1,POINTER2:P;          (*Declare the pointer variables*)
    POINTER3,TOPLIST:P;
    CH:CHAR;VOWELS:SET OF CHAR;   (*Declare data variables*)
    ENTRYMADE:BOOLEAN;            (*Declare flag*)

PROCEDURE MAKELIST;               (*Procedural heading*)
(* MAKELIST CREATES A LIST OF THE ALPHABET, LEAVING
   POINTER1 POINTING TO THE TOP OF THE LIST AT THE
   THE ELEMENT CONTAINING CHARACTER 'A'.
   NIL, AS THE END OF LIST INDICATOR, OCCUPIES THE
   FIELD 'NEXT' OF THE ELEMENT CONTAINING THE 'Z'. *)
BEGIN                              (*Begin compound statement*)
  POINTER1:=NIL;                   (*End of list indicator is NIL*)
  FOR CH:='Z' DOWNTO 'A' DO        (*Get data to fill the list*)
    BEGIN                          (*Begin FOR loop's statement*)
      NEW(POINTER2);               (*Create & point to target var*)
      POINTER2^.DATA:=CH;          (*Assign data to target variable*)
      POINTER2^.NEXT:=POINTER1;    (*Link nodes,first using NIL*)
      POINTER1:=POINTER2;          (*Adjust linking pointer*)
    END;                           (*End FOR statement's statement*)
END (* END PROCEDURE MAKELIST *);


PROCEDURE REMOVENODE;
(* REMOVENODE DELETES THOSE ELEMENTS OF THE LIST
   THAT CONTAIN VOWELS IN THE FIELD 'DATA' *)
BEGIN                       (*Begin procedure's compound statement*)
  POINTER1 := TOPLIST;      (*Fetch top of list pointer*)
  POINTER2 := POINTER1;     (*Start a pointer to walk thru list*)
  VOWELS := ['A','E','I','O','U','Y'];
  WHILE POINTER2 <> NIL DO              (*Set up repetitive loop*)
    IF POINTER2^.DATA IN VOWELS        (*Check component for removal*)
      THEN
        IF POINTER2 = TOPLIST                   (*If at beginning - *)
          THEN
            BEGIN
              POINTER1 := POINTER2^.NEXT;       (*Bump pointers*)
              POINTER2 := POINTER2^.NEXT;
              TOPLIST := POINTER2;              (*Adjust top of list*)
            END
```

```
          ELSE                              (*If not at beginning - *)
            BEGIN
              POINTER1^.NEXT := POINTER2^.NEXT; (*Link previous node*)
              POINTER2 := POINTER2^.NEXT;  (*Bump to continue search*)
            END
      ELSE   (*This node does not contain a vowel, so bypass it*)
        BEGIN
          POINTER1 := POINTER2;            (*Bump pointers to next*)
          POINTER2 := POINTER2^.NEXT;
        END;
  END (* END PROCEDURE REMOVENODE *);


PROCEDURE ADDTOIIST (CH:CHAR);     (*Procedural heading,importing CH*)

(* ADDTOLIST INSERTS A NODE INTO THE FORWARD SORTED
   LIST, MAINTAINING AN ASCENDING ORDER. THE ENTRY IS MADE
   WHETHER OR NOT THE CH:CHAR ALREADY EXISTS IN THE LIST*)

BEGIN
  POINTER1 := TOPLIST;             (*Fetch top of list*)
  ENTRYMADE := FALSE;              (*Indicate no entry made yet*)
  IF TOPLIST = NIL                 (*If list empty, enter CH here*)
    THEN
      BEGIN
        NEW(POINTER1);             (*Create new dynamic variable*)
        POINTER1^.DATA := CH;      (*Fill data field of node*)
        POINTER1^.NEXT := NIL;     (*Set end of list indicator*)
        TOPLIST := POINTER1;       (*Establish new top of list*)
        ENTRYMADE := TRUE;         (*Indicate entry made*)
      END

    ELSE                           (*If list not empty, search*)
      IF POINTER1^.DATA > CH       (*Check if CH should be first*)
        THEN
          BEGIN
            NEW (TOPLIST);         (*Create new toplist and variable*)
            TOPLIST^.DATA := CH;   (*Enter character into data field*)
            TOPLIST^.NEXT := POINTER1; (*Link new node to old one*)
            ENTRYMADE := TRUE;     (*Indicate entry made*)
          END

        ELSE    (*Search non-empty list, looking two nodes ahead*)
          WHILE ENTRYMADE <> TRUE DO
            BEGIN
              IF POINTER1^.NEXT <> NIL   (*If current node not end*)
                THEN
                  IF POINTER1^.NEXT^.DATA > CH   (*Check two ahead*)
                    THEN
                      BEGIN                   (*To enter node here*)
                        NEW (POINTER2);       (*Create variable*)
                        POINTER2^.DATA := CH; (*Fill with data*)
                        POINTER2^.NEXT := POINTER1^.NEXT; (*Link it*)
                        POINTER1^.NEXT := POINTER2; (*Link previous*)
                        ENTRYMADE := TRUE;    (*Indicate entry made*)
                      END
```

```
                     ELSE  POINTER1 := POINTER1^.NEXT (*Skip onward*)
                 ELSE  (*If at end of list, add entry to end*)

                   BEGIN
                     NEW (POINTER2);           (*Create new variable*)
                     POINTER2^.DATA := CH;     (*Fill with data*)
                     POINTER2^.NEXT := NIL;    (*And end of list*)
                     POINTER1^.NEXT := POINTER2; (*Link last node*)
                     ENTRYMADE := TRUE;        (*Indicate entry made*)
                   END;
             END (* begin *);
END (* END PROCEDURE ADDTOLIST *);



PROCEDURE WRITELIST;

(* WRITELIST FETCHES THE TOP OF THE LIST POINTER AND
   EITHER PRINTS THE MESSAGE 'LIST EMPTY';
   OR VERTICALLY PRINTS THE LIST ELEMENTS DATA. *)

BEGIN
  POINTER1 := TOPLIST;
  IF POINTER1 = NIL
    THEN
      WRITELN ('LIST EMPTY')
    ELSE
      WHILE POINTER1 <> NIL DO

        BEGIN
          WRITELN (POINTER1^.DATA);
          POINTER1 := POINTER1^.NEXT;
        END;

END (* END PROCEDURE WRITELIST *);
(*

   MAIN BODY OF PROGRAM POINTEREXERCISER

*)

BEGIN
  TOPLIST := NIL;           (*Initialize pointers to NIL*)
  POINTER1 := NIL;
  POINTER2 := NIL;
  WRITELIST;                (*This call produces: LIST EMPTY*)
  MAKELIST;                 (*This call creates linked list*)
  TOPLIST := POINTER1;      (*Save top of list pointer*)
  WRITELIST;                (*This call prints: entire alphabet*)
  REMOVENODE;               (*Remove several nodes, i.e. vowels*)
  WRITELIST;                (*This call prints: alphabet without vowels*)
  CH := 'Y';                (*Establish data to add to the list*)
  ADDTOLIST(CH);            (*Add a node, i.e. 'Y', to the list*)
  WRITELIST;                (*This call prints the consonants*)
END (* END PROGRAM POINTEREXERCISER *).
```

The procedure MAKELIST creates a forward-sorted linked list of the alphabet characters as depicted in Figure 5-1.

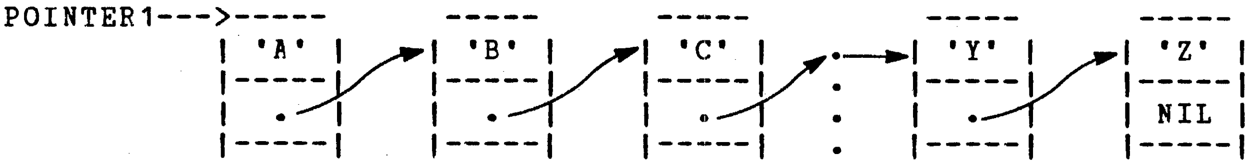


Figure 5-1 Linked List Creation

The procedure REMOVENODE removes those characters or nodes in the list holding vowels, leaving POINTER1 pointing to the 'Z' node; and TOPLIST adjusted downward to point to the character 'B' node. Note that no storage was released, only that the nodes containing vowels are no longer connective or considered part of the list. Refer to Figure 5-2.

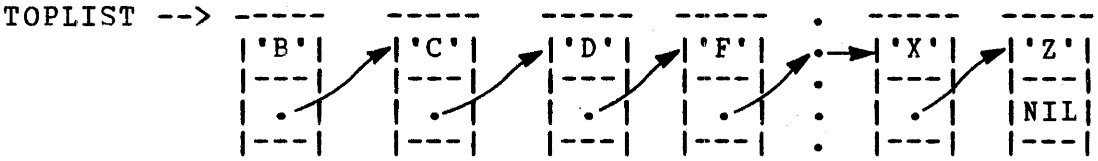


Figure 5-2 Linked List With Nodes Removed

The procedure ADDTOLIST adds the node for the character 'Y' back into the list to give a list of the consonant characters. This is depicted in Figure 5-3.

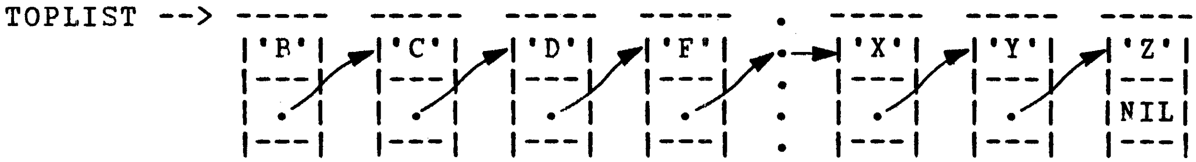


Figure 5-3 Node Addition to Linked List

## 5.4 VARIABLE SELECTORS

The basic operations on a variable, which is a datum whose value may be changed, are assignment of an initial or a new value to it, or a reference to its current value. Variables of any particular data-type may only assume values, undergo only certain operations, and be accessed by methods defined and meaningful for that data-type, as detailed in Section 5.3.

A named variable in a Pascal program is introduced, and given an identifier, by being declared in a parameter-list or in the variable declarations part of a block. A dynamic variable, which does not have a name of its own, is created by a call on the procedure NEW, using a named pointer variable.

Once a variable datum is introduced, by declaration or dynamic allocation, it can be selected for access.

Separately declared scalar variables, not part of another structure, are simply referenced by their identifiers. A scalar variable which is part of another structure is referenced as a component of the structure. A separately declared variable of a structured data-type may be referenced as the entire structure, with its identifier, or have one of its components referenced. Also, a structured data-type may be part of another structured-type, such that it can be reference in its entirety as the component of its parent structure, or have one of its components referenced.

The means to access a component of a structure, such as an array-component, for example, depends on the array-type declaration specifying index-types and component-types. Field identifiers are introduced within a record-type definition, and the means to access them depends upon a record-variable-selector or a WITH statement, which opens the scope to a record's type-definition, such that the field-identifiers become visible for direct reference within the scope of the WITH statement.

Dynamic variables are referred to as the targets of pointer variables.

The construct used to access named or targeted variables is a variable-selector. A variable-selector is a syntactical means of specifying one piece of data: a variable of a single simple scalar type, an entire structure, or a component of a structure. Very complex structures, where the intermingling of different data-types for fields in records, or targeted records or arrays of pointers, records of arrays, arrays of records, etc., require a more complex selector syntax. For this reason, the user will note the recursive use of selector in the syntax of the subcomponents of the general form of selector, itself. A detailed composite context-free syntax of variable selector is first provided.

### Variable-Selector (Summarized in Detail)

```
--->variable-identifier------------------------------------->
 |                      ^ ^ |                              |
 |                      | | |--> [ -->expression--> ] ---->|
 v-->field-identifier-->| | |       ^            |         |
                          | |       |<--- , <----/         |
                          | |                              |
                          | |--> . -->field-identifier---->|
                          | |                              |
                          | v--> ^ ----------------------->|
                          |                                |
                          |                                |
                          |<-------------------------------v
```

Taking the simple paths through this syntax graph clarifies its application. The variable-identifier by itself is the name of a declared variable, such that it is the selector of any scalar variable, including a pointer variable, or an entire structured variable such as an array-variable or a record-variable, or a file-variable. The field-identifier by itself is the name of a field in a record as it can be addressed within a WITH statement which has already specified a variable selector as the intended record, and whose record-type-definition has had its scope opened. The variable-identifier, when an array name, followed by square brackets enclosing one or more index-expressions is the selector of an array-component. The variable-identifier, when a record name, followed by a period and field-identifier is the selector of a field in a record variable. The variable-identifier, when a file variable, followed by ^, is the selector of its file-buffer variable (see Chapter 8). The variable-identifier, when a pointer variable, followed by ^, is the selector of its target variable. More complex selectors to access more deeply imbedded components in more complex structures can be formed by taking the path which can repeatedly loop back.

The more generalized syntax of selector is presented in the following four graphs:

### Variable-Selector (General Classifications)

```
---------------> identifier ----------------->
          |                           ^
          |----> array component ---->|
          |                           |
          |----> record field ------->|
          |                           |
          |----> pointer target ----->|
          |                           |
          V-----> file-buffer  ------>|
```

where the identifier is the name of a declared variable or field. The selector to specify or access any scalar variable, including a pointer variable, is its identifier. The selector to specify or access an entire structured variable, such as an array, record, or file, is its identifier. The selectors for an array-component, record-field, or pointer-target are detailed below. Note the recursive use of "variable-selector" in their syntax. In simplest form, the identifier of an array, record, or pointer variable becomes the selector referred to in the following graphs, respectively.

The syntax of selecting an array-component is:

### Array-Component (Selector)

```
---> variable-selector ---> [ ---> index-expression ---> ] --->
                               ^                      |
                               |                      |
                               |<------- , <---------v
```

The type of the "variable-selector" in this graph is an array-type; e.g., in simplest form, its the name of a variable-identifier or field-identifier of an array. The index-expression must be of the index-type of that array-type. In particular, if the index-type were defined as a subrange in the array-type definition, then the value of the corresponding index-expressions must lie within that subrange. The data-type of the array-component being selected is the component-type defined in the array-type definition. The array-component being selected is one of the elements of the array; the value of the index-expression(s) determines which one. The "variable-selector" in the syntax graph of array-component may also be a pointer-target, record-field, or another array-component.

The syntax of selecting a field of a record is:

### Record-field (Selector)

```
 ^--------------------------->|
 |                            |
 |                            v
--->variable-selector---> . ----->field-identifier--->
                      ^                            |
                      |                            |
                      <----------------------------v
```

The type of the "variable-selector" in this graph must be a record-type, e.g., in simplest form, the name of a variable-identifier or field-identifier which is itself a record.

The field-identifier in this graph must be the name of one of the fields in the associated field-list of that record-type. The data-type of the field being selected, is the type associated to that field name in the record-type definition. The "variable-selector" in the syntax graph of record-field can also be a pointer-target, array-component, or another record-field. Within a WITH statement that has the "variable-selector" in this graph specified, a record-field is selected without repeating the "variable-selector"; thereby the reason for the optional path in the graph which skips over the leading "variable-selector".

The syntax of selecting the target (dynamic variable) of a pointer is:

## Pointer-target (Selector)

```
--------> variable-selector ^ ------->
```

The type of the "variable-selector" in this graph must be a pointer-type, e.g., in simplest form, the name of a pointer variable. When the pointer variable is a component of an array, or a field in a record, the "variable-selector" in this graph is an array-component or record-field selector. To access a pointer-target, specification of a pointer-target selector consists of an appropriate variable-selector followed by an up arrow. The pointer-target is the dynamic variable that the pointer variable is pointing to. The data-type of the pointer-target being selected, is then the target-type of the pointer-type. The pointer-target is a dynamically allocated variable, with no declared identifier or name of its own; and can only be specified or accessed by means of a pointer-target selector. The "variable-selector" in the syntax graph of pointer-target can also be an array-component, record-field, or another pointer-target.

For example, given the type declarations:

```
    TYPE
        MATRICES = ^MATRIX;
        MATRIX = ARRAY[1..2, 1..3] OF REAL;
        DATUM = ^COORDINATE;
        COORDINATE = RECORD
                        X, Y : REAL;
                        NEXT : DATUM;
                     END;
```

and the variable declarations:

```
    VAR
        A,B : MATRIX;          {declaration of arrays A and B}
        I,J : INTEGER;         {declares integer variables I,J}
```

```
        XY, ZZ : COORDINATE;  {declares records XY and ZZ}
        POINT : DATUM;         {declares pointer variable POINT}
        VECTOR : MATRICES;    {declares pointer variable VECTOR}
        LOCATION : ^INTEGER;  {declaration of pointer LOCATION}
```

Typical examples of variable selectors are used in the following statement-sequences:

```
FOR I := 1 TO 2 DO
  FOR J:= 1 TO 3 DO
    B[I,J] := 1.0;       {B[I,J] is an array-component selector}
A := B;                  {A or B are entire array selectors}


XY.X := 1.0;             {XY.X selects field X in record XY}
XY.Y := 2.0;             {XY.Y selects field Y in record XY}
XY.NEXT := NIL;          {XY.NEXT selects field NEXT in record XY}


(* The following WITH statement can also initialize the same
   fields in the declared record variable XY without repeating
   the record-identifier in every field reference: *)


WITH XY DO               {XY is a record variable selector}
  BEGIN
    X := 1.0;            {field-identifier X is a record-field}
    Y := 2.0;            {field-identifier Y is a record-field}
    NEXT := NIL;         {field-identifier NEXT is a record-field}
  END;


ZZ := XY;                {ZZ and XY selects entire records}


NEW(LOCATION);          {LOCATION selects scalar pointer variable}
{LOCATION now points to pointer-target of type INTEGER}
LOCATION^ := 5;         {LOCATION^ selects a pointer-target}


NEW(VECTOR);            {VECTOR selects scalar pointer variable}
{VECTOR now points to pointer-target of array-type MATRIX}
FOR I := 1 TO 2 DO
  FOR J := 1 TO 3 DO
    BEGIN
      VECTOR^[I,J] := A[I,J];
{VECTOR^[I,J] selects an array-component of a pointer-target}
{A[I,J] selects an array-component of declared array variable A}

{The following two assignment statements are equivalent: }

      B[I,J] := VECTOR^[I,J];
{B[I,J] selects an array-component of declared array variable B}
```

```
      B[I,J] := VECTOR^[I][J];
 {VECTOR^[I][J] is a selector equivalent to VECTOR^[I,J]}
    END;


VECTOR^[2,3] := 6.6; {VECTOR^[2,3] selects an array-component}
A := VECTOR^;        {A and VECTOR^ select entire arrays}
DISPOSE(VECTOR);     {VECTOR selects scalar pointer variable}


NEW(POINT);          {POINT selects scalar pointer variable}
{POINT now points to a pointer-target of record-type COORDINATE}
POINT^.X := 3.0;     {POINT^.X selects field X of pointer-target}
POINT^.Y := 4.0;     {POINT^.Y selects field Y of pointer-target}
POINT^.NEXT := NIL;  {POINT^.NEXT selects field NEXT }
NEW(POINT);          {Create another pointer-target record}


(* The following WITH statement initializes the dynamic
pointer-target record of type COORDINATE without repeating
the record variable selector for each record-field: *)


WITH POINT^ DO      {POINT^ is a dynamic record variable selector}
  BEGIN
    X := 6.0;       {field-identifier X selects record-field}
    Y := 8.0;       {as does Y in dynamic record POINT^}
    NEXT := NIL;    {NEXT selects field in dynamic record POINT^}
  END;
```

The formats of variable selector become more complex as the data structures define more complex declarations, such as arrays of records, and records of arrays, etc.

For example, given the type declarations:

```
    TYPE
        POSITIONS = RECORD
                      FLAG : BOOLEAN;
                      X, Y : REAL;
                      QUADRANT : BYTE;
                    END;

        SERIES = ARRAY[1..10] OF POSITIONS;

        CAPSULE = RECORD
                    BESTFIT : SERIES;
                    WORSTFIT : SERIES;
                  END;
```

and the variable declarations:

```
    VAR
        AA : SERIES;         {declares an array of records}
        BB : POSITIONS;      {declares a record variable}
        CC,DD : CAPSULE;     {declares two records of arrays}
        I : 1..10;           {declares subrange integer variable}
```

The following statement-sequences reflect variable selectors used in specifying or accessing variables which are arrays of records or records of arrays, or their components.

```
WITH BB DO          {initialize a record variable}
  BEGIN
    FLAG := TRUE;
    X := 1.0;
    Y := 2.0;
    QUADRANT := 1;
  END;


FOR I := 1 TO 10 DO   {initialize an array of records}
  AA[I] := BB;      {AA[I] selects the ith record in array AA}


FOR I := 1 TO 10 DO   {reinitialize using record-fields}
  BEGIN   {record-field selectors in an array of records follow:}
    AA[I].FLAG := BB.FLAG;
    AA[I].X    := BB.X;
    AA[I].Y    := BB.Y;
    AA[I].QUADRANT := BB.QUADRANT;
  END;


CC.BESTFIT := AA;  {Both select entire arrays of records.}
                   {AA is an array of records: POSITIONS}
                   {CC.BESTFIT selects record-field BESTFIT}
                   {which is also an array of records: POSITIONS}


FOR I := 1 TO 10 DO
  BEGIN  {record-field selectors in records of arrays follow:}
    CC.WORSTFIT[I].FLAG := AA[10].FLAG;
    CC.WORSTFIT[I].X    := AA[10].X;
    CC.WORSTFIT[I].Y    := AA[10].Y;
    CC.WORSTFIT[I].QUADRANT := AA[10].QUADRANT;
  END;


DD := CC;      {Both select entire records of arrays of records}
```

# CHAPTER 6

# EXPRESSIONS, TYPE COMPATIBILITY, AND TYPE CONVERSIONS

## 6.1 EXPRESSIONS

An expression is programmed to represent a computable value at any one point in time during program execution.

In Pascal, an expression must be used in several other constructs. An expression is used:

- to set the value of a variable datum in an assignment statement of the form:

  ```
  variable_selector := expression;
  ```

- in an assignment statement within functions (to set the value of the function) of the form:

  ```
  function-identifier := expression;
  ```

- as a decision maker in the conditional and repetitive control statements of the form:

  ```
  IF expression THEN statement ELSE statement;

  REPEAT statement-sequence UNTIL expression;

  WHILE expression DO statement;

  FOR variable := expression DOWNTO expression DO statement;
                               TO
  ```

- to specify an actual argument value in an argument-list of a routine invocation in order to pass it to a value parameter;

- to specify an argument in the WRITE and WRITELN statements as a write-parameter;

- or as a means to specify an index-expression in an array-element reference.

This section defines how expressions are interpreted (or evaluated) and thereby describes how expressions may be formed by the programmer.

An expression is formed by one or more datum operands and/or operators which, when evaluated, results in a single value.

The value of an expression is computed by applying the operations (as dictated by the operators) on the data, proceeding from left to right sequentially, with modifications to this order of evaluation controlled by operator precedence rules and/or the presence of parentheses.

Only certain operations are defined for certain types of data; such that only certain types of operands may be associated with certain operators within an expression. The Pascal rules defining data type-compatibility are outlined in section 6.2.

A mixed mode arithmetic expression contains operands of two or more different data types. When a mixed mode arithmetic expression is evaluated, an operand may first be converted to a data type that is compatible to the type of another operand datum. The rules concerning this data type conversion are described in Section 6.1.1 and summarized in Section 6.3.

Whenever an expression is evaluated, applying the operators to operands yield an expression value of a certain data-type; which may be different from the operand type(s). Some operators serve as both arithmetic or set operators, such that some operations yield numeric or discrete scalar values, some yield set-values, depending on the operand data-types. Some operators yield Boolean values, as all comparison and logical operators do, and the set test membership does; although the operands may not be of Boolean type. Therefore, there are several kinds of expressions the user may wish to code or form.

They are:

- arithmetic expressions (operands may be mixed mode)
- relational expressions
- logical (or boolean) expressions
- set expressions
- set test membership expressions
- complete-expressions, which may be any one of the above, or a simple-expression of one or more terms or factors forming one of the above, or one of the above enclosed in parentheses.

To understand how to write expressions such that the expression written will be interpreted with the programmer's intended meaning, the rules of how an expression is evaluated with respect to operator-precedence having priority over the left to right order of interpretation must be known.

Syntactically, wherever the construct "expression" is required in another construct, the compiler will evaluate the written expression from left to right using the following priority rules, and syntax graphs which inherently define operator-precedences:

1. Operand Factors are evaluated. This includes obtaining values for literal constants, constant-identifiers, variables, function calls, parenthesized expressions, the logical negation operator NOT applied to its operand factor, and any set-constructors.

2. Terms are evaluated. If present, the multiplying operators are applied to factor values to form the value of terms. This includes applying the arithmetic operators *, /, DIV, and MOD to their operand factors; and applying the logical operator AND to its operand factors; and applying the set operator * to its operand factors.

3. Simple-expressions are evaluated. If present, the adding operators are applied to term values to form the values of simple-expressions. This includes applying the arithmetic operators unary plus, unary minus, +, or -, to their operand terms; applying the logical operator OR to its operand terms; and applying the set operators + and - to their operand terms.

4. Complete-expressions are evaluated. If present, the relational operators =, <>, <, >, <=, >= are applied to their simple-expression operands; and the set test membership operator IN is applied to its operands to form the value of a complete-expression.

The context-free syntax of expression is defined by the following four syntax graphs.

The syntax of a complete expression is:

## Complete-Expression

```
--->simple expr-------------------------------------->
                |   |  |   |  |  |   |       ^
                |   |  |   |  |  |   |       |

                =  <>  <  <=  >  >=  IN  simple expr

                |   |  |   |  |  |   |       |
                |   |  |   |  |  |   |       |
                v   v  v   v  v  v   v       |
                ---------------------------->|
```

The format of a complete-expression is any construct that consitutes a simple-expression (see syntax graph below) optionally followed by either a relational operator or the set test membership operator IN, followed by another simple expression.

The syntax of a simple expression is:

### Simple-Expression

```
  |--> + -->|
  |         |
--------------->term------------------->
                       ^   |  |   |
  |         |          |   |  |   |
  |--> - -->|          |   |  |   |

                     term  +  -  OR

                       ^
                       |   |  |   |
                       |   |  |   |
                       |   v  v   v
                       |<----------
```

The format of a simple expression is then an optional leading plus or minus sign, followed by a term, optionally followed by an adding operator (+, -, or OR), followed by a term; where the sequence from operator to ending term is repeatable.

The syntax of a term is:

### Term

```
--->factor----------------------------------->
             ^      |  |  |     |    |
             |      |  |  |     |    |
             |      |  |  |     |    |

           factor   *  /  DIV   MOD  AND

             ^
             |      |  |  |     |    |
             |      |  |  |     |    |
             |      v  v  v     v    v
             |<-----------------------
```

The format of a term is then a factor optionally followed by a multiplying operator (*, /, DIV, MOD, or AND) followed by a factor where the sequence from operator to ending factor is repeatable.

The syntax of factor is:

Factor

```
----------> unsigned constant ------------>
    |                                ^
    |                                |
    |---->  variable-selector  ----->|
    |                                |
    |------>   function-call   ----->|
    |                                |
    |---> ( --> expression --> ) --->|
    |                                |
    |------> NOT ----> factor ------>|
    |                                |
    V------>  set-constructor  ----->|
```

A factor represents a single operand, which may be a constant, a variable, a function call, another parenthesized expression, or logical negation, or a set-constructor.

The constant may be a literal constant as described in Section 3.3.4. or a constant-identifier defined in the CONST declarations part, as described in Section 4.2.2. The syntax of "constant" is given in Section 5.2.

A variable may be specified with a variable selector as described in Section 5.4.

In a function call, the beginning identifier must be the name of a defined function, optionally followed by an argument-list (see Section 9.3 on routines). The resultant value of the function call to be used as the operand value, has the data type of the function value defined in the declaration of the function-header.

Note that the parenthesized expression may be itself another "complete-expression"; or thereby, another "simple-expression", another "term", or another "factor". Parentheses are often used to ensure that an expression will be interpreted with the user's intended meaning.

The logical negation operator NOT applied to a factor, produces another factor, such that NOT has highest operator precedence within logical expressions; then AND, then OR.

In set-expressions, or set-test-membership expressions, a factor may be a literal set-constructor as described in Section 5.3.8.

To summarize, proceeding from left to right within an expression, the operand values of factors are obtained prior to evaluating the next operator within an expression. Parenthesized factors means that if the user encloses an expression in parentheses within a larger expression, the parentheses supersede the operator precedence rules. Note that the operator precedence in the language of Pascal is different from those established for other languages such as FORTRAN.

The rules of operator precedence are summarized in Table 6-1. Note that if all the operators used within an expression were of the same priority level, and no parentheses were used, interpretation of the expression would simply procede from left to right as it was written.

TABLE 6-1 OPERATOR PRECEDENCES

| PRECEDENCE | OPERATORS |
|---|---|
| Level 4 (Highest) | NOT |
| Level 3 | * / DIV MOD AND |
| Level 2 | + - OR |
| Level 1 (Lowest) | = <> < <= >= IN |

### 6.1.1 Arithmetic Expressions

Arithmetic expressions are formulas for computing numeric values.

An arithmetic expression consists of one or more numeric operands separated by arithmetic operators. There are six arithmetic operators. They are addition (+), subtraction (-), multiplication (*), real division (/), integer division with truncation (DIV), and integer remaindering, modulo, (MOD); where the value of A MOD B = (A - (B * K)) for integral K such that 0<= A MOD B < B.

Within an arithmetic expression, the multiplying operators *, /, DIV, and MOD have precedence over the adding operators + and -, or unary plus and unary minus.

Operands of the arithmetic operators must have numeric values of the numeric scalar data-types: BYTE, SHORTINTEGER, INTEGER, REAL, or SHORTREAL.

For example, given that A,B,C,D,E,F are variables containing numeric values of a numeric scalar type; the following expression is evaluated from left to right, as all the operators are on the same precedence level:

```
A + B - C + D
```

and is equivalent to:

```
( (A + B) - C ) + D
```

It is not equivalent to:

```
(A + B) - (C + D)
```

such that the user must parenthesize operands in any expression whose intended ordered meaning differs from the defined left to right order of interpretation.

Without parentheses, using constants in the above expression:

```
2 + 3 - 4 + 1
```

yields a value of 2, not 0.

Arithmetic expressions using operators of mixed precedence may also require parenthetical specification of its operands; for example:

```
A + B * C
```

is equivalent to:

```
A + ( B * C )
```

and is not equivalent to:

```
(A + B) * C
```

Without parentheses, using constants in the expression:

```
2 + 3 * 4
```

yields a value of 14, not 20.

When a unary plus or a unary minus appears in an arithmetic expression, they have the same precedence as the adding operators and are of lower priority than the multiplying operators. For example:

```
- A + B is interpreted as (-A) + B, not - (A + B)
```

whereas:

```
- A / B is interpreted as - (A / B), not (-A) / B
```

Successive arithmetic operands must be separated by operators; and two arithmetic operators may not be used in succession.

Also note that unary plus and minus are part of the syntax of simple-expression, not a term, nor a factor, such that we cannot write:

```
8 * +4 DIV -2
A / - B + - C - + D + + E - - F
```

but rather:

```
8 * (+4) DIV (-2)
A / ( - B ) + ( - C ) - ( + D ) + ( + E ) - ( - F )
```

Operands of the arithmetic operators in an arithmetic expression may be numeric valued:

literal constants, or constant-identifiers,

variables: including array-elements, record-fields, or pointer-target components;

function calls, whose function value is of some numeric scalar type

arithmetic expressions, or

arithmetic expressions enclosed in parentheses.

Once the expression is written in accordance with the established rules of left to right interpretation, operator precedence, and parenthetical priority; the expression can be evaluated as any mathematically equivalent expression. The evaluation may produce slightly different computational results, depending in part upon both the precision or conversion of input data, or the magnitude and accuracy of intermediate values obtained during the evaluation of the expression, mindful that reals are approximations of the real numbers input.

In nested parenthetical expressions, the innermost expressions are evaluated first. For example, the mathematical expression:

$$A - 6\,\frac{B^2}{C+D} * \frac{|E + F|}{2}$$

is equivalent to the arithmetic expression:

```
A - (6 * (SQR(B)/(C+D))) * (ABS(E+F) / 2))
```

which would obtain intermediate values for the function call SQR(B); (C+D); SQR(B)/(C+D); 6 * (SQR(B)/(C+D)); and the function call ABS(E+F); perform a real division on (ABS(E+F) / 2); multiply the intermediate values of the example's last two math terms; and subtract this result from the first math term A.

Whereas:

```
(A - (6 * (SQR(B)/(C+D)))) * (ABS(E+F) DIV 2)
```

obtains intermediate values for the function call: SQR(B); (C+D); (SQR(B)/(C+D)); (6 * (SQR(B)/(C+D)); substracting this value from A first and then multiplies this result by the value of the third math term in which an intermediate value had been obtained for ABS(E+F) to perform an integer division of (ABS(E+F) DIV 2), assuming E and F are integers.

A mixed mode arithmetic expression is an arithmetic expression containing numeric operands of two or more different numeric scalar types: BYTE, INTEGER, SHORTINTEGER, INTEGER, REAL, or SHORTREAL.

Type conversions are performed internally when mixed mode arithmetic expressions are evaluated. The value of an arithmetic expression, may therefore be dependant on this type conversion. Also when the value of an expression is assigned to a variable or function identifier in an assignment statement across the assignment operator (:=), a data type conversion may take place.

Tables 6-2 through 6-4 reflect the allowable operand types within an arithmetic expression and the resultant type of the value obtained from a mixed mode operation. Table 6-5 reflects the allowable numeric assignments across the assignment operator. Cautions are noted where truncation or range errors may occur. Where mixed mode operands are not allowed, such as: (shortreal DIV real), the table entry reflects the type as ERROR, and a compile-time compiler diagnostic error message would result: OPERAND TYPE CONFLICT.

### TABLE 6-2 ARITHMETIC MIXED MODE EXPRESSIONS (RESULTS USING OPERATORS + - *)

| LEFT OPERAND TYPE | RIGHT OPERAND TYPE-> | BYTE | SHORTINTEGER | INTEGER | SHORTREAL | REAL |
|---|---|---|---|---|---|---|
| BYTE | | BYTE | SHORTINTEGER | INTEGER | SHORTREAL | REAL |
| SHORTINTEGER | | SHORTINTEGER | SHORTINTEGER | INTEGER | SHORTREAL | REAL |
| INTEGER | | INTEGER | INTEGER | INTEGER | SHORTREAL | REAL |
| SHORTREAL | | SHORTREAL | SHORTREAL | SHORTREAL | SHORTREAL | REAL |
| REAL | | REAL | REAL | REAL | REAL | REAL |

### TABLE 6-3 ARITHMETIC MIXED MODE EXPRESSIONS (RESULTS USING REAL DIVISION OPERATOR)

| LEFT OPERAND TYPE | RIGHT OPERAND TYPE-> | BYTE | SHORTINTEGER | INTEGER | SHORTREAL | REAL |
|---|---|---|---|---|---|---|
| BYTE | | REAL | REAL* | REAL* | SHORTREAL | REAL |
| SHORTINTEGER | | REAL | REAL or SHORTREAL* | REAL or SHORTREAL* | SHORTREAL | REAL |
| INTEGER | | REAL | REAL or SHORTREAL* | REAL or SHORTREAL* | SHORTREAL | REAL |
| SHORTREAL | | SHORTREAL | SHORTREAL | SHORTREAL | SHORTREAL | REAL |
| REAL | | REAL | REAL | REAL | REAL | REAL |

*With real division operator /: BYTE/INTEGER/SHORTINTEGER, as operand types produce either REAL or SHORTREAL operations and results depending upon the type of assignment which is to occur after evaluation of the expression.

## TABLE 6-4 ARITHMETIC MIXED MODE EXPRESSIONS (RESULTS USING INTEGER DIVISION OPERATOR DIV and MOD)

| LEFT OPERAND TYPE | RIGHT OPERAND TYPE-> | BYTE | SHORTINTEGER | INTEGER | SHORTREAL | REAL |
|---|---|---|---|---|---|---|
| BYTE | | BYTE | SHORTINTEGER | INTEGER | ERROR | ERROR |
| SHORTINTEGER | | SHORTINTEGER | SHORTINTEGER | INTEGER | ERROR | ERROR |
| INTEGER | | INTEGER | INTEGER | INTEGER | ERROR | ERROR |
| SHORTREAL | | ERROR | ERROR | ERROR | ERROR | ERROR |
| REAL | | ERROR | ERROR | ERROR | ERROR | ERROR |

## TABLE 6-5 ASSIGNABLE MIXED MODES OF NUMERICS WITH ASSIGNMENT OPERATOR (:=)

| LEFT OPERAND TYPE | RIGHT OPERAND TYPE-> | BYTE | SHORTINTEGER | INTEGER | SHORTREAL | REAL |
|---|---|---|---|---|---|---|
| BYTE | | BYTE | BYTE* | BYTE* | ERROR | ERROR |
| SHORTINTEGER | | SHORTINTEGER | SHORTINTEGER | SHORTINT* | ERROR | ERROR |
| INTEGER | | INTEGER | INTEGER | INTEGER | ERROR | ERROR |
| SHORTREAL | | SHORTREAL | SHORTREAL | SHORTREAL | SHORTREAL | SHORTREAL |
| REAL | | REAL | REAL | REAL | REAL | REAL |

*Significant truncation occurs during execution, when assignment to a BYTE datum involves a value out of the BYTE range 0 to 255; or when assigning to a SHORTINTEGER datum, any INTEGER value greater than the maximum shortinteger (32,767) or less than the minimum negative shortinteger (-32,768). This condition may be trapped at run-time, if the program is compiled with the BOUNDS CHECK option, to produce the run-time error with the message:

VALUE RANGE ERROR.

Whereas the operators ( +, -, and * ) and the real division operator (/) are applicable to operands any scalar numeric (integer and real) type, the DIV and MOD operators require operands of any integer type. The second operand of /, DIV, and MOD cannot be zero. The second operand of MOD cannot be negative, or the runtime error VALUE RANGE ERROR occurs.

### 6.1.2 Relational Expressions

A relational expression is used to perform a comparison between two datum operands of compatible ordered data-types.

The value of a relational expression is a Boolean value, TRUE or FALSE.

A relational expression consists of two operands, separated by a relational operator. Two relational operators may not appear in succession. No spaces are allowed in the double-charactered relational operators <>, >=, or <=.

The six relational operators are:

| | |
|---|---|
| = | Equality |
| <> | Inequality |
| > | Greater Than |
| >= | Greater Than or Equal |
| < | Less Than |
| <= | Less Than or Equal |

The operands of a relational operator must be of compatible data types, e.g., we cannot compare a character to a number; or an arithmetic expression to a logical expression.

Relational expressions are evaluated from left to right, and as the relational operators have weakest operator precedence, it is not always necessary to parenthesize their operands. It is usually necessary to parenthesize relational expressions themselves, when they are operands to the logical operators.

That is, within a relational expression the value of the left operand has its relationship to the right operand tested to meet the condition specified by the relational operator. If this comparison or test validates the relationship, the value of the relational expression is TRUE, otherwise it is FALSE.

The relational operators =, <>, <=, and >= have been also defined to perform analgous operations on data which are of set-types. Set operations are detailed in Section 5.3.8. The relational operators as they pertain to sets are summarized in Table 5-4 in that section. Note that the relational operators < and > are not defined for sets. A relational expression with set-type operands and using the set relational operators =, <>, <=, and >= is referred to as a relational set-expression. If the evaluation of the relational set-expression validates the relationship between two set-type operands, the value of the relational set-expression is TRUE, otherwise it is FALSE.

The compatibly typed operands of a relational operator may be any two of the following constructs: (or any combination that ensures data-type compatibility of their operand values):

a constant: either a literal constant or a constant-identifier,

a variable: including an array-element, record-field, or pointer-target component,

a function call,

an arithmetic expression, or one enclosed in parentheses;

a set-expression, or one enclosed in parentheses; or

a set-constructor;

Tests for equality (=) or inequality (<>) may be performed on any two datum of the following compatible data-types:

| TYPE of Left Operand | Relational Operator = or <> | TYPE of Right Operand |
|---|---|---|
| CHAR | | CHAR |
| BOOLEAN | | BOOLEAN |
| numeric:<br>BYTE, SHORTINTEGER, INTEGER<br>REAL, SHORTREAL | | numeric:<br>BYTE, SHORTINTEGER, INTEGER<br>REAL, SHORTREAL |
| User-defined Enumeration | | Same User-defined Enumeration |
| Subrange | | Same/Enclosing type of subrange |
| SET type | | SET type, same base member type |
| ARRAY | | Identically-typed ARRAY |
| Array-element component-type | | Compatible to component-type |
| RECORD | | Identically-typed RECORD |
| RECORD with variant-part | | RECORD with same variant-part |
| Record-field of some type | | Type compatible with field type |
| Pointer | | Same Pointer type or NIL |
| Pointer target-type | | Type compatible to target-type |
| String Array | | String constant (same length) or identically-typed ARRAY |

Also see Section 6.4, Table 6-7 for cautions on ARRAY/RECORD structured comparisons, which are non-standard Pascal.

Tests for Greater Than or Equal (>=) or Less Than or Equal (<=) may be performed on operands of the following data-types:

| TYPE of Left Operand | Relational Operator <= or >= | TYPE of Right Operand |
|---|---|---|
| CHAR | | CHAR |
| BOOLEAN | | BOOLEAN |
| numeric: | | numeric: |
| BYTE, SHORTINTEGER, INTEGER | | BYTE, SHORTINTEGER, INTEGER |
| REAL, SHORTREAL | | REAL, SHORTREAL |
| User-defined Enumeration | | Same user-defined Enumeration |
| Subrange | | Same/Enclosing type of subrange |
| SET type | | SET type, same base member type |
| String Array | | String constant (same length) or identically-typed ARRAY |
| Array-element component-type | | Compatible to component-type |
| Record-field of some type | | Type compatible with field type |
| Pointer target-type | | Type compatible to target-type |

Note that the <= and >= operators are defined for standard scalar, enumeration, subrange, string or set types; but are not defined for whole arrays, whole records, or pointers. These operators, <= and >=, are also defined for array-elements, record-fields, pointer-targets, or pointer-target components, only if those subcomponents are of the standard scalar, enumeration, subrange, set or string types i.e., not whole arrays, whole records, or pointers.

Tests for Greater Than (>) or Less Than (<) may be performed on operands of the following data-types:

| TYPE of Left Operand | Relational Operator > or < | TYPE of Right Operand |
|---|---|---|
| CHAR | | CHAR |
| BOOLEAN | | BOOLEAN |
| numeric: | | numeric: |
| BYTE, SHORTINTEGER, INTEGER | | BYTE, SHORTINTEGER, INTEGER |
| REAL, SHORTREAL | | REAL, SHORTREAL |
| User-defined Enumeration | | Same User-defined Enumeration |
| Subrange | | Same/Enclosing type of subrange |
| String Array | | String constant (same length) or identically-typed ARRAY |
| Array-element component-type | | Compatible to component-type |
| Record-field of some type | | Type compatible with field type |
| Pointer target-type | | Type compatible to target-type |

Note that the < and > operators are defined for standard scalar, enumeration, subrange, and string types; but are not defined for whole arrays, whole records, pointers, or sets. These operators, < and >, are also defined for array-elements, record-fields, pointer-targets, or pointer-target components, only if those subcomponents are of the standard scalar, enumeration, subrange, or string types i.e., not whole arrays, whole records, pointers or sets.

A brief summary of valid comparisons (that can be made with the relational operators) between values of compatible data types are listed in Table 6-7.

Failure to use valid operands of compatible types results in a compiler diagnostic error message: OPERAND TYPE CONFLICT.

Given the declarations:

```
    CONST
      SPACE = ' ';
    VAR
      CH:CHAR; B:BYTE; I:INTEGER; J:SHORTINTEGER;
      REALS,REALS2 : ARRAY[1..10] OF REAL;
      SMALLRANGE : 1..50;
      S1,S2 : SET OF 1..50;
      REC1, REC2 : RECORD
                     NUMBER:INTEGER;
                     INITIAL:CHAR;
                   END;
      NAME : ARRAY[1..4] OF CHAR;
```

Some examples of relational expressions are:

```
'A' <> CH         {Character constant compared to CHAR variable}
CH = SPACE        {CHAR variable compared to character constant}
B <  55           {Byte variable compared to integer constant}
(I+J) > SQR(9)    {Arithmetic expression compared to function call}
REALS = REALS2    {Array compared to Array}
REALS[2] >= 3.2   {Array-element compared to real constant}
REC1 <> REC2      {Record variable compared to record variable}
REC2.NUMBER > J   {Record field compared to integer variable}
NAME <= 'JANE'    {String array compared to literal string}
SMALLRANGE = I    {Subrange variable compared to integer variable}
```

Some examples of relational set-expressions:

```
S1 =  S2               {Set-variable compared to set-variable}
S1 <> []               {Set-variable compared to null set}
S2 <= [1,2,8,9]        {Set-variable compared to set-constructor}
S1 >= [1,3] + [2,4]    {Set-variable compared to set-expression}
S2 - [7,9] <= S1       {Set-expression compared to set-variable}
S1 * S2 = [3,6,9]      {Set-expression compared to set-constructor}
(S1*S2)<>([1,2]+[2,3])  {Parenthesized set-expression comparison}
```

### 6.1.3 Logical (Boolean) Expressions

A logical expression is formed to test the truth of a condition; or of a compound condition, using the operator AND or OR; or to form the reverse of the truth of a condition, using the operator NOT.

The value of a logical expression is a Boolean value, TRUE or FALSE.

A logical expression, also called a Boolean expression, consists of a single logical operand, optionally preceded by NOT, or two or more logical operands separated by logical operators.

The logical operators are NOT, for logical negation; AND, for logical product (conjunction); or OR, for logical sum (disjunction). If A and B represent conditions which are either TRUE or FALSE, then logical operations are defined to mean:

| | |
|---|---|
| A AND B | is TRUE if both A and B are TRUE and neither is FALSE |
| A OR B | is TRUE if either A or B is TRUE, or if both are TRUE |
| NOT A | is TRUE if A is FALSE; or FALSE is A is TRUE. |

When a logical expression consists of a single operand, its data-type must be of Boolean type; i.e., and have a value of TRUE or FALSE. In a logical expression, two logical operands must be separated by a logical operator. The logical operators AND or OR are infix operators and must be surrounded by two logical operands. The logical operator NOT is a prefix operator, and must be followed by a logical operand.

Within a logical expression, logical negation NOT has precedence over logical product AND, and logical sum, inclusive OR. The logical product operator AND has precedence over the logical sum operator, inclusive OR.

As the logical operand of NOT must syntactically be a factor; when the condition to which NOT applies is a compound condition or a relational expression, it is required to be enclosed in parentheses. When the operand of NOT is a single operand factor such as a Boolean valued constant, variable, or function call; its operand need not be enclosed in parentheses. For example:

```
NOT A
NOT B
NOT TRUE
NOT EOF
NOT EOF(FILENAME)
NOT EOLN
NOT EOLN(FILENAME)
NOT (A AND B)
NOT (A OR B)
```

However, some conditions can be written so as to avoid the use of NOT. For example, the following logical expressions represent a relational expression as the operand of NOT, with C and D as operands of the relational operators; and these could be rewritten as relational expressions to test for the equivalent conditions:

```
NOT(C = D) could also be written as:  C <> D

NOT(C <= D) could also be written as:  C > D
```

Two logical operators may not appear in succession, except for the following two combinations:

```
AND NOT

OR NOT
```

All three logical operators may appear in one logical expression.

Logical expressions are evaluated from left to right unless an operator which has higher precedence is encountered.

For example, given that P,Q,S,T have values of BOOLEAN type:

The logical expression:

```
P AND Q OR S
```

is equivalent to:

```
(P AND Q) OR S
```

and the expression:

```
P OR Q AND S
```

is equivalent to:

```
P OR ( Q AND S )
```

The logical expression:

```
NOT P OR Q AND S
```

is equivalent to:

```
(NOT P) OR (Q AND S)
```

A truth table of Boolean operations in given Section 5.3.3 (see Table 5-1).

Operands of the logical operators NOT, AND, or OR may be any of the following Boolean valued constructs:

constant, e.g., either Boolean literal constant, TRUE or FALSE; or a constant-identifier;

variable:
including an array-element, record-field, or pointer-target component of type BOOLEAN;

function call;

logical expression (which always yields a Boolean value),

> Caution: Due to the logical operators having differing precedence it is best to enclose logical expressions in parentheses when they themselves are operands to other logical operators. See example above.

logical expression enclosed in parentheses,

relational expression (which always yields a Boolean value), enclosed in parentheses;

> Relational operators have lower precedence than the logical operators, so it is required to parenthetically enclose relational expressions as operands of the logical operators.
>
> For example, given that VALUE, A and B are scalar variables:

(VALUE < A) OR (VALUE > B) yields TRUE if VALUE outside A..B

(VALUE >= A) AND (VALUE <= B) yields TRUE if VALUE inside A..B

relational set-expression (which always yields a Boolean value), enclosed in parentheses;

Relational operators are analagous and have identical precedence to relational set operators used in relational set-expressions, so it is required to parenthetically enclose relational set-expressions as operands of the logical operators.

For example, given S1 and S2 are set-variables of type SET OF 1..50, a logical expression may be formed with two relational set-expressions:

```
(S1 <= [2,3,4]) AND (S2 <= [1,3,5])
```

set-test-membership expression (which always yields a Boolean value) enclosed in parentheses;

Relational operators have identical precedence to the set-test-membership operator IN, so it is best to parenthesize set-test-membership expressions as operands of the logical operators.

For example, given S1 and S2 are set-variables of type SET OF 1..50; and:

```
S1 := [2,4,5,6,8];
S2 := [1,3,5,7,9];
ITEM := 5;
{we may logically test for the conjunctive condition of
ITEM residing in both S1 and S2, by using two
set-test-membership expressions as the operands to the
logical operator AND.}
```

```
IF (ITEM IN S1) AND (ITEM IN S2) THEN ... ELSE ...
```

whereas:

```
IF ITEM IN (S1 + S2) THEN ... ELSE ...
```

{which tests for ITEM residing in the union of S1 and S2} {i.e., ITEM residing in either or both S1 and S2}

We cannot write an expression such as:

```
ITEM IN S1 AND S2
```

Note that an arithmetic expression may not be an operand to a logical operator.

Not using parentheses in mixed operator expressions, may result in a compile-time diagnostic error message: OPERAND TYPE CONFLICT. For example; given A,B,C, and D are scalar variables:

```
A <= B AND C >= D results in an OPERAND TYPE CONFLICT
```

due to the attempt to interpret B AND C as a logical expression, as AND has higher precedence than the relational operators.

### 6.1.4 Set Expressions and Set Test Membership Expressions

Analogous operations to arithmetic adding, subtracting, and multiplying numeric operands are available for sets (data of the set-type) to perform set union, set difference, and set intersection.

A set-expression consists of one or more set operands, separated by set operators. There are three set operators:

| | |
|---|---|
| + | set union |
| - | set difference |
| * | set intersection |

The value of a set-expression is a set.

Successive set operands must be separated by set operators; and two set operators may not appear in succession. The set operators are infix operators.

Operands of the set operators in a set-expression may be any of the following constructs, if of the set-type:

variable: a set-variable, including an array-element, record-field, or pointer-target component of the set-type;

set-expression;

set-constructor, or

or any of the above, enclosed in parentheses.

Set-expressions are evaluated from left to right; except for operator precedence. Within a set-expression, the set intersection operator (*), has precedence over the set union (+), or set difference (-), operators.

For example, given that X, Y, Z are set-typed operands: the following set-expressions are equivalent to those listed on the right:

```
X * Y + Z   is equivalent to     (X * Y) + Z
X + Y * Z                        X + (Y * Z)
X * Y - Z                        (X * Y) - Z
X - Y * Z                        X - (Y * Z)
X + Y - Z                        (X + Y) - Z
X - Y + Z                        (X - Y) + Z
X + Y - X + Z                    ((X + Y) - X) + Z
(X + Y) - (X + Z)                (X + Y) - (X + Z)
```

Some examples of set-expressions, using set-constructors:

```
[1,2,3] + [2,3,5]  is equivalent to, and yields: [1,2,3,5]
['A','B','C'] - ['C','D']  yields the set-value: ['A','B']
[1,2,3] * [2,3,5]          yields the set-value: [2,3]
```

Refer to Section 5.3.8 for other applicable examples of using sets and set operators in set-expressions.

A set-expression may be the right operand to the set test membership operator described below.

Given that S is a set of objects all of the same base member type T, then a given object of type T, it is either a current member of the set S, or it is not a member of the set S. We can test for this condition with a set test membership expression, using the set test membership operator IN:

```
object     IN       S
```

The operator IN is an infix operator. A set test membership expression tests to see if one datum operand on the left of IN, is a member of the set operand on the right of the operator IN.

The value of a set test membership expression is a Boolean value, TRUE or FALSE.

A set test membership expression consists of a left operand of some discrete scalar type, corresponding to the base member type of the right operand set-type, followed by the operator IN, followed by the right operand.

The right operand of IN may be any of the set-valued constructs:

variable: a set-variable, including an array-element, record-field, pointer-target component of the set-type.

set-expression, as described above; or a

set-constructor;

or any of the above enclosed in parentheses.

When the right operand of IN is a set of some base member type; the left operand of IN may be any of the following constructs:

constant: a literal constant, or a constant-identifier, of the base member type; or a user-defined enumeration type constant-identifier when the base member type is a user-defined enumeration type;

variable, of any discrete scalar type, including an array-element, record-field, or pointer-target component whose type is of the base member type;

function call, whose discrete scalar value is compatible to the base member type;

arithmetic expression: whose discrete scalar value is compatible to the base member type;

or any of the above enclosed in parentheses.

For example, given the declarations:

```
CONST
  SCORE = '_';
TYPE
  STAPLES = (FLOUR,SUGAR,EGG,BUTTER,SALT,MILK,
                                        SODA,POWDER,YEAST);
  PASTRY  = (CAKES,COOKIES,BREADS);
  OUNCES  = SET OF 1..16;
  AMOUNTS = ARRAY[STAPLES] OF OUNCES;
  RECIPES = ARRAY[PASTRY]  OF SET OF STAPLES;
VAR
  COOKIE, CAKE, BREAD : SET OF STAPLES;    {set-variables}
  SUPPLIES : AMOUNTS;       {array of sets of subranges}
  RECIPE : RECIPES;         {array of sets of enumerations}
  CAPS, LETTERS : SET OF CHAR;  {character set-variables}
  CH : CHAR;                {character variable}
  OZS, LBS : 1..16;         {integer subrange variables}
  OZ : OUNCES;              {set-variable}
```

then assuming these assignments:

```
COOKIE  := [FLOUR,SUGAR,EGG,BUTTER,SALT,SODA];
CAKE    := [FLOUR,SUGAR,EGG,BUTTER,MILK,SALT,POWDER];
```

```
BREAD   := [FLOUR,EGG,BUTTER,SALT,YEAST];
LETTERS := ['A'..'Z','a'..'z','_'];
CAPS    := ['A'..'Z'];
CH      := 'Z';
RECIPE[CAKES]   := CAKE;
RECIPE[COOKIES] := COOKIE;
RECIPE[BREADS]  := BREAD;
OZS := 3;
SUPPLIES[FLOUR] := [1..10];
OZ      := [1..8];
```

some examples of set-test-membership expressions are:

```
'B' IN ['A'..'Z']           {literal constant IN set-constructor}
SCORE IN  LETTERS           {constant-identifier IN set-variable}
SODA  IN  COOKIE            {enumeration constant IN set-variable}
CH    IN  CAPS              {character variable IN set-variable}
OZS   IN  SUPPLIES[FLOUR]   {scalar variable IN array-element set}
SQR(OZS) IN [1..16]         {function call IN set-constructor}
(OZS*2)  IN OZ              {arithmetic-expression IN set-variable}
SUGAR IN COOKIE + CAKE      {enumeration constant IN set-expr}
YEAST IN BREAD - CAKE       {enumeration constant IN set-expr}
SALT IN (BREAD*CAKE)        {enumeration constant IN set-expr}
FLOUR IN RECIPE[CAKES]      {enumeration constant IN array-element}
```

The set test membership operator IN has the same precedence as the relational operators. The relational operators =, <>, <=, >= are defined for sets, to form relational set-expressions discussed above in Section 6.1.2 and detailed in Section 5.3.8 on the set-type. The relational operators < and > are not defined for set operations. Refer to Table 5-4 for a summary of all operators pertaining to sets. As the set test membership operator IN has lower precedence, than the set operators, it may only be necessary to parenthesize set-expression operands of the operator IN, for clarity or readability.

A set-test-membership expression should be itself parenthesized when serving as an operand to the logical operators to test a condition with a logical expression (see Section 6.1.3).

## 6.2 TYPE COMPATIBILITY

In Pascal, there are several programming considerations concerning type-compatibility. Programming expressions with operand factors compatible to the expression operator, insuring assignment compatibility across the ":=" assignment operator, passing arguments to parameters, and planning the identity of type where necessary by establishing type-identifiers for separately declared variables, all require knowledge of Pascal's rules concerning type-compatibility.

Within an expression, as detailed in Section 6.1 above, operands of any operators must be of identical or compatible data types. When an operation is defined for a certain data-type, it can only be legally performed on two operands, if those operands are of identical or compatible types; to either each other or the operator requirements. The value of an evaluated expression may have a different type than its operands, or a type dependant on its operands or operators.

Aside from the compatibilities required in expressions, or across a particular expression operator, Pascal has two major degrees of type-compatibility. They are "identity" or "assignment-compatibility".

Passing an argument expression to a value-parameter only requires "assignment-compatibility" between the argument expression and value-parameter; whereas passing an argument variable to a variable-parameter requires "identity" of type between the argument variable and the variable-parameter. A function-argument name or a procedure-argument name being passed to its respectively appropriate formal routine parameter also requires that the parameter-lists of the routine-argument name and its formal routine are compatible, (see Section 9.6.5). Some argument-to-parameter type-compatibility checking may be suppressed when either value or variable parameter type-identifiers have been previously declared to be of universal (UNIV) type, requiring only the internal sizes of the arguments to be the same as the parameter internal sizes.

Within an assignment statement, we are also concerned with type-compatibility of the operands to the assignment operator (:=).

In an assignment statement of the form:

```
    variable_selector := expression;
```

an expression value of one type is "assignment-compatible" to a variable of another type, if the type of the former expression value is assignable to the latter variable type, either with some meaningful implicit data conversion and allowed by the rules of the language.

Within a function-definition, in an assignment statement of the form:

```
    function-identifier := expression;
```

an expression value of one type is "assignment-compatible" to a function-identifier of another type, if the type of the expression value is assignable to the defined type-identifier of the function value (in the function-header). (A function value may only be typed with a simple scalar type, such as CHAR, BOOLEAN, BYTE, SHORTINTEGER, INTEGER, REAL, SHORTREAL, or a user-specified subrange thereof, or a user-defined enumeration type, or a user-specified pointer-type type-identifier.)

Programming an expression, a routine invocation(passing arguments to parameters), an assignment statement, or a function definition assignment statement without adhering to the Pascal type compatibility conventions may result in a compile-time diagnostic error message:

```
    OPERAND TYPE CONFLICT
```

The following discussion summarizes Pascal type-compatibility as "identity" and "assignment-compatibility".

If two types are identical, they are assignment-compatible. The concept of assignment-compatibility includes identity.

Two types are identical, if the equality between them can be verified from source definitions without inspecting the internal structure of their definition.

We may introduce the name of a user-specified "type" (type_1 below) in a type-definition, which establishes the type-identifier (type_identifier_1 below) as available to subsequent code (within the scope of the type-definition), such as:

```
    TYPE
        type_identifier_1 = type_1;
```

Identity is programmable within a type-definition establishing identity between two type-identifiers:

```
    TYPE
        type_identifier_2  =  type_identifier_1;
```

or when two type-identifiers are identical to a third type-identifier, as in:

```
    TYPE
        type_identifier_2  =  type_identifier_1;
        type_identifier_3  =  type_identifier_1;
```

Identity is not established between two type-identifiers, by programming:

```
TYPE
    type_identifier_1   =  type_1;
    type_identifier_A   =  type_1;
```

even though both declarations have seemingly equivalent user-specified type_1 types.

When two type-identifiers are defined to be of different user specified types, they are of course not identical.

```
TYPE
    type_identifier_4   =  type_2;
    type_identifier_5   =  type_3;
```

A specific coding example illustrating these differences follows:

```
TYPE
    A1 = ARRAY[1..5] OF REAL;
    B1 = ARRAY[1..5] OF REAL;

    RANGE1 = 2..8;
    RANGE2 = 3..5;
    RANGE3 = 1..2;
    RANGE4 = 8..13;
```

The type-identifiers, A1 and B1, do not establish identical array-types and two different array variables declared to be of type A1 and B1 would not be compatible for comparisons of entire arrays, nor assignment compatible. This is because Pascal rules require identity of type of structured-types for entire arrays (and record-types for entire records) in order for them to be assignment-compatible. (The compatibility of any array-element is dependant on the component-type of its array-type; and in this case, an array-element in either an array of type A1 or an array of type B1, is compatible to each other and would be compatible to any type REAL is compatible to; e.g. REAL or SHORTREAL).

However, subrange variables of the type-identifiers RANGE1 through RANGE4, although not identical, are compatible due to the additional rules governing assignment-compatibility of subrange types, e.g., their enclosing type is identically type INTEGER.

Different data declared in separate variable declarations with identical type-identifiers are of identical type and are therefore identically compatible.

```
VAR
    var1  :  type_identifier_1;
    var2  :  type_identifier_1;
    var3  :  type_identifier_1;
```

Likewise, different data declared in the same single group variable declaration; to be of the type established by a previously defined type-identifier, are of identical type; and therefore identically compatible:

```
VAR
    var1, var2, var3  : type_identifier_1;
```

Also, different data declared in the same single group variable declaration, to be of the type whose user-specified definition follows the colon, are of identical type; and therefore compatible:

```
VAR
    var4, var5, var6  : type_2;
```

However, we do not gain identically typed data with separate declarations not using type-identifiers, such as:

```
VAR
    var7  :  type_1;
    var8  :  type_1;
    var9  :  type_1;
```

even though all three user-specified type_1's seemingly appear to be equivalent. Var7, var8, and var9 are not of identical type, are declared in separate variable declarations, and therefore are NOT compatible.

Identity of type is also not established, even if, in the same scope, the following two group variable declarations followed a type-definition, equating type_identifier_1 to type_1. For example,

```
TYPE
    type_identifier_1 = type_1;
VAR
    var1, var2, var3 : type_identifier_1;
    var4, var5, var6 : type_1;
```

Var4, var5, var6 are not of identical type to var1, var2, var3; even though type_identifier_1 = type_1 exists in the TYPE declarations part (see Compatibility rules below). Var1, var2, var3 are compatible, being of identical type, amongst themselves. Var4, var5, var6 are compatible, being of identical type, amongst themselves.

If, in the same scope, the following variable declarations followed the type-definition, equating type_identifier_2 to type_identifier_1, then all six variables would be of identical type and therefore compatible. For example:

```
    TYPE
        type_identifier_2 = type_identifier_1;
    VAR
        var1, var2, var3 : type_identifier_1;
        var4, var5, var6 : type_identifier_2;
```

In this case, var4, var5, var6 are compatible to var1, var2, var3; because the variable declarations are establishing identity of type amongst the variables with identically typed type-identifiers. For example,

```
    TYPE
        AAA   =  ARRAY[1..5] OF REAL;
        TTT   =  AAA;          {Establishes type-identity of types}
    VAR
        X1    : AAA;
        X2,X3 : AAA;
        X4    : TTT;    X5   : ARRAY[1..5] OF REAL;
        X6,X7 : ARRAY[1..5] OF REAL;
```

The arrays X1, X2, X3, and X4 are of identical types and therefore compatible; but not with X5, X6, X7. The arrays X6 and X7 are only identically compatible to each other, and are NOT compatible to the other arrays. However, the elements of all of the above arrays, as the component-type is REAL in each case, are all compatible to each other and to any datum of type REAL or a type compatible to REAL.

Each of the predefined type-identifiers define types which are not identical to each other.

That is, the following type-identifiers and types are not identical to each other:

BOOLEAN
BYTE
CHAR
INTEGER
REAL
SHORTINTEGER
SHORTREAL
TEXT
user-defined enumeration type
subrange-type of some enclosing type
SET OF some base member type
array-type[index-type(s)] of some component type
record-type of some field-list,
  with either or both a fixed-part or variant-part
pointer-type to some target-type
file-type of some component type

but a few of them, or their components, are assignment-compatible to each other.

The syntax of each of these types is detailed in Chapter 5, and the file-type (and TEXT) is detailed in Chapter 8; where the text and sample examples illustrate programming considerations particular to data of each type.

Elements of these types may have relationships whereby two datum as components may be considered compatible, e.g., in an array of records (where the array has a component-type declared as a record-type type-identifier), the array-element is compatible to a record variable whose type has also been declared as the identical record-type type-identifier. Similarly, in a record with a field which is an array, that field (if declared to be of an array-type type-identifier), is compatible to an array whose type has also been declared as the identical array-type type-identifier.

To summarize, although two data are declared to be of two types which seem to be equivalent because they define the same range of values (magnitude and precision) and internal structure, (storage size and binary representation within that storage); those two data may not be considered of compatible types in Pascal, unless the rules for identity or assignment-compatibility have been adhered to.

We shall now define Pascal type-compatibility rules regarding "identity" and "assignment-compatibility".

Two data, each having a data-type, are of identical-type if one of the following conditions for Pascal "identity" of type is satisfied.

1. Both types are identical when:

    Both types are of type BOOLEAN;

    Both types are of type CHAR;

    Both types are of type BYTE;

    Both types are of type SHORTINTEGER;

    Both types are of type INTEGER;

    Both types are of type REAL;

    Both types are of type SHORTREAL;

    Both types are of type TEXT;

   or both types are of identical user-defined type-identifiers; i.e., both datum were typed with (associated to) an identical type-identifier; such as <u>one</u> of the following:

    user defined enumeration type type-identifier
    subrange type-identifier
    set-type type-identifier
    array-type type-identifier
    string-array type-identifier
    record-type type-identifier
    pointer-type type-identifier
    file-type type-identifier

   (In all of the above identities of rule #1, the identical type-identifier may have been associated to the two datum in either:

    the variable declarations of a VAR declarations part;
    the parameter declarations of a parameter-list;
    the function-header which defined the function-value
        type-identifier; or an
    expression evaluation yielded a value of identical type;

   or both datum received their typing in the same single group variable declaration (VAR declarations part); where the typing was specified as either a "type" or a type-identifier).

2. A component of a structured-type is identically compatible to any datum of identical type to the type-identifier of the component-type of the structure. This includes the components of arrays, records, or files.

We shall now define assignment-compatibility.

A datum or expression of one type is assignable to a datum of another type, if the assignment of a datum or expression of the former type to a variable (or function-identifier within a function-definition) to a datum of the latter type is meaningful, perhaps with an internal data conversion.

Two datum, or a datum and an expression value, each having a data-type, are assignment compatible if one of the following conditions for Pascal assignment-compatibility is satisfied:

1. Both types are assignment-compatible if both types are identically compatible, as defined above; but not including file-types. In addition to identity, the following are assignment-compatible:

2. A BYTE type datum or expression is also assignable to a variable (or function-identifier within a function-definition) of type SHORTINTEGER, INTEGER, REAL, or SHORTREAL.

3. A SHORTINTEGER type datum or expression is also assignable to a variable (or function-identifier within a function-definition) of type BYTE, INTEGER, REAL, or SHORTREAL.

4. An INTEGER type datum or expression is also assignable to a variable (or function-identifier within a function-definition) of type BYTE, SHORTINTEGER, REAL, or SHORTREAL.

5. A REAL type datum or expression is also assignable to a variable (or function-identifier within a function-definition) of type SHORTREAL. REAL is not assignable to BYTE, SHORTINTEGER, or INTEGER.

6. A SHORTREAL type datum or expression is also assignable to variable (or function-identifier within a function-definition) of type REAL. SHORTREAL is not assignable to BYTE, SHORTINTEGER, or INTEGER.

7. A subrange-type, S1, of enclosing type, T1, is assignable to to type T2, if the enclosing type T1 is assignable to type T2.

8. If S1 is a subrange of enclosing type T1, and S2 is of enclosing type T2, and type T1 is assignable to type T2, then S1 is assignable to S2.

9. The empty set [] is assignable to any set-type. Set-variables, set-expression values, and set-constructors are assignment-compatible if their base-member-types are assignment-compatible.

10. The pointer constant NIL is assignable to any pointer-type.

11. A literal-string constant or named constant string is assignment-compatible to any string variable of the same fixed-length; not necessarily having identical index-types.

## 6.3 DATA TYPE CONVERSION

Data type conversion is the process of forcing operands of an operator to be of compatible types.

Within arithmetic expressions, the compiler automatically performs the following conversions:

1. Converts (expands) all integer variable values to the length of the longest operand of each operator
2. Compiles all literal integer constants to be of type INTEGER
3. Compiles all literal real constants to be of type REAL

These conversions are transparent to the user. The compiler implicitly converts the different real data types, REAL and SHORTREAL. The user may explicitly convert real variables to compatible types by using the built-in type changing functions, SHORTEN and LENG. Refer to section 5.3.7 for details on LENG amd SHORTEN. Also, the user may explicitly convert integer values to REAL with the standard function, CONV; and integer values to SHORTREAL with the standard function, SHORTCONV. Refer to section 5.3.4 for details on CONV and SHORTCONV.

Also, a data type conversion may take place across the assignment operator (:=) from its right operand to its left operand, when its operands are assignment compatible, as defined above in Section 6.2. Refer to Table 6-5 for a summary of assignment data type conversions.

## 6.4 SUMMARY OF EXPRESSION OPERATORS AND OPERANDS

Every operator is either a prefix, in which case it is written before its operand, or an infix, in which case it is written between its two operands. The operators "+" and "-" can be both prefixes and infixes. For some operators there are restrictions on the acceptable data types of the operand or operands. These are summarized in Table 6-6. Even when the operators are given operands of their required compatible data types, sometimes the result of an operation has a different data type. These conditions are also listed in Table 6-6.

## TABLE 6-6 SUMMARY OF OPERATORS

| OPERATOR | PREFIX OR INFIX | TYPE OF OPERAND(S) | TYPE OF RESULT | DESCRIPTION |
|---|---|---|---|---|
| * | I | numerical* | numerical | multiplication |
| * | I | SET | SET | intersection |
| / | I | numerical* | REAL or SHORTREAL | real division |
| DIV | I | BYTE, SHORTINTEGER or INTEGER | INTEGER or SHORTINTEGER | integer division, with truncation |
| MOD | I | BYTE, SHORTINTEGER or INTEGER | INTEGER or SHORTINTEGER | integer remainder (modulo) |
| + | I,P | numerical* | numerical | addition |
| + | I | SET | SET | set union |
| - | I,P | numerical* | numerical | subtraction |
| - | I | SET | SET | set difference |
| IN | I | left:ordinal<br>right: SET | BOOLEAN | membership test |
| =,<> | I | all types, except FILEs See Table 6-7 | BOOLEAN | test equality or inequality |
| <=,>= | I | various types See Table 6-7 not pointers, nor whole arrays/records | BOOLEAN | test for comparison |
| <,> | I | various types See Table 6-7 not sets, not pointers, not whole arrays, not records | BOOLEAN | test for strong comparison |
| NOT | P | BOOLEAN | BOOLEAN | negation |
| AND | I | BOOLEAN | BOOLEAN | conjunction |
| OR | I | BOOLEAN | BOOLEAN | inclusive OR |

*numerical data that are BYTE, INTEGER, SHORTINTEGER, REAL, SHORTREAL types

Comparisons can be made between data of certain types, either "identically" typed structures, or "assignment-compatible" scalars, or a variety of string structures, and a brief generalized summary of those that are valid is listed in Table 6-7. Structure comparisons are non-standard Pascal, and in this implementation of Pascal, structured comparisons (arrays and records) are performed binarily on a byte by byte basis, regardless of alignment gaps imbedded in their internal storage; such that they may only be useful in comparing structures without alignment gaps. Also see Section 10.6.1 on Internal Data Storage Requirements.

TABLE 6-7 SUMMARY OF VALID COMPARISONS

| DATA TYPE | =,<> | <=,>= | <,> |
|---|---|---|---|
| Compatible Predefined Type | OK | OK | OK |
| User-Enumeration | OK | OK | OK |
| Subrange | OK | OK | OK |
| SET | OK | OK | none |
| ARRAY | OK | none | none |
| RECORD | OK | none | none |
| Pointer | OK | none | none |
| String-Literal or constant (same lengths) | OK | OK | OK |
| String variable & same-length string-literal or constant | OK | OK | OK |
| String variable & identically-typed variable | OK | OK | OK |

# CHAPTER 7
# PASCAL EXECUTABLE STATEMENTS

## 7.1 INTRODUCTION

An executable statement is an instruction to operate on the data of a program or to transfer control to another place in the program. Pascal statements are executed in sequence, except where control is explicitly transferred elsewhere.

Pascal provides a rich variety of executable statements, both simple and structured.

The simple statements are the empty statement, the assignment statement, the procedure call, and the GOTO statement. These statements are called simple statements because they cannot be divided into smaller statements. All other statements are considered structured statements.

Structured statements may contain simple-statements or other structured statements, such as the compound statement which provides the mainframe for the body of a program, module, or routine.

A conditional statement, either the IF or CASE statements, chooses whether to execute one, or one of two, or one of many statements.

A conditional repetition statement, either the WHILE or REPEAT statement, may contain other statements which are to be executed repeatedly while or until some condition is true.

A controlled repetition statement, the FOR statement, executes a statement repeatedly, for a predeterminable number of repetitions.

The WITH statement, by delineating an expanded scope, facititates the programming of record-field selectors, to access the fields of a record-variable.

Variations of the procedure-call statement included in the Pascal repertoire, not covered in this chapter, are the additional statements afforded the user by predefined routines, such as NEW, MARK, RELEASE, and DISPOSE (see Section 5.3.11 on the pointer-type), for programming dynamic data structures; and for I/O, the routines: RESET, REWRITE, READ, READLN, WRITE, WRITELN, PAGE, GET and PUT (see Chapter 8 on the file-type and I/O). Also, language extensions afforded by the Perkin-Elmer Prefix and SVC support routines are described in Sections 10.3 and 10.4.

The syntax of a statement is:

Statement

```
---> label ---> : --->|
 |                 ^  |
 |                 |  |
 V---------------->|  |
                      |
|<--------------------V
|
V--------------------------------------->
    |                            ^
    |---> compound-statement --->|
    |                            |
    |-----> case-statement ----->|
    |                            |
    |-----> for-statement ------>|
    |                            |
    |------> if-statement ------>|
    |                            |
    |----> while-statement ----->|
    |                            |
    |----> repeat-statement ---->|
    |                            |
    |-----> with-statement ----->|
    |                            |
    |--> assignment-statement -->|
    |                            |
    |-----> procedure-call ----->|
    |                            |
    V-----> goto-statement ----->|
```

Any statement within the body of a block may be labeled with an unsigned integer and colon, if that label has been previously declared in the LABEL declarations part of the block. Every label, so declared, must be used on exactly one statement within the statement sequence of that block. However, the compound statement serving as the body of a block is never labeled.

## 7.2 SIMPLE STATEMENTS

Simple statements are those which cannot be further subdivided. The simple statements are the:

- empty statement,
- assignment statement,
- procedure call, and
- GOTO statement.

### 7.2.1 The Empty Statement

The empty statement is simply a void, wherever a statement is allowed, and/or a void between two statement separators.

While on the surface it appears that an empty statement is a ludicrous concept, there are some instances in which the construct is useful.

For example, when developing a program, it is often desirable to simulate a yet unimplemented procedure. To do this, it is necessary only to write the procedure heading followed by a body consisting of an empty compound statement:

```
PROCEDURE FORM_FEED_CHECK (LINE_NUMBER:INTEGER);
BEGIN
END;
```

Other common occurrences of the empty statement are introduced by extraneous semicolons, for example, (assuming I and J are integer variables and P1 is a pointer-variable):

```
BEGIN
   I:=0;
   J:=0;
   P1:=NIL;
END;
```

Here, the semicolon following the NIL is unnecessary; it merely introduces an empty statement between itself and the END. The semicolon after NIL is unnecessary because the next keyword END; serves the purpose of ending the compound statement.

The introduction of this empty statement has no effect on the efficiency of the program. In many cases, the use of the empty statement in this manner may serve to alleviate subsequent reprogramming of the statement just prior to END.

If another statement were to be added after the "P1 := NIL" statement in the example above and the semicolon were not already present after NIL, a semicolon would have to be inserted to serve as a statement separator.

Note the use of the empty statement below, where the statement labeled 301 serves simply as a statement on which to hang a label for the GOTO statement to escape to the exit.

```
PROGRAM ALPHABET_NUMBER(INPUT,OUTPUT);
LABEL 301;
VAR CH : CHAR;
     I : INTEGER;
BEGIN
  READ(CH);
  IF NOT (CH IN ['A'..'Z','a'..'z'])
     THEN GOTO 301
     ELSE
          BEGIN
          IF CH IN ['A'..'Z'] THEN
            I := ORD(CH) - ORD('A') + 1;
          IF CH IN ['a'..'z'] THEN
            I := ORD(CH) - ORD('a') + 1;
          END;
  WRITELN(I);
301: ;
END.
```

The empty statement can also be used as a case-labeled statement in the CASE statement, when one or more of the multiple alternatives, or the OTHERWISE clause, requires no action at all. For example, the application below, using the CASE statement, writes back only the even-numbered units digit, other than zero, of the integer read in.

```
PROGRAM PRINT_UNITS_DIGIT(INPUT,OUTPUT);
VAR  I : INTEGER;
BEGIN
  READ(I);
  I := I MOD 10;
  CASE I OF
    0 : ;
    2,4,6,8 : WRITELN(I);
    OTHERWISE  ;
  END;
END.
```

### 7.2.2 The Assignment Statement

The assignment statement is used to assign values to variables; or within function-definitions to assign values to the function-identifier. It has the syntax:

<u>Assignment-Statement</u>

```
---->  variable-selector  -------> := ---> expression --->
 |                              ^
 |                              |
 V---> function-identifier --->|
```

The variable-selector names a previously declared or dynamic variable into which is placed, the value of the expression on the right of the assignment operator symbol, :=. Within a function-definition, the function-identifier is the name of the function currently being defined. This second form of the assignment statement establishes the value of the expression on the right of := as the value of the function named on the left of the assignment operator. This second form of the assignment statement may not occur outside the function-definition. Note that the assignment operator symbol is a double-character symbol, the colon and equal sign, and is read as "becomes". No space may appear between the colon and the equal sign.

The data type of the value of the expression must be assignment compatible with the data type of either the variable-selector or function-identifier on the left of the assignment operator (see Section 6.2 on type-compatibility). If they are not of assignment-compatible types, a compile-time user diagnostic error message is displayed in the compiled-program listing below the errant line, with the message OPERAND TYPE CONFLICT.

When the assignment statement is executed, the expression is evaluated (see Chapter 6) and its value is assigned to, or placed into, the datum indicated by the variable-selector or function-identifier.

Run time errors would occur, for example, if the variable were a subrange type, and the value of the expression did not lie in that subrange; and the program had been compiled under the compiler-option BOUNDSCHECK. The run time error warning of this condition contains the message VALUE RANGE ERROR.

For an example of the assignment statement:

```
    VARIABLEA := 65;
```

replaces the current value of variable VARIABLEA, with the value 65; and the assignment statement:

```
    VARIABLEA := VARIABLEB;
```

replaces the current value of variable VARIABLEA, with the current value of variable, VARIABLEB, assuming they are both declared with types that permit assignability between them.

Another example of assigning an expression value to a variable:

```
    VARIABLEA := VARIABLEA + VARIABLEB - 25;
```

assigns the accumulated value of VARIABLEA plus VARIABLEB minus 25, to VARIABLEA.

An example of using the assignment statement within a function definition follows.

```
    FUNCTION CUBE(NUMBER:REAL):REAL;
      BEGIN
        CUBE := NUMBER*NUMBER*NUMBER;
      END;
```

In this case, the function-identifier, CUBE, is assigned or "becomes" the value of the expression NUMBER*NUMBER*NUMBER; which when CUBE is subsequently referenced in another expression as a function-call, that value is returned as the value of the function CUBE. For example,

```
    POLYNOMIAL := A*CUBE(X)+B*SQR(X)+C*X+D;
```

returns for CUBE(X), the value X*X*X, during the expression evalutation in the assignment statement for POLYNOMIAL.

### 7.2.3 The Procedure Call Statement

Large complex programs can be subdivided by the programmer into smaller units of coding that perform simpler or common portions of the overall programming problem. These units are called routines. A routine may be a function, which, when referenced within an expression, computes a value. A routine may be a procedure, which, when called into execution by the procedure call statement, performs some operation or set of actions.

Whereas a function-call is syntactically a variant of a "factor" within an expression (see Chapter 6), a procedure-call is syntactically a variant of "statement". Note that although both a function-call and procedure-call have identical syntax, where they are used is different.

The programmer defines the procedures, giving them names with identifiers and defining their set of expected parameters, if any, in its procedure-header statement, described in Chapter 9. Definition of the procedures occurs in the routine-declaration part of a block, as described in Section 4.2.5. A procedure call statement calls into execution a previously declared procedure by means of its name, i.e., using the identifier of the procedure-header that named it in its definition.

If the procedure definition specified a parameter-list, i.e., a list of expected parameter data, then the procedure call must specify an appropriate argument-list. If the procedure definition did not specify a parameter-list, then the procedure call must not specify an argument-list.

The syntax of a procedure call statement is:

### Procedure-Call

```
---> identifier --------------------------------->
                    |                      ^
                    |                      |
                    |                      |
                    v---> argument-list --->
```

where the identifier is the name of a procedure, previously declared or defined in a routine declarations part, i.e., the identifier must be visible in the current scope of where the procedure-call statement is being written; and where the syntax of the argument-list is:

### Argument-List

```
  ^---------------------------------------------------->|
  |                                                     |
  |            |---> variable-selector --->|            |
  |            |                           |            |
  |            |---> procedure-argument -->|            |
  |            |                           |            V
-----> ( ----->|                           |-----> ) ----->
           ^   |                           |   |
           |   |---> function-argument --->|   |
           |   |                           |   |
           |   |-------> expression ------>|   |
           |                                   |
           |<--------------   ,  <-------------V
```

in which variables, procedure or function names, and/or expressions are the arguments, as the actual data or identifiers being passed to the called procedure.

The format of a procedure call statement is, then, the procedure identifier, optionally followed by an argument-list. The format of an argument-list is one or more arguments, each separated from the other by a comma, with the entire list enclosed in parentheses.

An argument-list defines in a one-to-one correspondence positionally and by data type the arguments of this particular call on the procedure. The number of arguments, then, must exactly equal the number of parameters defined in the parameter-list specified in the procedure definition. The arguments are substituted for the parameters before the procedure is executed.

Each argument being passed to a parameter must be compatible entities. The argument corresponding to a variable (VAR) parameter must be a variable, as specified by a variable-selector (see Section 5.4). The argument corresponding to a value parameter must be an expression (see Chapter 6). The argument corresponding to a formal procedure parameter must be a procedure-argument name. The argument corresponding to a formal function parameter must be a function-argument name.

The syntax of a procedure-argument is:

### Procedure-Argument

```
---> procedure-identifier --->
```

The syntax of a function-argument is:

### Function-Argument

```
---> function-identifier --->
```

The argument data types of variables and expression values must be compatible with the corresponding parameter types except that:

- an argument corresponding to a value-parameter of a string type may be a string of any length, and
- an argument corresponding to a universal parameter (UNIV) may be a variable or expression value that occupies the same number of storage locations as the parameter type.

Execution of a procedure call statement obtains the selected arguments, passes them to the parameters of the procedure, and executes the compound statement of the body defined in the procedure called. Once the procedure body is executed, execution control returns to the next sequential statement following the procedure-call statement, unless execution control was transferred elsewhere by a GOTO statement. Refer to Chapter 9 for details on encoding a procedure definition with parameter declarations, and examples of routine invocations with argument specifications.

Examples of procedure call statements, assuming P1, P2, P3, P4, and P5 are previously declared with procedure-headings, are:

```
P1;                    (*procedure call without arguments*)

P2(A+G-1,2*B,4.0); {procedure call with expression arguments}

P3(V1,V2,V3);     {procedure call with variable V1,V2,V3 args}

P4(P1);                {procedure call with procedure-argument}

P5(P1,2*B,V2,FUNC1); {procedure call with many kinds of args}
```

### 7.2.4 The GOTO Statement

The GOTO statement is a simple-statement syntactically, but as its purpose is to direct the transfer of execution control, its placement requires an understanding of structured-statements and routines, and the reader may wish to return to this section later. The labeling of any statement (with certain restrictions concerning prior label declarations, scope, statement levels, and procedure levels) makes it possible to directly transfer execution control to that labeled-statement by referring to that label in a GOTO statement.

The syntax of the GOTO statement is:

GOTO-Statement

```
---> GOTO ---> label --->
```

where the label is an unsigned integer (between 0 and the value of MAXINT, inclusively) and has the syntax:

label

```
---> digits --->
```

where digits is formed by up to ten consecutive decimal digits, and the label is defined on a labeled-statement i.e., the digit-sequence of the label and its subsequent colon prefixes either a simple-statement or a structured-statement, such as:

```
     label :  simple-statement;
```

or

```
     label :  structured-statement;
```

Refer briefly to the syntax graph of "statement" in Section 7.1. The label used in a "GOTO label" statement is not to be confused with a case constant on a case-labeled statement, nor a constant label on a record variant.

A statement label, in order to appear on a labeled-statement, must first be declared as a "label" in the LABEL declarations part of the block in which the labeled-statements appear. Refer briefly to Section 4.2.1 which describes the LABEL declarations part.

For example:

```
PROGRAM SUM_THE_ELEMENTS(INPUT,OUTPUT);

LABEL  1000,2000,3000,9000,9999;

VAR LOTS : ARRAY[1..5] OF INTEGER;
    I, COUNT, NEGSUM, POSSUM : INTEGER;
BEGIN
  FOR I := 1 TO 5 DO
    READ(LOTS[I]);  {read 5 integers from 1 line on INPUT}
  NEGSUM := 0;
  POSSUM := 0;
  COUNT := 0;
  FOR I := 1 TO 5 DO
    BEGIN
      IF LOTS[I] = 0 THEN GOTO 9000;  {escape from for-loop}
      IF LOTS[I] < 0 THEN GOTO 1000;
      GOTO 2000;
      1000 : NEGSUM := NEGSUM + LOTS[I];
      WRITE(LOTS[I]);
      GOTO 3000;
      2000 : BEGIN
               COUNT := COUNT +1;
               POSSUM := POSSUM + LOTS[I];
             END;
      3000 : ;
    END;
  WRITELN('NEGSUM = ',NEGSUM,'POSSUM = ',POSSUM);
  GOTO 9999;
  9000: WRITELN('WARNING: ZERO ENTRY - TERMINATED SUM');
  9999: ;
END.
```

Label 1000 labels a simple assignment statement, label 2000 labels a structured compound statement, and label 3000 labels an empty-statement; all within the same statement-sequence. Labels 9000 and 9999 label statements in the outermost statement-sequence of the body of the block.

The "GOTO 9000" statement escapes from the for-loop by going to a labeled-statement outside its statement-sequence; to its enclosing statement-sequence. The "GOTO 1000" statement causes a direct transfer of execution control to the statement labeled 1000, so that effectively only the negative entries triger a WRITE. The "GOTO 2000" statement skips over the additional processing for negatives and the WRITE. The "GOTO 3000" statement, by 3000 being at the end of the statement-sequence of the for-loop, allows the for-loop to continue after summing negatives. The "GOTO 9999" statement simply skips over the abnormal termination message mechanism of statement 9000.

Although this example, as with most programs, can be written without the use of the GOTO statement; it presents the associative process of label-declarations, labeling statements, and the placement and effect of GOTO statements.

Standard Pascal governs the use of GOTO statement by the following rule. If a label prefixes a statement S, that label is only allowed in GOTO-statements, which are:

- in the statement S; or
- in the statement-sequence (if any) in which S is immediately contained; and
- if that statement-sequence is the statement-sequence of the compound-statement that forms the body of a block, in the procedure and function declarations of that block.

For an example of the first rule, considered the following structured-statement labeled 77, below. A GOTO 77 is allowed (legal Pascal but not recommended) within that statement.

```
LABEL 77;
...
BEGIN
...
77:  WHILE expression DO
       BEGIN
         statement_1;
         statement_2;
         IF expression THEN GOTO 77;
         statement_4;
         GOTO 77;  {Caution: looping back like this requires
                    user discretion in first adjusting values
                    being evaluated in the WHILE expression,
                    or the program may loop forever}
         ...
       END;
...
END;
```

Note that, in the above example, the labeling "77:" was not placed on the component compound-statement of the WHILE statement, which would also be legal. If the labeling "77:" was placed on that compound-statement, the WHILE expression would not be reevaluated after execution of either GOTO 77.

To apply the first rule to labeled simple-statements, note that of the four simple-statements, the only one that can contain a GOTO statement is the GOTO statement itself. The only instance of the first rule applied to simple-statements, although legal (no compile-time error message occurs) is to be avoided. That is a GOTO statement going to itself.

```
LABEL 8;
...
BEGIN
...
8 : GOTO 8;     {Although legal Pascal, hangs up a program}
...
END;
```

The second rule states that when either a simple-statement or a structured-statement with a label, is contained in a statement-sequence, a "GOTO label" statement is allowed in that same statement-sequence. For example:

```
LABEL  10,11,12;
...
BEGIN
  IF expression THEN GOTO 10;
  statement_1;
  GOTO 10;
  12 : statement_2;
  GOTO 11;
  10 : statement_3;
  statement_4;
  GOTO 12;
  11 : statement_5;
END;
```

A GOTO statement can be used to jump (forward or backward) from its placement to another statement within the same executing statement level, or from an inner statement level structure to an outer statement level structure, i.e., within the same statement-sequence, or an enclosing statement-sequence.

Backward jumps are not encouraged as it weakens the strengths of a well structured program. As will be seen, the purpose of the GOTO statement is not a proviso for handconstructed linear programming loops; but rather useful for escaping from a Pascal structured-statement loop, to skip over the remaining statements in a statement-sequence, as an abnormal exit from a procedure, or as an abnormal exit due to function aborts.

It is legal Pascal to jump from an inner statement structure level to an outer, but not the reverse.

This implementation of Pascal does not give an error message for example, if the user jumps into a structured-statement such as:

```
BEGIN
  GOTO 100;
  IF I <> -1 THEN
  100: I := -1
   ...
END.
```

or

```
BEGIN
WHILE expression DO
  BEGIN
    statement_1;
    13 : statement_2;
    statement_3;
  END;
...
GOTO 13;
...
END.
```

but this is an invalid use of the GOTO statement, not legal Pascal, and not recommended, and may cause infinite looping in the previous example.

The third rule states: that if the statement-sequence in which the labeled-statement is contained is the statement-sequence that forms the body of a block, then a "GOTO label" statement is allowed in the function and procedure declarations of that block. This means that a "GOTO label" statement within a procedure may refer to a label, not only within its own body, but in an enclosing procedure, provided that the label prefixes a simple-statement or structured-statement at the outermost level of nesting of the block of the enclosing procedure (or main program).

You can jump out of a procedure to a lower procedural level; i.e., from an inner nested routine to an enclosing procedure, but not from an enclosing routine into an inner nested routine.

For example, an invalid use of the GOTO statement, which would result in a compile-time error message, is:

```
PROGRAM TABOO;
  PROCEDURE BADENTRY;
     LABEL 666;
     BEGIN
       ...
       666: statement_1;
       ...
     END;
BEGIN
  ...
  GOTO 666;
  ...
END.
```

An example of using GOTO statements, amongst routines:

```
PROGRAM HOPALONG;
LABEL 900,777,888,999;
  PROCEDURF EXITJUMP;
    LABEL 200,300;
      PROCEDURE SKIPOUT;
        LABEL 500;
        ...
        BEGIN {skipout}
          ...
          {Conditionally jumps to normal routine exit}
          IF expression THEN GOTO 500;
          ...
          {takes abnormal exit to enclosing routine}
          GOTO 200;
          500 : ;
        END;
   BEGIN {exitjump}
     ...
     SKIPOUT;      {calls a routine with extra exit to 200}
     ...
     {Conditionally jumps to normal routine exit}
     IF expression THEN GOTO 300;
     ...
     {Takes abnormal exit to program statement-sequence}
     GOTO 900;
     200 : statement_n;
     ...
     {Conditionally jumps to main program statement 888}
     IF expression THEN GOTO 888;
     ...
     {Conditionally jumps to program termination}
     IF expression THEN GOTO 999;
     ...
     300 : ;
   END;
BEGIN {main program}
  ...
  IF expression THEN GOTO 999; {conditionally abort program}
  ...
  EXITJUMP;               {calls a routine with extra exits}
  ...
  {Conditionally aborts program at 777 or 888}
  IF expression THEN GOTO 777 ELSE GOTO 888;
  900 : statement;
  ...
  777 : GOTO 999;   {skips over abnormal termination of 888}
  888 : statement;  {provides a labeled statement prior end}
  999 : ;          {provides a label at program termination}
END.
```

Pascal, as both a sequential and block-structured programming language, executes statements one after another in the textual order in which they are written; beginning with the first statement of the body of the main program which is a compound-statement itself. Procedure level and statement level is zero.

The compound-statement serving as the body of a block in either a program, module, function, or procedure is never labeled. Execution enters the body of any block, to execute the statement-sequence of its compound-statement by means of the normal Pascal execution control mechanisms.

A statement level is increased by one, as another structured-statement within a structured-statement is written to be executed. Execution of each of the structured-statements is described in the remainder of this chapter. Statement-sequence execution continues, until the last statement in the statement-sequence of the main program is completed, where the program terminates.

If a procedure-call is written, the procedure level is increased at the place of invocation by one. The procedure-call causes execution control to start executing the main compound-statement body of the procedure block. Within a routine, any calls on nested procedures increases the procedure level by one, at the place of invocation. Exiting a procedure returns execution control to the next sequential statement following the procedure-call invocation.

The statement-parts of functions are executed during expression evaluations, which contain their invocation. Exiting a function returns control to the expression evaluation in progress.

Execution of externally linked modules, occurs upon invocation, as either a procedure or function, depending on which type of routine the module has been programmed to serve as; and exits accordingly.

This is the normal flow of execution of a Pascal program without using the GOTO statement.

Within a single compilation unit (either a program or a module) it is legal to jump from an inner routine level to an outer routine level, or from an inner structured-statement level to an outer statement level, but not the reverse.

Just as the GOTO statement may not be used to jump into internal routines, it cannot be used to jump into external modules. Additionally, a GOTO statement cannot be used to jump from an external module, whether the module is serving as a procedure or a function, to any statement in the main program (or anywhere outside of itself).

The GOTO statement is not usually used within a function definition to goto to the end of the function, because the function call having been referenced anywhere within an expression is expecting some resultant value returned, i.e., the function isn't being executed as a "statement". Going to a labeled empty statement at the end of a function-definition without having set the function-identifier to a value is returning an undefined value as its result, to some unsuspecting expression evaluation in progress. However, functions may receive certain arguments for which its value is not computable, and the function-definition may be programmed for this condition, e.g. after giving an error message to identify the reason for aborting, to GOTO to a labeled empty statement (or a final labeled statement) at the end of the main program to terminate the main program. For example:

```
PROGRAM ESCAPE(INPUT,OUTPUT);
LABEL 9999;
...
FUNCTION SQUAREROOTE(NUMBER:REAL) : REAL;
VAR X,Y : REAL;
  BEGIN
    IF NUMBER < 0
       THEN
         BEGIN
           WRITELN('NEG VALUE GIVEN TO SQUAREROOT=',NUMBER);
           GOTO 9999;
         END
       ELSE
         BEGIN
           ...
           {compute squareroot of number}
           SQUAREROOTE := expression;
           ...
         END;
       END;
BEGIN {main program}

...

9999: WRITELN('PROGRAM TERMINATION');
END.
```

The label declarations part shall specify all labels that prefix a statement in the corresponding statement part. Each declared label shall prefix one statement in that statement part. A compile-time error message is given if a statement is labeled and no corresponding label has been declared, or a label has been declared and no statement, or more than one statement has been labeled with that label.

Note that a label, once declared, has a scope slightly disimilar to the scope of a declared identifier, i.e., the declared label of an outer block cannot be used to label a statement in an inner nested routine; as it is no longer visible for the purposes of prefixing statements. (However, it is visible for the purposes of going to wherever it prefixes a statement; i.e., it is visible for using in a "GOTO label" statement.) To prefix a statement, a label must be used to label a statement in the body of the same block containing the LABEL declarations part declaring it as a label.

It is legal to use the same label number in two different block scopes. If a certain label number is declared and prefixed on a statement, in an outer block; and the same label number is declared and prefixed on a statement in an inner block; additional complexities arise to avoid the ambiguity caused by a GOTO that label. When a label as in use in an outer block, was redeclared in an inner block and used to label a statement in the inner block, a GOTO statement in the inner block goes to the inner block statement prefixed with that label, not to a labeled-statement in an outer block using a similar label. As this implementation of Pascal extends the standard limit of labels from four to ten digits, it is not necessary for a programmer to repeat the declaration of the same integral value of a label.

## 7.3 STRUCTURED STATEMENTS

The various structured statements available in Pascal are individually described in the following sections. They are the:

- compound statement,
- IF statement,
- CASE statement,
- FOR statement,
- REPEAT statement,
- WHILE statement, and the
- WITH statement.

The compound statement allows a sequence of statements to be written anywhere a single statement is required. The IF and CASE statements are decision making mechanisms to choose from one, two, or more alternatives based on some condition, i.e., they provide for conditional execution of one or more statements. The FOR statement provides for controlled repetitive execution, and not based on some conditions but rather on a predetermined number of consecutive values. The REPEAT and WHILE statements provide mechanisms for conditional repetitive execution. The WITH statement identifies a record variable selector and delineates an expanded scope so that record field references may be easily specified without requiring tedious repetition of the record variable selector.

Structured statements may contain simple statements or other structured statements.

### 7.3.1 The Compound Statement

In Pascal, when more than a single statement is written, as when a sequence of several instructions is to be executed, the compound statement may be used to enclose them, and in most cases is required. This presents the series of statements as a single syntactical entity, a "statement". The compound statement has the syntax:

Compound-Statement

```
--->BEGIN--->statement-sequence--->END--->
```

where the statement-sequence has the syntax:

Statement-Sequence

```
-----statement------>
   ^             |
   |             |
   |             v
   <-----;<------
```

As will be shown, the compound-statement is sometimes "required" in all of the other structured statements (CASE, FOR, IF, WHILE, WITH statements), except for the REPEAT statement. Wherever the syntax of the other structured statements show the construct, "statement", a compound-statement must be used when that "statement" needs to be programmed as more than one; but note that the REPEAT statement syntactically allows a statement-sequence between its keywords:

    REPEAT statement-sequence UNTIL expression

As noted earlier, the statement body of a Pascal program, module, or routine is itself a compound statement. The compound statement serving as the body of block is never labeled with a statement-label for the purpose of going to it with a GOTO statement (see Section 7.2.4). Also a compound statement is itself a statement and, therefore, can be contained in other compound statements or can contain other compound statements.

The format of the compound statement is a statement sequence in which each statement is separated from the other by a semicolon, and the entire sequence is enclosed in the keywords, BEGIN and END. Within the statement sequence of a compound statement, the semicolon serves as a statement separator. The last statement of the sequence need not end with a semicolon, as the keyword, END, serves this purpose. However, if the semicolon is present, as may often happen from the habit of ending a statement, it is treated as introducing the empty statement. The empty statement has no effect. The statement-sequence may also be a single statement, and the single statement may be the empty statement (see example in 7.2.1).

The statements within the statement-sequence of a compound-statement are executed in textual order, except as modified by execution of a GOTO statement.

### 7.3.2 The IF Statement

As a decision making mechanism in programming, it is often necessary to be able to choose to execute a statement or sequence of statements based upon some logical condition. The conditional statement provided in Pascal for this purpose is the IF statement. The IF statement provides for the conditional execution of one statement, which may in turn be a compound statement, or provides the conditional execution of one of two statements, either one or both of which may be a compound statement.

The syntax of the IF statement is:

IF-Statement

```
--->IF-->expression-->THEN-->statement--------------------------->
                                       |                     ^
                                       |                     |
                                       |                     |
                                       v--->ELSE-->statement->
```

The first format of the IF statement is then the keyword, IF; followed by an expression that produces a Boolean result of true or false; followed by the keyword, THEN; followed by a statement. If the user has a sequence of statements to be executed following the keyword, THEN, ensure that they are presented as a compound statement, and surround them in the keywords, BEGIN and END. In this format of the IF statement, when no ELSE clause is present, the execution of the IF statement proceeds as follows. The expression is evaluated, and if the condition is true, the statement following the keyword, THEN, is executed. Otherwise, if the condition if false, the statement following the keyword, THEN, is not executed. In either case, execution continues at the next sequential statement following the IF statement.

Another format of the IF statement includes the ELSE clause. This allows the decision based upon a condition to choose between executing two statements. Its format is the keyword IF; followed by an expression that produces a Boolean result of true or false;

followed by the keyword, THEN; followed by a statement; followed by the keyword ELSE; followed by a statement. Do not place a semicolon before the ELSE clause as that would indicate the end of the IF statement prior to the presentation of the ELSE clause. Each statement may be a compound statement when there is a sequence of statements to be executed upon the occurence of either the THEN or the ELSE condition. When the ELSE clause is present in this format of the IF statement, execution proceeds as follows: The expression is evaluated, and if its value is true, the statement following the THEN is executed. Otherwise, if its value is false, the statement following the ELSE is executed. In either case, program execution continues at the next sequential statement following the IF statement.

The general form of an IF statement to enhance readibility may indent both the "THEN" and paired "ELSE" with respect to the "IF" beginning the statement. (This indentation, as for all indentation is good practice but is not required by the language or the compiler).

```
IF expression
   THEN  statement1
   ELSE  statement2
```

The general form of an IF statement without an ELSE clause is:

```
IF expression
   THEN statement
```

If the statements within an IF statement are compound statements, the general form may be indented as follows:

```
IF expression
   THEN
     BEGIN
       statement1;
       statement2;
       ...
       statementi;
     END
   ELSE
     BEGIN
       statement1;
       statement2;
       ...
       statementj;
     END
```

and the variations:

```
IF expression
   THEN
     statement
   ELSE
     BEGIN
       statement1;
       statement2;
       ...
       statementm;
     END
```

or

```
IF expression
   THEN
     BEGIN
       statement1;
       statement2;
       ...
       statementn;
     END
   ELSE
     statement
```

If a statement were to follow any of the above IF statements, of course, a semicolon would be added at the end of the formats shown, in order to separate the IF statement from its succeeding statement.

Some examples:

```
{An IF statement with no ELSE clause}

IF CURRENT_VALUE > MAXVALUE_RECORDED
   THEN
     MAXVALUE_RECORDED := CURRENT_VALUE;

{An IF statement with an ELSE clause}

IF NUMBER < 0
   THEN
     WRITELN('NEGATIVE NUMBER')
   ELSE
     WRITELN('ZERO OR POSITIVE NUMBER');

{A nested IF statement}

IF NUMBER = 0
   THEN
     EMPTYCOUNT := EMPTYCOUNT + 1
   ELSE
     IF NUMBER > 0
        THEN PCSITIVECOUNT := POSITIVECOUNT + 1
        ELSE NEGATIVECOUNT := NEGATIVECOUNT + 1;
```

```
{An IF statement with compound-statement statements}

IF CH IN ['0'..'9']
   THEN
     BEGIN
       DIGIT := ORD(CH) - ORD('0');
       SUM   := SUM + DIGIT;
     END
   ELSE
     BEGIN
       IF CH IN ['A'..'Z']
          THEN
            BEGIN
              LETTER := CH;
              LFLAG  := TRUE;
            END
          ELSE
            BEGIN
              LFLAG := FALSE;
              CH    := ERRORCHAR;
            END;
       WRITELN(CH);
     END;
```

The IF statements can be nested, of course, but caution must be used so as not to present an ambiguous situation, such as:

```
IF expr1
  THEN
    IF expr2
      THEN
        statement1
      ELSE
        statement2;
```

It is not clear upon which expression the execution of statement2 depends: expr1 or expr2. To make the execution of statement2 dependent on expr1, the user writes:

```
IF expr1
  THEN
    BEGIN
      IF expr2
        THEN
          statement1
    END
  ELSE
    statement2;
```

To clearly make the execution of statement2 dependent on expr2, the user writes:

```
IF expr1
  THEN
    BEGIN
      IF expr2
        THEN
          statement1
        ELSE
          statement2
    END;
```

An IF statement with no ELSE clause cannot be followed by the keyword ELSE.

An an ELSE clause becomes paired with the nearest preceding unpaired THEN, an IF statement which contains an ELSE clause and also a nested IF statement (with no ELSE clause, necessary), creates the ambiguity of a dangling ELSE. Another method to prevent ambiguity is to insert an empty compound statement wherever a dangling ELSE clause causes confusion, such as:

```
IF expr1
  THEN
    IF expr2
      THEN
        statement1
      ELSE BEGIN END
  ELSE
    statement2;
```

In this example, the execution of statement2 depends upon expr1.

Also note that no statement separator, i.e., the semicolon, is to be used prior to the keyword, ELSE, in an IF statement (see the syntaxgraph above). In the example above, note that there is no semicolon following statement1 and no semicolon following the empty compound statement.

### 7.3.3 The CASE Statement

It is often necessary to have a decision making mechanism to perform one of many actions dependent upon some general condition, i.e., to program a logical switch with a multiplicity of choices.

The CASE statement allows us to program which one of several statements is to be executed depending on the value of some variable condition.

The syntax of the CASE statement is:

## CASE-Statement

```
--->CASE--->expression--->OF--->labeled statements--->END--->
```

where the expression, called the case-selector, is representative of the variable condition upon which the decision of which one of the labeled-statements to execute depends.

The labeled-statements is a list of several statements each of which is labeled by one or more constants, where the constants are of the same (or compatible) data type as the expression.

For each value, or group of values, that the case-selector may assume and for which a different logical path is to be taken, the programmer can label a case-labeled statement.

A CASE statement executes the statement that has a constant label matching the current run time value of the expression. The OTHERWISE clause can be used to specify that its associated statement is to be executed if the case-selector expression has a value at run time that does not match one of the labels.

Therefore, the syntax of the construct, labeled-statements, is:

## Labeled-Statements

```
  ----->  OTHERWISE  ------------->|              ---> ; -->|
  ^                                |              ^         |
  |                                V              |         V
----->enumeration constant----> : --->statement---------------->
 ^ ^                       |                    |
 | |                       |                    |
 | |<--------- , <---------V                    |
 |                                              |
 |<---------------------- ; <-------------------V
```

The format of the CASE statement is the keyword, CASE; followed by an expression that yields a value of ordinal type, (a discrete scalar enumeration type other than real or shortreal); followed by the keyword, OF; followed by the case labeled-statements; and ending with the keyword, END.

The format of the case labeled-statements is either the keyword, OTHERWISE, followed by a statement; or one or more enumeration constants (of compatible type to the case-selector) separated by commas when more than one, followed by a colon, followed by a statement.

Each case labeled-statement, when a series of actions must be performed, can be a compound statement. The case-labeled statement can also be an empty statement (see last example in Section 7.2.1 on the empty statement).

We refer to the enumeration constants construct in the graph as case labels. As noted, the case selector expression and the case labels must be of compatible types, and the case labels must be distinct, without duplication within the same CASE statement.

A sample of one of many general forms that a CASE statement may take is:

```
    CASE case-selector-expression OF

      case-label1, case-label2, case-label3 : BEGIN
                                                statement-sequence
                                              END;
      case-label4 : BEGIN
                      statement-sequence;
                    END;
      case-label5, case-label6 : statement ;
      case-label7 : ;
      case-label8 : statement;
      case-label9 : BEGIN
                      statement-sequence;
                    END;
      ...
      case-labeli : statement;
      OTHERWISE BEGIN
                  statement-sequence
                END;
    END;        {end of entire sample CASE statement form}
```

In this implementation of Pascal, there may be a maximum of 128 case labels given in any one CASE statement; where each of their ordinal values lies in the range 0..127. Integer case labels must be in the range 0..127. Also, the enumeration constant-identifiers of a user-defined enumeration type may serve as case labels (see an example below).

When a variable condition, for which a logical switch is to be programmed with a CASE statement, arises, and that variable may assume values outside the range 0..127, that variable condition must first be adjusted into a case-selector variable/expression which will only assume the ordinal values 0..127. When more than 128 values are to be handled, a nested CASE statement in an OTHERWISE clause may handle a differently adjusted case-selector to differentiate switching on another set of 128 values outside the first range 0..127. Also, the CASE statement may be used in combination with the IF statement to arrange for a variety of values to be logically differentiated.

Likewise, when the situation arises for a variable condition which may assume negative values, the variable containing negative values must first be adjusted into a case-selector which can only assume the ordinal values 0..127. Case labels cannot be negative constants.

Execution of the CASE statement begins with the evaluation of the expression, following the keyword, CASE. This expression is not usually a Boolean expression, as the IF statement provides a mechanism for dual alternative decisions. If the current run time value of the case-selector matches that of one of the labels, the statement following the matching case label is executed. If no match is made and there is no OTHERWISE clause statement to execute, a run time error occurs, with the message CASE LABEL ERROR.

Examples:

```
VAR
  MONTH : 1..12 ; NUMBER_OF_DAYS : 1..31; OVERTIME_RATE:REAL;
  YEAR  : INTEGER; DAY : (WEEKDAY,WEEKEND,HOLIDAY);  CH:CHAR;
BEGIN
  READ(MONTH);
           {set NUMBER_OF_DAYS depending on value of MONTH}
  CASE MONTH OF
    1,3,5,7,8,10,12 : NUMBER_OF_DAYS := 31;
    4,6,9,11 : NUMBER_OF_DAYS := 30;
    2 : IF (YEAR MOD 4=0) AND (YEAR<>1900)
          THEN NUMBER_OF_DAYS := 29
          ELSE NUMBER_OF_DAYS := 28
  END;
   .
   .
   .          {set OVERTIME_RATE depending of value of DAY}
  CASE DAY OF
    WEEKEND : OVERTIME_RATE := 1.7;
    HOLIDAY : OVERTIME_RATE := 2.0;
    OTHERWISE OVERTIME_RATE := 1.5
  END;
 .
 .
 .
END
```

Note that no colon follows the keyword OTHERWISE.

{Example of CASE statement used in combination with IF statement}

```
IF CH IN ['A'..'Z']
  THEN
  CASE CH OF
    'A','E','I','O','U' : VOWELCOUNT:=VOWELCOUNT+1;
    OTHERWISE CONSONANTS:=CONSONANTS+1;
    END {end of case-statement}
```

```
    ELSE
      WRITELN('CH NOT LETTER');
```

Note that only single-character character-literals or character-constants may be used as case labels in a CASE statement with a case-selector of character type; but that strings may not be used as case labels. That is, we cannot write:

```
VAR STRING:ARRAY[1..3] OF CHAR;

 ...

CASE STRING OF
'YES' : statement1;
'NO ' : statement2;
 OTHERWISE statement3
END;
```

Subrange variables may also be used as case-selectors.

### 7.3.4 The FOR Statement

The FOR Statement provides controlled repetitive execution of a statement (which may be a compound-statement containing a sequence of statements) for a predetermined programmable number of times, where the number of repetitions does not depend on the effect of the repeated statement within the for-controlled loop.

The FOR statement specifies a control-variable and an initial value and final value of a progression, which determines a collection of successive (or predecessor) values that will be attributed to that control-variable for each iteration of the loop. The number of repetitions is the number of adjacent values being delineated by the initial and final values of the progression.

The syntax of the FOR statement is:

FOR-Statement

```
---> FOR --> identifier --> := --> expression1 ----> TO ---->|
                                                 |           |
                                                 |->DOWNTO-->|
                                                             |
                                                             v
              <--- statement <--- DO <--- expression2 <---
```

where the identifier is that of a declared variable, called the FOR-Statement control-variable, which must be an enumeration type that is discrete scalar and ordered (not real nor shortreal).

The control-variable identifier must have been declared in the VAR declaration part of the block which immediately contains the FOR statement that is using it as a control-variable.

Expression1 and Expression2, as the initial value and final value, respectively, must yield values of the same or compatible type as the type of the control-variable.

The repeatable statement, which may be a compound-statement containing a sequence of statements, is to be executed repeatedly; once for each successive value of the range of values expressed by the initial value "TO" the final value in an ascending progression; or for each predecessor value of the range of values expressed by the initial value "DOWNTO" the final value in a descending progression. For each iteration of the loop, the control-variable is implicitly assigned a value of the progression, and is available for reference within the for-controlled loop.

The format of the FOR statement is then the keyword, FOR; followed by a control-variable identifier; followed by the assignment operator (:=); followed by an initial value expression beginning a progression; followed by either the keyword, TO, indicating an ascending progression, or the keyword, DOWNTO, indicating a descending progression; followed by the final value expression; followed by the keyword DO and ending with the statement that is to be repeated.

The general form of encoding a FOR statement is:

```
FOR control-variable := expression1 TO expression2 DO
    statement;
```

or

```
FOR control-variable := expression1 DOWNTO expression2 DO
    statement;
```

The statement to be repeated may be a compound-statement containing a sequence of statements, such that the general form is:

```
FOR control-variable := expression1 TO expression2 DO
  BEGIN
    statement1;
    statement2;
    ...
    statementn;
  END;
```

or

```
FOR control-variable := expression1 DOWNTO expression2 DO
  BEGIN
    statement1;
    statement2;
    ...
    statementn;
  END;
```

Note that in the above general forms, the semi-colon at the end of the repeated statement is serving as a statement separator from the next to come statement and is not actually part of the required syntax of the FOR-Statement itself, (see syntax graph above).

Execution of the FOR statement begins with evaluations of both the initial value and final value expressions. These values of the expressions are remembered internally during execution of the FOR statement, but not after it has completed execution. These values expressing a range, determine a collection of values for the control-variable. The keywords TO or DOWNTO, between the initial and final values determine a directional order among these values. In an ascending progression, if the initial value of expression1 is greater than the final value of expression2, the collection of values is empty, and therefore the statement to be repeated is not executed at all. In a descending progression, if the initial value of expression1 is less than the final value expressed by expression2, the collection of values is empty, and therefore the statement to be repeated is not executed at all. If the initial value of expression1 is equal to the final value expressed by expression2, then the collection of values of the progression contains only one value, and therefore the statement to be repeated is only executed once. The statement to be repeated is executed once for each value in the determined collection of values, taken in order, with the control-variable set to that value.

The expressions are evaluated just once prior to any execution of the repeatable statement upon entry into the FOR statement. They are not reevaluated on each cycle; as the repetitive execution control is being effected by implicitly assigning a value from the collection of values to the control-variable and executing the body of the FOR statement once for each assignment. Therefore, the number of times that the statement following the DO will be repeatedly executed is the number of adjacent values in the expressed range inclusively i.e., when we express the range 3 TO 5, or 5 DOWNTO 3, -5 TO -3, or -3 DOWNTO -5, the statement to be repeated will be executed three times, when we express the range 1 TO 2, 2 DOWNTO 1, -2 TO -1, or -1 DOWNTO -2, the statement to be repeated will be executed two times.

Completion of the entire FOR statement execution occurs when the repeatable statement has been executed with the control-variable containing the value of the final value expressed by expression2. However, no assumptions can be made as to the contents of the control-variable after execution of the FOR statement, as the control-variable becomes undefined upon termination of the FOR statement execution (other than by leaving the for-controlled loop by a GOTO-statement leading out of it).

There are several programming considerations concerning the FOR statement control-variable, in addition to the fact that it must have been declared in the block immediately containing the FOR statement using it as a control-variable. Its value is accessible within the for-controlled loop as a read-only variable for reference, but it is not available to be written into or have its value changed.

The FOR statement control-variable may not be interfered with while it is effectively controlling the repetitive execution of the statement within the FOR statement.

Therefore, the variable selected as the FOR statement control-variable has the following programming restrictions, within the FOR controlled loop. That is, within the statement (or compound-statement containing a sequence of statements) following the DO of a FOR statement, the following restrictions must be adhered to.

- A control-variable cannot be assigned a value in an assignment statement.
- A control-variable cannot be passed as a VAR variable-parameter to a procedure or function.
- A control-variable, V, cannot be read into by a READ(V), or a READLN(V).
- A control-variable cannot be re-used as another control-variable in a FOR statement nested within a FOR statement already using it;
- A procedure-call on a procedure violating any of these restrictions is not allowed; i.e., a procedure which attempts to change the control-variable value cannot be invoked; and
- A function-reference to a function violating any of these restrictions is not allowed; i.e., a function which attempts to change the control-variable value cannot be invoked.

For example, given the declarations:

```
     TYPE
          COLORS = (GREEN, RED, WHITE, BLUE, YELLOW, BLACK);

     VAR
          CH : CHAR;
          B  : BYTE;
          J  : SHORTINTEGER;
          I,VALUE  : INTEGER;
          FLAG : BOOLEAN;
```

```
     COLOR : COLORS;
```

some examples of FOR statements:

```
{To print the capitals of the alphabet vertically}

FOR CH := 'A' TO 'Z' DO
    WRITELN(CH);

{To print the small-letters of the alphabet horizontally}

FOR CH := 'a' TO 'z' DO
    WRITE(CH);

{To test the output of Boolean values in text}

FOR FLAG := FALSE TO TRUE DO
    WRITELN(FLAG);

{To print the powers of 2 representable as an integer}
{followed by the largest positive integer MAXINT}

VALUE := 1;
FOR B := 0 TO 30 DO  {integers and byte, compatibly typed}
  BEGIN
    WRITELN('2 **',B,' = ',VALUE);
    VALUE := VALUE * 2;
  END;
WRITELN('(2 ** 31) - 1 = ',MAXINT);   {note B now undefined}

{FOR statement using user-defined enumeration typed range}

FOR COLOR := RED TO BLUE DO      {Print "U.S.A." 3 times}
  WRITELN('U.S.A.');  {COLOR, due to type, cannot be output}

{A FOR statement follows using variables in expressions}

J := MAXSHORTINT;
B := 255;
FOR I := B TO  J - 32468  DO
  WRITELN(I);

{To print the printable characters of the ASCII set}
{the following FOR statement expressions call functions}

FOR CH := CHR(126) DOWNTO CHR(32) DO
    WRITELN(CH);
```

FOR statements are often used for array-element processing. For example, given the additional declarations:

```
CONST LIMIT = 100;

TYPE
     SUBSCRIPTS = 1..LIMIT;
     TABLETYPE  = ARRAY[SUBSCRIPTS] OF REAL;
```

```
          HUE = GREEN..BLACK;

     VAR
          TABLE : TABLETYPE;
          INDEX : SUBSCRIPTS;
          TABLE1, TABLE2 : ARRAY[1..2,'A'..'Z',HUE] OF REAL;
          SHADE : HUE;

{To initialize a single-dimensioned array: }

     FOR INDEX := 1 TO LIMIT DO
         TABLE[INDEX] := 0.0;

{Nested FOR statements used for multi-dimensioned arrays:}

     FOR I := 1 TO 2 DO
       FOR CH := 'A' TO 'Z' DO
         FOR SHADE := GREEN TO BLACK DO
            TABLE2[I,CH,SHADE] := 0.0;

{Another example of nested for-controlled loops: }

     VAR I,J,PROCESS_COUNT,DETAIL_COUNT : INTEGER;

     BEGIN
     PROCESS_COUNT := 0;
     DETAIL_COUNT  := 0;
     FOR I := 1 TO 100 DO
       BEGIN
         PROCESS_COUNT := PROCESS_COUNT + 1;
         FOR J := 1 TO 5 DO
             DETAIL_COUNT := DETAIL_COUNT + 1
       END
     WRITELN(PROCESS_COUNT,DETAIL_COUNT);
     END
```

Note that in the preceeding example, although I and J no longer have defined values after execution, because they were control-variables, PROCESS_COUNT = 100 and DETAIL_COUNT = 500, after execution.

### 7.3.5 The REPEAT Statement

Conditional repetitive statement execution is provided by the REPEAT statement. The REPEAT statement allows the execution of one or more statements until a termination condition is met. That is, the sequence of statements between the keywords REPEAT and UNTIL are repeatedly executed (except as modified by the GOTO-statement) until the expression is evaluated as TRUE. Different from the FOR statement, the REPEAT statement is used for repetitive execution countrol when at the time of entering the statement, it is not known exactly (or expressible) how many iterations are necessary. The statements are executed at least once, and then the condition is checked. The condition need not necessarily be defined upon entry into the REPEAT statement, as long as one or more of the statements in the statement-sequence establish the condition's value prior to the keyword UNTIL. The syntax of the REPEAT statement is:

REPEAT-Statement

```
---> REPEAT -----> statement -----> UNTIL ---> expression --->
              ^                 |
              |                 |
              |<----- ; <------v
```

where the expression must yield a Boolean value, either true or false.

The format of the REPEAT statement is the keyword REPEAT followed by one or more statements. When more than one statement follows the keyword REPEAT, each statement must be separated from its preceding statement with a statement separator, the semicolon. The last statement of the statement sequence need not necessarily be terminated with a semi-colon but if ";" preceeds the keyword UNTIL it will be treated as an empty-statement and have no effect. This statement-sequence is followed by the keyword UNTIL and an expression which produces a value of type Boolean.

Execution of the REPEAT statement proceeds as follows. The statements that follow the keyword REPEAT are executed at least once. Then the expression is evaluated. If the expression is FALSE, the statements following the keyword REPEAT are executed again. This process continues repeatedly until the expression becomes TRUE which is evaluated at the end of an iteration. The expression's value is adjustable from within those statements in the loop. When the expression is evaluated after statement-sequence execution and becomes true, program execution of the REPEAT statement ceases and continues at the next sequential statement following the REPEAT-statement.

The general form of the REPEAT statement is:

```
REPEAT
   Statement
UNTIL expression
```

or: (Notice no compound statement is necessary between "REPEAT" and "UNTIL")

```
REPEAT
   Statement1;
   Statement2;
   Statement3;
UNTIL expression
```

Example:

```
{The following reads integers, discarding negative values,  until
a positive integer is read} {assumes VAR ITEM:INTEGER; }

    REPEAT
       READ (ITEM)
    UNTIL ITEM>0 {Note that the value of ITEM was not known upon
                  entry into the REPEAT};


{The following example prints out  all  such  discarded  negative
values, and the first positive number on input that occurs}

    REPEAT
       READ (ITEM);
       WRITELN (ITEM);
    UNTIL ITEM >= 0;


    {The following example reads two integers, the first a base}
    {and the second a power;      and with a REPEAT statement, }
    {repeatedly multiples the preceding base*number by the base}
    {to achieve a powers table, UNTIL the number of times>power}
    {As this example types VALUE as INTEGER, the input base and}
    {and power is restricted such that base**power < MAXINT    }
    {A more inclusive powers program can be written using REALs}


    PROGRAM POWERS(INPUT,OUTPUT);
    VAR BASE, NUMBER, POWER, EXPO, VALUE : INTEGER;
    BEGIN
      READ(BASE,POWER);
      NUMBER := 1;
        EXPO := 1;
      REPEAT
        VALUE := BASE * NUMBER;
        WRITELN(BASE,' **',EXPO,' = ',VALUE);
        NUMBER := VALUE;
        EXPO := EXPO + 1;
        UNTIL EXPO > POWER;
    END.
```

### 7.3.6 The WHILE Statement

The WHILE statement is also a conditional repetitive statement. It will test a condition (a Boolean expression), to conditionally execute a statement, which may in turn be a compound statement. It will cease repeating execution when the condition becomes false between iterations. The condition must have a well defined value prior to entry into the WHILE statement. The condition may be adjusted from within the statement-sequence of the loop formed by a compound statement incorporated in the WHILE statement. The syntax of the WHILE statement is:

## WHILE-Statement

```
---> WHILE ---> expression ---> DO ---> statement --->
```

where the expression must yield a value of Boolean type, either true or false.

The format of the WHILE statement is then the keyword, WHILE, followed by an expression that produces a Boolean value, followed by the keyword, DO, followed by a statement.

Execution of the WHILE statement begins with the evaluation of the Boolean expression after the keyword, WHILE. If it is FALSE, then nothing more occurs, and the next sequential statement is executed. If the expression is TRUE, then the statement following the keyword, DO, is executed. After each execution of the statement, the expression is again evaluated. Execution of this statement is repeated until the expression is evaluated as FALSE. However, the expression is only evaluated between interations of the entire executing statement after DO. That is, if the statement after DO were a compound-statement whose statement-sequence adjusts the value of the condition in the expression, execution of that statement-sequence does not halt at the statement of adjustment; but rather after the entire statement-sequence has completed execution. The statement-sequence of a compound-statement in a WHILE statement is repeatedly executed (except as modified by a GOTO statement in the statement-sequence) while the Boolean expression remains TRUE.

For example, the general form of a WHILE statement is:

```
WHILE expression DO
  Statement
```

Most WHILE statements will contain a compound-statement after the DO, and be of the general form:

```
WHILE expression DO
  BEGIN
    Statement1;
    Statement2;
    ...
    Statementn;
  END
```

Given a record-type declaration (ENTRY) containing a pointer-type field (NEXT), and a declared pointer-variable (LIST):

```
TYPE LINK = ^ENTRY;
     ENTRY = RECORD
             CHARACTER : CHAR;
             NEXT : LINK;
             END;
VAR LIST : LINK;
    FOUND : BOOLEAN;  CH : CHAR;
```

A forward linked list of many dynamic records, created by NEW(LIST) calls, having had field values for CHARACTER and NEXT previously established, is easily searched for the record containing a specific value of CH by controlling the repetition of the IF statement with a WHILE statement that ceases iterations when either FOUND becomes TRUE, or the end of the list is reached when LIST = NIL.

```
CH := 'B'; FOUND := FALSE;
WHILE (NOT FOUND) AND (LIST <> NIL) DO

     IF LIST^.CHARACTER = CH
        THEN FOUND    := TRUE
        ELSE LIST := LIST^.NEXT;
```

There are some important differences between the WHILE statement and the REPEAT statement.

The statement in the WHILE statement, if it is to express more than one action, must be presented as a compound statement; and the repetitive statement is only executed after the condition of the expression is examined.

In the REPEAT statement, the sequence of the repetitive statements need not be formed into a compound statement when there is more than one. Also, the repetitive statement-sequence in the REPEAT statement is executed at least once before the terminating condition is examined.

The WHILE statement forms a loop where the condition is evaluated at the beginning of the loop; prior to ever executing it; (or prior to each interation); whereas the REPEAT statement forms a loop where the condition is evaluated at the end of loop, after each interation takes place. When the values of variables contributing to the value of the expression are indeterminable prior to the loop, use the REPEAT statement instead of the WHILE statement. The WHILE statement is used in a situation where the user has preset variables contributing to the value of the expression prior to entering the repetitive statement.

For example, the previous example on the REPEAT statement called PROGRAM POWERS can be rewritten using the WHILE statement because the values used in the controlling expression are in fact known prior to the requirement for repetition:

```
PROGRAM POWER (INPUT,OUTPUT);
VAR BASE,NUMBER,POWER,EXPO,VALUE : INTEGER;
BEGIN
  READ(BASE,POWER);
  NUMBER := 1;
  EXPO   := 1;
   WHILE EXPO < POWER DO
     BEGIN
       VALUE := BASE*NUMBER
       WRITELN(BASE, '**',EXPO,'=',VALUE);
       NUMBER := VALUE;
       EXPO := EXPO+1;
       END;
END.
```

## 7.3.7 The WITH Statement

It is often necessary to program frequent references to many fields within a variable of the record type. This occurs, for example, upon record initialization, and examination of or manipulation of the data in its fields.

Pascal provides a special statement to facilitate these record field references, so that tedious and repetitious recoding of the record variable selector as a prefix to the record field identifier is not necessary in a field reference. This special statement is the WITH statement.

The WITH statement specifies one or more record variable selectors, and a statement in which record fields of those records specified may be referenced without qualifying those references with the selectors of the record variables.

The syntax of the WITH statement is:

<u>WITH-Statement</u>

```
---> WITH ---> variable-selector ---> DO ---> statement --->
           ^                      |
           |                      |
           |<--------- , <--------V
```

where the variable-selector is that of a previously declared variable identifier of the record-type or a dynamic variable of the record-type. (See Section 5.4 on variable-selectors). The statement following "DO" may be, as is the usual case, a compound statement.

The format of the WITH statement is, then, the keyword, WITH; followed by one or more record variable-selectors, each separated from the other by a comma; followed by the keyword, DO; followed by a statement, which may contain direct field references without repeating their associated record-variable-selector as a prefix to the field references.

The general form of the WITH statement is:

```
WITH record-variable-selector DO
   Statement
```

When the WITH controlled statement is a compound statement the general form is:

```
WITH record-variable-selector DO
    BEGIN
      Statement1;
      Statement2;
      ...
      Statementn;
    END
```

The effect of a WITH statement is to enlarge the scope or visibility of certain declarations. When there is only one record variable selector in a WITH statement, the scope of the declarations of field names of that record type is expanded to include the statement controlled by the WITH statement. Any other definitions of the same names that were available at the beginning of the WITH statement are overridden during the WITH statement.

When there is more than one record-variable-selector specified in the WITH statement, the scopes of those variables are opened, and also nested, in the order in which they are listed from the left to right as indicated in the following WITH statement:

```
WITH REC1,REC2,...RECn DO statement;
```

is equivalent to:

```
WITH REC1 DO

  WITH REC2 DO
    .
    .
    .
    WITH RECn DO statement;
```

If REC1 and REC2 both have a field named FIELD, then a simple implicit reference to FIELD within the WITH controlled statement would refer to the field named FIELD in the record named REC2.

Because of the rules of nested scopes, in this case, where both records have fields of the same name, the user must explicitly reference REC1.FIELD to obtain the field named FIELD in REC1 in the WITH controlled statement.

Execution of the WITH statement begins with the evaluation of the addresses of the record-variable-selectors listed in it. For any implicit references to those variables, it is assumed that their addresses are not being changed within the WITH controlled statement. Explicit references to what are nominally the same data may occur in the WITH controlled statement. Any computation which changes the variable address affects the explicit references but not the implicit references.

The example given in Section 7.3.6 on the WHILE statement can thus be rewritten as:

```
CH := 'B'; FOUND := FALSE;
WHILE (NOT FOUND) AND (LIST <> NIL) DO

  WITH LIST^ DO      {LIST^ selects a dynamic record variable}
    IF CHARACTER = CH    {CHARACTER is field LIST^.CHARACTER}
      THEN FOUND := TRUE
      ELSE LIST := NEXT;       {NEXT is the field LIST^.NEXT}
```

Some additional examples of the WITH statement selecting declared record variables and field references are given in Section 5.3.10 on the record-type.

# CHAPTER 8
# FILE-TYPE AND I/O

## 8.1 INTRODUCTION TO PASCAL FILES

Data that is maintained or processed in a form outside of the program and memory at run-time, is called a file.

A Pascal program may communicate with its external environment by input and output (I/O) from and to files, declared as file-variables. Pascal programs may receive, display, or maintain information external to themselves. To temporarily process large volumes of data (internal file) or for that data to be retained from one program execution to another (external file), Pascal provides the ability to declare either internal or external file-variables. These file-variables are given names as Pascal identifiers, which are not the same as an OS/32 file-descriptor.

Pascal requires the user to define file-variables, their associatively required file-type, and use several standard predefined routines to perform I/O (input and output) on files.

Pascal treats files as abstractions of magnetic tapes, i.e., sequential files; and also provides a special type of file for line-structured and text formatted data, which are called text files.

A <u>Pascal file</u> is viewed as a sequence of components, all of the same component-type. Each component on the file may be any type, simple or structured, other than a file-type itself. That is, we may have a file of integers, reals, characters, arrays, or records, etc., but we may not have a file of files.

Pascal also provides a <u>text file</u> containing component characters, utilizing text buffered I/O in blocks of up to 256-characters, and implicitly structured into variable-length lines, containing formatted integers, reals, or strings; where the "text file" of characters has a special property of an end-of-line marker between lines.

A Pascal file component may be any type other than another file.

A Pascal file is considered an open-ended collection of its components with an undetermined length, i.e., the declaration of a file-type or a file-variable does not predetermine the length of the file. However, the size and structure of each component is determined by the component-type of the file. At some points in time, the file may contain no components at all; in which case the file is said to be empty. An end-of-file condition determines the end of data existing on the file, or the empty condition.

The components in a file are accessed sequentially; at any one instance, only one component is directly accessible.

Associating a file-type to a variable-identifier, declares that variable as a file-variable. Declaring a file-variable automatically introduces a Pascal entity called a file-buffer variable which is an internal variable of the same data-type as the component-type of the file. It is through this buffer, that each component of the file shall pass during input or output. If a file-variable, f, is declared; then the buffer variable is denoted, f^. The buffer variable can be viewed as a window through which the programmer can inspect (read) existing components of the file or append (write) new components to the file. The standard file handling procedures automatically obtain or send components from or to the file through this file buffer, i.e., the window is automatically moved as we access or append the next component to a file.

Each file to be used within a program unit, must be declared in the appropriate VAR declaration section of a block, with the exception of the standard predefined text files, INPUT and OUTPUT. See Section 8.4 below.

Two distinctions are made of files. They may be external files or internal files. Files may exist either temporarily as an internal file, or more permanently as an external file. Both genre of files are declared as file-variables with their file-type in the VAR declaration part of a block (except for INPUT and OUTPUT).

To differentiate an internal file from an external file, an external file name must be listed in the PROGRAM header file-name-list (see Chapter 9) and an external file must be declared in the VAR declaration part of the outermost block of the main program. An external file-variable is then global in scope as a global variable to the entire program (but external files are not visible to external modules, unless they are passed as arguments to VAR variable parameters of the modules).

Examples are given in Section 9.1.1 on the file-name-list of the PROGRAM header and Section 9.2.2 on passing files to VAR parameters of MODULEs.

An internal file may be declared in the VAR declaration part of any block, and therefore an internal file-variable declared within a module, procedure, or function VAR declaration part becomes local to that module, procedure, or function (similar to a local variable). A file-variable that is not listed in the PROGRAM header file-name-list but is declared in the outermost block VAR declaration part of the main program is an internal file and is global in scope to the program (similar to a global variable). A file-variable which is to serve as an internal file must not be listed in the PROGRAM header file-name-list.

File-variables may be passed as arguments to VAR variable parameters of modules, procedures or functions. They cannot be passed as arguments to value parameters of a routine. In fact, as modules are separately compileable units, file-variables declared in the main program, must be passed as arguments to VAR variable parameters of modules, in order for those file-variables to be referenceable in the module.

The logical unit to which a Pascal file-variable name is associated is determined by the compiler.

External files are assumed to exist prior to and after the execution of the program referencing them. The association of external files to logical units is governed by the position of the external file name in the PROGRAM header file-name-list, i.e., the first file is associated to logical unit 0, the second file to logical unit 1, etc., and with the maximum number of file names allowed in the file-name-list being 32, the last maximum file would be associated with logical unit 31 (see Chapter 9). At run time, the user must assign logical units that have been associated with Pascal external file-names to actual files prior to starting the execution of the user program task (see Chapter 1). The associations of Pascal external file-names to logical unit numbers made by the compiler are listed in the compiled-program listing.

Internal files do not exist prior to nor after execution of a program. Internal files have their contents created by the programmer, but are allocated and assigned as OS/32 temporary files by compiler-generated code in the user program when the block in which they are declared becomes activated. Likewise, they are destroyed (deassigned and deallocated) when the block becomes inactive. As internal files are treated as any other variable local to a given routine, "n" recursive calls on a routine which declares an internal file will cause the attempt to create "n" internal files. The compiler assigns an internal file to any logical unit available at the time the routine containing the declaration of the internal file becomes activated. If no free logical unit is obtainable in the established user task, a run time error occurs.

## 8.2 The FILE-TYPE

A Pascal file-type is a structured data-type which is considered a sequence of possible components, whose data type is the component-type of the file. The component-type may or may not be itself structured, but it may not be the file-type.

The syntax of the construct, file-type, is:

## File-Type

```
--->FILE-->OF--->component-type--->
 |                               ^
 |                               |
 v--->file-type-identifier----->|
```

To establish a file-type, we first ascertain that either a Pascal predefined type-identifier or a previously defined component-type type-identifier is available to "type" or describe the components of the file. For example, a component-type type-identifier may be defined by:

```
    TYPE  type-identifier = component-type;
```

where the identifier on the left of the equal sign is the introduction of the name of the file's component-type, and the type on the right side of the equal sign is any type other than a file-type. For example an array-type and two component-types are defined below to be used in future examples:

```
    TYPE   CHARS = ARRAY[1..20] OF CHAR;
            {establish a record-type type-identifier}
           CLIENT =  RECORD
                        NAME:CHARS;
                        ADDR:CHARS;
                        ORDERNO:INTEGER
                     END;
            {establish an array-type type-identifier}
           ORDER  = ARRAY [1..10] OF INTEGER;
```

The introduction of a definition of a file-type type-identifier, still within the TYPE definition part is expressed as:

```
    TYPE  file-type-identifier = file-type;
```

where the identifier introduced on the left of the equal sign becomes the name of the file-type being defined; and the file-type is of the form "FILE OF component-type."

Once the component-types are available, the file-types: CLIENTFILE, ORDERFILE, and INITFILE may be defined as follows:

```
    TYPE
         CHARS  = ARRAY[1..20] OF CHAR;
         CLIENT = RECORD
                     NAME:CHARS;
```

```
                    ADDR:CHAR;
                    ORDERNO:INTEGER
                  END;
         ORDER    = ARRAY [1..10] OF INTEGER;
         CLIENTFILE = FILE OF CLIENT;
         ORDERFILE = FILE OF ORDER;
         INITFILE = FILE OF INTEGER;
```

Note that INITFILE is using a standard predefined type-identifier, INTEGER, as a component-type.

A file must be declared in the same manner as a variable.

The name of the external file listed in the file-name-list of the PROGRAM header must be re-introduced in the outermost VAR declarations of the program block and typed with a file-type. The name of an internal file is introduced with a new identifier, and that identifier must be associated with a file-type within a VAR variable declaration, in any block. For example:

```
    VAR file-identifier : file-type;
```

where the file-identifier prior to the colon is the name of the file, and the file-type is either a previously defined file-type-identifier or of the form: "FILE OF component-type". The file-identifier then becomes a file-variable.

Every identifier that is declared to be a variable of the file-type, if it is an identifier of an external file, must also have that identifier listed in the file-name-list of the program header statement; whereas internal files need not have their names included there.

For example, using the file-type-identifiers defined above, CLIENTFILE and ORDERFILE, the following file-variables can be declared:

```
    VAR OLDCLIENTS,NEWCLIENTS : CLIENTFILE;

        OLDORDERS,NEWORDERS : ORDERFILE;
```

Files to be passed to routines or external modules must use the above method to achieve identity of type between the argument file and the file-variable-parameter.

If the file-types CLIENTFILE or ORDERFILE had not been previously defined, the file-variables can be declared if the component-type definitions ORDER and CLIENT are available. This would be accomplished then by the following examples:

```
    VAR OLDCLIENTS,NEWCLIENTS : FILE OF CLIENT;
```

```
        OLDORDERS,NEWORDERS : FILE OF ORDER;
```

Files which are not going to be passed to variable-parameters may use the above method, as file-variables have no other operations applicable to file-variables; other than specifying their name as arguments to the Pascal I/O routines.

If the component-types CLIENT or ORDER had not been available, it is also possible to declare the file-variable thusly:

```
    VAR OLDCLIENTS,NEWCLIENTS : FILE OF RECORD
                                         NAME:CHARS;
                                         ADDR:CHARS;
                                         ORDERNO:INTEGER;
                                        END;
    OLDORDERS,NEWORDERS : FILE OF ARRAY[1..10] OF INTEGER;
```

but it is usually preferable to define component-types separately, making component-type type-identifiers available to define the types of separately declared variables while also assuring Pascal type-compatibility between them and the file's components.

Such file-variable declarations produce a special kind of variable called a file-buffer variable, which is the "identical" type as that of the file's component-type listed in the file-type definition. This file-buffer serves as storage to hold each of the file's components as they are transmitted between the file and the program. At any one moment, only one component at a time occupies and is available in this file-buffer. It is, therefore, the means of accessing the file's component and is referenceable by means of the construct, file-buffer selector. The syntax for selecting the file-buffer variable is:

<u>File</u>-<u>Buffer</u> (<u>Selector</u>)

```
---> file-identifier ---> ^ --->
```

where the file-buffer selector consists of the file-variable identifier followed by an up arrow. That is, if a file-variable, f, is declared; the file-buffer variable is referencable as f^.

For example, in order to access the components of the previous file-variable examples (using RESET statements first to prepare the files for reading and which will, by definition, fetch the first component [if it exists on the file] into the file-buffer variable), we actually access the value in the file-buffer variable by the references specified by a file-buffer selector as shown in the assignment-statements below the RESETs:

```
{file-variables only used in Pascal I/O routine calls}
{or passed to variable-parameters of routines}

RESET (OLDCLIENTS);
RESET (NEWCLIENTS);
RESET (OLDORDERS);
RESET (NEWORDERS);

{file-buffer variables accessible for assignment }

VARIABLE1 := OLDCLIENTS^;

VARIABLE2 := NEWCLIENTS^;

VARIABLE3 := OLDORDERS^;

VARIABLE4 := NEWORDERS^;
```

The file-buffer selectors, OLDCLIENTS^, NEWCLIENTS^, OLDORDERS^, and NEWORDERS^, specify the file-buffer variables containing components from their respective files.

Files are acted upon from within the program only by procedures. There are no arithmetic, relational, set, or Boolean operators or assignments that can be performed on the file-variables denoting files. However, the file-buffer variable can be operated on or with just as any separately declared variable of its type, the component-type of the file. The file-buffer variable is accessible with a file-buffer selector, but the file-buffer contents become defined on input with calls on Pascal input routines, and are to be programmed with contents either with assignment prior to a PUT call, or is utilized to transfer data specified in calls on other Pascal output routines.

In summary, an entire Pascal file can be created and identified by defining a component-type, defining a file-type (and thereby associating it with the component-type), declaring a file-variable name associated with a file-type (and thereby the component-type), and then appending new values through the file-buffer reference to the empty file. Given a previously written file, components are read in the same order as they were written. For this reason, Pascal files are treated as sequential files, and their components can only be sequentially accessed in the order in which they are written.

## 8.3 INPUT/OUTPUT ROUTINES

Input and output from and to Pascal named files, (both Pascal non-text files and text files) is accomplished by means of the following predefined procedures: GET, PUT, RESET, REWRITE, READ, and WRITE; and the function EOF. The procedures PAGE, READLN and WRITELN and the function EOLN are additionally and especially provided for text files.

These procedures provide the detailed interaction necessary to communicate through the operating system, via the logical units associated to Pascal named files, to actual devices or disc files assigned to those logical units.

The seven predefined routines available for both Pascal files and text-files are defined in the list directly below. The four additional text file routines are given in the following Section 8.4.

## EOF

If f is a file-variable, the function EOF(f) returns the Boolean value of true if the file f is positioned at its end-of-file, otherwise, the Boolean value of false is returned. When the implicit form of EOF is referenced, without specifying a file-variable f, the end-of-file condition on the textfile INPUT is queried.

## RESET

If f is a file-variable, RESET(f) positions the file to its beginning for the purpose of reading. RESET prepares for queries on EOF(f). EOF(f) becomes false and the file is in a read-only state unless the file is empty in which case EOF(f) becomes true. If the file is not empty, the file-buffer variable f^ is assigned the value of the first component of the file f; otherwise f^ is not defined. An automatic unprogrammable reset is performed on textfile INPUT, whenever INPUT is listed in the user's PROGRAM header with similar preparations for EOF and INPUT^. Therefore, after a RESET, the programmer should usually query EOF or EOF(f) prior to reading a previously written textfile, INPUT or f, respectively, to make sure the file is not empty, or at "end-of-file".

## REWRITE

If f is a file-variable, REWRITE(f) positions the file to its beginning for the purpose of writing and the file is in a write-only state. EOF(f) becomes true.

## GET

If f is a file-variable, GET(f) advances the file to the next component of f, i.e., the file-buffer variable f^ is assigned the value of this component. If no next component exists, EOF(f) becomes true and the value of f^ is undefined. That is, the resultant f^ after a GET(f) is defined only if EOF(f) is false prior to the execution of GET(f). If EOF(f) is true when GET(f) is attempted, a run-time error occurs. A file to which GET(f) is applied must be RESET prior to the first execution of GET(f) to prepare that file for reading. If the file has not been reset, a run-time error occurs at the time GET(f) is attempted.

## PUT

If f is a file-variable, PUT(f) appends the value of the buffer variable f^ to the file f. The file must be prepared for writing at first by a call to the standard procedure REWRITE prior to the first execution of PUT(f). If the file has not been rewritten, a run-time error occurs at the time the PUT(f) is attempted. If the file f was not empty at the time it was rewritten, calls to PUT(f) will replace the values of the file components previously in existance. If the file f is at its end-of-file, i.e., EOF(f) is TRUE, then PUT(f) appends the contents of the buffer variable f^ to the file.

## READ

If v is a variable of type T and f is a file whose type is f_type = FILE OF T, then READ(f, v) will assign to v the contents of the current file component of f, i.e., f^; and advance the file to the next component. That is, READ(f,v) is executed as if it were an invocation of this procedure:

```
PROCEDURE READ_T(VAR ff: f_type; VAR xx: t);
  BEGIN xx := ff^; get(ff); END;
```

READ is defined in this manner, to detail the fact that the variable v is passed to a _variable-parameter_, xx, to which the file-buffer variable is assigned, thereby requiring Pascal "identity" of type between the component-type of a _non-textfile_ and the variable being read into. For example, differing from the WRITE routine which allows "assignment-compatibility" between the expression written out and the component-type of the _non-textfile_ f; the READ routine reading in from a FILE OF INTEGER can only read in an INTEGER, not into a variable of type BYTE, nor SHORTINTEGER. See WRITE routine below.

Textfiles, on the other hand, are treated differently, allowing any type of integer variable, either BYTE, SHORTINTEGER, or INTEGER; or subrange thereof, to cause the reading in of a character-coded decimal literal-integer from a textfile. Textfile I/O calls on READ only require "assignment-compatibility" of type respective to whether characters, formatted integers, or formatted reals are to be read in.

## WRITE

If f is a file-variable and x is an expression of the file component type, WRITE (f,x) will assign the value of x to buffer variable f^ and append (or replace) the file component with the value of f^. That is,

```
WRITE (f,x)
```

is equivalent to

```
BEGIN f^ := x; PUT(f) END
```

WRITE is defined in this manner, to detail the fact that the expression value x, (even if it is only a variable), need only be "assignment-compatible" to the component-type of the non-textfile f; as the WRITF routine treats expression x as a value-parameter and assigns it to f^, thereby the requirement for "assignment-compatibility" of type between expression x and the component-type of non-textfile f. For example, differing from the arguments to a READ call, which require "identity" of type to the file's component-type; calling the WRITE routine to output to a FILE OF INTEGER, would allow the expression x to additionally be of assignment-compatible type to the file's component-type, thereby allowing expressions of type BYTE, SHORTINTEGER, or INTEGER with the smaller integers being elongated in the output stream as a full four-byte integer.

Textfiles are treated differently, in that any expression of BYTE, SHORTINTEGER, or INTEGER , or subranges thereof, would be output as a character-coded formatted decimal literal-integer. Textfile I/O calls on WRITE require of its expression arguments an "assignment-compatibility" of type respective to whether a character, string, Boolean value, formatted integer, or formatted real, is to appear on the textfile.

Additionally, the predefined procedures READ and WRITE provide the flexibility of performing multiple read and write sequences in one call. The forms for using READ and WRITE for reading in multiple variables or writing out multiple expressions follow. The rules regarding type-compatibility are stated here as those required for non-textfiles. I/O for textfiles is described in the following Section 8.4.

For non-textfiles, the procedure call READ(f,v1,...vn), where v1...vn are variables of "identical" type as the component-type of the file-variable, f, is the same as:

```
BEGIN
  READ(f,v1);
  .
  .
  READ(f,vn)
END;
```

Likewise, for non-textfiles, the procedure call

WRITE(f,e1,...,en), where e1...en are expressions of "assignment-compatible" type to the component-type of the file-variable, f, is the same as:

```
     BEGIN
       WRITE(f,e1);
       .
       .
       WRITE(f,en)
     END;
```

Although transparent in the user-level Pascal coding of a READ or WRITE procedure-call statement, the data being transferred between the file and either the variables or the expressions is actually being processed through the file's associated file-buffer variable, f^. That is, we READ(f,v) or WRITE(f,e), always specifying the non-textfile f as the first argument to READ or WRITE, and follow the file-identifier selector f with the variable(s) or expression(s), respectively. There are no implicit forms for optionally dropping f, for READs or WRITEs to non-textfiles, as there are for textfiles.

## 8.4 TEXT FILES

Pascal provides a special file-type for files whose components are characters containing character-formatted integers, reals, or (on output) strings of data within lines of text; to additionally provide efficiencies in displaying/processing text information. This standard Pascal file-type is denoted by the predefined type-identifier, TEXT.

The type TEXT is equivalent to a FILE OF CHAR with a special property. This attribute is the addition of a line marker indicator and the buffering of character text information of up to 256 characters. Therefore, conceptually TEXT files are strings of characters interspersed with line markers and thereby organized into lines of arbitrary length. In practice, the implementation restricts the maximum length of any string between two line markers to 256 characters.

Variables that are attributed to be of the file-type TEXT are file-variables; and these file-variables are referred to as text files. Although the physical text file I/O may occur in line lengths of up to 256 characters, the text file file-buffer variable f^ performs as a window into the buffered text one character at a time. Each component, f^, of a text file, f, is of CHAR type.

A text file has its text subdivided into a sequence of lines. Each line contains zero or more values formatted in type CHAR, and each line is separated from the other by a line marker represented by an internal line control character (the space character).

This is why a call on GET(f) or PUT(f), when f is a textfile, gets or puts one character at a time, through f^ into or from the text file line-buffering scheme. The READ/WRITE routines, within lines, can format characters/strings(on output) and convert numerics into character-formatted entities from and to text files. Therefore, the READ/WRITE routines are defined in terms of equivalences using f^ and GET/PUT character interactions with the text file. The READLN/WRITELN routines also accomplish what READ and WRITE do, and additionally handle and direct vertical line control.

The actual demarkation between logical records, breaks between cards, or carriage-return character on a terminal all serve to end a line, externally; and this end-of-line condition between lines is internally recorded on a text file to Pascal code by EOLN or EOLN(f) going true and the appendage of the internal line control character (a space) after the text of the line.

The actual external end-of-line characters are seemingly transparent to the incoming data stream as some formatted data are read from a TEXT file. Reading formatted integers or reals from a textfile with READ or READLN skip leading spaces and line markers occuring prior to the numeric. However, a single character read attempt on a TEXT file when the text file-buffer variable window is positioned at a line marker returns the character space. That is, when programming text file I/O, a character at a time, the space character serving as a line marker is viewable and must be programmed for as desired. Recognition of when line markers are occuring is supplied to the running program through the use of a standard function EOLN. EOLN(f) returns the value true and the file-buffer variable is a space character when the current character available from a text file is a line control character. As other spaces can occur in text file input, the line marker space character is that which occurs when EOLN becomes true. Otherwise, EOLN(f) is false. When a text file is empty, EOF(f) is true.

Note: the maintenance and recognition of line markers is provided by the language only for files declared to be of type TEXT. Therefore a file f declared

```
        VAR    f:  FILE OF ARRAY [1..256] OF CHAR;
```

is <u>not</u> equivalent to

```
                    VAR    f:  TEXT;
```

The standard routines EOF, RESET, REWRITE, GET, PUT, READ and WRITE (described above in Section 8.3) all also apply to TEXT files. One additional predefined routine, PAGE, provides page ejection on text files. Three additional predefined routines are defined by the language to deal with the line markers. They are EOLN, READLN, and WRITELN as summarized below:

## EOLN

If f is a text file, EOLN(f) returns the Boolean value true if the file-buffer variable f^ corresponds to the position of a line marker and false otherwise.

## READLN

If f is a text file, READLN(f) causes the window to be advanced to the beginning of the next line of the text file; bypassing any end-of-line line-marker of a previous line; i.e., the file-buffer variable f^ becomes the first character of that line, after effectively skipping a line on the file. See Section 8.6 for the thorough details on the various forms and argument-lists that can be specified for reading from text files with READLN.

## WRITELN

If f is a text file, WRITELN(f) terminates the current line of f by generation of a line marker; which will cause any previous WRITEs having had their text buffered without any line-markers to appear on output on the line terminated by WRITELN. See Section 8.8 for thorough details on the various forms and argument-lists that can be specified for writing to text files with WRITELN.

## PAGE

Pascal offers the procedure PAGE to provide page ejection forms control during the writing to text files. Its predefined identifier is PAGE. The procedure call PAGE or PAGE(f), where f is a text file-variable, causes the OUTPUT file or the text file f, respectively, to advance to top-of-form.

Calls on these four predefined routines EOLN, READLN, WRITELN, and PAGE are valid only with files of type TEXT.

When used with TEXT files, the standard procedures READ, READLN, WRITE, and WRITELN may accept arbitrary numbers of arguments of certain various types, whereas when the READ and WRITE procedures are used with non-textfiles, the READ and WRITE routines may only accept arbitrary numbers of arguments of the same "identical" type to the "component-type" of the non-textfile.

Two standard file-variables are predefined with the identifiers INPUT and OUTPUT so that when present in a program header statement, they have the predeclared attribute of being text files. That is, the user can assume that the following implicit variable declaration is available:

```
    VAR INPUT,OUTPUT :  TEXT;
```

Listing either INPUT or OUTPUT or both as external files in the file-name-list of the program header causes the following actions to be taken by the compiler:

- an implicit declaration of the subject file as a variable of type TEXT is made.
- if the standard file INPUT is listed in the program header, code is generated to RESET that file prior to execution of any statements in the main body of the program.
- if the standard file OUTPUT is listed in the program header, code is generated to REWRITE that file prior to execution of any statements in the main body of the program.

When the standard file INPUT is listed in the program header, the standard routines EOF, EOLN, READ, and READLN may be referenced without a file-variable as an argument. The standard file INPUT is implicitly referenced in such calls.

When the standard file OUTPUT is listed in the program header, the standard procedures PAGE, WRITE and WRITELN may be referenced without a file-variable as an argument. The standard file OUTPUT is implicitly referenced in such calls.

Unless changed by a redeclaration of their identifiers for some other purpose and/or coupled with their extraction from use in a PROGRAM header file-name-list, the identifiers INPUT and OUTPUT are normally used as text files. Certain file handling procedure call argument-lists that do not contain a file-variable imply that either the standard text file INPUT is meant for read operations (READ or READLN, respectively); or that the standard text file OUTPUT is meant for write operations (WRITE or WRITELN, respectively). Also both functions, EOF and EOLN, may be called without any argument to obtain the EOF or EOLN status on the text file INPUT.

When the identifier INPUT is present in a PROGRAM header file-name-list, the standard text file INPUT is automatically reset by the compile-generated code at the beginning of the main program. This automatic reset, upon starting the user program task, causes the first component to be requested from the logical unit associated to file INPUT through the operating system. When the identifier OUTPUT is present in a program header file-name-list, the standard text file OUTPUT is automatically rewritten (given a REWRITE command) by compiler-generated code in the user program. This implicit rewrite, upon starting the user program task, to the file OUTPUT occurs prior to execution of any statements in the main body of the program.

In interactive applications, it is recommended that the standard textfile INPUT not be used as an interactive device, e.g. with the file OUTPUT. Doing so may cause a misleading sequence of input and output requests the interactive device assigned to the logical unit associated with file INPUT. Using any other text file name an an interactive device avoids this problem by allowing the user to control the reset operation. For an example of textfile interactive I/O addressing this problem inherent in the nature of Pascal definitions, see the last example of this chapter, in Section 8.9 on Data Transfer Examples.

The standard procedures READLN and WRITELN are specifically provided for text files and can only be applied to text files; however, the standard procedures READ and WRITE, apply both to files and text files. PAGE only applies to text files.

## 8.5 READING FILES USING RESET AND GET

In order to begin reading a file (other than text file INPUT), the standard procedure RESET is called. The effect of the procedure call RESET(f), where f is a file-variable, depends on whether or not the file is empty. If the file is empty, the internal file-buffer is undefined and the Boolean function EOF(f) becomes true. This is a condition which may be tested by the programmer and is why it is best to test EOF after calling RESET and before calling GET, READ, or READLN. If the file is not empty the internal file-buffer receives, or is assigned to, the first file-component available to the programmer for processing; and the Boolean function EOF(f) becomes false. Then to obtain the next and all subsequent components on the file the procedure call GET(f) may be used. For example, to read and process the components on file-variable, f:

```
BEGIN
  RESET(f); {Note: this assigns the first component of f^}
  WHILE NOT EOF(f) DO
  BEGIN
    .
    . (*process f^*)
    VARIABLE:=f^;
    GET(f)
  END;
END
```

Therefore, the definition of the standard procedure RESET requires one argument, as in RESET(f), where f is the identifier of a variable which is of the file-type. The procedure call RESET(f) where f is a file-variable, resets the current position of the file to the beginning of the file and assigns the value of the first component of the file to the file-buffer variable, f^. The Boolean function EOF(f) yields false if the file, f, is not empty. If the file, f, is empty the file-buffer variable, f^, becomes undefined and EOF(f) becomes true.

- RESET(f) is necessary for initialization before reading from the file, f. [The textfile INPUT is automatically RESET.]

- Once RESET(f) has been applied to the file, f, it is considered to be in a "read-only" state; i.e., GET(f) may be used on f but not PUT(f). READ may be used on f but not WRITE, and if f is a textfile, READLN may be used but not WRITELN.

The effect of the procedure call GET(f) is to assign the value of the next available component on the file to the file-buffer variable, f^. If the component exists on the file, the Boolean function EOF(f) remains false, and the component value is assigned to the file-buffer variable. If the component does not exist on the file, the Boolean function EOF(f) becomes true (a condition that can be tested by the programmer) and the file-buffer variable becomes undefined. Refer to the example above under the RESET procedure.

Therefore, the definition of the standard procedure GET requires one argument, as in GET(f), where f is an identifier of a variable which is of the file-type. The procedure call, GET(f), operates as follows. If the Boolean function EOF(f) is false prior to the execution of the procedure call GET(f), the current position of the file, f, is advanced. If a "next-component" exists on the file, f, the component's value is assigned to the file-buffer variable, f^, and the Boolean function EOF(f) remains false. If no "next component" exists, then the Boolean function EOF(f) becomes true, and the file-buffer variable becomes undefined. It is considered an error if the Boolean function EOF(f) is not false prior to an attempt to read the file with the execution of a GET(f).

## 8.6 READING FILES USING READ AND READLN

The standard procedure READ is called to input data from files. The file must have been given an initial RESET, (other than textfile INPUT), and the EOF of the file should be false. The syntax of the procedure call on the standard procedure READ is:

<u>READ</u>

```
---> READ ---> parameter-list --->
```

where the syntax of the parameter-list defined below requires a different argument-list in the call, depending on whether READ is being applied to files other than text files or solely to text files. The READ statement must always specify the file-variable as the first argument in its parameter-list for non-textfiles; but for text files, it has two forms: to imply the textfile INPUT when the first argument is not a file-variable or to specify a textfile by using the file-variable as the first argument.

## Reading from Pascal files which are not textfiles:

The syntax of the READ parameter-list, when applied to other than text files is:

### READ-Parameter-List (for non-textfiles)

```
---> ( --->file-variable---> , --->variable-selector---> ) --->
                                |                    |
                                <-------- , <--------V
```

In this case, the format of the call on READ, requires an argument-list corresponding to the above parameter-list. This list contains first, the file-variable identifier of the file to read from, followed by a comma, followed by one or more variable-selectors, each separated from the other by a comma. The data read from the file will be read directly into the variables specified by the variable-selector(s), thereby the requirement for "identity" of type to the file's component-type.

As noted earlier, the READ procedure call statement, depending on their various forms is defined as follows; where f represents a file, other than a text file, and v represents a variable; where each variable is of the "identical" type as the component-type of the file f.

```
    READ(f,v)             is equivalent to:   BEGIN
                                                v:=f^;
                                                GET(f)
                                              END

    and

    READ(f,v1,...,vn)     is equivalent to:   BEGIN
                                                READ(f,v1);
                                                .
                                                .
                                                .
                                                READ(f,vn)
                                              END
```

READLN is not used to read from non-textfiles.

## Reading from Pascal textfiles with READ:

The syntax of the READ parameter-list when applied to text files is different in that the file-variable, whose presence is optional, defaults to the standard file INPUT when a file is not specified in the list. Also, the variable-selectors may denote variables of various certain types. When f is a text file the variables may be of the character-type (or a subrange-variable of host character-type), the integer-types (or a subrange-variable of host integer-type), or the real types, SHORTREAL or REAL. The syntax of the READ parameter-list when applied to text files is:

READ-Parameter-List (for text files)

```
--> ( ------------------------------->variable-selector---> ) -->
      |                            ^  |                    |
      |                            |  |                    |
      V-->file-variable--> , ->|  <-------- , <--------V
```

In this case, the format of the call on READ requires an argument-list corresponding to the above defined parameter-list. That is, the argument-list of READ contains first, although optional, a file-variable identifier of the text file to be read from, and, if present, is followed by a comma. Then this is followed by one or more variable-selectors, each separated from the other by a comma, and to which the data read in will be assigned. The entire list of arguments is enclosed in parentheses.

The effect of the execution of the READ procedure call statement, depending on their various forms and the data types of the variables, is as follows. Where f stands for the text file, and v stands for the variable(s); and the variables are of "assignment-compatibility" respective to the formatted integers, reals, or characters on the textfile:

```
    READ(f,v1,...,vn)      is equivalent to:   BEGIN
                                                 READ(f,v1);
                                                 .
                                                 .
                                                 .
                                                 READ(f,vn)
                                               END
```

and

```
    READ(v1,...,vn)      is equivalent to:   BEGIN
                                                 READ(INPUT,v1);
                                                 .
                                                 .
                                                 .
                                                 READ(INPUT,vn)
                                               END
```

and the variables need only be of any integer-type, any real-type, or character-type respective to whether literal integers, literal reals, or characters are being read in from the textfile.

### Reading characters from textfiles:

For textfiles, ONLY if v is of character-type CHAR, (or a subrange-variable of host character-type), are the above textfile READ(f,v) or READ(v) enacted by the following equivalences:

```
    READ(f,v)                is equivalent to:   BEGIN
                                                   v:=f^;
                                                   GET(f)
                                                 END
```

and

```
    READ(v)                  is equivalent to:   BEGIN
                                                   v:=INPUT^;
                                                   GET(INPUT)
                                                 END
```

The above definitions mean, each character on the textfile is read one at a time, and in inputting character-data from textfiles, no leading spaces or EOLN markers are skipped by the READ routine as it will do when reading formatted integers or reals on a textfile. While reading from Pascal textfiles a character at a time through a character-variable, every character occuring on the textfile is actually input, even a character for the EOLN marker (which although occuring on the textfile as a carriage-return is converted into a space character). This means, if the user did not program around the EOLN character in a loop reading characters from a textfile, and simply echoed a WRITE/WRITELN of the characters read in, and output them to a textfile, a space character would appear in that output wherever a carriage-return or other end-of-line was encountered in the input stream.

READLN can also read characters from a textfile and is detailed below after READ.

### Reading formatted integers from textfiles:

If v specifies a variable of integer-type (or a subrange-variable of host integer-type), a textfile READ(f,v) or implicit READ(v) [implying from textfile INPUT], causes the sequence of characters that form an optionally signed integer to be read from the text file, f or INPUT, respectively. The conversion of this external representation to an internal integer value takes place; and the integer value is assigned to the variable, v. Preceding spaces and end-of-line indicators are skipped. Reading stops when either the end-of-file is reached or when the file-buffer variable, f^, contains a character that does not form part of an optionally signed integer. A run time error occurs if the sequence of characters does not form an optionally signed integer.

READLN can also read formatted integers from textfiles and is detailed below after READ.

### Reading formatted real numbers from textfiles:

If v specifies a variable of REAL or SHORTREAL type, READ(f,v) or READ(v) [implying textfile INPUT], causes the sequence of characters that form an optionally signed real number to be read from the text file, f or INPUT, respectively. The conversion of this external representation to an internal real number takes place, and the real number is assigned to the variable, v. Preceding spaces and end-of-line indicators are skipped. Reading ceases as soon as end-of-file is reached or the file-buffer variable, f^, contains a character that does not form part of an optionally signed real number. A run time error occurs if the sequence of characters does not form an optionally signed real number.

READLN can also read formatted real numbers from textfiles and is detailed below.

### Differences between READ/WRITE and READLN/WRITELN:

When applied to text files, the READ and WRITE procedures usually concern the input or output of data across a single line. Because text files are structured into multiple lines with an end-of-line (EOLN) line marker indicator separating the lines, Pascal provides the two standard procedures, READLN and WRITELN, for the specific purpose of handling the line marker on text files. On input, the EOLN condition requires the bypassing of the EOLN indicator. On output, an EOLN condition in the output data stream requires a mechanism to effect skipping to the next line of text. READLN and WRITELN perform these important functions. READLN and WRITELN, therefore, must only be applied to text files. READ also skips EOLNs prior to reading integers or reals, but not when reading single characters.

On input, READLN, with an appropriate argument-list, will not only read the information on the line as the READ procedure does but will also advance the text file to the next line, making the first character of the next line the next available character to be read, or skipped.

On output it must be noted, that for text files, data output by the WRITE procedures is actually buffered by the Pascal I/O scheme, and only the use of WRITELN, aside from a buffer-full condition, triggers the actual physical transfer of data to the text file. In effect, WRITELN causes the output by activating an SVC, (Supervisor Call) whereas WRITE does not.

Therefore, the user controls text file input or output and may specify, by judicious use of READ and READLN, and WRITE and WRITELN, the exact format of text transfer, both horizontally within a single line and vertically with multiple lines a particular data stream, output to a textfile.

A description of READLN immediately follows and the description of WRITELN is presented following the procedure WRITE.

### Reading textfiles with READLN

The syntax of the procedure call on the standard procedure READLN, which may only be applied to text files, is:

### READLN

```
--->READLN--------------------------->
          |                       ^
          |                       |
          V--->  parameter-list--->
```

The format of the procedure call on READLN, is the procedure name, READLN, optionally followed by a argument-list which corresponds to a READLN parameter-list. The effect of the execution of a READLN call, when no argument-list is present, is to scan over the remainder of a line of the input stream on the default file-variable, INPUT, bypassing the EOLN character and leaving the next available character for input at the first character position on the next line. A READLN call with no argument-list effectively provides a skip to next line on the text file INPUT. The syntax of the READLN parameter-list, which may only be applied to text files, is:

### READLN-Parameter-List

```
                       --------------------------------->|
                       ^                                 |
                       |                                 v
--> (--->file-variable---> , ------>variable-selector-----> ) -->
      |                       ^  ^                     |
      |                       |  |                     |
      v----------------------->  <--------- , <-------v
```

In this case, the format of the call on READLN requires an argument-list corresponding to the above defined parameter-list. That is, the argument-list of READLN, contains first, although optional, a file-variable identifier of the text file to be read from, and if present, followed by a comma (whenever one or more variable-selectors follow). Then this is optionally followed by one or more variable selectors, each separated from the other by a comma, and to which the data will be assigned. The entire list of arguments is enclosed in parentheses. When the file-variables is omitted, the default text file, INPUT, is assumed. When the file-variable is present in a READLN argument-list, but no variables are present, the effect of READLN is to skip a line on text file, f. The effect of the execution of the READLN procedure call statements, depending on their various forms and the data types of the variables is as follows; where f stands for the text file, and v stands for the variable(s).

READLN is equivalent to:

```
BEGIN
 WHILE NOT EOLN(INPUT)
  DO
  GET(INPUT);{scan line}
GET(INPUT) {bypass EOLN}
END
```

READLN(f) is equivalent to:

```
BEGIN
  WHILE NOT EOLN(f) DO
    GET(f);
  GET(f) {bypass EOLN}
END
```

READLN(f,v) is equivalent to:

```
BEGIN
  READ(f,v);
  READLN(f)
END
```

READLN(f,v1,...vn) is equivalent to:

```
BEGIN
  READ(f,v1,...,vn);
  READLN(f)
END
```

READLN(v) is equivalent to:

```
BEGIN
  READ(INPUT,v);
  READLN
END
```

READLN(v1,...,vn) is equivalent to:

```
BEGIN
  READ(INPUT,v1,...,vn);
  READLN
END
```

In the above equivalence definitions, the actual data read in for the variable(s) vn depends upon the data types of the variable(s); as was described in detail in the section above; on the READ routine applied to textfiles. That is, these variables need only be "assignment compatible" to the <u>respective</u> formatted integers, reals, or characters on the textfile being read in. For example, if v is of character-type, or subrange-variable of host character-type, one character on the textfile may be read in; if v is of any integer-type, BYTE, SHORTINTEGER, or INTEGER; or subrange-variable of host integer-type, an optionally signed integer on the textfile is read; if v is of any real-type, SHORTREAL or REAL; then an optionally signed real number on the textfile is read.

## 8.7 WRITING FILES USING REWRITE AND PUT

In order to construct or write a file, the standard procedure REWRITE is used to position a file at its beginning component and put the file into a "write-only" state. The effect of the procedure call REWRITE(f), where f is a file-variable, is to rewind the file without actually erasing any previous contents of the file, f, externally. REWRITE(f) effectively makes file, f, appear empty having zero components. The Boolean function EOF(f) becomes true indicating the position is at end-of-file. The file-buffer variable, f^, associated with the file f, becomes undefined. The next component written to the file by means of another standard procedure, PUT, would place that component in the first position on the file. All subsequent components written to the file by means of PUT would be placed at the end of the file in sequential order. For example, to prepare a file for writing, the user writes:

```
    REWRITE(f);
```

and EOF(f) remains true while the file is used for output.

Therefore, the definition of the standard procedure REWRITE requires one argument, as in REWRITE(f), where f is an identifier of a file-type variable. The procedure call REWRITE(f) where f is a file-variable, discards the current value of the file, f, so that a new file can be generated. The Boolean function EOF(f) becomes true and the file-buffer variable, f^, becomes defined.

- REWRITE(f) is necessary for initialization of the file, f, before generating the file, f. [The textfile OUTPUT automatically recieves a REWRITE.]

- Once REWRITE(f) has been applied to file, f, it is considered to be in a "write-only" state; i.e., PUT(f) may be used on f but not GET(f). WRITE may be used in f but not READ, and if f is a text file, WRITELN may be used but not READLN.

To use PUT, to either write the first component to the beginning of the file or to append a new component at the end of a file, the programmer must first insure that data has been assigned to the internal file-buffer variable. For example, the user writes:

```
    f^:= expression
```

If file, f, has been prepared for writing by means of the procedure call REWRITE(f), and is in the write-only state, and the file-buffer variable contains data to be written to the file as a file component, then the standard procedure PUT may be called. Then the effect of the procedure call PUT(f) is to physically append the current value of the file-buffer variable; f^, to the file, f. The Boolean function EOF(f) remains true, and the value of the file-buffer variable, f^, becomes undefined. Prior to the use of PUT(f), the condition of EOF(f) under any circumstances, either writing the first component or appending a new component, is true. For example, to generate a new file of the same component-type as the data elements in an ARRAY indexed by I from 1 to 100, the user may write:

```
BEGIN
  REWRITE(f);
  WHILE EOF DO
  FOR I:=1 TO 100 DO
    BEGIN
      f^:= ARRAY [I];
      PUT(f)
    END
END
```

Therefore, the definition of the standard procedure PUT requires one argument, as in PUT(f), where f is an identifier of a variable which is of the file-type. The procedure call PUT(f) operates as follows. If the Boolean function EOF(f) is true prior to the execution of the PUT(f), the value of the file-buffer variable, f^, is appended to the file, f; EOF(f) remains true; and the file-buffer variable, f^, becomes undefined. It is considered an error if EOF(f) is not true prior to the execution of PUT(f) or if the file-buffer variable, f, is undefined prior to the execution of PUT(f).

## 8.8 WRITING FILES USING WRITE AND WRITELN

The standard procedure, WRITE, is called to output data to files. The file must have been given an initial REWRITE and EOF should be true. The textfile OUTPUT is automatically given a REWRITE at the beginning of any Pascal program listing OUTPUT in the Program header. The syntax of the procedure call on the standard procedure, WRITE, is:

WRITE

```
---> WRITE ---> parameter-list --->
```

where the syntax of the parameter-list, defined in the WRITE procedure, requires a different argument-list in the call, depending on whether WRITE is being applied to files other than text files or to text files.

### Writing to Pascal files which are not text files:

The syntax of the WRITE parameter-list, when applied to files other than text files is:

**WRITE-Parameter-List** (for non-textfiles)

```
---> ( ---> file-variable ---> , ---> expression ---> ) --->
                                ^                |
                                |                |
                                |<----   ,  <----V
```

In this case, the format of the call on WRITE requires an argument-list corresponding to the above parameter-list. The argument-list of WRITE, for non-textfiles must contain first, a file-variable identifier of the file to be written to, followed by a comma, followed by one or more expressions, each separated from the other by a comma; the entire list enclosed in parentheses.

The effect of the execution of the WRITE procedure call statement, depending on their various forms, is as follows, where f stands for a non-textfile, and e stands for an expression which is "assignment-compatible" to the component-type of the file, f:

```
    WRITE(f,e)             is equivalent to:   BEGIN
                                                 f^:=e;
                                                 PUT(f);
                                               END

    WRITE(f,e1,...,en)     is equivalent to:   BEGIN
                                                 WRITE(f,e1);
                                                 .
                                                 .
                                                 .
                                                 WRITE(f,en);
                                               END
```

WRITELN is not used to write to non-textfiles.

### Writing to Pascal text files with WRITE:

The syntax of the WRITE parameter-list, when applied to text files, is different in that the file-variable, whose presence is optional, defaults to the standard file OUTPUT when not specified in the list. Also, instead of just supplying the expression whose value is to be output, the parameter-list for text file

WRITEs allow expressions with additional formatting attributes specified. Therefore, following the optional file-variable, the parameter-list for text file WRITEs contains one or more write-parameters.

Within the construct, write parameter, the user may format to a text-oriented file, such types of data as integer or real numeric data, character data, Boolean data, and string data. The syntax of the WRITE parameter-list, when applied to text files, is:

### WRITE-Parameter-List (for text files)

```
---> ( ------------------------------>write-parameter----> ) --->
        |                         ^ ^                  |
        |                         | |                  |
        v--->file-variable-> ,-- >| <-------  , <------v
```

In this case, the format of the call on WRITE requires an argument-list corresponding to the above defined parameter-list. That is, the argument-list of WRITE must contain first, although optional, a file-variable identifier of the text file to be written to, followed by a comma when present. This is followed by one or more write-parameters, each separated from the other by a comma with the entire list enclosed in parentheses. When the file-variable is not present, the file OUTPUT is assumed.

Again, the effects of the execution of the WRITE procedure call statement, depending on their various forms, are defined as follows; where f stands for the text file-variable and pn stands for a write-parameter:

```
WRITE(f,p1,...pn)      is equivalent to:   BEGIN
                                             WRITE(f,p1);
                                             .
                                             .
                                             .
                                             WRITE(f,pn)
                                           END


WRITE(p1,...pn)      is equivalent to:   BEGIN
                                           WRITE(OUTPUT,p1);
                                           .
                                           .
                                           .
                                           WRITE(OUTPUT,pn)
                                         END
```

However, only when the write-parameter is an expression of type CHAR, with no formatting attributes specified for spacing the character in a textfile field, are the WRITE(f,p) or WRITE(p) calls enacted by the following equivalences.

```
    WRITE(f,p)             is equivalent to:   BEGIN
                                                 f^:=p;
                                                 PUT(f)
                                               END


    WRITE(p)               is equivalent to:   BEGIN
                                                 OUTPUT^:=p;
                                                 PUT(OUTPUT);
                                               END
```

The various forms of write-parameters are defined below, and the prose and examples describe how write-parameters appear on the external textfiles in formatted textfile fields.

The syntax of a write-parameter is:

Write-Parameter

```
                                      ^----------------->|
                                      |                  |
                                      |                  v
 --->expression1--->: -->expression2---->: expression3 ------>
                  |                                         ^
                  V---------------------------------------->|
```

where expression1 is the expression whose value is to be written to the text file.

Expression1 may be of any integer-type, or character-type, or subrange thereof, or any real-type, Boolean-type, or string-type. Expression2 is an integer-type expression specifying the field-width parameter, total-width; and expression3 is an integer-type expression specifying the field-width parameter, fractional-digits. This last field applies only when expression1 is real-type data. Total-width means the number of character positions to be occupied on the output file, in a textfile field. Fractional digits means the number of character positions to be occupied by the fractional portion of a real number to be displayed in a textfile field.

Formatting data to a text file is dependent on the form of each write-parameter given as an argument to WRITE, (or WRITELN). In accordance with the syntax graph above for "write-parameter", a write-parameter may be specified by:

```
expression

expression : total-width

expression : total-width : fractional-digits
```

where the value of the expression is to be formatted and written to a text file and that expression1 may be of type:

| | |
|---|---|
| CHAR | {character data} |
| ARRAY[i..j] OF CHAR | {string-type data} |
| BOOLEAN | {Boolean-type data} |
| BYTE, SHORTINTEGER, or INTEGER | {integer-type data} |
| REAL or SHORTREAL | {real-type data} |

The total-width and fractional-digits expressions are of any integer-type; and specify a number of field positions:

total-width

> specifies the exact number of character positions to be written, i.e., a minimum field width. When representation of the value of the expression requires more positions than total-width, the output field is expanded to get the value written, regardless of an inadequately specified total-width (except for Booleans and strings). If representation of the value of the expression requires less positions than specified in total-width, an appropriate number of spaces are written to the left of the representation, i.e., the value is usually right-justified within the field width specified by total-width (except for scientific notation of reals).

fractional-digits

> specifies the number of decimal digits of precision to be displayed in the output of a real number; and only applies to reals.

See examples, below. Note that the examples below, call the WRITE routine to demonstrate text file formatted output to a textfile field. Executing the examples as shown and in sequence using WRITE would actually produce the textfile fields side by side, not one per line. However, replacing the calls on WRITE with calls on WRITELN produces the example outputs on one per line. The text file-variable is represented by f, and if omitted implies the file OUTPUT.

### {Write-parameters formatting character-type data to text files}

If the expression to be output is character-type data, and no total-width is specified in the write-parameter, the default value of total-width is one, i.e., a single character is written. If total-width is specified in a write-parameter for character-type data, the representation written to the text file, is ("total-width" - 1) spaces followed by the character value of the expression; i.e., the single character is displayed right-justified in a field of total-width positions.

For example, given the declaration: VAR CH:CHAR; and CH := 'A';

```
                                |column1 of textfile field
                                |
                                v
    WRITE(f,CH)       produces: A
    WRITE(f,CH:1)     produces: A
    WRITE(f,CH:2)     produces:  A
    WRITE(f,CH:3)     produces:   A
    WRITE(f,CH:10)    produces:          A
    WRITE(f,'B':10)   produces:          B
```

### {Write-parameters formatting string-type data to text files}

If the expression to be output is string-type data, and no total-width is specified in the write-parameter, the default value of total-width is n, i.e., the number of components in the string. The string may be a literal-string, constant, or variable of the type ARRAY[i..j] OF CHAR, i.e., a single-dimensioned character array of n components. If total-width is specified in a write-parameter for string-type data, the representation written to the text file, is ("total-width" - n) spaces followed by the n characters in the string; i.e., the string is displayed right-justified in a field of total-width positions. If the total-width specified is inadequate, the textfile field is not increased and the string is truncated from the right.

For example, given a declaration: VAR STRING:ARRAY[1..5] OF CHAR; and:

STRING := 'HELLO';

```
                                    |column1 of textfile field
                                    |
                                    v
    WRITE(f,STRING)       produces: HELLO
    WRITE(f,STRING:5)     produces: HELLO
    WRITE(f,STRING:6)     produces:  HELLO
    WRITE(f,STRING:7)     produces:   HELLO
    WRITE(f,STRING:10)    produces:      HELLO
    WRITE(f,'HELLO':10)   produces:      HELLO
    WRITE(f,STRING:3)     produces: HEL
    WRITE(f,'HELLO':1)    produces: H
```

### {Write-parameters formatting Boolean-type data to text files}

If the expression is Boolean-type, the internal form of the Boolean value of the expression is converted into the appropriate character strings 'TRUE ' or 'FALSE' corresponding to that value. If no total-width is specified in the write-parameter for a Boolean valued expression, the default of total-width is five positions. If total-width is specified and is greater than five, then the words TRUE or FALSE are preceded by ("total-width" - 5) spaces, i.e., the strings 'TRUE ' and 'FALSE' are displayed right-justified in a field of total-width positions. If total-width is inadequately specified as less than 5 positions the field width is not expanded and the strings 'TRUE ' or 'FALSE' are truncated from the right.

For example, given the declaration: VAR FLAG,SIGN : BOOLEAN;
and:

```
    FLAG := true;
    SIGN := false;
                                  |column1 of textfile field
                                  |
                                  v
    WRITE(f,FLAG)       produces: TRUE
    WRITE(f,FLAG:5)     produces: TRUE
    WRITE(f,FLAG:6)     produces:  TRUE
    WRITE(f,FLAG:7)     produces:   TRUE
    WRITE(f,FLAG:10)    produces:      TRUE

    WRITE(f,SIGN)       produces: FALSE
    WRITE(f,SIGN:5)     produces: FALSE
    WRITE(f,SIGN:6)     produces:  FALSE
    WRITE(f,SIGN:7)     produces:   FALSE
    WRITE(f,SIGN:10)    produces:      FALSE

    WRITE(f,FLAG:3)     produces: TRU
    WRITE(f,SIGN:3)     produces: FAL

    WRITE(f,FLAG:1)     produces: T
    WRITE(f,SIGN:1)     produces: F

    WRITE(f,FLAG,SIGN)  produces: TRUE FALSE
    WRITE(f,SIGN,FLAG)  produces: FALSETRUE

    WRITE
      (f,FLAG,' ',SIGN) produces: TRUE  FALSE
    WRITE
      (f,SIGN,' ',FLAG) produces: FALSE TRUE

    WRITE(f,TRUE,FALSF) produces: TRUE FALSE

    WRITE(f,7>5)        produces: TRUE
    WRITE(f,25<=10)     produces: FALSE

    WRITE(f,
       (7>5)AND(25<=10)) produces: FALSE
```

### {Write-parameters formatting integer-type data to text files}

If the expression to be output is integer-type data, the decimal representation (in characters) of the integer value is output to the text file. If no total-width is specified in the write-parameter of an integer valued expression, the default value of total-width is ten, i.e., the integer is displayed right-justified in a field of ten positions (with insignificant leading zeroes suppressed; and if the value is negative, the integer will be preceded by a minus sign). If total-width is specified and is greater than the number of positions required to represent the integer; then ("total-width" - that number - 1) spaces will first be written. Then, if the integer is less than zero ( a negative) the character "-" is written, otherwise a space character (for positives) is written. Then the characters required to represent the decimal value of ABS(expression) will be written. If total-width is specified and is less than the number of positions required to represent the integer; the output field is expanded to get the integer written. For example:

```
{Declaring:}  VAR IPOS,INEG:INTEGER; B:BYTE; SI:SHORTINTEGER;

{& assigning}IPOS := 32768; INEG := -IPOS; B := 255; SI := 32767;

                                  |column1 of textfile field
                                  |
                                  v
    WRITE(f,IPOS)       produces:      32768
    WRITE(f,INEG)       produces:     -32768
    WRITE(f,SI)         produces:      32767
    WRITE(f,B)          produces:        255

    WRITE(f,IPOS:10)    produces:      32768
    WRITE(f,INEG:10)    produces:     -32768

    WRITE(f,IPOS:6)     produces:  32768
    WRITE(f,INEG:6)     produces: -32768

    WRITE(f,SI:6)       produces:  32767
    WRITE(f,-SI:6)      produces: -32767

    WRITE(f,IPOS:3)     produces: 32768
    {If width specification inadequate, value gets written}
    WRITE(f,INEG:3)     produces: -32768

    WRITE(f,MAXINT)     produces: 2147483647
    {exception: large 10-digit negatives occupy 11 positions}
    WRITE(f,-MAXINT)    produces: -2147483647   {exception}

    WRITE(f,25)         produces:         25
    WRITE(f,25:5)       produces:    25
    WRITE(f,25:2)       produces: 25

    WRITE(f,-B)         produces:       -255
    WRITE(f,-B:5)       produces:  -255
    WRITE(f,-B:4)       produces: -255
```

### {Write-parameters formatting real-type data to text files}

If the expression is real-type data, (REAL or SHORTREAL), the write-parameter given as an argument to WRITE (or WRITELN), may be expressed by any of the three forms:

```
expression                                  {for scientific notation}

expression : total-width                    {for scientific notation}

expression : total-width : fractional-digits  {for fixed-point}
```

If the expression is real-type data, a decimal representation (in characters) of the value of the expression is written to the text file. That value is rounded to the specified (or implied default) number of significant decimal digits of precision before being written to the text file.

There are two formats in which reals may be displayed on a text file. They are a floating-point or fixed-point representation as follows.

As Perkin-Elmer Pascal additionally provides the SHORTREAL type occupying half the storage as REAL, and containing fewer digits of precision; there are two different floating-point types of representation: one for SHORTREAL values and one for REAL values.

- floating-point (or scientific notation) representation

```
        for SHORTREALs:     sd.ddddddEsdd

        for REALs:          sd.dddddddddddddddEsdd
```

where s is indicative of sign, and d is a decimal digit.

The display always starts in column 1 of a textfile field of total-width positions. The default field total-width for a SHORTREAL is 14. The default field total-width for a REAL is 24.

In both REAL and SHORTREAL floating-point formats the following apply. The leading s for sign is the minus character "-" for negatives, but a space character (not the "+") is given for the leading sign character s of positives.

That is, in both SHORTREAL and REAL seven positions are always occupied by the leading sign indicator s, the first digit d, the decimal point ".", and the ending exponent letter E, the exponent's sign character "+" or "-" and two digits of exponent.

The letter E begins the exponent field, where sign is indicated by s, which is the character "+" for positive exponents (large in magnitude reals) or the character "-" for negative exponents (small in magnitude reals). Leading zeroes in the exponent are not suppressed.

For example, with a total-width default of 14 positions for the field, of a SHORTREAL, the default allows the number of digits to the right of the decimal point, to default to seven, i.e., total-width SHORTREAL default of 14, so 14-7 = 7 places after the decimal point.

For example, with a total-width default of 24 positions for the field, of a REAL, the default allows the number of digits to the right of the decimal point, to default to seventeen, i.e., total-width REAL default of 24, so 24-7 = 17 places after the decimal point.

The user may specify total-width in the write-parameter of a real-valued (either REAL or SHORTREAL) expression, so the number of precision digits displayed following the decimal point may be at least one, and expand (display more digits than one past the decimal point) when the user-specified total-width increases from 8 and up. When a write-parameter contains a user-specified total-width following a real- or shortreal-valued expression (and does not specify a fractional-digits field) the floating-point representation of the real number is output in a field of the user-specified "total-width" positions not the default widths.

The exponential floating-point (or scientific notation) representation of reals or shortreals are written to text files with the following formatting action; i.e., the following sequence of characters are written:

the sign character, "-" if expression < 0, otherwise a space;
the leading decimal-digit character of the ABS(expression);
the character "." for a decimal point;
the next ("total-width" - 7) digits of the ABS(expression);
the character "E";
the sign of the exponent, "-" if exponent < 0, otherwise a "+";
the two digits of the exponent, with leading zero, if required.

The other form for outputting real numbers is called:

- fixed-point representation and is displayed in the form:

sDDDDDDDDDD.dddddd

where s is indicative of sign, either a space character for positives, or the character "-" for negatives; and D is a decimal digit of the integer part of a real number, and d is a decimal digit of the fractional part of a real number. The only form of the write-parameter that will direct the output of a REAL or SHORTREAL value to be written to a text file in fixed-point representation is:

**expression : total-width : fractional-digits**

The number of displayed D's is determined by the magnitude of the value of the real-valued "expression". The number of displayed d's is determined by the value of the "fractional-digits" specified in the write-parameter. In this format, the representation of the real-valued expression, or shortreal-valued expression, is displayed right-justified in a field of "total-width" positions, where the field is filled on the left with as many spaces, as is required.

The fixed-point decimal representation of real-valued expressions, either REAL or SHORTREAL (no difference, as the user specifies the output format), is formatted as follows, only when the user has specified "fractional-digits" in addition to "total-width" in the write-parameter.

A minimum number of character positions, determined by the number of characters required left of the decimal point for the integer part of the magnitude of the value plus the specified number of "fractional-digits" plus one, will be computed. If the value of the "expression" is negative, one more is added to this minimum. If the specified "total-width" is greater than that minimum number of character positions so computed, then ("total-width" - minimum) spaces will first be written; i.e., where the total-width field specification and real value permit, the representation of the real is displayed right-justified within that field. If the expression is negative, the character "-" is next written to precede the decimal representation of the real number magnitude. Then, as many D decimal-digit characters as is required to represent the integer part of the real number are written, followed by the character ".", followed by "fractional-digits" d decimal-digit characters to represent the fractional part of the real number. If the total-width specified is inadequate, to at least represent the magnitude of the real, the field will be expanded to output a minimum representation of the real number.

The following sequence of characters are written to a text file for the fixed-point decimal representation of a real-valued or shortreal-valued expression.

```
if total-width >= minimum, ("total-width" - minimum) spaces;
if expression < 0, the character "-";
the first integer digits of the decimal representation of:
        TRUNC(ABS(expression + 0.5E-"fractional-digits")
the character ".", for a decimal point;
the next "fractional-digits" digits of precision.
```

For example, given the declarations:

```
VAR PI,RP,RN,R1,R2,R3,R4 : REAL;
    SPI,SRP,SRN,SR1,SR2,SR3,SR4 : SHORTREAL;
```

and assigning values to these real-typed variables:

```
  BEGIN
    PI := 3.141592653589793;
    RP := MAXINT;
    RN := -2.0E00;
    R1 := 789.1025;
    R2 := 6789.5;
    R3 := 32768.66666666;
    R4 := -600000.75;
    {and the shortreals: }
    SPI := PI;
    SRP := MAXSHORTINT;
    SRN := -2.0E+00;
    SR1 := 789.1025;
    SR2 := 6789.5;
    SR3 := 32768.66666666;
    SR4 := -600000.75;
  END;
```

Then, real-valued expressions may be formatted as write-parameters to be output to text file fields (with either WRITE or WRITELN):

```
    (*   floating-point    *)      |column1 of textfile field
    (* scientific notation *)      |
                                   v
    WRITE(f,PI)          produces:  3.14159265358970027E+00
    WRITE(f,PI:24)       produces:  3.14159265358970027E+00

    WRITE(f,SPI)         produces:  3.1415920E+00
    WRITE(f,SPI:14)      produces:  3.1415920E+00

    {outputting REALs in floating-point representation}

    WRITE(f,PI:8)        produces:  3.1E+00
    WRITE(f,PI:9)        produces:  3.14E+00
    WRITE(f,PI:10)       produces:  3.142E+00
    WRITE(f,PI:11)       produces:  3.1416E+00
    WRITE(f,PI:12)       produces:  3.14159E+00
    WRITE(f,PI:13)       produces:  3.141593E+00
    WRITE(f,PI:14)       produces:  3.1415927E+00
    WRITE(f,PI:15)       produces:  3.14159265E+00
    WRITE(f,PI:16)       produces:  3.141592654E+00
    WRITE(f,PI:17)       produces:  3.1415926536E+00
    WRITE(f,PI:18)       produces:  3.14159265359E+00
    WRITE(f,PI:19)       produces:  3.141592653590E+00
    WRITE(f,PI:20)       produces:  3.1415926535897E+00
    WRITE(f,PI:21)       produces:  3.14159265358970E+00
    WRITE(f,PI:22)       produces:  3.141592653589700E+00
    WRITE(f,PI:23)       produces:  3.1415926535897003E+00
    WRITE(f,PI:24)       produces:  3.14159265358970027E+00
    WRITE(f,PI:25)       produces:  3.141592653589700270E+00
```

```
{outputting SHORTREALs in floating-point representation}

                             |column1 of textfile field
                             |
                             v
WRITE(f,SPI:8)     produces:  3.1E+00
WRITE(f,SPI:9)     produces:  3.14E+00
WRITE(f,SPI:10)    produces:  3.142E+00
WRITE(f,SPI:11)    produces:  3.1416E+00
WRITE(f,SPI:12)    produces:  3.14159E+00
WRITE(f,SPI:13)    produces:  3.141592E+00
WRITE(f,SPI:14)    produces:  3.1415920E+00
WRITE(f,SPI:15)    produces:  3.14159200E+00
WRITE(f,SRP:14)    produces:  3.2767000E+04

{Other REALs in floating-point format}

WRITE(f,RP:16)     produces:  2.147483647E+09

WRITE(f,RN:8)      produces: -2.0E+00

WRITE(f,-2.0E3:8)  produces: -2.0E+03

WRITE(f,R1:24)     produces:  7.891025000000000040E+02
WRITE(f,R2:24)     produces:  6.789500000000000000E+03
WRITE(f,R3:24)     produces:  3.276866666666600009E+04
WRITE(f,R4:24)     produces: -6.000007499999999972E+05

(* fixed-point representation *)

                             |column1 of textfile field
     { REALs }               |
                             v
WRITE(f,R1:24:4)   produces:                 789.1025
WRITE(f,R2:24:4)   produces:                6789.5000
WRITE(f,R3:24:4)   produces:               32768.6667
WRITE(f,R4:24:4)   produces:             -600000.7500

WRITE(f,R1:14:2)   produces:         789.10
WRITE(f,R2:14:2)   produces:        6789.50
WRITE(f,R3:14:2)   produces:       32768.67
WRITE(f,R4:14:2)   produces:     -600000.75

                             |column1 of textfile field
    { SHORTREALs }           |
                             v
WRITE(f,SR1:24:4)  produces:                 789.1023
WRITE(f,SR2:24:4)  produces:                6789.5000
WRITE(f,SR3:24:4)  produces:               32768.6640
WRITE(f,SR4:24:4)  produces:             -600000.7500

WRITE(f,SR1:14:2)  produces:         789.10
WRITE(f,SR2:14:2)  produces:        6789.50
WRITE(f,SR3:14:2)  produces:       32768.66
WRITE(f,SR4:14:2)  produces:     -600000.75
```

```
(* REAL results with inadequate field specifiers *)

                                   |column1 of textfile field
                                   |
                                   v
WRITE(f,RN:1)         produces: -2.0E+00
WRITE(f,RN:1:1)       produces: -2.0

WRITE(f,R1:3:2)       produces: 789.10
WRITE(f,R2:3:2)       produces: 6789.50
WRITE(f,R3:3:2)       produces: 32768.67
WRITE(f,R4:3:2)       produces: -600000.75

(* SHORTREAL results with inadequate field specifiers *)

WRITE(f,SRN:1)        produces: -2.0E+00
WRITE(f,SRN:1:1)      produces: -2.0

WRITE(f,SR1:3:2)      produces: 789.10
WRITE(f,SR2:3:2)      produces: 6789.50
WRITE(f,SR3:3:2)      produces: 32768.67
WRITE(f,SR4:3:2)      produces: -600000.75
```

The Write-parameter need not specify the optional fields: total-width (or fractional digits for reals); when the default values suffice.

For this implementation, the default values of total-width are:

| | |
|---|---|
| BYTE | 10 |
| SHORTINTEGER | 10 |
| INTEGER | 10 |
| SHORTREAL | 14 |
| REAL | 24 |
| BOOLEAN | 5 |
| CHAR | 1 |
| String | Length (the number of characters in the string) |

## **Writing to Pascal text files with WRITELN:**

The syntax of the procedure call on the standard procedure WRITELN which may only be applied to text files, is:

### **WRITELN**

```
--->WRITELN-------------------------------->
             |                          ^
             |                          |
             |                          |
             V---> parameter-list--->|
```

The format then of the procedure call on WRITELN, is the procedure name WRITELN followed optionally by an argument-list corresponding to a WRITELN parameter-list. The effect of the execution of a WRITELN call when no argument-list is present, and if no previous WRITES have been performed, is to output on the default file-variable OUTPUT an end-of-line indicator, which effectively produces a line-feed or skip to the next line on the OUTPUT file. The effect of the execution of a WRITELN call when no argument-list is present and if previous calls on WRITE still have information buffered for output is to output remaining information and then effect a line-feed or skip to the next line on the default file-variable, OUTPUT.

The syntax of the WRITELN parameter-list, which may only be applied to text files, is:

### WRITELN-Parameter-List

```
                            ---------------------------->
                            ^                           |
                            |                           v
-----> ( --->file-variable---> ,----->write-parameter-----> ) -->
          |                      ^ ^                  |
          |                      | |                  |
          v--------------------->| <------ , <--------v
```

In this case, the format of the call on WRITELN, requires an argument-list corresponding to the above defined parameter-list. The argument-list of WRITELN contains first, although optional, a file-variable identifier of the text file to be written to. If the text file is present, it is followed by a comma (whenever one or more write-parameters follow). Then this is optionally followed by one or more write-parameters, each separated from the other by a comma, the entire list enclosed in parentheses. When the file-variable is not present, the standard text file OUTPUT is assumed. The syntax and format of write-parameters are exactly as described in the previous information on the WRITE procedure's parameter-list.

The effect of the execution of the WRITELN procedure call statement, depending on their various forms, are as follows; where f stands for the text file-variable and pn stands for a write-parameter:

| | | |
|---|---|---|
| WRITELN | produces the effect: | outputs buffered text, if any, and skips line (sends EOLN indicator) to default OUTPUT. |
| WRITELN (f) | produces the effect: | outputs buffered text, if any, and skips line (sends EOLN indicator) |

to text file, f.

```
WRITELN (f,p)          is equivalent to:     BEGIN
                                              WRITE (f,p);
                                              WRITELN (f)
                                             END

WRITELN (f,p1,...,pn) is equivalent to:      BEGIN
                                              WRITE (f,p1,...,pn);
                                              WRITELN (f)
                                             END

WRITELN (p)            is equivalent to:     BEGIN
                                              WRITE (OUTPUT,p);
                                              WRITELN
                                             END

WRITELN (p1,...,pn)    is equivalent to: BEGIN
                                          WRITE(OUTPUT,p1,...,pn);
                                          WRITELN
                                         END
```

In the above equivalences the actual data written for the write-parameter(s), pn, depend upon the data types of the expressions within the write-parameter construct and the additional specifications of total-width (and fractional-digits for reals) that may be included in the write-parameter. Refer to the above in Section 8.7 for details. Essentially, the default specifications regarding field-widths would allow character data to be output as one character, Boolean data to be output as the sequence TRUE or FALSE, string data to add that string's sequence to the output stream, and integer or real numeric data to be output as their respective optionally signed character integer or real/decimal representations.

## 8.9 DATA TRANSFER EXAMPLES

Data transfers with text files can be entirely programmed by the use of the standard procedures READLN and/or WRITELN. However, text files, depending on their user-planned input or output, may at times require using both READ and/or WRITE routines and the READLN and/or WRITELN routines. To illustrate the use of the combination of calls to READ and READLN or to WRITE and WRITELN, the following examples are offered.

If a text file of sequential lines with its planned format to consist of four variables on each line, the first variable is a code, 1,2, or 3; the second variable is a character-type, integer-type, or real-type, respective to the first variable's code. The third variable also is a code, 1,2, or 3, and the fourth variable has the same method of typing as the second variable, respective to whatever the code of the third variable is. For example:

```
 ------------------------------------
|    V1    |   V2    |    V3    |  V4  |
| INTEGER |  DATA   | INTEGER | DATA |
|   CODE   |         |   CODE   |      |
 ------------------------------------
```

READ is used below to read in each variable from the line, and READLN is used when input is ready to advance to the next whole line. For purposes of simplicity, it is assumed that the input data stream is on a previously written textfile (non-terminal) on the predefined text file, INPUT; i.e., so that no file-variable identifier is present in the READ or READLN procedure-calls; and an automatic RESET is performed on INPUT prior to executing any statements in the main program body. If the textfile to be read from is a terminal, I/O might not be controlled with a WHILE loop querying EOF.

```
    PROGRAM INDATA(INPUT,OUTPUT);
    VAR CHARVAR:CHAR;I,CODE,INTVAR:INTEGER;REALVAR:REAL;
    BEGIN
    WHILE NOT EOF DO
      BEGIN
        FOR I := 1 TO 2 DO
          BEGIN
            READ (CODE) ;
            CASE CODE OF
              1: BEGIN READ(CHARVAR); {process character} END;
              2: BEGIN READ(INTVAR); {process integer} END;
              3: BEGIN READ(REALVAR); {process real} END;
              OTHERWISE BEGIN WRITELN('ERR=',CODE) END
            END;
          END;
        READLN;     {advance to next line}
      END;
    .
    .
    .
   END.
```

The above coding for textfile I/O would require the character to be read into CHARVAR to be immediately following the integer, on the external file, allowing for no space between them (as textfile READ does not skip leading spaces on a character-read as it does for reals and integers). But there must be a space on the textfile being read between the integer CODE and a real number to be read into REALVAR or between the integer CODE and an integer number to be read into INTVAR, and vice-versa between the data and a subsequent CODE; or the data could not be distinguished from the integer CODE, or vice-versa the CODE from the data.

In the following example, the textfile is presumed to have been written (or input data positioned) such that one or more spaces exist between the CODEs and data variables; and the planned format to be read is:

```
 -----------------------------------------
|    V1    | |    V2    | |    V3    | |  V4  |
| INTEGER  | |   DATA   | | INTEGER  | | DATA |
|   CODE   | |          | |   CODE   | |      |
 -----------------------------------------
```

In the above diagram the data codes and data variables are depicted as having either a space or EOLN between them, such that the following reprogramming of INDATA would apply.

```
 PROGRAM INDATA2(INPUT,OUTPUT);
 VAR CHARVAR:CHAR;I,CODE,INTVAR:INTEGER;REALVAR:REAL;
     CH:CHAR;
 BEGIN
 WHILE NOT EOF DO
   BEGIN
     FOR I := 1 TO 2 DO
       BEGIN
         READ (CODE) ;
         CASE CODE OF
           1: BEGIN
                CH := ' ';
                WHILE CH = ' ' DO
                  BEGIN
                    READ(CH);
                  END;
                READ(CHARVAR);
              {process character}
              END;
           2: EEGIN READ(INTVAR); {process integer} END;
           3: BEGIN READ(REALVAR); {process real} END;
           OTHERWISE BEGIN WRITELN('ERR=',CODE) END
         END;
       END;
     READLN;      {advance to next line}
   END;
 .
 .
 .
END.
```

In the following example READLN suffices to bypass EOLN's after each line. Given a text file of sequential lines exists with the line format consisting of only two variables, a CODE and DATA; the line format is such that the first variable on a line is an integer code, 1, 2, or 3; and the second variable on the line is a character-type, integer-type, or real-type, respective to whatever the code of the first variable is. For example:

```
 --------------
|   V1   |      |
|INTEGER| DATA |
|  CODE  |      |
 --------------
```

Only READLN is used after the CODE is obtained, and READLN accomplishes this specific application best. Again, for purposes of simplicity, the input data stream is assumed to be on the text file, INPUT. [The query on NOT EOF assumes INPUT is a previously written textfile, not an interactive terminal.]

```
VAR CHARVAR:CHAR;I,CODE,INTVAR:INTEGER;REALVAR:REAL;
BEGIN
WHILE NOT EOF DO
  BEGIN
    READ (CODE);
    CASE CODE OF
      1:READLN(CHARVAR);  {assuming no space prior char}
      2:READLN (INTVAR);
      3:READLN (REALVAR)
    END;
  END;
END;
```

To illustrate the effect of the combination of uses of WRITE and WRITELN, the following example is provided. The standard text file, OUTPUT, is being used. The sequence of statements:

```
WRITELN ('8');
WRITE ('A', 'B', 'C');
WRITELN ('D');
WRITELN;
WRITELN ('E', 'F', 'G');
```

would output:

```
8
ABCD

EFG
```

Another example; the sequence: WRITE ('MSG'); WRITELN; is equivalent to:

```
WRITELN ('MSG');
```

Both would output:

```
MSG
```

and advance to the next line of output.

## Example: Interactive Textfile I/O

It is possible to use Pascal standard textfile I/O to write programs which can be run either interactively or in batch mode. To do this, one must pay careful attention to the defined time order in which events happen by definition. Here is an example of a simple program which receives lines of input from the terminal and responds to them.

The response to a command, for this example, is very simple: the program counts the number of occurrences of the letter "X" in the input command, and reports it. The program can end in two ways; by reading a command line in which the last character is a period, or by reaching end of file on its command input.

In this example implementation, LINE is a buffer controlled by the program, containing the command line. Because the line may be less than the maximum width, which is chosen to be 80 characters, processing the line depends on knowing the value of LENGTH, which is the number of characters actually in the line. (It may be zero.) Procedure SCAN transfers characters from the text buffer to LINE; this procedure is designed on the assumption that it starts at the beginning of a line.

```
PROGRAM COUNTX(COMMAND, OUTPUT);

CONST
  LINE_MAX = 80;

TYPE
  LINE_INDEX = 1 .. LINE_MAX;
  LINE_INDEX_0 = 0 .. LINE_MAX;
  LINE_TYPE = ARRAY [LINE_INDEX] OF CHAR;

VAR
  LINE: LINE_TYPE;
  COMMAND: TEXT;
  LENGTH: LINE_INDEX_0;
  END_OF_INPUT: BOOLEAN;

PROCEDURE SCAN(VAR BUFF: LINE_TYPE; VAR BUFLEN: LINE_INDEX_0);
  BEGIN
  BUFLEN := 0;
  WHILE (BUFLEN < LINE_MAX) AND NOT EOLN(COMMAND) DO BEGIN
    BUFLEN := BUFLEN + 1;
    READ(COMMAND, BUFF[BUFLEN]);
    END;
  END;

FUNCTION NUM_X(BUFF:LINE_TYPE; BUFLEN:LINE_INDEX_0):LINE_INDEX_0;
  VAR I, XCOUNT: LINE_INDEX_0;
  BEGIN
  XCOUNT := 0;
  FOR I := 1 TO BUFLEN DO
    IF BUFF[I] = 'X' THEN XCOUNT := XCOUNT + 1;
  NUM_X := XCOUNT;
  END;

FUNCTION THRU(BUFF: LINE_TYPE; BUFLEN: LINE_INDEX_0): BOOLEAN;
  BEGIN
  THRU := (BUFLEN > 0) AND (BUFF[BUFLEN] = '.');
  END;

BEGIN
WRITELN('PLEASE ENTER A COMMAND LINE');
RESET(COMMAND);
END_OF_INPUT := EOF(COMMAND);
WHILE NOT END_OF_INPUT DO BEGIN
  SCAN(LINE, LENGTH);
  WRITELN(NUM_X(LINE, LENGTH):5, ' X''S IN THIS LINE');
  END_OF_INPUT := THRU(LINE, LENGTH);
  IF NOT END_OF_INPUT THEN BEGIN
    READLN(COMMAND);
    END_OF_INPUT := EOF(COMMAND);
    END;
  END;
END.
```

# CHAPTER 9
# PROGRAM, MODULES, PROCEDURES, AND FUNCTIONS

## 9.1 PROGRAM AND PREFIX SYNTAX

A Pascal main program consists of an optional source prefix of declarations, a program-heading, followed by a block and concluded by a period, its terminating full stop, and is formally defined by the syntax:

Program

```
   ----> prefix --->|
   ^                |
   |                v
-----------------------> program-heading ---> block ---> .
```

The Perkin-Elmer predefined Prefix is required for those programs utilizing the additional language extensions for operating system services, provided for by the predefined Prefix (see Section 10.3 or Appendix N).

The user also can modify the predefined Prefix or program a user-written prefix to declare a collection of constants, type-identifiers and externally available routines for a compilation-unit.

A prefix can be placed ahead of either a program or a separate module heading. If the same prefix preceded a PROGRAM and a MODULE heading, then those declarations would be visible to both compilation-units. The context-free syntax of a prefix is:

Prefix

```
  |<--- const definitions <---  |<- ; <-- procedure-heading <---
  |                          ^  |                             ^
  V                          |  V                             |
----------------------------------------------------------------->
  ^                          |  ^                             |
  |                          |  |                             |
  |<--- type definitions <---V  |<- ; <-- function-heading <---V
```

The prefix consists entirely of declarations; it has no body; and does not form a block, although its declared identifiers have a global scope, to its subsequent code.

The syntax of a prefix in Perkin-Elmer Pascal, consists of one or more Constant Definition parts, or one or more Type Definitions parts, and these parts may be intermixed; and may also have individual routine header declarations, separated by a semicolon. Once the routine PROCEDURE/FUNCTION header declarations begin, no further constant or type definitions may follow between them and the PROGRAM header, as part of the prefix. The prefix procedure-heading (Section 9.4.1) or function-heading (Section 9.5.1) declarations, may occur in any order, (differing from Pascal R00 order restrictions) and declare their named routines as external routines using Pascal R01 linkage conventions for EXTERN routines. Use of the directives EXTERN or FORTRAN or a block definition is not allowed.

The prefix constant-identifiers and type-identifiers become visible in scope from their point of introduction to the entire compilation unit. The prefix routine header declarations define external routine names and their parameter interfaces, making the routine names visibly callable in routine-invocations. Users who add to or alter the routine header declarations in the Prefix provided, must also generate or have available external routines of the same names, with corresponding parameter-lists, and link to their object code at task establishment time. Refer to Chapter 10 for details on the predefined Prefix and the language extensions available to programs that include the predefined Prefix. See Chapter 4 for constant/type definitions parts.

The definition of the program-heading follows in Section 9.1.1. The definition of a block is discussed in Section 2.1.2 as a basic Pascal language concept and syntactically detailed in Section 4.1. An overview of the PROGRAM block and its routines is given in Section 9.1.3 and detailed throughout this chapter.

Briefly, a block contains optional label, constant, type, variable, and routine-declaration parts, to declare statement labels and to declare identifiers as the names of constants, types, variables, and routines; and the block contains a body, which is an executable compound statement housing a sequence of executable statements which may operate on declared data.

### 9.1.1 Program-Heading

The syntax of the program-heading is:

Program-Heading

```
                             -------------------->
                             ^                   |
                             |                   v
---> PROGRAM ---> identifier ---> file-name-list ---> ; --->
```

The identifier after the word symbol, PROGRAM, is the name of the main program, and if its identifier is longer than eight characters, the object program label, as listed in a OS/32 Link map, is truncated to the first eight characters.

The file-name-list has the syntax:

File-Name-List

```
---> ( ---> file-identifier ---> ) --->
       ^                       |
       |                       |
       <--------    , <-------v
```

The file-identifier is the name of a file-variable which by virtue of its appearance in the file-name-list is an external file. It is a Pascal identifier, not an OS/32 file-descriptor of the form "voln:filename.ext". These user-specified file names (other than INPUT or OUTPUT) must also be declared as a file variable in the VAR variable declarations part of the outermost block of the main program. An external file is one which may exist prior to or after the execution of a user program.

The format of the program-heading is the word PROGRAM, followed by the identifier of the program, optionally followed by the file-name-list, and separated from its block with a semicolon.

The format of the file-name-list is one or more file-identifiers, each separated by a comma, with the entire list enclosed in parentheses.

Any user-specified external file-identifier to be able to be referenced in the Pascal RESET, REWRITE, GET, PUT, READ, WRITE, READLN and WRITELN statements must be declared as a file-variable, a variable of the file-type; and must be listed in the main program file-name-list. No duplicates can be listed. The order in which the file-identifiers are listed determines the logical unit (lu) numbers of the external files (which the compiler uses in generating object code). That is, the first file-identifier is associated with lu 0, the second file-identifier is associated with lu 1, and so on. The maximum number of external files that can be listed in the file-name-list is 32.

A sample of a program-heading with no file-name-list is:

```
PROGRAM ACOUNTER;
```

which establishes the identifier ACOUNTER as the name of a program with no external files to be used. Internal files could still be declared in a variable-declarations-part of any block and used within its scope. A sample of a program-heading with the predefined text-file identifiers declared is:

```
PROGRAM TRANSACT (INPUT,OUTPUT);
```

which establishes the predefined text files INPUT and OUTPUT as available file-identifiers. INPUT is assigned to lu 0 and OUTPUT, to lu 1. If the program-heading were:

```
PROGRAM TRANSACT (OUTPUT, INPUT);
```

OUTPUT would be assigned to lu 0 and INPUT would be assigned to lu 1. These are predefined text-file identifiers, and the user need not declare their type as TEXT, nor their identifiers as file-variable identifiers since this is assumed in Pascal. A sample of a program-heading with several user-named files, in addition to the predefined INPUT and OUTPUT textfiles, is:

```
PROGRAM SORT(MASTER,UPDATE,INPUT,DELETE,OUTPUT);
```

In this example, the user declares the identifiers: MASTER, UPDATE, and DELETE in the file-name-list, as external filenames. They also must be declared as file variables in the variable-declarations-part of the outermost block of the main program, unlike INPUT and OUTPUT. For example:

```
PROGRAM SORT(INPUT, MASTER, OUTPUT, UPDATE, DELETE);
VAR MASTER,UPDATE,DELETE:TEXT;
BEGIN ...

END.
```

Other examples of declaring file-types other than text files are in Chapter 8. For files whose components are other than text-file information, a file-type other than TEXT must be used. For external files in the file-name-list, their file-variable declaration must take place in the outermost block of the main program. For example:

```
PROGRAM UPDATER (MASTERFILE, CHANGEFILE);

TYPE COMPONENTS = RECORD
                    NAME:  CHAR;
                    IDNUM:  INTEGER;
                    END;
VAR MASTERFILE,CHANGEFILE:FILE OF COMPONENTS;
```

```
    BEGIN ...

    END.
```

In this example the file variables MASTERFILE and CHANGEFILE, are declared and able to be referenced as external files. These identifiers have a scope similar to a global variable. Internal files are not to be declared in the file-name-list. Refer to Chapter 8. If the user only declares the file identifiers in the file-name-list of the program-heading and does not declare those identifiers as file variables in the outermost block of the main program, appropriate error messages are generated in the compiled program listing.

To force the compiler to associate certain external files to particular logical units, dummy file-identifiers and associated file-variables could be specified. To enforce the association of MASTERFILE and CHANGEFILE of the previous example to lu 2 and lu 4, respectively, the following code names dummy files and declares external file-identifiers as file variables.

```
PROGRAM UPDATE(DUMFILE1,DUMFILE2,MASTERFILE,DUMFILE3,CHANGEFILE);

    TYPE COMPONENTS = RECORD
                        NAME:  CHAR;
                        IDNUM:  INTEGER;
                      END;
    VAR MASTERFILE,CHANGEFILE:FILE OF COMPONENTS;
        DUMFILE1:TEXT;
        DUMFILE2:TEXT;
        DUMFILE3:TEXT;

    BEGIN  ...

    END.
```

File variables, must be passed to an external module, as variable (VAR) parameters, not value parameters (see Section 9.2.2). That is, there is no file-name-list construct within a MODULE-heading.

### 9.1.2 Program Block and its Routines: Procedures and Functions

Portions of the overall programming problem to be solved can be encoded into units called routines. Several Pascal programming considerations are:

- A main program constitutes the outermost block, in which the definition of a routine constitutes another block nested within the program block. This block structuring defines a scope, or area of visibility, in which a group of identifiers have their definition, visibility, existence, and use.

Declarations in the outermost block have a global scope, and declarations within a routine have a scope local to the block of that routine. Execution begins, after initializing the Pascal run time memory and internal system tables, by executing the body of the main program block. Activation of a routine block and execution of its body occurs upon a routine-invocation with a Pascal procedure-call statement or function-reference within an expression.

- A routine can be either a procedure that performs an operation, or a function that computes a value. A routine can be internal in the main program or external to the main program. See Section 9.1.3 below on external routines.

- A routine declaration is encoded to define its name, and interfacing parameter-list, to be referenced later to invoke the routine. See Section 9.3 on routine declarations.

- A routine declaration becomes a routine definition when its block has become defined, or the nature of its location for the purpose of executing it, has been established by a directive. Procedure definitions are detailed in Section 9.4 and function definitions are detailed in Section 9.5.

- Parameter data identifiers are specified in the parameter-list of the routine declaration so they can serve as vehicles to receive, be referenced, or transmit actual argument data when the routine is invoked. Parameter identifiers are not visible to outer enlosing blocks of the routine. They are visible to the body of the routine block and to any nested routines declared within the routine. There are four kinds of parameters: variable, value, formal procedure, and formal function. See Section 9.6.1.

- A routine invocation calls the routine into execution; activating the parameters and local data of the routine block, transfers execution control to execute the body of the routine block, after passing actual data arguments to the parameters of the routine. See Section 9.6.2 on routine invocation.

- Actual argument data or routine names are passed to the routine upon invocation. There are four kinds of arguments, respective to the kinds of parameters. They are variables, expressions, procedure-argument names and function-argument names. See Section 9.6.3 on argument specification.

- Compatibility of parameter declarations in a routine declaration and the arguments specifications in a routine invocation must be enacted by the programmer and is enforced by the compiler. See argument to parameter type-compatibility rules in Section 9.6.5.

- The environment of an invocation determines which set of data, that identifiers are referring to, as currently viewable in scope, and currently existent due to local routine and recursive calls, is actually passed to and received from a routine at its invocation. See Section 9.6.6.

- Procedures or functions can be nested within a routine definition. See Section 9.6.7 on nesting routines.
- A procedure or function can recursively call itself. See Section 9.6.8 on recursion.

### 9.1.3 Routines and Modules External to the Main Program

Other units of executable object code, aside from the main program, can be separately compiled (not necessarily by the Pascal compiler) and linked to the main program at task establishment time. This Pascal implementation provides several mechanisms for a main program to incorporate those other units within its scope; i.e., make their names referenceable and communicate through an interface defined by parameter-lists. Two actions are required to perform this function:

- the main program unit to which external code is to be linked must declare the name of the external routine and the interface to it through PROCEDURE or FUNCTION declarations (see Sections 9.3, 9.4, and 9.5). The interface is programmed through compatible parameter-lists, for modules (see rule 7 of Section 9.6.5), or adherence to Pascal linkage conventions for CAL routines (see Section 10.7), or Pascal-FORTRAN linkage conventions (see Section 10.8). Argument specifications in invocations of external procedure/function then must also be compatible to the foregoing parameter-list declaration (See rules 1 through 7 of Section 9.6.5).
- if the external routine is to be written in Pascal and compiled by the Pascal compiler, it must be encoded so the compiler will recognize that it is to be referenceable from another compilation unit. For this purpose, a compilation unit is available called a MODULE (see Section 9.2), so the compiler can differentiate whether it is compiling the main program or a module program unit and can generate appropriate linkage. Additional programming considerations are given in Section 9.2.2.

Within a main program or a module, the declaration of an external routine informs the compiler that the block associated with that routine will be provided at task establishment time and it is to generate the necessary object code for linkage to the code for that block.

Routines are made known as external in the source by replacing the routine block with an appropriate directive, either EXTERN or FORTRAN, described below.

A routine declaration that contains the EXTERN directive makes the routine name visible within its immediately enclosing block containing the declaration. The parameters in the declaration heading serve to specify the number and type of arguments expected when this routine is referenced. The compiler generates appropriate code calling sequences to permit linkage of a module or any other properly prepared external code body that conforms to the linkage conventions of this implementation. There is no cross compilation-unit type-checking in effect, so the user must assure that compatible parameter-lists are declared to effect proper interface linkage to MODULEs, or adhere to Pascal linkage conventions for CAL written external routines declared with EXTERN.

A routine declaration that contains the directive FORTRAN makes the routine name visible within its immediately enclosing block containing the declaration. The parameters in the declaration heading serve to specify the number and type of arguments expected when this routine is referenced. The compiler generates appropriate code calling sequences that permits linkage of a subroutine compiled by FORTRAN VII or any routine available in the mathematical functions of the FORTRAN VII RTL. There is no cross compilation-unit type-checking in effect, so the user must assure that appropriate parameter-lists are declared to effect proper interface linkage.

## 9.2 MODULES

A Pascal externally compiled routine, called a module, whose object code is to be linked to the main program at task establishment time is formally defined by the syntax:

Module

```
    ----> prefix --->|
    ^                |
    |                v
-------------------------> module-heading ---> block ---> .
```

A module permits externally compiled Pascal code to be referencable from the main program or even from another module as a procedure or a function. This capability is useful in large applications that are advantageously programmed in a modular fashion.

A module compilation-unit consists of an optional source prefix of declarations, its module-heading, followed by the module block, and concluded by a period, its terminating full stop.

The module header must be preceded by the Perkin-Elmer predefined Prefix if the additional prefix language extensions provided by the predefined Prefix are referenced (see Chapter 10). The user also can also write a prefix, so that if a main program were compiled with a specific prefix, its associated modules also would require compilation with that prefix if the modules were to have available any common constant, type, or routine declarations. Declaring new prefix routine headers declares those routines as EXTERN external routines.

The definition of the module-heading (except for the keyword, MODULE) is analogous to that of either the procedure-heading or the function-heading, depending on whether the external module is serving as an external procedure or an external function, respectively. A module-heading of a module which is a function, must additionally specify a function-value type-identifier. The syntactical structure of the parameter-list is also different for an external module than for internal routines in that a module parameter-list cannot contain a formal routine parameter declaration. Procedure and function names cannot be passed as arguments to external modules.

Also, whereas the module-heading construct includes the semicolon that separates it from its block, the procedure-heading and function-heading syntax exclusively define the semicolon which separates the heading from its block as they may also be used to declare formal routine parameters.

### 9.2.1 Module-Heading

A module-heading defines the module identifier and its parameter-list, if parameters are to be defined. The module-heading also must define a function-value type-identifier if the module is designed and to be referenced as a function, as opposed to a procedure, from the calling program unit. The syntax of the module-heading is:

<u>Module-Heading</u>

```
                                      ----------------------->|
                                      ^                       |
                       module-        |                       v
->MODULE->identifier-->parameter-list--> : -->type-identifier-->;
```

The identifier is the name of the module and can be used in procedure-call statements or function references in the program to call the module when the module identifier has also been declared in a corresponding procedure or function routine EXTERN declaration in the main program.

The format of a module-heading is the keyword, MODULE, followed by the identifier name of the module, optionally followed by a module-parameter-list (see graph below), optionally followed by a colon and function-value type-identifier (required if the module is a function), and separated from subsequent code with a semicolon. The colon and type-identifier of the module's function-value is used only if the module is designed, declared and referenced as a function in the calling program unit. The colon and function-value type-identifier cannot be present if the module is designed and declared and referenced as a procedure from the main program.

The syntax of the module-parameter-list is:

Module-Parameter-List

```
 ^---------------------------------------------------------->|
 |                                                           |
 |   ^----->|                 ^------>|                      |
 |   |      |                 |       |                      |
 |   |      V                 |       V                      V
-->(--->VAR--->identifier-->:--->UNIV--->type-identifier--->)--->
    ^        ^           |                               |
    |        |<--  , <---V                               |
    |                                                    |
    |<---------------------- ; <-------------------------V
```

A module parameter-list, when present, defines the module parameter data by identifiers and their data types with type-identifiers. These definitions determine on which parameters the module can operate, and import actual data into either the value or variable parameters and also export changed values through those that are variable parameters. Each parameter is specified by its parameter identifier and its type identifier following the colon.

The format of the module-parameter-list is a list of parameter declarations, each separated by a semicolon, with the entire list enclosed in parentheses. The format of a parameter declaration in a module-parameter-list is one or more parameter identifiers, each separated by a comma, followed by a colon and the type-identifier of the preceding parameter identifiers. These are optionally preceded by the keyword, VAR, when the parameter(s) represent a variable (to which the module can export a changed value). The parameter identifiers not preceded by the keyword, VAR, are those through which the module can import values but remain unchanged upon exit; i.e., value parameters. The colon and the type-identifier which follows either one or more parameter-identifiers defines the data type of the preceding parameters. The type-identifier can optionally be preceded by the keyword, UNIV, when the parameter identifier(s) are of universal type, meaning the data-type checking is to be relaxed when passing actual argument data to UNIV parameters. UNIV typed parameters may receive an argument of the same size. With UNIV, only the argument-to-parameter compatibility of datum sizes is checked, not the actual data types. The entire sequence of either an individual or group value parameter declaration, or an individual or group VAR declaration, including their type-identifier, is repeatable after the semicolon separator, and the entire parameter-list ends with a right parenthesis.

A module is a Pascal routine compiled separately from the main program. The compiler generated object code of a module can be executed only by linking it to a main program. Declaring the module identifier as a procedure or a function EXTERN directs that this linkage is required by the main program. A module that has been designed and coded to be linked to a main program, may then itself become a calling program unit and declare other EXTERN modules. Communication of data between both the main program and the module, whether the module is referenceable as a procedure or function, occurs through the corresponding parameter-lists of the external module heading and internal procedure/function headings coordinated with the argument list of the procedure call or function reference.

For details on parameters refer to Section 9.6.1. The specific difference between a module parameter-list and a routine parameter-list is that formal procedures and formal functions are not allowed in the definition of a module parameter-list. That is, the names of procedures or functions cannot be passed as arguments to modules.

### 9.2.2 Module Declaration, Linkage Declaration, and Invocation

The separately compiled Pascal code of a module is headed by a module-heading defining its identifier and parameters followed by a block and concluding with a period. In both of the procedure or function general formats below, note that within a block the keywords LABEL, CONST, TYPE, and VAR (and semicolons) are not to be present if no definitions follow, whereas the keywords BEGIN and END must be present to form the body of the module.

The general form of a module serving as an external Pascal procedure is depicted:

```
{ Optionally precede the module-heading with any necessary
  TYPE predeclarations used within the heading and/or
  optionally include the predefined Prefix }

MODULE NAME ( list of parameters with their types ) ;
  LABEL {define local statement-labels, if any} ;
  CONST {define local constants,if any} ;
  TYPE  {define local types,if any} ;
  VAR   {define local variables,if any} ;
  {optionally define any local (nested) routines}
  BEGIN
          { Body of module }
  END .
```

The general form of a module serving as an external Pascal function is depicted as:

```
    { Optionally precede the module heading with any necessary
      TYPE  prefix declarations  used  within the heading and/or
      optionally include the predefined Prefix }

    MODULE NAME(list of parameters and types):function type;
      LABEL {define local statement-labels, if any} ;
      CONST {define local constants,if any} ;
      TYPE  {define local types,if any} ;
      VAR   {define local variables,if any} ;
      {optionally define any local (nested) routines}
      BEGIN
      .           { Body of module }
      .
      .
      NAME := expression ;  {assign result to value of function}
      END .
```

If the module is designed and referenceable as an external procedure, the module identifier and parameter-list also must be declared [usually in the routine declarations part of a block] in the calling program unit with a procedure-heading, followed by a semicolon, the keyword EXTERN, and another semicolon (see Section 9.4 and 9.5). If the module is designed and can be referenced as an external function, the module identifier, parameter-list, and function value type-identifier similarly must be declared [usually in the routine declarations of a block] in the calling program unit with a function-heading followed by a semicolon, the keyword EXTERN, and semicolon (see Section 9.4 and 9.5). When the module has its associated EXTERN declaration available in the routine-declarations part of a program block, its identifier is referenceable within the scope of that declaration, in a procedure-call statement or a function reference detailed in Section 9.6.2. Also, for prefix external routine declarations, see Section 9.1. The format of a routine invocation is:

Procedure-Call or Function-Reference

```
---> identifier ------------------------->
                 |                      ^
                 |                      |
                 v--> argument-list-->|
```

When invoking a module, the identifier is the name of the declared external module and the argument-list is a list, enclosed in parentheses, specifying the actual data to be passed to the module's parameters. Arguments are specified in a one-to-one correspondence to the module's expected set of parameters defined in the parameter-list of both the external module heading and associated internal procedure/function-heading EXTERN declaration statement. Refer to Section 9.6.3.

The compiler checks for argument compatibility to the parameter-list in the internal procedure/function-heading EXTERN declaration. No cross compilation-unit type-checking is performed between the parameter-lists of the associated procedure/function-heading EXTERN declaration and the external module-heading. The user must assure this correspondence as the compiler is performing separate compilations of the main program, or calling compilation units, and the called modules. Inaccuracies might result in run time errors.

Uniquely identify external module (procedure or function) identifiers within the first eight characters because only the first eight characters are significant to the operating system link loader and when OS/32 Link is resolving external references. That is, the module name identifier, may be greater than eight characters, but only the first eight characters are used to form an actual extern/entry object label used for external object linkage.

An example of a main program with several external modules performing tasks in segments follows. For each external module serving as a procedure, there is an associated EXTERN procedure declaration within the main program. In a system referencing external functions, for each external module serving as a function, there would be an associated EXTERN function declaration within the main program.

When external files are to be visible and the file-variables are to be passed to an external module, they must be declared as variable (VAR) parameters in both the module-heading and the routine external declaration.

**Example:**

```
PROGRAM SYSTEM3 (FILE0,FILE1,FILE2);
  CONST ...
  TYPE  ...
  VAR   FILE0,FILE1,FILE2:TEXT;
  PROCEDURE SEGMENT1 (VAR FILE0,FILE1,FILE2:TEXT); EXTERN;
  PROCEDURE SEGMENT2 (VAR FILE0,FILE1:TEXT); EXTERN;
  PROCEDURE SEGMENT3 (VAR FILE1,FILE2:TEXT); EXTERN;
  BEGIN
    .
    .
    .
    SEGMENT1(FILE0,FILE1,FILE2);
    SEGMENT2(FILE0,FILE1);
    SEGMENT3(FILE1,FILE2);
    .
    .
    .
  END.
```

Then each separate and individually compilable module would respectively be headed with the following module headers:

```
    MODULE SEGMENT1(VAR FILE0,FILE1,FILE2:TEXT);

    MODULE SEGMENT2(VAR FILE0,FILE1:TEXT);

    MODULE SEGMENT3(VAR FILE1,FILE2:TEXT);
```

In this case, since FILE1 is an external file to be passed to each module as a VAR variable file-variable, it can be written to or read from in the body of either the main program or any of the modules.

Example:

```
    RESET (FILE1);              or     REWRITE(FILE1);
    READLN (FILE1,DATA_REC);           WRITELN(FILE1,DATA_REC);
```

Refer to Chapter 8 for further details on files and I/O.

When the previously described system uses external files in both the program and modules, and the files and their file-variable identifiers are of a type other than TEXT, file-variable identifiers must be declared in the module-heading with their type. A prefix of at least the TYPE declarations is necessary so the file-variable parameter type is defined prior to the module-heading. A similarly structured main program that addresses more complex structured files can be depicted:

```
    PROGRAM  SYSTEM2(FILE0,FILE1,FILE2);
      TYPE NAMEARRAY = ARRAY[1..19] OF CHAR;
      TYPE COMPONENTS = RECORD
                          NAME : NAMEARRAY;
                          IDNUM : INTEGER;
                        END;
           ACCOUNTS   = RECORD
                          IDNUM : INTEGER;
                          COMPENS : REAL;
                        END;
           DEPENDANTS = RECORD
                          IDNUM : INTEGER;
                          DEPNUM : INTEGER;
                        END;
          COMPS = FILE OF COMPONENTS;
          ACCTS = FILE OF ACCOUNTS;
          DEPS  = FILE OF DEPENDANTS;

      VAR FILE0:COMPS;
          FILE1:ACCTS;
```

```
    FILE2:DEPS;

PROCEDURE SEGMENT1(VAR FILE0: COMPS;
                   VAR FILE1: ACCTS;
                   VAR FILE2: DEPS); EXTERN ;

PROCEDURE SEGMENT2(VAR FILE0: COMPS;
                   VAR FILE1: ACCTS); EXTERN ;

PROCEDURE SEGMENT3(VAR FILE1: ACCTS;
                   VAR FILE2:DEPS) ; EXTERN ;
BEGIN
  .
  .
  .
SEGMENT1(FILE0,FILE1,FILE2);      {module invocation}
SEGMENT2(FILE0,FILE1);            {module invocation}
SEGMENT3(FILE1,FILE2);            {module invocation}
  .
  .
  .
END.
```

Then each of the separate and individually compiled modules would be headed with the following module-headings after preceding each module-heading with at least a prefix that defines the type-identifiers being referred to in the module-heading parameter-list.

A user-written prefix consisting of the following TYPE declarations should precede each example module-heading for SEGMENT1, SEGMENT2, and SEGMENT3 prior to its separate compilation, even under a batch compilation:

```
TYPE NAMEARRAY = ARRAY [1..19] OF CHAR;
TYPE COMPONENTS = RECORD
                    NAME : NAMEARRAY;
                    IDNUM : INTEGER;
                  END;

    ACCOUNTS =    RECORD
                    IDNUM : INTEGER;
                    COMPENS : REAL;
                  END;

    DEPENDANTS = RECORD
                    IDNUM : INTEGER;
                    DEPNUM : INTEGER;
                  END;

    COMPS = FILE OF COMPONENTS;
    ACCTS = FILE OF ACCOUNTS;
    DEPS  = FILE OF DEPENDANTS;
```

The following external module-headings refer to the above types by type-identifiers within the module parameter-list.

```
MODULE SEGMENT1(VAR FILE0:COMPS;VAR FILE1:ACCTS;
                VAR FILE2:DEPS);


MODULE SEGMENT2(VAR FILE0:COMPS; VAR FILE1:ACCTS;
                VAR FILE2:DEPS);


MODULE SEGMENT3(VAR FILE1:ACCTS; VAR FILE2:DEPS);
```

No prefix is needed if neither the module-heading nor the block of the module makes any references to prefix identifiers afforded by the predefined Perkin-Elmer Prefix or if the module-heading does not use user-specified identifiers for its parameter type-identifiers which would require the presence a user-written prefix. Note that SEGMENT3 could have used a different prefix due to its module-heading not declaring FILE0 and not referencing COMPS. See Section 1.10 for sample examples on linking modules.

When a module uses both the predefined Prefix and also requires type-identifiers to type its parameters, those type-identifier definitions cannot follow the Prefix containing routine header declarations but must be placed ahead of the predefined Prefix or at least ahead of the routine header declarations of the Prefix. (See syntax of prefix above in Section 9.1).

For additional examples of modules and associative procedure declarations, see Section 9.4.3 on external procedures.

For additional examples of modules and associative function declarations, see Section 9.5.3 on external functions.

A subtle restriction regarding Pascal I/O is placed on the Pascal coding in a module, due to the fact that the module-heading does not allow a mechanism to declare whether the standard predefined Pascal text file identifiers are to be used or not. Because of this, the _implicit_ forms of the Pascal READ, WRITE, READLN, WRITELN, etc., statements are not available in a module. If the user wishes to program Pascal I/O to the textfiles INPUT and OUTPUT which have been declared in his main program header as external files and also do I/O to them in modules, then, INPUT and/or OUTPUT must be passed to VAR variable parameters of type TEXT listed in the module-parameter-list. Also, only the _explicit_ forms (where the file-variable is the I/O routine's first argument) of the Pascal I/O statements are available in modules. The user must explicitly code the name of the VAR parameter-identifier, chosen to represent INPUT, as a file-variable in the READ, READLN, etc. read statements; or explicitly code the name of the VAR parameter-identifier, chosen to represent OUTPUT, as a file-variable in the WRITE, WRITELN, etc., write statements.

## 9.3 ROUTINE-DECLARATIONS

Routines can be defined in the routine-declarations part of the declarations of any block, prior to its main compound statement. A routine-declaration declares the identifier of the routine and parameter data and, generally, the block that operates on them. A routine-declarations part has the syntax:

Routine-Declarations

```
       <---;<---procedure---
       |                   ^
       v                   |
--->---------------------------->
       ^                   |
       |                   v
       <---;<---function----
```

A routine can be a procedure or a function. Each procedure and/or function declaration is separated (or ended) with a semicolon. Optionally the entire routine-declaration part of a block can be nonexistent. A routine declaration becomes a routine definition when its block (or its externality directed) has become defined.

## 9.4 PROCEDURES

A procedure declaration generally consists of a procedure-heading that declares its identifying name and its list of expected parameters (if any), and a block that is to be executed when the procedure is called from another place in the program. A procedure also can be an external EXTERN procedure declaration, an external FORTRAN procedure declaration, or a forwardly defined FORWARD procedure declaration. In these cases, the block is replaced with a directive, EXTERN, FORTRAN, or FORWARD. The syntax of a Pascal procedure is a procedure-heading, a semicolon, and either a block or directive, as follows:

Procedure

```
                                  |---> EXTERN  --->|
                                  |                 |
                                  |---> FORTRAN --->|
                                  |                 |
---> procedure-heading ---> ; --->|--->  block  --->|--->
                                  |                 |
                                  |---> FORWARD --->|
```

The above procedure declaration is followed by a semicolon as defined by the routine-declarations syntax-graph in Section 9.3.

### 9.4.1 Procedure-Heading

A procedure-heading declares the procedure identifier name and its interfacing parameter-list (when one is intended). The syntax of the procedure-heading is:

Procedure-Heading

```
                               ^------------------>|
                               |                   |
                               |                   V
---> PROCEDURE ---> identifier ---> parameter-list ---------->
```

The format of a procedure-heading is the keyword, PROCEDURE, followed by the identifier of the procedure, optionally followed by a parameter-list. The procedure-heading can be used to declare/define a procedure in a routine-declarations part or to declare formal procedures as parameters inside a parameter-list.

The optional path omitting the parameter-list in the above syntax graph must be taken in those procedure-headings heading the full definition of a previously declared FORWARD procedure.

The identifier is the name of the procedure and can be used, (if visible in scope), in subsequent procedure-call statements to call the procedure. This parameter-list syntax is:

Parameter-list (Internal Procedures and Functions)

```
 ^------------------------------------------------------------>|
 |                                                             |
 |   /-------------> procedure-heading ------------------>|    |
 |   |-------------> function-heading ------------------->|    |
 |   |                                                    |    |
 |   | /--->|                   /---->|                   |    |
 |   | |    |                   |     |                   |    |
 |   | |    V                   |     V                   V    V
-->(---->VAR--->identifier-->:--->UNIV--->type-identifier-->)--->
     ^        ^           |                               |
     |        |<--- , <---V                               |
     |                                                    |
     |<--------------------- ; <--------------------------V
```

A parameter-list when present defines the procedure's parameter data by identifiers and their data types with type-identifiers. No data type is necessary for a formal procedure, but if it is to receive a routine name, whose routine definition expects an argument list, then the formal procedure heading must define a compatible parameter-list. These parameter definitions determine on which parameter identifiers the procedure can operate or reference in its code. Parameters are vehicles for actual data the procedure can import from or export to the place of its invocation. Parameters are declared in a parameter-list and are available for reference by the procedure in addition to other identifiers visible in its scope, such as: any existent global data; or any locally defined data in the routine's declarations section; or any data local to outer blocks. What is not visible to the routine are those identifiers of nested routines or of inner nested scopes. See Section 2.1.3 on scope.

The format of a procedure parameter-list is a list of parameter declarations, each separated by a semicolon, with the entire list enclosed in parentheses. A parameter declaration can be a routine parameter, (a procedure-heading or function-heading in the graph); or a value or variable parameter represented by an identifier and following a colon, an identifier of its data type. A value or variable parameter group declaration has one or more identifiers, each separated by a comma with the entire group optionally preceded by the keyword, VAR, when the group is variable parameters not value parameters. Variable parameters represent variables (to which the routine can return a value). The value or variable parameter identifier (or group) is followed by a colon, and this is followed by a type-identifier establishing the data type of the parameter-identifier(s) preceding the colon. The type-identifier optionally can be preceded with the keyword, UNIV, when the parameter identifier(s) are of universal type, meaning the data type-checking is to be relaxed when passing actual argument data in a routine invocation to UNIV parameters. UNIV type parameters may receive an argument of the same size. With UNIV, only the argument-to-parameter compatibility of datum sizes is checked, not the actual data types. See Section 9.6.1 for parameter declarations and definitions of value, variable, or formal routine parameters.

### 9.4.2 Procedure Definitions

The general procedure definition form within a declarations part of a block is depicted as follows. Within a block, note that the keywords LABEL, CONST, TYPE, or VAR and their separating semicolons are not to be present if no definitions are present. The keywords BEGIN and END must be present to form the body of a procedure. For an internal procedure definition, the block is defined after the procedure-heading, and separating semicolon:

```
PROCEDURE NAME1 ( list of parameters with their types ) ;
  LABEL {define local statement-labels, if any} ;
  CONST {define local constants,if any} ;
  TYPE  {define local types,if any} ;
  VAR   {define local variables,if any} ;
  {optionally define any local (nested) routines}
  BEGIN
        { Body of procedure }
  END ;
```

For an external module that will serve as a procedure:

```
PROCEDURE NAME2(list of parameters with their types);EXTERN ;
```

For an external FORTRAN subroutine that serves as a procedure:

```
PROCEDURE NAME3(list of parameters with their types);FORTRAN;
```

For a forward declaration of an internal Pascal procedure:

```
PROCEDURE NAME4(list of parameters with their types);FORWARD;
```

Later in the same declarations part, its block is defined:

```
PROCEDURE NAME4;
  LABEL {define local statement-labels, if any};
  CONST {define local constants, if any} ;
  TYPE  {define local types, if any} ;
  VAR   {define local variables, if any} ;
  {optionally define any local nested routines}
  BEGIN
          { Body of procedure }
  END;
```

Any Pascal procedure, to be callable from another place in the program, must be declared in a routine-declarations part of a block (and visible in scope) prior to its call. (External Prefix routine-declarations, globally visible in scope, are also callable.)

The declaration of an internal procedure can be a complete definition of the procedure consisting of the procedure-heading and its block, or the procedure-heading and a forward declaration. If the procedure definition is a forward declaration, it is first presented by the procedure-heading followed by the keyword FORWARD. If a routine is referenced before its block is completely defined, it must be introduced in this way. Later the routine can be completed by repeating its name, without its associated parameter-list, followed by its block.

If the keyword FORWARD appears in a procedure declaration, the declaration of the procedure name is visible in the block containing the declaration. When two procedures mutually reference each other, first one must be forwardly declared in this way to be available by declaration to the second. Later in the same block there must be a deferred definition of the first procedure with the same initial identifier. This allows indirect recursive programming. The two routines invoking each other this way are said to be mutually recursive. Refer to Section 9.6.8 for information on direct recursion.

If the directive EXTERN appears in a procedure declaration, the block containing its local declarations part and statement body will be provided by an external MODULE or other means; i.e., assembler level routine, which will be linked at task establishment time.

If the directive FORTRAN appears in a procedure declaration, the body of that routine will be provided by a Fortran-compiled subroutine or assembler-level procedure using Pascal-FORTRAN calling conventions, that will be linked at task establishment time.

Some examples of procedure definitions reflecting the use of the procedure-heading in program PROCDEFS are:

```
PROGRAM PROCDEFS;
  VAR CHER:CHAR;     {global variable declarations}
      L,M,N,P:INTEGER;  Q,R,S:REAL;

                   {internal procedure definition}

     PROCEDURE XYZ(VAR X:REAL;Y,Z:REAL);
       BEGIN
        X := Y+Z;
       END;

                   {external procedure declaration}

     PROCEDURE ABC(ARG1:CHAR);EXTERN;

                   {external FORTRAN declaration}

     PROCEDURE DEF(A,B:INTEGER;VAR C,D:UNIV INTEGER);FORTRAN;

                   {internal procedure forward declaration}

     PROCEDURE TEST1(PARAM:REAL);FORWARD;

                   {internal declaration referencing TEST1}

     PROCEDURE TEST2(PARAM:REAL);
       VAR K:REAL;
       BEGIN
         K := 1.0 + PARAM;
         TEST1(K);
         .
         .
         .
       END;
```

Later in the same routine-declarations part of the block, the procedure, TEST1, must be fully defined:

```
    PROCEDURE TEST1;
      VAR J:REAL;
      BEGIN
       .
       .
       .
       TEST2(J);
       .
       .
       .
      END;
```

These procedures can be referenced in the body of the main program with appropriate procedure-call statements, assuming the following global data definitions are available: VAR CHER:CHAR; L,M,N,P:INTEGER; and Q,R,S:REAL.:

```
    BEGIN       { main program PROCDEFS }
     .
     .          {assign values to CHER,L,M,N,P,Q,R,S}
     .
    XYZ(Q,R,S);
    ABC(CHER);
    ABC('A');
    DEF(L,M,N,P);
    TEST2(3.2);
    TEST1(5.46);
     .
     .
     .
    END.
```

Procedure XYZ is a simple procedure definition, importing a changeable variable parameter X, and two value parameters, Y and Z. Whichever argument is imported into XYZ for the parameter X, it will be exported with a changed value; i.e. X = Y+Z. The procedure invocation XYZ(Q,R,S); lists Q,R, and S as arguments to XYZ, and after execution, that procedure returns Q = R+S, and R and S remain unchanged. Although they could have been referenced, and had a local change of value as Y and Z within the routine XYZ, upon exit, R and S have the same values they had upon invocation of XYZ.

In the example procedure declarations, procedure ABC represents the declaration of an externally compiled Pascal module with a value parameter ARG1 of character-type. Because ARG1 is a value parameter, the second call on ABC sends a literal-constant character 'A' expression as an argument to ABC. Procedure DEF is an external FORTRAN routine with two value parameters, A and B; and two variable parameters, C and D.

Because the procedures, TEST1 and TEST2, reference each other and both are visible and can be referenced from the body of the block containing their declarations; i.e., the main program, one of them must be represented in a forward declaration so the procedure identifier and its parameter interface are available to the other. Once the second procedure, TEST2, is defined, the forwardly declared procedure, TEST1, can be given its full definition by repeating its name; i.e., procedure-heading without parameter-list, and defining its block. This permits mutual recursion. Refer to Section 9.6.8 for information on direct recursion in Pascal.

### 9.4.3 External Procedures

Any user written procedure that is externally compiled as a Pascal module must be declared in the routine-declarations part of a block (other than prefix routines) and be visible in scope to be callable. This declaration must consist of a procedure-heading, specifying an identifier name and corresponding set of parameters, a semicolon, the word EXTERN, and a semicolon.

Given a module that serves as a procedure, such as:

```
MODULE FINDMATCH (VALUE1:INTEGER;VAR ENTRY:BOOLEAN);
  VAR I:INTEGER; TABLE:ARRAY [ 1..50 ] OF INTEGER;
  BEGIN
    { Compute values for table, then: }
    ENTRY := FALSE;
    FOR I:=1 TO 50 DO
        IF VALUE1 = TABLE[I] THEN
           ENTRY:= TRUE
           ELSE
           BEGIN END;
  END.
```

In the routine-declarations part of a block of the calling program unit that will reference the module as an external procedure, FINDMATCH, it must be declared as an external procedure, such as:

```
PROCEDURE FINDMATCH(VALUE1:INTEGER;VAR ENTRY:BOOLEAN);EXTERN;
```

Any FORTRAN procedure externally compiled to be callable, must be declared in the routine-declarations part of a block. This declaration must consist of a procedure-heading, specifying an identifier name, and parameter-list (if any), a semicolon, and the keyword FORTRAN, and a semicolon. If an external subroutine, written in and compiled by FORTRAN were:

```
SUBROUTINE TRIANGLE(DATUM1,DATUM2,DATUM3)
REAL DATUM1
DOUBLE PRECISION DATUM2,DATUM3
DATUM1 = (DATUM2*DATUM3)/2
RETURN
END
```

Within the routine-declarations part of the block in which the external routine TRIANGLE is referenced or is to be visible, a declaration for the external FORTRAN procedure is specified as:

```
PROCEDURE TRIANGLE(VAR X1:SHORTREAL; X2,X3:REAL);FORTRAN;
```

Then the external FORTRAN procedure would be referenceable in a Pascal procedure-call statement, as in the body of the following:

```
PROGRAM LINKUP(INPUT,OUTPUT);
VAR AREA:SHORTREAL;
    BASE:REAL;
    HEIGHT:REAL;
PROCEDURE TRIANGLE(VAR X1:SHORTREAL; X2,X3:REAL);FORTRAN;
BEGIN
  .
  .
  .
  HEIGHT:=4.5;
  BASE:=60.8;
  .
  .
  .
  TRIANGLE(AREA,BASE,HEIGHT);
  WRITELN('AREA1 IS ',AREA);
  .
  .
  .
  TRIANGLE(AREA,7.3,8.1);
  WRITELN('AREA2 IS ', AREA);
  .
  .
  .
  READLN(BASE);
  READLN(HEIGHT);
  TRIANGLE (AREA,BASE,HEIGHT);
  WRITELN ('AREA3 IS', AREA);
END.
```

The FORTRAN data type DOUBLE PRECISION is equivalent to the

Pascal data type REAL; and the FORTRAN data type REAL is equivalent to the Pascal data type SHORTREAL.

Any assembler language procedure, to be callable from a Pascal program, must be declared in the routine-declarations part of a block, (other than prefix routines). This declaration must consist of a procedure-heading, specifying its identifier name, and parameters (if any), a semicolon, the keyword either FORTRAN or EXTERN, depending on whether the routine uses FORTRAN or Pascal calling conventions, respectively (see Section 9.4.2); and separating semicolon.

## 9.5 FUNCTIONS

A function is a routine that computes a value. The value e of a function f is defined by an assignment operation,

```
    f := e;
```

within the function block. Reference to f within an expression that is inside the definition of f, is interpreted as a recursive function call rather than a reference to a previously assigned value of f (refer to Section 9.6.8). For example, in an assignment statement within the block of the function-definition, such as:

```
    f := f + e;
```

The function-value type-identifier and the type of the expression, e, must be of "assignment-compatible" types, and the function-value type-identifier, (i.e., the types allowed for function-values) must be scalar: ordinal or user-defined enumerations, or real, or pointer types; not the structured-types.

A function declaration generally consists of a function-heading that declares its identifying name and its expected set of parameters, a function-value type-identifier, and a block or a directive which indicates that the block is elsewhere. A function-definition defines the block that is to be executed to obtain the function value when the function is referenced from another place in the program.

A "function" in Pascal can be an internal function with its block immediately defined following the "function-heading;" or an external EXTERN function declaration, an external FORTRAN function declaration, or a forwardly defined FORWARD function declaration. In these latter cases, the block is omitted and replaced with a keyword specifying where the function declaration being presented, has its block defined elsewhere. The formal syntax of a function in Pascal is a function-heading, a semicolon, and either a block or a directive, as follows:

Function

```
                                           |---> EXTERN  --->|
                                           |                 |
                                           |---> FORTRAN --->|
                                           |                 |
---> function-heading ---> ; --->|---> block   --->|--->
                                           |                 |
                                           |---> FORWARD --->|
```

The above "function" declaration is followed by a semicolon, as depicted in the graph of routine-declarations in Section 9.3.

### 9.5.1 Function-Heading

A function-heading declares the function name identifier and its parameter-list, when one is specified, and the type-identifier that determines the data-type of the function's value. The syntax of the function-heading is:

Function-Heading

```
                       ^-------------------------------------->|
                       |                                       |
                       |                                       V
->FUNCTION->identifier-->parameter list->:->type-identifier----->
```

The identifier following the word FUNCTION introduces the name of of the function and it is used in any subsequent function references to call or invoke the function. The call specifies actual data arguments passed to its parameter-list, if a parameter-list is defined for the function in its function-heading; otherwise when no parameter-list is defined, the call must not supply arguments. The parameter-list syntax is discussed in Section 9.4.1 (where its syntax-graph also depicts a non-existant "parameter-list"); and the contents of "parameter-list" are detailed in Section 9.6.1 on parameter declarations.

In the case of a function-heading having no parameter-list, the colon and function value type-identifier is also usually required in order to define the type of the function-value; (other than when a previously declared FORWARD function is becoming fully defined).

The optional path in the syntax graph of Function-Heading above, can only be taken, and must be taken, when coding the function-heading to introduce the full definition of a previously declared FORWARD function.

It is good programming practice, to only declare data parameters of a function to be value parameters, not variable parameters; but the language of Pascal does allow variable parameters to also be declared in a function parameter-list.

The format of the function-heading is the keyword, FUNCTION, followed by the identifier of the function, optionally followed by a parameter-list, followed by a colon. The colon is followed by a type-identifier which determines the data type of the value that the function will compute.

### 9.5.2 Function Definition

The general form of a function definition within a routine-declarations part of a block is depicted below. Note that within a block, the words LABEL, CONST, TYPE, or VAR, and separting semicolons are not to be present if no definitions are present; whereas the words BEGIN and END must be present to form the body of the function.

For an internal function definition, the block is defined after the function-heading, and separating semicolon:

```
FUNCTION NAME1(list of parameters and types):function-type;
  LABEL {define local statement-labels, if any} ;
  CONST {define local constants, if any} ;
  TYPE  {define local types, if any} ;
  VAR  {define local variables, if any } ;
  { define any local(nested) routine declarations, if any }
  BEGIN
  .         { Body of function }
  .
  .
  NAME1 := expression ;  {assign result to value of function}
  END ;
```

For an external module which will serve as a function reference:

```
FUNCTION NAME2(list of parameters and types):function-type;EXTERN;
```

For an external FORTRAN routine to serve in a function reference:

```
FUNCTION NAME3(list of parameters and types):function-type;FORTRAN;
```

For a function whose block will be forwardly defined:

```
    FUNCTION NAME4(list of parameters and types):function-type;FORWARD;
```

where later in the same declarations part, the function is defined by:

```
    FUNCTION NAME4;
      LABEL {define local statement-labels, if any};
      CONST {define local constants, if any} ;
      TYPE  {define local types, if any};
      VAR   {define local variables, if any};
      {optionally define any local nested routines}
      BEGIN
      .       { Body of function }
      .
      .
      NAME4 := expression ;  {assign result to value of function}
      END;
```

When a forward declaration receives its actual block definition, the parameter-list, colon and function type-identifier, must not be respecified.

Any user written Pascal function that is to be referenceable from another place in the program must be <u>declared</u> in a routine declarations part of a block (and visible in scope) prior to its reference. (External Prefix routine-declarations, globally visible in scope, are also callable.)

A function declared in the outermost block of the main program, is globally visible for reference over the entire program unit. A function declared in a routine, as a nested routine within a routine, is locally visible for reference to the body of that routine and to any routines on the same level of nesting in that routine, and to any routines nested within itself.

The declaration of an internal function can be a complete declaration consisting of the function-heading and its block, or it can be the function-heading and a forward declaration. If the function's block definition is a forward declaration, it is first presented by the function-heading, including its parameter-list, and semicolon, followed by the keyword, FORWARD, followed by another semicolon. If a function routine is referenced before its block is completely defined, it first must be introduced this way.

If the keyword FORWARD appears in a function declaration, the definition of the function name is visible in the block containing the declaration. Later in the same block, there must be a deferred declaration of the same function with the same name, followed by its defining block.

If the function is an external function, its block is replaced with either the directive EXTERN or FORTRAN.

If the directive EXTERN appears in a function declaration, the body of that function will be provided by an external MODULE or other means; i.e., assembler level routine using Pascal calling conventions, that will be linked at task establishment time.

If the directive FORTRAN appears in a function declaration, the body of that function will be provided by a FORTRAN-compiled, or assembler-level function using Pascal-FORTRAN calling conventions, such as a FORTRAN RTL math routine, that will be linked at task establishment time.

Some examples of function definitions, reflecting the use of the function-heading, are specified in program FUNCDEFS:

```
PROGRAM FUNCDEFS(INPUT,OUTPUT);
TYPE REALARRAY=ARRAY[1..200] OF REAL;    {type definitions}
VAR  I,LOW,HIGH:INTEGER;         {gobal variable declarations}
     TPOINTS:REALARRAY; AREA,VOLUME,SIDE,DATAX:REAL;
     SOLUTION:REAL;

                 {internal function definition}

FUNCTION TRULY_IN_RANGE(DATUM:REAL):BOOLEAN;
  CONST LOWER=0.0; HIGHER=5.6E6;
  BEGIN
     TRULY_IN_RANGE := (DATUM > LOWER) AND (DATUM < HIGHER);
  END;

                {external function declaration}

FUNCTION INTEGRAL(LO,HI:INTEGER;FARRAY:REALARRAY):REAL;EXTERN;

            {external function FORTRAN declaration}

FUNCTION DSIN(RADIAN:REAL):REAL;FORTRAN;

            {internal forward function declaration}

FUNCTION POLYNOMIAL(A,B,C,D:REAL;X:REAL):REAL;FORWARD;

FUNCTION CUBE(NUMBER:REAL):REAL;
  BEGIN
    CUBE := NUMBER*NUMBER*NUMBER;
  END;

{Later in the declarations, POLYNOMIAL must become defined:}

    FUNCTION POLYNOMIAL;
      BEGIN
       POLYNOMIAL := A*CUBE(X)+B*SQR(X)+C*X+D;
      END;
```

The above declared functions can be referenced; i.e., called into execution, in the body of the program with function references as follows:

```
    BEGIN   {main program FUNCDEFS}
     READ(DATAX);
     WHILE NOT EOF DO
       BEGIN
         IF TRULY_IN_RANGE(DATAX) THEN
            BEGIN
                  {Process DATAX}
            LOW:=TRUNC(DATAX);
            HIGH:=ROUND(DATAX);
            FOR I:=1 TO 200 DO
                 BEGIN
                 TPOINTS[I] := DSIN(DATAX);
                 DATAX:=DATAX - 0.005;
                 END;
            AREA := INTEGRAL(LOW,HIGH,TPOINTS);
            WRITELN(AREA);
            END;
         READ(DATAX)
       END;
     .
     .
     .
     VOLUME := CUBE(SIDE);
     SOLUTION := POLYNOMIAL(5.4,4.3,2.1,3.2,DATAX);
     .
     .
     .
    END.
```

### 9.5.3 External Functions

Any user written function externally compiled as a Pascal module must be declared in the routine-declarations part of a block (other than prefix routines) and be visible in scope to be referenceable. This declaration consists of a function-heading, (specifying its identifier name and corresponding parameter-list), a colon and function type-identifier), a semicolon, the word EXTERN, and another semicolon.

Given a module that will serve as a function, such as:

```
    MODULE CHECKRANGE(INPUTDATA:REAL;
                      VAR LOWRANGE,HIRANGE:REAL):BOOLEAN;
      BEGIN
        IF (INPUTDATA > LOWRANGE) AND (INPUTDATA < HIRANGE) THEN
           CHECKRANGE := TRUE
           ELSE
           CHECKRANGE := FALSE;
      END.
```

Then, in the routine-declarations part of the block in which the external function is referenced or is first to become visible in scope, it must be declared as an external function, such as:

```
    FUNCTION CHECKRANGE(INPUTDATA:REAL;
                        VAR LOWRANGE,HIRANGE:REAL)
                        :BOOLEAN;EXTERN;
```

This external function declaration associated with an external MODULE must assure that parameter-lists are compatible so the position and data types of each parameter match. The parameter identifiers need not be identical. The external module, CHECKRANGE, could be declared in a main program or calling program unit, as:

```
    FUNCTION CHECKRANGE(INDATA:REAL; VAR LOWEND,HIEND:REAL)
                        :BOOLEAN; EXTERN;
```

It is not good programming practice to declare variable parameters in any function-declaration but it is allowed by the language of Pascal. Usual concepts of functions would design them to receive argument data for parameters in the form of value parameters.

Any FORTRAN function externally compiled by FORTRAN VII, to be referenceable within a Pascal program, must be declared in the routine-declarations part of a Pascal block. This declaration must consist of a function-heading, specifying its identifier name, parameters (if any), a colon and its function type-identifier, a semicolon, the keyword FORTRAN, and a semicolon.

Any external assembler language function to be referenceable from another place in the main program also must be declared in the routine-declarations part of a block, (other than prefix routine-declarations). This declaration must consist of a function-heading specifying its identifier name, parameters(if any), colon and function type-identifier, followed by a semicolon, and either FORTRAN or EXTERN, depending on whether the routine uses FORTRAN or Pascal calling conventions, respectively; and separating semicolon. An example of assembler-level written routines utilizing the FORTRAN linkage conventions are the FORTRAN RTL math routines.

Several math functions are provided by the FORTRAN VII RTL to Pascal by this method. Refer to Section 3.5.9 for such external FORTRAN declarations necessary for accessing the standard RTL math routines as external FORTRAN functions. See other external function declaration examples in Section 9.5.2.

## 9.6 PROGRAMMING ROUTINES

There are several concepts to be detailed for programming routines in Pascal, such as parameter declaration, routine invocation, argument specification, argument passing to parameters, the requirements for compatibility of parameters and arguments, the routine invocation environment, nesting of routines, and routines recursively calling themselves. These concepts are individually discussed in the remaining sections.

### 9.6.1 Parameter Declaration

The programmer specifies the parameter data for a routine. This is programmed by listing their user-specified identifiers and their type-identifiers in the parameter-list within parentheses immediately following the name of the routine written after the words PROCEDURE or FUNCTION, at the routine's declaration. The syntax of a parameter-list is:

Parameter-List (Internal Procedures and Functions)

```
 ^------------------------------------------------------------>|
 |                                                             |
 |   /-------------> procedure-heading ------------------>|    |
 |   |-------------> function-heading ------------------->|    |
 |   |                                                    |    |
 |   | /--->|                    /---->|                  |    |
 |   | |    |                    |     |                  |    |
 |   | |    V                    |     V                  V    V
-->(----->VAR--->identifier-->:--->UNIV--->type-identifier-->)--->
    ^          ^           |                                |
    |          |<--- , <---V                                |
    |                                                       |
    |<----------------------- ; <---------------------------V
```

The syntax of a module-parameter-list used in both the MODULE header and associative external declarations is:

Module-Parameter-List (External Procedures or Functions)

```
 ^----------------------------------------------------------->|
 |                                                            |
 |   ^----->|                 ^------>|                       |
 |   |      |                 |       |                       |
 |   |      V                 |       V                       V
-->(--->VAR--->identifier-->:--->UNIV--->type-identifier--->)--->
    ^         ^          |                                |
    |         |<-- , <---V                                |
    |                                                     |
    |<---------------------- ; <--------------------------V
```

Within the parameter-list one or more entities as declared parameters of the routine. These parameter declarations are visible within the block associated to the routine being declared with this parameter-list. That is, the parameter identifiers may be used and referenced in the block to which they belong, established by where are declared. When programming the block of the routine, the parameters may be assumed to come into existence with values being passed in each invocation with actual argument data or actual routine names.

There are four kinds of Pascal parameters called variable parameters, value parameters, or formal routine parameters, either a formal procedure parameter or a formal function parameter. Variable parameters are detailed in Section 9.6.1.1 below. Value parameters are detailed in Section 9.6.1.2 below. Formal routine parameters are detailed in Section 9.6.1.3 below.

Note the difference in parameter-list syntax above for external routines and internal routines. Formal routine parameters, as represented by procedure-heading and function-heading in internal routines are not allowed in module-parameter-lists.

Also, once a user-specified name is given to a value or variable parameter, no duplicates can be listed. That is, no two parameters can be specified by the same identifiers.

In any parameter-list, the keyword UNIV provides a mechanism to partially override the compiler type-checking of the argument and parameter correspondence of either value or variable parameters. A parameter is of the universal type if its type-identifier is preceded with the keyword UNIV. The word UNIV suppresses compatibility checking of argument types being passed to the parameters. An argument of type T1 is compatible with a parameter of universal type T2 if both types are represented by the same number of storage locations: e.g., INTEGER and SHORTREAL data types. Inside the given routine, the parameter is considered to be of nonuniversal type T2. Outside the routine call, the argument is considered to be of nonuniversal type T1.

The parameters and any local variables declared within a routine are considered temporary variables and exist only while it is being executed.

Since a parameter-list can contain a procedure-heading or a function-heading, to declare a formal routine parameter, note that the procedure-heading or function-heading can again contain another parameter-list; such as when a dummy formal routine parameter declaration declares another dummy formal routine as one of its parameters.

#### 9.6.1.1 Variable Parameters

A variable parameter represents a variable through which the routine can not only receive an argument variable but also return a changed value in the argument variable. Its parameter identifier comes into visibility to the block of the routine in whose parameter-list it is being introduced. At its declaration within a parameter-list it is prefixed with the keyword, VAR. The variable parameter group declaration:

```
    VAR v1, v2, ... vn : T
```

where v1, v2, ... vn are variable parameters, and T is a type-identifier, is equivalent to:

```
    VAR v1:T; VAR v2:T;..., VAR vn:T
```

In a variable parameter group declaration, the identifier following the colon must be a type-identifier that establishes the data-type of any preceding variable parameter identifiers that were preceded by the keyword VAR.

The argument passed to a variable parameter must be a variable, as specified by its variable-selector (See Section 5.4). That is, the argument in a corresponding position to a variable parameter cannot be an expression, nor can it be a routine name.

To be compatible, the argument variable passed to a variable parameter must be "identical" in type to the type-identifier of the parameter. This is, of course, unless the variable parameter declaration contains the word UNIV preceding the parameter type-identifier. Then, in the case of UNIV, the argument variable passed to the variable parameter may be a variable of the same internal storage size.

### 9.6.1.2 Value Parameters

A value parameter represents the name of a value to be imported into a routine at the time the routine is called into execution. The user-specified name given to such an expected value is a value parameter identifier. The value parameter identifier comes into visibility at the point of its declaration, and can be referenced within the block of the routine it is being declared for, and any of its inner nested blocks. The value parameter identifier is not visible in scope to any outer blocks for reference. The user must specify the data type of the value parameter identifier with a type-identifier within the parameter-list, where it is being introduced.

An actual argument expression value will be passed to the value parameter identifier from the argument-list of a routine invocation when the routine is called into execution. If a variable is passed to a value parameter, the routine may change the value of the value parameter within its block but the variable passed, upon return to the caller, is unchanged in value. That is, the value of the variable so passed is not changed by the routine upon exit, even if the routine changes the value of the value parameter identifier.

A value parameter is not preceded with the keyword, VAR, which is only used to declared parameters as variable parameters. The value parameter group declaration:

```
    v1,v2,...,vn:T
```

where v1,v2,...,vn are value parameter identifiers, and T is a type-identifier, is equivalent to:

```
    v1:T;v2:T,...,vn:T
```

In a value parameter group, the identifier following the colon defines the data type of each identifier in the preceding identifier list. The preceding list of value parameter identifiers are separated by commas (if more than one). Whether the parameters defined by that group are value parameters or variable parameters is determined by the absence or presence keyword VAR, respectively.

The argument passed to a value parameter must constitute a defined expression. It cannot be a routine name, but it can be a variable, because an expression can contain a defined variable. However, the variable so passed to a value parameter is only used as a read-only entity for its current value and any change made to the value parameter is not exportable to the passed in variable.

To be compatible in type, in Pascal, the expression passed to a value parameter need not be "identical" in type to the type-identifier of the value parameter; but must be at least "assignment compatible" to the type-identifier established for the value parameter. There are two exceptions to this rule in this implementation of Pascal. One is, of course, when the value parameter declaration contains the word UNIV preceding the parameter type-identifier. Then, in the case of UNIV, the argument expression passed to the value parameter may be of a type different from the parameter type-identifier as long as its internal storage size is the same. The other exception is that variable-length strings may be passed to value parameters of any string-type. See rule 2 of type-compatibility of arguments to parameters in Section 9.6.5 below.

#### 9.6.1.3 Formal Routine Parameters

A formal routine parameter is a dummy routine name and specified dummy parameter-list. It has no other local declarations nor a body of its own. The dummy name and parameter-list serves as a placeholder and referenceable identifier within the routine for which it is being declared. A formal routine parameter declaration can be either a formal procedure or a formal function.

A formal procedure is declared by listing its procedure-heading within a parameter-list. A formal function is declared by listing its function-heading within a parameter-list.

Formal routines have no actual declarations part nor statement body themselves. They are simply given a nomenclature with which they can be referenced and an optional parameter-list that outlines the acceptable interface of the formal routine. An actual routine name which has a compatible parameter-list will be passed to the dummy formal routine parameter declaration, upon invocation of the routine it belongs to. A formal routine name, which has had either directly or indirectly an actual routine name passed to it, can be passed to another formal routine parameter. A predefined Pascal routine name cannot be passed to a formal routine parameter, neither the predefined function-identifiers nor the predefined procedure identifiers.

Inside the routine, for which formal routines are declared parameters, when the dummy formal routine in invoked in a routine-invocation, the actual routine that has been passed along to the dummy formal routine parameter is actually executed.

A procedure-heading inside a parameter-list defines a formal procedure parameter declaration. A function-heading inside a parameter-list defines a formal function parameter declaration.

Parameter identifiers within the parameter-lists of formal routine parameter declarations are visible only within that dummy parameter-list as placeholders and as they are not visible outside the dummy parameter-list, any user-specified identifier may name them, as they are never referenced again, only passed another compatibly typed parameter-list in its entirety.

### 9.6.2 Routine Invocation

Once a routine is declared by a declaration, whether a procedure or function, it can be referenced by either a procedure-call statement or a function-reference, respectively.

As the routine-declaration makes the routine name visible in scope from its point of introduction, to itself, its nested routines, and its immediately enclosing block; a routine-invocation of that routine name may occur in itself (from the body of its own block), from the body of its nested routines, or from the routines declared on the same procedure level of its immediately enclosing block or from the body of its immediately enclosing block.

In a routine-invocation a user-specified argument-list supplies the actual data to be operated on by the routine. The syntax of either type of routine invocation is:

Procedure-Call or Function-Reference

```
---> identifier ---------------------------->
                  |                        ^
                  |                        |
                  V---> argument-list --->|
```

The identifier is the name of a predefined Pascal procedure or a user-written declared procedure if the routine-invocation is that of a procedure with a procedure-call statement. If the the routine-invocation is a function-reference within an expression, the identifier is the name of a predefined Pascal function or a user-written declared function.

The argument-list is a list enclosed in parentheses, specifying the actual data to be passed to the routine's parameters. Arguments are listed in a one-to-one correspondence to the routine's expected set of parameters defined in the parameter-list of the routine declaration. If we had the following routine declaration:

```
PROGRAM SAMPLE1;
  VAR A,B,C,X:INTEGER;                {Global variables}
  PROCEDURE MEDIAN;  {Example declaration without parameters}
    BEGIN
      X:=(A+B+C) DIV 3;
    END;
```

then within the body of the block containing that declaration, the routine could be invoked:

```
BEGIN
  .
  .     {Example of routine invocation without argument-list}
  .
MEDIAN;
  .
  .
  .
END.
```

If the previous example were further generalized to be able to perform a median operation at several different times:

```
PROGRAM SAMPLE2;
  VAR A,B,C,D,X:INTEGER;
  PROCEDURE MEDIAN(VAR M:INTEGER;J,K,L:INTEGER);
    BEGIN
      M := (J+K+L) DIV 3;
    END;
```

then the routine invocations must specify actual data arguments in correspondence to the parameter-list of the procedure declaration as in the following example code:

```
BEGIN  {begin main program SAMPLE2}
.
.
.
A:= 3;
B:= 4;
C:= 5;
D:= 6;
.
.
.
MEDIAN(X,A,B,C);      { Routine invocation with arguments }
.
.
.
       { X now contains the median of A,B,C }
.
.
.
MEDIAN(X,B,C,D);      { Routine invocation with arguments }
.
.
.
       { X now contains the median of B,C,D }
.
.
.
END.   { end main program SAMPLE2 }
```

Similarly, the process of computing a median value of three values could be programmed as a function instead of a procedure. Then the routine invocation would be in the form of a function-reference within an expression. In the following example, MEDIAN is declared as a function in program SAMPLEFUNC, and referenced within an expression of the assignment statements as a function:

```
    PROGRAM SAMPLEFUNC;
      VAR A,B,C,D,X:INTEGER;
      FUNCTION MEDIAN(J,K,L:INTEGER):INTEGER;
        BEGIN
          MEDIAN := (J + K + L ) DIV 3;
        END;
      BEGIN   {begin main program SAMPLEFUNC}
        .
        .
        .
        A:= 3;
        B:= 4;
        C:= 5;
        D:= 6;
        .
        .
        .
        X:=MEDIAN(A,B,C);    {X now contains median of A,B,C}
        .
        .
        X:=10+MEDIAN(B,C,D); {X now contains 10+ median of B,C,D}
        .
        .
        X:=MEDIAN(24,95,33); {X now contains median of numbers}
      END.     {end of main program SAMPLEFUNC}
```

### 9.6.3 Argument Specification

When a procedure is invoked by a procedure-call statement, or a function is invoked by a function-reference in an expression, arguments are specified by listing them in an argument-list following the name of the routine.

The syntax of an argument-list, which may be part of a routine-invocation, is:

<u>Argument-list</u>

```
  ^--------------------------------------------------->|
  |                                                    |
  |            |---> variable-selector --->|           |
  |            |                           |           |
  |            |---> procedure-argument -->|           |
  |            |                           |           V
-----> ( ----->|                           |-----> ) ----->
           ^   |                           |  |
           |   |---> function-argument --->|  |
           |   |                           |  |
           |   |-------> expression ------>|  |
           |                                  |
           |<--------------  ,  <-------------V
```

Within a procedure-call statement or function-reference, a list of arguments (enclosed in parentheses) is specified in a one-to-one correspondence to the declared parameter-list of the routine being invoked.

In Pascal, there are four kinds of arguments, respective to the four kinds of parameters. An argument list may contain a variable, expression, procedure-argument, or function-argument.

Correspondingly positioned and "identically" typed, a variable may be passed to a VAR variable parameter. See Section 9.6.1.1. The argument variable is specified by a variable-selector (see Section 5.4).

Likewise, correspondingly positioned and "assignment-compatible" typed, an expression may be passed to a value parameter. See Section 9.6.1.2.

Also, correspondingly positioned, and compatibly typed in regard to parameter-lists, an actual routine name, either a procedure-argument or function-argument may be passed to a formal procedure or formal function parameter, respectively. See Section 9.6.1.3.

See Section 9.6.5 for type-compatibility rules between arguments and parameters, and particularly rule 7 for parameter-list compatibility rules.

The syntax of a procedure-argument is:

Procedure-argument

```
    ---> procedure-identifier --->
```

The identifier is the name of a previously declared procedure, whose declaration is visible in scope to the routine invocation. A procedure-argument in an argument-list positionally corresponds to a formal procedure parameter in a parameter-list.

The syntax of a function-argument is:

Function-argument

```
    ---> function-identifier --->
```

The identifier is the name of a previously declared function, whose declaration is visible in scope to the routine invocation. A function-argument in an argument-list positionally corresponds to a formal function parameter in a parameter-list. An actual function name so passed to a formal function, must be of a function who has compatible parameter-list to the parameter-list declared for the formal function.

In the following example procedure definition, procedure EXCHANGE declares two variable parameters, so the values of the arguments are not only passed to this routine upon invocation but will be exchanged upon exit from the routine:

```
PROGRAM PASSINGARGS;

VAR ITEM1,ITEM2:REAL;

PROCEDURE EXCHANGE(VAR A,B:REAL);
{declares A and B as variable parameters}
  VAR HOLD:REAL;
  BEGIN {begin EXCHANGE}
    HOLD := A;
    A:=B;
    B:=HOLD;
  END; {end EXCHANGE}

BEGIN     {begin main program}
  .
  .
  .
  ITEM1:=500.45;
  ITEM2:=3.14;
          {ITEM1 and ITEM2 are passed as argument variables}
  EXCHANGE(ITEM1,ITEM2);
          {ITEM2 is now 500.45 and ITEM1 is 3.14}
  .
  .
END.
```

To illustrate the difference in value parameters and variable parameters, the procedure EXCHANGE is redefined below to receive and set the value of the variable parameter equal to the value parameter. Upon the first invocation of EXCHANGE and return to the main program body, only the value of the argument variable ITEM2 passed to the variable parameter B is changed.

The value of the variable ITEM1 is unchanged. Upon the second invocation of EXCHANGE and return to the main program body, as an argument expression of literal-constants is passed to the value parameter A, the argument variable ITEM2 passed to the variable parameter B is changed upon return to the value of the argument expression passed to A.

```
PROGRAM PASSARGS;

VAR ITEM1,ITEM2:REAL;

PROCEDURE EXCHANGE(A:REAL; VAR B:REAL);
  {A is a value parameter and B is a variable parameter}
  BEGIN  {begin EXCHANGE}
    B:=A;
  END;   {end EXCHANGE}

BEGIN
  .
  .
  .
  ITEM1:=2.3;
  ITEM2:=4.5;
    {ITEM1 is passed to a value parameter, }
    {ITEM2 is passed to a variable parameter}
  EXCHANGE(ITEM1,ITEM2);
    {ITEM2 now contains 2.3}
    {ITEM1 remains the same 2.3}
  .
  .
  .
    {Below, the expression 67.48+15.6 is passed  }
    {to a value parameter, }
    {and ITEM2 is passed to a variable parameter  }
  EXCHANGE(67.48+15.6,ITEM2);
    {ITEM2 now contains 83.08}
  .
  .
  .
 END.
```

The following example defines a procedure, TALLY, with a parameter-list containing a variable parameter, FINAL; a formal procedure, COMPUTE; and a formal function, TAXED. Upon invocation in the main body of the program, arguments are specified and listed in a one-to-one correspondence of order and data types with the parameter-list of the invoked routine.

That is, in the first call on TALLY, the variable TAX1 is passed to FINAL, the procedure-argument COMPUTE_WITH_ROUNDING is passed to the formal procedure COMPUTE, and the function-argument TAXED_WITH_ROUNDING is passed to the formal function TAXED. In the second call on TALLY, the variable TAX2 is passed to FINAL, the procedure-argument COMPUTE_ WITH_TRUNCATION is passed to the formal procedure COMPUTE, and the function-argument TAXED_WITH_TRUNCATION is passed to the formal function TAXED.

```
PROGRAM CHECKTAX(INPUT,OUTPUT);  {Checks tax on incomes}
VAR TABLE:ARRAY[1..1000]  OF REAL; INCOME,DEDUC:REAL;
    TAX1,TAX2:REAL;  INDEX:1..1000;  I:INTEGER;
PROCEDURE TALLY (VAR FINAL : REAL;
                 PROCEDURE COMPUTE(Y,Z:REAL; VAR X:INTEGER);
                 FUNCTION TAXED(PC:REAL;TX:INTEGER):INTEGER);
  VAR TXINC:INTEGER;  PCENT:REAL;
  BEGIN
    COMPUTE(INCOME,DEDUC,TXINC);
    IF TXINC <= 2000 THEN FINAL := 0.0 ELSE
      BEGIN
        INDEX:= TXINC DIV 2000;
        PCENT:=TABLE [INDEX];
        FINAL := TAXED(PCENT,TXINC);
      END;
  END;

  PROCEDURE COMPUTE_WITH_ROUNDING(BEFORETAX,EXPENSE:REAL;
                                  VAR TAXABLE:INTEGER);
    BEGIN
      TAXABLE := ROUND(BEFORETAX) - ROUND(EXPENSE);
    END;

  PROCEDURE COMPUTE_WITH_TRUNCATION(BEFORETAX,EXPENSE:REAL;
                                    VAR TAXABLE:INTEGER);
    BEGIN
      TAXABLE := TRUNC(BEFORETAX) - TRUNC(EXPENSE);
    END;

  FUNCTION TAXED_WITH_ROUNDING(PERCENT:REAL;TAXABLE:INTEGER):
                               INTEGER;
    BEGIN
      TAXED_WITH_ROUNDING := ROUND(PERCENT * TAXABLE);
    END;

  FUNCTION TAXED_WITH_TRUNCATION(PERCENT:REAL;
                                 TAXABLE:INTEGER): INTEGER;
    BEGIN
      TAXED_WITH_TRUNCATION := TRUNC(PERCENT * TAXABLE);
    END;

BEGIN { begin main program CHECKTAX }
  .
  .      { Read in or compute values for TABLE of percents }
  .
  READ(INCOME,DEDUC);
  TALLY(TAX1,COMPUTE_WITH_ROUNDING,TAXED_WITH_ROUNDING);
  .
  .
  TALLY(TAX2,COMPUTE_WITH_TRUNCATION,TAXED_WITH_TRUNCATION);
  .
  .
  .
  WRITELN(TAX1:15:2,' COMPUTED WITH ROUNDING');
  WRITELN(TAX2:15:2,' COMPUTED WITH TRUNCATION');
END.
```

An example of passing function-arguments to formal function parameters of a function follows.

Two different methods are used to calculate the integration of a function to approximate the area under a curve: a trapezoidal method and a simple method, so two specific routines were declared as:

```
FUNCTION TRAPEZOID(M1,N1:REAL; FUNCTION FUNC(X:REAL):REAL):REAL;
CONST LENGTH = 0.000005;
VAR SUM,X1,X2:REAL;
BEGIN {Body of function TRAPEZOID}
  X1 := M1; X2 := X1 + LENGTH;
  SUM := 0.0;
  REPEAT
    SUM := SUM + (((FUNC(X1) + FUNC(X2))/2.0) * LENGTH);
    X1 := X1 + LENGTH;
    X2 := X2 + LENGTH;
  UNTIL  X2 > N1;
  TRAPEZOID := SUM;
END;
```

and:

```
FUNCTION SIMPLE(M1,N1:REAL; FUNCTION FUNC(X:REAL):REAL)
                :REAL;
CONST LENGTH = 0.000005;
VAR SUM,X1,X2:REAL;
BEGIN {Body of function SIMPLE}

  SUM:= 0.0;
  X1:=M1; X2:=X1 + LENGTH;
  REPEAT
    SUM := SUM + (FUNC((X1+X2)/2.0)*LENGTH);
    X1 := X1 + LENGTH;
    X2 := X2 + LENGTH;
  UNTIL X2 > N1;
  SIMPLE := SUM;
END;
```

Also assumed available is the function, DSQRT, after:

```
FUNCTION DSQRT(Z:REAL):REAL;
```

If there were several specific functions that computed the value of a simple function, given one parameter of type REAL; such as:

```
    FUNCTION SEMICIRCLE(XX:REAL):REAL;
      BEGIN   {Body of function SEMICIRCLE}
        SEMICIRCLE := DSQRT(1.0 - SQR(XX));
      END;

    FUNCTION PARABOLA(YY:REAL):REAL;
      BEGIN   {Body of function PARABOLA}
        PARABOLA := SQR(YY);
      END;

    FUNCTION HYPERBOLA(ZZ:REAL):REAL;
      BEGIN   {Body of function HYPERBOLA}
        HYPERBOLA := (1.0 / ZZ);
      END;
```

then, in the routine invocations of TRAPEZOID and SIMPLE, any of the function names, SEMICIRCLE, PARABOLA, or HYPERBOLA, could be passed to either of them as function-arguments.

```
{assumes       variable       declarations       of:       VAR
POINT1,POINT2,AREAT1,AREAS1,AREAT2,AREAS2,AREAT3,AREAS3:REAL}
    BEGIN {Body of block}
      .
      .
      .
      READLN(POINT1);
      READLN(POINT2);
      .
      .
      .
      AREAT1 := TRAPEZOID(POINT1,POINT2,SEMICIRCLE);
      AREAS1 := SIMPLE(POINT1,POINT2,SEMICIRCLE);
      .
      .
      AREAT2 := TRAPEZOID(POINT1,POINT2,PARABOLA);
      AREAS2 := SIMPLE(POINT1,POINT2,PARABOLA);
      .
      .
      AREAT3 := TRAPEZOID(POINT1,POINT2,HYPERBOLA);
      AREAS3 := SIMPLE(POINT1,POINT2,HYPERBOLA);
      .
      .
      .
      WRITELN('SEMICIRCLE INTEGRALS ARE:',AREAT1,AREAS1);
      WRITELN('PARABOLA INTEGRALS ARE:',AREAT2,AREAS2);
      WRITELN('HYPERBOLA INTEGRALS ARE:',AREAT3,AREAS3);
    END;
```

### 9.6.4 Argument Passing to Parameters

When a routine is invoked, the arguments specified in the argument-list of the invocation are passed to the parameters declared in the parameter-list of the routine's declaration.

For a value parameter, the argument expression is evaluated and its value is assigned to the parameter.

For a variable parameter, the datum selected by the argument variable is found by its address, and during execution of the invoked procedure, every action that involves that parameter is performed on the selected datum, as its actual address has been passed. If the argument variable selector requires indexing into an array or finding the target of a pointer, these actions take place before execution of the routine, and their computed addresses are not changed during execution.

If the parameter is a formal routine, any use of the parameter is an indirect use of the argument that was passed to the parameter. An invocation using the identifier of a formal routine parameter is thus indirectly an invocation of some declared routine. If the indirectly invoked routine accesses any non-local datum then that datum would have been accessible to the routine when its name was passed to a formal parameter.

### 9.6.5 Compatibility of Parameters and Arguments.

The argument-list must be compatible with the parameter-list of the routine being invoked. This argument to parameter type compatibility is defined by the following rules:

1. The number of arguments must equal the number of parameters. Each argument must be compatible with the parameter holding the corresponding place in the list. Compatibility of individual arguments with the corresponding parameters is defined by rules 2 through 6.

2. The argument corresponding to a value parameter must be an expression (not a routine name) and can be an expression whose value is of a data-type which is "assignment compatible" to the type-identifier of the parameter. In this implementation of Pascal, we permit two exceptions to this rule. One exception to this rule, is if the parameter type-identifier is preceded by the word UNIV. If the parameter is declared as UNIV, the argument corresponding to it must be of the same internal size to permit compatibility. The other exception concerns string arguments passed to string value parameters. A string argument, i.e., an argument whose type is a one-dimensional array of characters, either a literal-string or a variable of string-type, of any length may be passed to a value parameter. If the argument string is shorter than the value parameter string, then the value parameter string contains undefined values in its string beyond the length of the passed string. If the argument string is greater in length than the length of the value parameter, then the value parameter simply does not contain the extra characters attempted to be passed beyond the length of the value parameter. Although these variable length argument strings can be passed to value parameters in

this implementation of Pascal, the "assignment compatibility" rules of Pascal are in effect, i.e., only strings of the same fixed length can be assigned. This means that the programmer must program for this condition, by planning how to determine the end of, or the length of, such a variable length string so passed. See Section 5.3.9.1 on strings. For example, any routine which is programmed to receive a variable length string, as a value parameter, might consider also having either a parameter to specify length, or have an imbeded end of string character indicator planned to be passed in all strings, and check for that character inside the routine.

3. The argument corresponding to a variable parameter must be a variable and cannot be an expression nor a routine name. The type of the argument variable and that of the variable parameter type-identifier must be "identical". The only exception to this rule is if the parameter type-identifier had been preceded with the word UNIV to relax type-checking. If the parameter is declared as UNIV, the argument is compatible to it if both the parameter and the argument are the same size.

4. The argument corresponding to a formal procedure must be a procedure-argument name. It must be the name of a procedure, either actual or formal; and it cannot be the name of a function. It cannot be a variable nor an expression. Whether the procedure name being passed to a formal procedure parameter is an actual procedure name or a formal procedure name, in either case, the parameter-list of the named procedure-argument must be compatible with the parameter-list of the formal procedure to which that procedure-argument name is being passed. Compatibility rules for two parameter-lists are detailed in rule 7 below.

5. The argument corresponding to a formal function must be a function-argument name. It must be the name of a function, either actual or formal. and it cannot be the name of a procedure. It cannot be a variable nor an expression. Whether the function name being passed to a formal function parameter in either case, the parameter-list of the named function-argument must be compatible to the parameter-list of the formal function to which that name is being passed. Also, the value type of the function-argument must be identical to the value type of the formal function. Compatibility rules for two parameter-lists are detailed in rule 7 below.

6. When a variable is a field of a PACKED record, or a component of a PACKED array, that variable cannot be the argument corresponding to a variable parameter.

7. Two parameter-lists are compatible if they satisfy these conditions:

   a) The number of parameters is the same in each list,

b) A value parameter in one list positionally corresponds to a value parameter in the other list, and the type-identifiers of the value parameters are identical.

c) A variable parameter in one list positionally corresponds to a variable parameter in the other list, and the type-identifiers of the variable parameters are identical.

d) A formal procedure in one list positionally corresponds to a formal procedure in the other list, and the parameter lists of the two formal procedures are compatible (as defined herein by a through f of rule 7).

e) A formal function in one list positionally corresponds to a formal function in the other list and the parameter lists of the two formal functions are compatible, (as defined herein by a through f of rule 7). Also, the function-value type-identifiers of the two formal functions must be identical.

f) A parameter which is declared UNIV in one list must positionally correspond to a parameter in the other list which is also declared UNIV.

### 9.6.6 Environment of an Invocation

Any invocation of a routine has a set of statically allocated data that it can potentially access. This is the environment of the invocation and includes all constants that are visible to the routine, and all global variables. For non-global local variables, the definition of the environment of a particular invocation is more complex. Routines can be called recursively, so there can be several variables in existence to which the same name can refer, and routines can be invoked indirectly by invocation of formal routines. The environment of a routine invocation is determined by these rules:

1. When a routine is invoked directly, its environment contains:

   - a new copy of every variable whose declaration belongs to the procedure,
   - a copy that is in the environment where the routine is invoked for every non-local name of a variable that is visible in the routine.

2. When a routine is passed to a formal routine it carries its environment along with it. The environment of the formal routine is the environment that the argument would have if it were being invoked directly instead of being passed along.

3. The environment of a routine that is invoked indirectly is that environment that was determined for the routine when it was passed as an argument.

### 9.6.7 Nesting Routines

Within the routine declarations part of any block where procedures and functions are defined, definitions of routines can be nested within the routine being defined with a block.

Example:

```
PROGRAM NESTING;
  VAR A,B,C:INTEGER;

  PROCEDURE OUTER1 (Y,Z:REAL; VAR X:INTEGER);
    VAR L,M : INTEGER;
        FLAG: BOOLEAN;

    PROCEDURE INNER1 (D:INTEGER;VAR SETFLAG:BOOLEAN);
      VAR ITEM1:INTEGER;

      BEGIN           {begin body of INNER1}
        ITEM1:= D;
        IF ITEM1 > 0 THEN
           SETFLAG := TRUE
           ELSE
           SETFLAG := FALSE;
      END;            {end of procedure INNER1 definition}

    BEGIN   {begin body of OUTER1}
      .
      .
      .
      INNER1(B,FLAG);         {Invocation of local routine}
      .
      .
      .
    END;              {end of procedure OUTER1 definition}

  BEGIN  {begin body of main program}
    .
    .
    .
    OUTER1(6.8,9.1,A);    {Invocation of local OUTER1}
    .
    .
    .
  END.  {end of main program}
```

The maximum level of nesting inner procedure or function definitions within the block of a routine definition in this implementation of Pascal is 14; but when discussed as the number of nesting levels within a program it is 15, as a total nesting limit of blocks is 16. For example, the PROGRAM NESTING constitutes one block, the program block; having no procedure level (listed as 0); the PROCEDURE OUTER1 constitutes another block, a local routine to the program body and the first routine block within the program block; OUTER1 having procedure level 1; and the PROCEDURE INNER1 constitutes another block, a nested routine within procedure OUTER1; and INNER1 has procedure level 2; and there may up to a total of 14 nested levels within a routine; (e.g., although not shown above, a PROCEDURE INNER2 within INNER1, INNER3 within INNER2, etc., until the deepest nested routine INNER14 within INNER13 became declared).

There may be any number of routine declarations on any one procedure level. That is, for example, there may be any number of routine declarations on the same procedure level as OUTER1; and again, any number of routine declarations on the same procedure level of INNER1, etc.

The compiled-program listing reflects the procedure/function routine nesting level for each source line of a program under the column heading PRC LVL.

The PRC LVL of a program block is counted and listed as 0, so for each listed line that is part of the program block, a 0 under PRC LVL, is listed.

For each listed line contained in the first routine level, of a routine local to the main program, a 1 under PRC LVL, is listed.

For each more deeply nested routine level within a routine, the nesting level under PRC LVL increases by 1 up to 15.

### 9.6.8 Recursion

A routine is said to be recursive if it is defined in terms of itself. Pascal routines can recursively call themselves. Given a procedure definition of procedure P, procedure P can be recursively called in a procedure-call statement within the body of procedure P.

Example:

```
PROCEDURE P(PARAM1:REAL;VAR PARAM2:REAL);
  VAR L1,L2:REAL;
  BEGIN
    ...
    P(L1,L2);     {Direct recursive call on procedure P}
    ...
  END;
```

Procedure P is said to be directly recursive. Indirect recursion occurs when one procedure calls another, where the calling procedure is called again. This indirect type of recursion is programmable in Pascal by use of the FORWARD directive explained in Section 9.4.2.

Each time a procedure is recursively executed a new set of local data and parameters is internally created although they have the same names as coded. With one finite recursive procedure-call statement, an infinite process can be coded. Therefore, recursion termination must be subjected to some preprogrammed condition inherent in the application or designed into the procedure's code where the repetition caused by the recursive call is small enough to be practical. This is because additional storage is required during execution of each recursive call.

Recursion is useful in certain processing on linked lists, traversing tree data structures based on some underlying order of the traversal, or computing certain functions that are recursively defined.

Similarly, in Pascal, functions can recursively invoke themselves, as in the following function F, which computes the factorial of the value parameter integer X.

```
FUNCTION F(X:INTEGER): INTEGER;
BEGIN
  IF X = 0 THEN  F := 1
           ELSE  F := X*F(X-1);
END;
```

Referencing the function F within the expressions of the following assignment statements in the main program would produce the results indicated in the comments:

```
PROGRAM FACTORIAL(INPUT,OUTPUT);
VAR A,B,C,D,Y: INTEGER;
FUNCTION F(X:INTEGER): INTEGER;
  BEGIN
    IF X = 0 THEN
       F:=1
       ELSE
       F:=X*F(X-1);     {Recursive invocation of function F}
  END;
BEGIN
  A:=F(2);      {value in A becomes 2}
  B:=F(3);      {value in B becomes 6}
  C:=F(4);      {value in C becomes 24}
  READ(Y);      {read into Y an integer from INPUT}
  D:=F(Y);      {value in D becomes Y!}
  WRITELN (A,B,C,D);
END.
```

# PART III
# RUN TIME SUPPORT INFORMATION

# CHAPTER 10
# RUN TIME SUPPORT INFORMATION AND LANGUAGE EXTENSIONS

## 10.1 INTRODUCTION

The Perkin-Elmer Pascal system provides the user with a compiler to compile programs written in the Pascal language, as described in Chapters 2 through 9. Additionally, there are the language extensions for direct I/O, SVCs, and other OS services; afforded by the Prefix routines and SVC capability, described in this chapter. An object library is provided for the runtime support of the user compiled program, and is linked to during task establishment as briefly described in Chapter 1. Details on task establishment are described in the OS/32 LINK Reference Manual.

Chapter 10 discusses the following topics:

- The Pascal Runtime Library and its components
- Using the Pascal Prefix routines
- Using the SVC Capability
- Register Usage in the Executing Pascal Task
- Memory Utilization at Runtime
    - Internal Data Storage Representations
        - Alignment
        - Size
        - Packed Structures
    - Memory Overview (Initial State)
    - Memory Overview (Running State)
    - Stack and Heap Management

and describes both the Pascal routine interfaces amongst themselves, and the Pascal-FORTRAN interface, as follows:

- Pascal Linkage Conventions, passing arguments to parameters, calling sequences, receiving sequences, and exits used to and from:
    - internal procedures and functions

  - routines declared as external with the EXTERN directive:
    - * externally compiled PASCAL modules
    - * external CAL routines, using Pascal linkage conventions
    - * external SVC Support routines
  - Prefix routines, declared with procedure/function headings prior to the PROGRAM or MODULE header.
- Pascal-FORTRAN interface to routines declared as external with the FORTRAN directive:
  - external FORTRAN subprograms: subroutines and functions
  - external FORTRAN subprograms using FORTRAN I/O
  - external FORTRAN Runtime Library routines (no argument data-type checking)
  - external CAL routines (declared with FORTRAN directive), using FORTRAN linkage conventions

  Also: external CAL routines (declared EXTERN), referencing FORTRAN subprograms is discussed in Section 10.8.

Run time error messages generated while executing Pascal compiled code or the Run Time Library support routines are described in detail in the latter part of Appendix G under RUN TIME ERROR MESSAGES.

## 10.2 RUN TIME LIBRARY

During execution of an established Pascal task, the Pascal Run Time Library provides a variety of run time support routines. The Pascal Run Time Library is conceptually classified into six major parts; as follows:

- The Pascal Initializer and Common Error Message Routines (PASINIT group)
- The Pascal Task Pausing/EOT/Error Handler (PAS.ERR)
- The RELIANCE-Pascal Interface/Error Handler (PAS.REL)
- The Pascal Prefix Support Routines (PASPREF group)
- The Pascal SVC Support Routines (PASSVC group)
- The Pascal Library Routines (PASLIB group)

The entire Pascal Run Time Library is provided on one object file, PASRTL.OBJ, and is linked to a Pascal compiled program during task establishment, with an OS/32 LINK command:

```
LIBRARY [voln:]PASRTL.OBJ
```

Having linked the user compiled object program to PASRTL.OBJ, a Pascal task always contains routines in PASINIT, always contains the routines of either PAS.ERR or PAS.REL, but not both; and usually contains several routines from PASLIB. Additionally, if the original program source had referenced the standard Prefix definitions, and was compiled with the Prefix, then the established task contains several run time support routines from PASPREF. If the original program source used the Pascal routines to issue SVC calls, then the established task contains several run time support routines from PASSVC.

The Pascal RTL is re-entrant, and subsets of it may be used in building shared segments.

### 10.2.1 The Pascal Initializer and Common Error Message Routines

The PASINIT group contains two routines:

| | |
|---|---|
| P$INIT | Start of any Pascal program |
| P$ERMES | Send error message |

The Pascal Initializer, P$INIT, initializes the memory management mechanisms for the task. That is, it organizes the task workspace, in coordination with the user-specified MEMLIMIT option and the task's UTOP and CTOP; sets the initial values of internal pointers into this workspace, such as a local base, global base, and stack limit, which control storage allocations for dynamic variables in a heap and other data in a stack. P$INIT then intializes the workspace contents to zero.

An alternative version of P$INIT is provided within the same object program as P$FORT. It differs from the principal version of P$INIT, in that it additionally reserves space for the FORTRAN Static Communications Area (SCA) and calls the routine .INITSCA from the FORTRAN VII RTL to initialize the FORTRAN SCA.

Any compiled object containing external references to routines declared with the FORTRAN directive will force resolution of the external reference to P$INIT to select the appropriate version, since the P$FORT containing its P$INIT precedes the basic P$INIT on the PASRTL.OBJ file.

Note that if P$FORT is included in a shared segment then the principal version of P$INIT should not also be included in that shared segment. The attempt to do so results in a multiply defined symbol and an error message from OS/32 LINK.

In an OS environment, run-time errors in compiled code cause control to pass to an illegal instruction, and the run-time support contains an illegal instruction handler which calls P$ERMES.

In a Reliance environment, run-time errors cause direct calls to P$ERMES.

### 10.2.2 The Pascal Task Pausing/EOT/Error Handler (PAS.ERR)

Programs intended to run in an OS/32 environment (non-Reliance environment) are compiled and linked to use the routines in PAS.ERR for error handling, task pausing, and task termination; at run time.

The PAS.ERR group contains the following routines:

| | |
|---|---|
| P$ERR | Enable illegal instruction traps |
| PASERROR | Handle illegal instruction traps |
| P$PAUS | Pause |
| P$TERM | End of Task |
| P$SEND | Send message |

PASERROR is a part of P$ERR and is not visible directly to the outside world. P$PAUS, P$SEND, and P$TERM are provided in both PAS.ERR and PAS.REL, as weak entry points. When a Pascal program is compiled to run in an OS environment, (without the compiler option RELIANCE) the compiled code includes a call to P$ERR, which has the result at LINK time that the above OS-adapted versions of P$PAUS, P$TERM, and P$SEND are incorporated in the task.

### 10.2.3 The RELIANCE-Pascal Interface/Error Handler (PAS.REL)

Programs intended to run in a RELIANCE environment are compiled with a user-specified compiler-option, "RELIANCE", so that the compiled object program when task established, uses the routines in PAS.REL for error handling, task pausing, and task termination; at run time.

The PAS.REL group contains the following routines:

| | |
|---|---|
| P$PAUS | Pause |
| P$TERM | End of Task |
| P$SEND | Send message |

When a Pascal program is compiled to run in a Reliance environment, the compiled code includes a directive to LINK to include PAS.REL. The result is that the Reliance-adapted P$PAUS, P$TERM, and P$SEND are incorporated in the task. P$PAUS, P$TERM, and P$SEND have weak entry labels.

Note that PAS.REL and PAS.ERR routine groups cannot both be included in the same shared segment. The attempt to do so results in multiply defined symbols and an error message from OS/32 LINK.

### 10.2.4 The Pascal Prefix Support Routines (PASPREF group)

The Pascal Prefix Support Routines, in the PASPREF group, on PASRTL.OBJ, provide the run time support for tasks using the standard Perkin-Elmer Prefix routines. The Perkin-Elmer Pascal language extensions, afforded by these routines, are discussed in Section 10.3 on Using the Pascal Prefix.

This PASPREF group contains 20 user-callable object modules, using Pascal linkage conventions for external routines.

These routines are selectively incorporated in the user task, during the link to the PASRTL.OBJ file, for programs which include the Prefix and reference the routines.

In Pascal R01 and up, the names of the entry points are the Pascal R01 routine names, truncated if necessary to be no more than 8 characters long.

TABLE 10-1. Pascal PREFIX RUN TIME SUPPORT ROUTINES

| <u>Pascal Prefix Routine Name</u> (As callable in Pascal code) | <u>Support Routine ENTRY</u> (As listed in LINK MAP) |
|---|---|
| OPEN | OPEN |
| CLOSE | CLOSE |
| ALLOCATE | ALLOCATE |
| RENAME | RENAME |
| REPROTECT | REPROTEC |
| DELETE | DELETE |
| CHANGE_ACCESS_PRIVILEGES | CHANGE_A |
| CHECKPOINT | CHECKPOI |
| FETCH_ATTRIBUTES | FETCH_AT |
| REWIND | REWIND |
| WRITE_FILE_MARK | WRITE_FI |
| BACK_RECORD | BACK_REC |
| BACK_FILE_MARK | BACK_FIL |
| FORWD_RECORD | FORWD_RE |
| FORWD_FILE_MARK | FORWD_FI |
| BREAKPOINT | BREAKPOI |
| START_PARMS | START_PA |
| TIME | TIME |
| DATE | DATE |
| EXIT | EXIT |
| Non-user Prefix support routine: | P$IOFUN |

Note that these routines are now provided as external routines in the Pascal Run Time Support Library, linked to at task establishment time. Four routine names differ from Pascal R00; in order to uniquely identify the routines within the first eight characters of the name and the corresponding entry point. They are:

| Pascal R00 Prefix Routine Name | Pascal R01 Prefix Routine Name |
|---|---|
| BACKSPACE_RECORD | BACK_RECORD |
| BACKSPACE_FILE | BACK_FILE |
| FORWARD_RECORD | FORWD_RECORD |
| FORWARD_FILE | FORWD_FILE |

Note that WRITE_FILE_MARK, BACK_RECORD, BACK_FILE_MARK, FORWD_RECORD, and FORWD_FILE_MARK additionally call a common RTL internal routine, P$IOFUN, (not user-callable), which resides just subsequent to the PASPREF routines on the PASRTL.OBJ file.

### 10.2.5 The Pascal SVC Support Routines (PASSVC group)

The Pascal SVC Support Routines, PASSVC, are the run time object support routines for a set of basic SVC calls.

The Pascal source for the interface type-definitions and routine-declarations for this basic set of SVC calls is available on the file, SMPLSVCS.PAS, as provided with the product. They are also detailed in TABLE 10-5, and Table 10-6, in Section 10.4 on Using the SVC capability.

The user is given the capability to code SVC's within his Pascal program utilizing predefined source interfaces, see Section 10.4.

To implement additional SVC capabilities, object support routines may be added to the PASSVC group in the Pascal Run Time Support Library, and their routine names must be selected so as not to duplicate any preexisting name in the Library. Such additional object support routines may be coded in CAL, using Pascal linkage conventions, and appended to PASRTL.OBJ.

The basic set of SVC calls supported have identical Pascal declared routine names and ENTRY point names in the corresponding object support routine. They are listed below in Table 10.2 with the OS/32 SVC being supported.

TABLE 10-2. Pascal RUN TIME SUPPORT SVC ROUTINES

| Pascal SVC Routine Name and Support Routine Entry | OS/32 SVC Supported or System Service Request |
|---|---|
| SVC1 | SVC 1 Input/Output Request |
| SVC3 | Ends Task Execution via P$TERM |
| SVC5 | SVC 5 Fetch Overlay |
| SVC7 | SVC 7 File Handling Services |
| SVC2PAUS | Pauses Task Execution via P$PAUS |
| SVC2AFLT | SVC 2, Code 4: Set Status |
| SVC2FPTR | SVC 2, Code 5: Fetch Pointers |
| SVC2LOGM | SVC 2, Code 7: Log Message Option X'00' or X'80' |
| SVC2FTIM | SVC 2, Code 8: Interrogate Clock |
| SVC2FDAT | SVC 2, Code 9: Fetch Date |
| SVC2TODW | SVC 2, Code 10: Time of Day Wait |
| SVC2INTW | SVC 2, Code 11: Interval Wait |
| SVC2PKNM | SVC 2, Code 15: Pack ASCII numeric to binary |
| SVC2PKFD | SVC 2, Code 16: Pack File Descriptor |
| SVC2PEEK | SVC 2, Code 19: Peek |
| SVC2TMAD | SVC 2, Code 23, Option X'00' |
| SVC2TMWT | SVC 2, Code 23, Option X'80' |
| SVC2TMRP | SVC 2, Code 23, Option X'40' |
| SVC2TMLF | SVC 2, Code 23, Option X'20' |
| SVC2TMCA | SVC 2, Code 23, Option X'10' |
| SVCINITQ | Initialize Task Queue, set TSW Z bit |
| SVCTASKQ | Fetch a task queue parameter from the Task Queue |
| FROMUDL | Access UDL |
| TOUDL | Modify UDL |

The PASSVC routines are written in assembler-language using Pascal linkage conventions for external routines, and are callable with a Pascal procedure-call statement.

In order to be callable, they must have been declared with parameter lists denoting their interface and with the directive EXTERN in a routine declarations part of a block prior to their invocation.

As the parameter-lists contain certain type-identifiers, pertinent type-definitions must also be made in a type-definition part prior to the routine-declarations part containing the external declarations of the SVC routines. Their declarations are listed in Table 10-5 and Table 10-6.

The user may wish to extract only those SVC interfaces which pertain to those he'd be using in his program (See Section 10.4 on Using the SVC Capability). SVC2LOGM cannot be used under RELIANCE.

A sample example of an SVC support routine expecting no parameters, follows. When writing external routines where arguments will be passed to parameters, refer to the Pascal linkage conventions discussed in Section 10.7.

The external routine declaration required for SVC2PAUS is:

```
     PROCEDURE SVC2PAUS;EXTERN;    {Note no parameter-list}
```

An invocation of SVC2PAUS occurs with the Pascal procedure-call statement:

```
     SVC2PAUS;                     {Pause the Pascal program}
```

Therefore, an associative CAL written SVC support routine might be as follows: [although for the product to support both OS/32 and RELIANCE the Pascal SVC support routine, SVC2PAUS, links to P$PAUS to pause; either the P$PAUS of PAS.ERR (under OS/32) or the P$PAUS of PAS.REL (under RELIANCE)].

```
SVC2PAUS  PROG A Sample CAL SVC-support Routine for OS/32 Pauses
          PURE
          ENTRY SVC2PAUS
ERR       EQU   X'8804'        Define ERR mnemonic as instruction
LERREX    ERR   8,0            R1 field = 8 gives STACK OVERFLOW
*Pascal error handler will trap illegal instruction ERR
SVC2PAUS  ST    R15,4(R2)      Save return address
          CLHI  R0,12+4(R2)    Test for stack/heap collision
          BCS   LERREX         Adding to stack causes overflow
          LIS   R3,1           Obtain data to form OS SVC parblk
          STH   R3,16(R2)      Store code on stack to form parblk
          SVC   2,16(R2)       Issue the Pause SVC under OS/32
*Upon Operator entry of OS/32 CONTINUE command; return to caller
          L     R15,4(R2)      Fetch return address
          L     R2,0(R2)       Restore old Local Base
          BR    R15            Return to caller
R0        EQU   0
R2        EQU   2
R3        EQU   3
R15       EQU   15
          END
```

Note that Pascal R01 and up differs from Pascal R00 in that five additional routines replace the earlier SVC2TIMR routine. They are:

SVC2TMAD
SVC2TMWT
SVC2TMRP
SVC2TMLF
SVC2TMCA

Additionally two routines, FROMUDL and TOUDL are added to access or modify the UDL. Several new type-definitions are included. Also, the interfaces to several routines have been refined. See Section 10.4. For example:

| | |
|---|---|
| SVC3 | its argument must be an integer restricted to a byte value. |
| SVC2FPTR | no longer expects an argument (SVC2FPTR has no parameter-list). |
| SVC2PKFD | 5th argument is now an enumeration, not a BOOLEAN. |
| SVC2PEEK | Three new types which outline the Peek SVC parameter blocks are available. |
| SVCINITQ | restricts its first parameter to be of type QSIZE_TY. |

### 10.2.6 The Pascal Library Routines (PASLIB group)

The Pascal Library group, PASLIB, is the run time object support library of routines, which perform detailed functions to enact Pascal language features.

Compiler-generated external references to these routines are laid down in the compiled object program where necessary to support Pascal language features. At task establishment time, any such external references in a compiled object program are resolved against the file PASRTL.OBJ.

Differing from the Pascal R00 implementation, Pascal R01 and up selectively links in only those PASLIB routines needed by the compiled object program.

PASLIB contains routines to effectively perform or provide heap management, linkage to FORTRAN subprograms, copying and comparison of structured variables, set operations, Pascal file-name (non-text) input and output, text-file input, and text-file output.

These routines are provided as required runtime support and are not intended to be user-callable, so that they are briefly listed by name and functionality, without detailing their interface. Internally, a variety of interface linkages are in use, differing from the linkage conventions to Pascal external routines.

The PASLIB group contains over 60 routines to support Pascal language features. They are listed in sections pertaining to their functionality and are not necessarily in this order on the file PASRTL.OBJ.

Refer to Appendix L for a complete list of the Pascal Run Time Library routines as they appear in the order that they are contained on PASRTL.OBJ.

## TABLE 10-3. Pascal RUN TIME LIBRARY (PASLIB group) ROUTINES

Linkage to FORTRAN subprograms:

| | |
|---|---|
| P$FORT | FORTRAN subprogram call<br>Contains a weak entry P$INIT |

Heap management:

| | |
|---|---|
| P$NEW | NEW |
| P$DISP | DISPOSE |
| P$MARK | MARK |
| P$REL | RELEASE |
| P$SPAC | STACKSPACE |
| P$$REMV | Called from within the RTL |

Copy and comparison of structured variables:

| | |
|---|---|
| P$STCPY | Structured copy (fullwords) |
| P$STCMP0 | Structured compare; whole number of fullwords |
| P$STCMP1 | Structured compare; one byte remainder |
| P$STCMP2 | Structured compare; two bytes remainder |
| P$STCMP3 | Structured compare; three bytes remainder |
| P$FILCPY | Copy file component (arbitrary size) |

Set operations

| | |
|---|---|
| P$SCOMP | Set compare |
| P$SAND | Set product |
| P$SOR | Set sum |
| P$SDIF | Set difference |

Routines for Input, non-text

| | |
|---|---|
| P$READ | READ: not text |
| P$RESET | RESET: not text |
| P$GET | GET: not text |

Text input

| | |
|---|---|
| P$RESETT | RESET: text-file |
| P$READBY | READ byte |
| P$READSI | READ shortinteger |
| P$READI | READ integer |
| P$READSR | READ shortreal |
| P$READR | READ real |
| P$READCH | READ character |
| P$$RDINT | Called from within the RTL |
| P$GETT | GET: text-file |
| P$READLN | READLN |

TABLE 10-3. Pascal RUN TIME LIBRARY (PASLIB group) (continued)

Routines for Output, non-text:

| | |
|---|---|
| P$PUT | PUT: not text |
| P$WRITE | WRITE: not text |

Routines for Output, Text:

| | |
|---|---|
| PWRT.INT | integer routine entries: |
| P$WRITBY | WRITE byte |
| P$WRITSI | WRITE shortinteger |
| P$WRITI | WRITE integer |
| P$WRITSR | WRITE shortreal |
| P$WRITR | WRITE real |
| P$WRITCH | WRITE character |
| P$WRITB | WRITE Boolean |
| P$WRITS | WRITE string |
| P$PAGE | PAGE |
| P$PURGE | Flush the last line of text |
| P$PUTT | PUT: text-file |
| P$WRITLN | WRITELN |

I/O programs common to text and ordinary (non-text) files:

| | |
|---|---|
| P$REWRIT | REWRITE: text or non-text |
| P$IFCB | Initialize file block, internal file |
| P$EFCB | Initialize file block, external file |
| P$CLOSE | Close an internal file |
| P$$REWD | Called from within the RTL |

I/O error servicing routines:

| | |
|---|---|
| P$FCBERR | Called from within the RTL |
| P$$SVC1 | Called from within the RTL |
| P$$SVC7 | Called from within the RTL |
| P$GETER1 | Called from within the RTL |
| P$GETER2 | Called from within the RTL |
| P$PUTERR | Called from within the RTL |
| P$NUMERR | Called from within the RTL |

Duplicates on PASRTL.OBJ of object programs from FORTRAN VII RTL:

| | |
|---|---|
| .ATOD | ASCII to Double, Called by P$READR |
| .DTOA | Double to ASCII, Called by P$WRITR |
| .ATOF | ASCII to Floating, Called by P$READSR |
| .FTOA | Floating to ASCII, Called by P$WRITSR |
| .INITSCA | Initializer of FORTRAN SCA, Calls .CPLUB |
| .CPLUB | Called by .INITSCA |

## 10.3 USING THE Pascal PREFIX

A Perkin-Elmer extension to Pascal allows a prefix of declarations to be placed prior to either a PROGRAM or MODULE header. A prefix may contain a mixture of constant or type declarations, followed by routine procedure or function headings. The constant, type, and routine identifiers become globally visible to the following compilation-unit and the routines are assumed to be external.

The extended Pascal Language features provided by those routines declared in the Perkin-Elmer Pascal Prefix allow the user to:

- Open a file or device
- Close a file or device
- Allocate a file
- Rename a file
- Reprotect a file
- Delete a file
- Change access privileges to a file or device
- Checkpoint a file or device
- Fetch the attributes of an assigned file or device
- Write a file mark
- Rewind a file
- Backspace a record
- Backspace to a file mark
- Forward space a record
- Forward space to a file mark
- Breakpoint
- Obtain Start Parameters
- Obtain the time
- Obtain the date
- Exit the program with a specified return code

Device and file handling service requests for OS/32 contiguous or indexed files, or device and file positioning requests may be performed with Pascal procedure-call statements to the Prefix routines; supplying key values as user specified arguments in the call. Additionally, some miscellaneous routines may be called to pause at a breakpoint, obtain the start-parameters, time and date; and exit the Pascal program with a specified return code.

The Prefix source declarations are listed collectively in Table 10-4 and are available on the file PREFIX.PAS.

The user may use the Pascal {$INCLUDE (fd)} option just prior to the PROGRAM or MODULE header to easily include the Prefix source prior to any compilation-unit(see Chapter 1). For example:

```
{$INCLUDE (voln:PREFIX.PAS)}  or  {$INCLUDE (voln:PREFIX.PAS)}
PROGRAM name(file-name-list);     MODULE name(module-param-list);
```

The object routines to support the Perkin-Elmer Prefix are contained in the Pascal Runtime Library, on the file PASRTL.OBJ, which is linked to at task establishment time. Refer to the PASPREF group described above in Section 10.2.4.

## TABLE 10-4. THE PERKIN-ELMER Pascal PREFIX SOURCE

```
CONST     CR = '(:13:)'; FF = '(:12:)';
TYPE      LUNIT = 0..255;
          IDENTIFIER = ARRAY [1..19] OF CHAR;
          FILE_TYPE = (CONTIGUOUS,INDEXED);
          ACCESS_PRIVILEGE = (SRO,ERO,SWO,EWO,SRW,SREW,ERSW,ERW);
          ATTRIBUTE_BLOCK = PACKED RECORD
                              DEVICE_CODE: SHORTINTEGER;
                              PRECL: SHORTINTEGER;
                              VOLUME_NAME: ARRAY [1..4] OF CHAR;
                              FILE_NAME: ARRAY [1..8] OF CHAR;
                              EXTENSION: ARRAY [1..3] OF CHAR;
                              FILE_CLASS: CHAR;
                              FILE_SIZE: INTEGER
                              END;
          STRING8 = ARRAY [1..8] OF CHAR;
          PARM_POINTER = ^PARM_STRING;
          PARM_STRING = ARRAY [1..132] OF CHAR;

PROCEDURE OPEN (LU:LUNIT; ID:IDENTIFIER; AP:ACCESS_PRIVILEGE;
                KEYS:SHORTINTEGER; VAR STATUS:BYTE);
PROCEDURE CLOSE (LU:LUNIT; VAR STATUS:BYTE);
PROCEDURE ALLOCATE (FT:FILE_TYPE; ID:IDENTIFIER;
                    KEYS:SHORTINTEGER;
                    SIZE,DATA_BLOCK,INDEX_BLOCK:INTEGER;
                    VAR STATUS:BYTE);
PROCEDURE RENAME (LU:LUNIT; ID:IDENTIFIER; VAR STATUS:BYTE);
PROCEDURE REPROTECT (LU:LUNIT; KEYS:SHORTINTEGER;
                     VAR STATUS:BYTE);
PROCEDURE DELETE (ID:IDENTIFIER; KEYS:SHORTINTEGER;
                  VAR STATUS:BYTE);
PROCEDURE CHANGE_ACCESS_PRIVILEGE (LU:LUNIT;
                                   AP:ACCESS_PRIVILEGE;
                                   VAR STATUS:BYTE);
PROCEDURE CHECKPOINT (LU:LUNIT; VAR STATUS:BYTE);
PROCEDURE FETCH_ATTRIBUTES (LU:LUNIT;
                            VAR BLOCK:ATTRIBUTE_BLOCK;
                            VAR STATUS:BYTE);

PROCEDURE REWIND          (LU:LUNIT; VAR STATUS:SHORTINTEGER);
PROCEDURE WRITE_FILE_MARK (LU:LUNIT; VAR STATUS:SHORTINTEGER);
PROCEDURE BACK_RECORD     (LU:LUNIT; VAR STATUS:SHORTINTEGER);
PROCEDURE BACK_FILE_MARK  (LU:LUNIT; VAR STATUS:SHORTINTEGER);
PROCEDURE FORWD_RECORD    (LU:LUNIT; VAR STATUS:SHORTINTEGER);
PROCEDURE FORWD_FILE_MARK (LU:LUNIT; VAR STATUS:SHORTINTEGER);

PROCEDURE BREAKPOINT (LN:INTEGER);
PROCEDURE START_PARMS (VAR PTR:PARM_POINTER);
PROCEDURE TIME (VAR BUFR:STRING8);
PROCEDURE DATE (VAR BUFR:STRING8);
PROCEDURE EXIT (EOT:BYTE);
```

The constant-identifiers defined in the standard Prefix are:

CONST CR = '(:13:)'; FF = '(:12:)';

where:

> CR is the constant-identifier of a character whose value is the ASCII carriage return (X'0D'), and
>
> FF is the constant-identifier of a character whose value is the ASCII form-feed (X'0C').

The type-identifiers defined in the standard Prefix are as follows:

> LUNIT types logical unit specifiers, as the subrange 0 to 255;
>
> IDENTIFIER types a 19-character string-array to contain a complete OS/32 unpacked file descriptor of the form "voln:filename.ext/c", or a device mnemonic of the form "name:                ". A string argument passed to a parameter of type IDENTIFIER, may be specified as a variable array of characters or a string-constant (literal string-constant or named string-constant). The routines assume that the string contains at least 19 characters, and that the file-descriptor must be in the first 19 characters. When the file-descriptor is less than 19 characters, there may be blanks before and after the file descriptor. The argument may also be a local or global variable typed as IDENTIFIER. Unused characters must be blank filled.
>
> FILE_TYPE is an enumeration of the possible OS/32 file kinds, CONTIGUOUS or INDEXED.
>
> ACCESS_PRIVILEGE is an enumeration of the possible OS/32 file or device access privileges.
>
> ATTRIBUTE_BLOCK is a template for the information returned by a fetch attributes request.
>
> STRING8 types buffers used for returning the date and time, in an ASCII format.
>
> PARM_POINTER types the variable returned pointing to the OS START command parameter string.
>
> PARM_STRING types the buffer containing the OS START parameters; as a string of 132 characters.

The procedure declarations part of the standard Prefix defines 20 routine names and their parameters. These 20 routines may be classified into three related groups: SVC 7 file-handling service requests, SVC 1 file-positioning and command functions; and miscellaneous operating system requests.

The file-handling service requests (in the SVC 7 group) are:

OPEN
CLOSE
ALLOCATE
RENAME
REPROTECT
DELETE
CHANGE_ACCESS_PRIVILEGE
CHECKPOINT
FETCH_ATTRIBUTES

As will be detailed below, each routine performs the operation indicated by their procedure name. The argument variable passed to each routine's last parameter, STATUS, is the variable parameter status used to return the OS/32 defined SVC 7 status byte. In calls which involve packing an unpacked file descriptor, namely OPEN, ALLOCATE, RENAME and DELETE, the status will be returned as X'FF' if an error is encountered in the packing process.

The routines in the SVC 1 command functions group are:

REWIND
WRITE_FILE_MARK
BACK_RECORD
BACK_FILE_MARK
FORWD_RECORD
FORWD_FILE_MARK

The SVC 1 command function routines perform the operation indicated by their procedure name. For each routine, the logical unit number is the value of the first argument; and the halfword OS SVC 1 status is returned in the second argument.

These routines are included primarily to perform operations on files or devices which are otherwise accessed through external routines. Use of these procedures with Pascal defined files must be done with care since they cannot communicate with the Pascal file control blocks. Thus, use of these routines on Pascal file-variables may cause erroneous results of the standard functions EOF and EOLN. Additionally, in this case, the expected sequence of I/O data transfers may be altered by use of these routines to manipulate files. Pascal files should normally be positioned using the RESET and REWRITE procedures (see Chapter 8).

The miscellaneous operating system/service requests are:

BREAKPOINT START_PARMS TIME DATE EXIT

The extended language features to Pascal, afforded by the Prefix routines, may be invoked with the following Pascal procedure-call statements and arguments, as detailed below.

### 10.3.1 Open

A call on:

```
PROCEDURE OPEN (LU:LUNIT; ID:IDENTIFIER; AP:ACCESS_PRIVILEGE;
                KEYS:SHORTINTEGER; VAR STATUS:BYTE);
```

is of the form:

```
OPEN(arg1,arg2,arg3,arg4,arg5);
```

where:

arg1 is an integer expression (0 to 255) which is the logical unit.

arg2 is a string (a variable array of characters, a named or literal string-constant), which contains in its first 19-characters, the device mnemonic name or unpacked file-descriptor of the file to open. Leading spaces within the file-descriptor are skipped.

arg3 is an expression whose value is of type ACCESS_PRIVILEGE; specifying the device's or file's access privileges and must be, or evaluate to, one of the following enumeration constant-identifiers:

SRO,ERO,SWO,EWO,SRW,SREW,ERSW,ERW

arg4 is a shortinteger expression specifying write/read keys of the file to be opened. For example, such as expressed below in hexadecimal:

| | |
|---|---|
| #0000 | Unconditionally unprotected |
| #0700 | Protected write key = #07, unprotected read |
| #0089 | Unprotected write, protected read key = #89 |
| #3245 | Protected write key = #32, protected read key = #45 |

arg5 specifies a variable of type BYTE, which receives the SVC 7 status byte; or the value X'FF' when an error is encountered in the file-descriptor SVC 2 packing process. Zero indicates no error.

Action: OPEN assigns the file or device specified by the string in arg2 to the logical unit specified by arg1. OPEN assumes that the file or device exists, (see ALLOCATE below). Arg3 must be one of the constants enumerated by the type ACCESS_PRIVILEGE. The arg4 KEYS must match the existing write/read keys of the specified file. Keys are ignored, when opening a device, and therefore may be specified as zero. Status is returned in arg5.

An example:

```
VAR FD:IDENTIFIER; OK:BYTE;

BEGIN
  FD := 'MAG1:            ';
  OPEN(2,FD,SRW,0,OK);
  IF OK = #FF THEN BREAKPOINT(LINENUMBER);
  IF OK <> 0 THEN EXIT(OK);
END

BEGIN
  FD := 'M300:FILENAME.EXT/P';
  OPEN(8,FD,ERW,#3245,OK);
  IF OK = #FF THEN BREAKPOINT(LINENUMBER);
  IF OK <> 0 THEN EXIT(OK);
END
```

### 10.3.2 Close

A call on: PROCEDURE CLOSE (LU:LUNIT; VAR STATUS:BYTE);

is of the form:

**CLOSE(arg1,arg2);**

where:

arg1 is an integer expression (0 to 255) which is the logical unit.

arg2 specifies a variable of type BYTE, which receives the OS/32 SVC 7 status byte. Zero indicates no error.

Action: CLOSE deassigns the file or device currently assigned to the logical unit specified by arg1. Status is returned in arg2.

An example:

```
VAR OK:BYTE;

BEGIN
  ...   {Assuming a file/device is open on lu 2}

  CLOSE(2,OK);
  IF OK <> 0 THEN WRITELN('CLOSE ERROR=',OK);
END
```

### 10.3.3 Allocate

A call on:

```
PROCEDURE ALLOCATE (FT:FILE_TYPE; ID:IDENTIFIER;
                    KEYS:SHORTINTEGER;
                    SIZE,DATA_BLOCK,INDEX_BLOCK:INTEGER;
                    VAR STATUS:BYTE);
```

is of the form:

```
ALLOCATE(arg1,arg2,arg3,arg4,arg5,arg6,arg7);
```

where:

arg1 is an expression which specifies the file-type as an enumeration-constant; and must be, or evaluate to, one of the following enumeration-constant identifiers:

CONTIGUOUS
INDEXED

arg2 is a string, ( a variable array of characters, a named or literal string-constant), which contains in its first 19 characters the unpacked file-descriptor of the file to be allocated. Leading spaces within the file-descriptor will be skipped.

arg3 is a shortinteger expression specifying the write/read keys with which this file will be protected, and must be respecified in future accesses (e.g., via the SVC routines described below in Section 10.4 or other Prefix routines). For example, such as expressed below in hexidecimal:

| | |
|---|---|
| #0000 | Unconditionally unprotected |
| #0700 | Protected write key = #07, unprotected read |
| #0089 | Unprotected write, protected read key = #89 |
| #3245 | Protected write = #32, protected read key = #45 |

arg4 is an integer expression specifying a number of sectors as the physical file size, if arg1 is CONTIGUOUS. If arg1 is INDEXED, arg4 is an integer expression specifying the logical record length of the file.

arg5 is an integer expression specifying the data block size, as a multiple of 256 bytes; if arg1 is INDEXED. Arg5 is ignored, if arg1 is CONTIGUOUS, but some value must be present in the call.

arg6 is an integer expression specifying the index block size, as a multiple of 256 bytes (e.g. 1); if arg1 is INDEXED. Arg6 is ignored if arg1 is CONTIGUOUS, but some value must be present in the call.

arg7 specifies a variable of type BYTE, which receives the SVC 7 status byte; or the value X'FF' when an error is encountered in the file-descriptor SVC 2 packing process. Zero indicates no error.

Action: ALLOCATE creates a disc file with the name specified in arg2 of the arg1 file-type, either CONTIGUOUS or INDEXED. The arg3 keys specifies the file's write/read keys. If the file-type selected is CONTIGUOUS, the arg4 SIZE must specify the number of physical sectors to be allocated; and arg4 and arg5 are ignored (they must be specified however to maintain the correspondence of actual arguments and procedure parameters required by the language). If the file-type selected is INDEXED, the arg4 SIZE specifies the logical record length of the file's records. Arg5 specifies the data blocking factor and arg6 specifies the index blocking factor. Status is returned in arg7.

Examples:

```
VAR FD:IDENTIFIER; OK:BYTE;

BEGIN
  FD := '     ENTRIESX.SRC/P';
  ALLOCATE(CONTIGUOUS,FD,#0304,16000,0,0,OK);
  IF OK = #FF THEN BREAKPOINT(LINENUMBER);
  IF OK <> 0 THEN EXIT(OK);
  ...
  FD := 'REPORT02.LST/P     ';
  ALLOCATE(INDEXED,FD,#0807,132,1,1,OK);
  IF OK = #FF THEN BREAKPOINT(LINENUMBER);
  IF OK <> 0 THEN EXIT(OK);
  ...
END
```

Note: A temporary file may be created by performing an allocate and assign combination SVC 7, via the SVC7 routine described in Section 10.4. The Pascal internal file-variables (see Chapter 8) are also created as temporary files.

### 10.3.4 Rename

A call on:

```
PROCEDURE RENAME (LU:LUNIT; ID:IDENTIFIER; VAR STATUS:BYTE);
```

is of the form:

```
RENAME(arg1,arg2,arg3);
```

where:

arg1 is an integer expression (0 to 255) which is the logical unit.

arg2 is a string (a variable array of characters, a named or literal string-constant), which contains in its first 19 characters the unpacked file-descriptor with which to rename the file. Leading spaces within the 19-character file-descriptor are skipped. Only the filename and extension fields are required.

arg3 specifies a variable of type BYTE, which receives the OS SVC 7 status byte, or the value X'FF' when an error is encountered in the file-descriptor SVC 2 packing process. Zero indicates no error.

Action: RENAME causes the file (with access-privilege ERW) currently assigned to the logical unit of arg1 to be renamed to the string contained in arg2. Status is returned in arg3.

An example:

```
VAR FD : IDENTIFIER; OK:BYTE;

BEGIN
  FD := '     FILENAME.LST  ';
  RENAME(8,FD,OK);
  IF OK <> 0 THEN BREAKPOINT(LINENUMBER);
  ...
END

BEGIN
  RENAME(4,'SYSFILE1.PAS       ',OK);
  IF OK <> 0 THEN BREAKPOINT(LINENUMBER);

  ...
END
```

### 10.3.5 Reprotect

A call on:

```
PROCEDURE REPROTECT (LU:LUNIT; KEYS:SHORTINTEGER;
                     VAR STATUS:BYTE);
```

is of the form:

```
REPROTECT(arg1,arg2,arg3);
```

where:

arg1 is an integer expression (0 to 255) which is the logical unit.

arg2 is a shortinteger expression specifying the new write/read keys of the file currently assigned to the logical unit specified by arg1. For example, such as expressed below in hexidecimal:

| | |
|---|---|
| #0000 | Unconditionally unprotected |
| #0700 | Protected write key = #07, unprotected read |
| #0089 | Unprotected write, protected read key = #89 |
| #3245 | Protected write key = #32, protected read key = #45 |

arg3 specifies a variable of type BYTE, which receives the OS SVC 7 status byte. Zero indicates no error.

Action: REPROTECT causes the write/read keys of the file (with access-privilege ERW) currently assigned to arg1 to be changed to the value specified by the arg2 keys. Status is returned in arg3.

An example:
```
VAR OK:BYTE;

BEGIN
  REPROTECT(8,#3245,OK);
  IF OK <> 0 THEN BREAKPOINT(LINEUMBER);
  ...
END
```

### 10.3.6 Delete

A call on:

```
PROCEDURE DELETE (ID:IDENTIFIER; KEYS:SHORTINTEGER;
                  VAR STATUS:BYTE);
```

is of the form:

```
DELETE(arg1,arg2,arg3);
```

where:

arg1 specifies a string (a variable array of characters, a named or literal string-constant), which contains in its first 19 characters the unpacked file-descriptor of the file to be deleted. Leading spaces within the file-descriptor are skipped.

arg2 is a shortinteger expression specifying the write/read keys of the file currently being deleted. For example:

| | |
|---|---|
| #0000 | Unconditionally unprotected |
| #0700 | Protected write key = #07, unprotected read |
| #0089 | Unprotected write, protected read key = #89 |
| #3245 | Protected write = #32, protected read key = #45 |

arg3 specifies a variable of type BYTE, which receives the OS SVC 7 status byte, or the value X'FF' when an error is encountered in the file-descriptor SVC 2 packing process. Zero indicates no error.

Action: DELETE causes the file specified in arg1 to be deleted. The argument arg2 must contain a value which matches the file's current write/read keys. Status is returned in arg3.

```
An example:  VAR OK : BYTE;

             BEGIN
               FD := 'M300:FILENAME.EXT/P';
               DELETE(FD,#3245,OK);
               IF OK <> 0 THEN BREAKPOINT(LINENUMBER);
               ...
             END
```

### 10.3.7 Change Access Privilege

A call on:

```
PROCEDURE CHANGE_ACCESS_PRIVILEGE (LU:LUNIT;
                                   AP:ACCESS_PRIVILEGE;
                                   VAR STATUS:BYTE);
```

is of the form:

```
CHANGE_ACCESS_PRIVILEGE(arg1,arg2,arg3);
```

where:

arg1 is an integer expression (0 to 255) which is the logical unit.

arg2 is an expression of type ACCESS_PRIVILEGE. It must be or evaluate to, one of the following enumeration constant-identifiers:

SRO,ERO,SWO,EWO,SRW,SREW,ERSW,ERW

as is compatible with the current access-privilege.

arg3 specifies a variable of type BYTE, which receives the OS SVC 7 status byte. Zero indicates no error.

Action: CHANGE_ACCESS_PRIVILEGE causes the current access privileges of a file or device assigned to arg1 to be changed to that specified in arg2. The value of arg2 must be one of those declared in the enumeration ACCESS_PRIVILEGE. Status is returned in arg3.

```
An example:  VAR OK:BYTE;

             BEGIN
               CHANGE_ACCESS_PRIVILEGE(2,SRO,OK);
               IF OK <> 0 THEN BREAKPOINT(LINENUMBER);
               ...
             END
```

### 10.3.8 Checkpoint

A call on:

```
PROCEDURE CHECKPOINT (LU:LUNIT; VAR STATUS:BYTE);
```

is of the form:

```
CHECKPOINT(arg1,arg2);
```

where:

arg1 is an integer expression (0 to 255) which is the logical unit.

arg2 specifies a variable of type BYTE, which receives the OS SVC 7 status byte. Zero indicates no error.

Action: CHECKPOINT causes any remaining buffered data to be copied to the file or device specified by the logical unit of arg1. Status is returned in arg2.

An example:

```
VAR OK : BYTE;

BEGIN
  ...   {Assuming buffered I/O to lu 2}

  CHECKPOINT(2,OK);
  IF OK <> 0 THEN BREAKPOINT(LINENUMBER);
  ...
END
```

### 10.3.9 Fetch Attributes

A call on:

```
PROCEDURE FETCH_ATTRIBUTES (LU:LUNIT;
                            VAR BLOCK:ATTRIBUTE_BLOCK;
                            VAR STATUS:BYTE);
```

is of the form:

```
FETCH_ATTRIBUTES(arg1,arg2,arg3);
```

where:

arg1 is an integer expression (0 to 255) which is the logical unit.

arg2 specifies a variable of type ATTRIBUTE_BLOCK, which receives the attribute information of the file or device currently assigned to the logical unit specified by arg1.

arg3 specifies a variable of type BYTE, which receives the OS SVC 7 status byte. Zero indicates no error.

Action: FETCH_ATTRIBUTES returns the attribute information of the file or device currently assigned to arg1 in the variable variable arg2. FETCH_ATTRIBUTES returns information concerning whether or not a unit is assigned, device type, device attributes, record length, filename, and file size. Status is returned in arg3.

An example:

```
VAR OK:BYTE; FILEINFO:ATTRIBUTE_BLOCK;

BEGIN
  FETCH_ATTRIBUTES(2,FILEINFO,OK);
  IF OK <> 0 THEN BREAKPOINT(LINENUMBER);
  IF FILEINFO.FILE_NAME = 'FILENAME'
     THEN ...
     ELSE ...
END
```

### 10.3.10 Rewind

A call on:

PROCEDURE REWIND (LU:LUNIT; VAR STATUS:SHORTINTEGER);

is of the form:

REWIND(arg1,arg2);

where:

arg1 is an integer expression (0 to 255) which is the logical unit.

arg2 specifies a variable of type SHORTINTEGER, which receives the OS SVC 1 status halfword, both the device-independant status byte (left byte) and the device-dependant status byte (right byte). Zero indicates the rewind has succeeded.

Action: REWIND rewinds the file or device currently assigned the logical unit specified by arg1. Status is returned in arg2.

An example:

```
VAR ACK:SHORTINTEGER;

BEGIN
  REWIND(2,AOK);
  IF AOK <> 0 THEN BREAKPOINT(LINENUMBER);
  ...
END
```

### 10.3.11 Write File Mark

A call on:

`PROCEDURE WRITE_FILE_MARK (LU:LUNIT; VAR STATUS:SHORTINTEGER);`

is of the form:

`WRITE_FILE_MARK(arg1,arg2);`

where:

arg1 is an integer expression (0 to 255) which is the logical unit.

arg2 specifies a variable of type SHORTINTEGER, which receives the SVC 1 status halfword, both the device-independent status byte (left byte) and the device-dependent status byte (right byte). Zero indicates no error.

Action: WRITE_FILE_MARK writes a file mark on the file or device currently assigned to the logical unit specified by arg1. Status is returned in arg2.

An example:

```
VAR AOK:SHORTINTEGER;

BEGIN
  WRITE_FILE_MARK(2,AOK);
  IF AOK <> 0 THEN BREAKPOINT(LINENUMBER);
  ...
END
```

### 10.3.12 Back Record

A call on:

`PROCEDURE BACK_RECORD (LU:LUNIT; VAR STATUS:SHORTINTEGER);`

is of the form:

`BACK_RECORD(arg1,arg2);`

where:

arg1 is an integer expression (0 to 255) which is the logical unit.

arg2 specifies a variable of type SHORTINTEGER, which receives the SVC 1 status halfword, both the device-independent status byte (left byte) and the device-dependent status byte (right byte). Zero indicates no error.

Action: BACK_RECORD backspaces one record on the file or device currently assigned to the logical unit specified by arg1. Status is returned in arg2.

An example:

```
VAR AOK:SHORTINTEGER;

BEGIN
  ...
  BACK_RECORD(2,AOK);
  IF AOK <> 0 THEN BREAKPOINT(LINENUMBER);
  ...
END
```

### 10.3.13 Back File Mark

A call on:

PROCEDURE BACK_FILE_MARK (LU:LUNIT; VAR STATUS:SHORTINTEGER);

is of the form:

BACK_FILE_MARK(arg1,arg2);

where:

arg1 is an integer expression (0 to 255) which is the logical unit.

arg2 specifies a variable of type SHORTINTEGER, which receives the SVC 1 status halfword, both the device-independent status byte (left byte) and the device-dependent status byte (right byte). Zero indicates no error.

Action: BACK_FILE_MARK backspaces to file mark on the file or device currently assigned to the logical unit specified by arg1. Status is returned in arg2.

An example:

```
VAR AOK:SHORTINTEGER;

BEGIN
  ...
  BACK_FILE_MARK(2,AOK);
  IF AOK <> 0 THEN BREAKPOINT(LINENUMBER);
  ...
END
```

### 10.3.14 Forward Record

A call on:

PROCEDURE FORWD_RECORD (LU:LUNIT; VAR STATUS:SHORTINTEGER);

is of the form:

FORWD_RECORD(arg1,arg2);

where:

arg1 is an integer expression (0 to 255) which is the logical unit.

arg2 specifies a variable of type SHORTINTEGER, which receives the SVC 1 status halfword, both the device-independent status byte (left byte) and the device-dependent status byte (right byte). Zero indicates no error.

Action: FORWD_RECORD forwardspaces one record on the file or device currently assigned to the logical unit specified by arg1. Status is returned in arg2.

An example:

```
VAR AOK:SHORTINTEGER;

BEGIN
  FORWD_RECORD(2,AOK);
  IF AOK <> 0 THEN BREAKPOINT(LINENUMBER);
  ...
END
```

### 10.3.15 Forward File Mark

A call on:

PROCEDURE FORWD_FILE_MARK (LU:LUNIT; VAR STATUS:SHORTINTEGER);

is of the form:

FORWD_FILE_MARK(arg1,arg2);

where:

arg1 is an integer expression (0 to 255) which is the logical unit.

arg2 specifies a variable of type SHORTINTEGER, which receives the SVC 1 status halfword, both the device-independent status byte (left byte) and the device-dependent status byte (right byte). Zero indicates no error.

Action: FORWD_FILE_MARK forwardspaces to file mark on the file or device currently assigned to the logical unit specified by arg1. Status is returned in arg2.

An example:

```
VAR AOK:SHORTINTEGER;

BEGIN
  FORWD_FILE_MARK(2,AOK);
  IF AOK <> 0 THEN BREAKPOINT(LINENUMBER);
END
```

### 10.3.16 Breakpoint

A call on: PROCEDURE BREAKPOINT (LN:INTEGER);

is of the form:

BREAKPOINT(arg1);

where:

arg1 is an integer expression whose value will be output in a log message prior to pausing the task.

Action: BREAKPOINT causes execution of the Pascal task to be suspended through an OS SVC 2 pause. The value of the argument is logged on the user's console device along with the hexidecimal address of the call. The Pascal task will resume execution with the Pascal statement logically following the BREAKPOINT call when the OS CONTINUE command is given. The BREAKPOINT procedure is most commonly used in conjunction with the function LINENUMBER thusly:

```
BREAKPOINT (LINENUMBER)
```

Execution of this statement will generate a break at the current line number of the source program, during run time. As the line number is reflected in the message, the user may determine where in his source, or which programmed breakpoint is occurring.

The message output is of the form:

```
LINE xxxxx, ADDR yyyyyy BREAKPOINT
```

where, if the argument is the function LINENUMBER, xxxxx is the source line number of line containing the call on BREAKPOINT, and yyyyyy is the machine address of the breakpoint in compiled-code.

### 10.3.17 Start Parameters

A call on: PROCEDURE START_PARMS (VAR PTR:PARM_POINTER);

is of the form:

START_PARMS(arg1);

where:

arg1 specifies a pointer-variable of type PARM_POINTER, which receives a pointer to the start-parameters with which this task was started.

Action: START_PARMS returns a pointer to an array of up to 132 characters terminated with a carriage return. This character string is that entered with the OS START command, for example:

```
START ,T,1,2,3,4
ST ,A B C
ST
```

ending with a carriage-return, and is accessible in a targeted array by a call on START_PARMS. The Pascal program may access the elements of this array with a selector of the form POINTER^[I] where POINTER is a variable of type PARM_POINTER. A carriage-return character terminates the meaningful data in POINTER^.

An example:
```
{CONST  CR = '(:13:)' from Prefix}
VAR POINTER : PARM_POINTER; I : INTEGER;

BEGIN
  START_PARMS(POINTER);
  I := 1;
  WHILE POINTER^[I] <> CR DO BEGIN
        {examine start-parameters}
        IF POINTER^[I] = 'T' THEN ... ;
        I := I+1;
      END;
END;
```

### 10.3.18 Time

A call on: PROCEDURE TIME (VAR BUFR:STRING8);

is of the form:

TIME(arg1);

where:

arg1 specifies a variable of type STRING8, which receives the current time-of-day in an eight-character ASCII format.

Action: TIME returns in arg1, a character string which is the current system clock time. The time is formatted as hh:mm:ss.

An example:
```
VAR CLOCK:STRING8;

BEGIN
  TIME(CLOCK);
  WRITELN(CLOCK);    {writes 20:30:00 for 8:30p.m.}
END.
```

### 10.3.19 Date

A call on: PROCEDURE DATE (VAR BUFR:STRING8);

is of the form:

DATE(arg1);

where:

arg1 specifies a variable of type STRING8; which receives the current date, in either the format as mm/dd/yy or dd/mm/yy depending upon which form for dates has been system generated in the OS.

Action: DATE obtains the current date in either the form month/day/year or the form day/month/year as an eight character ASCII string; and returns it in arg1.

```
An example:  VAR TODAY : STRING8;

             BEGIN
               DATE(TODAY);
               WRITELN('Today is: ',TODAY);
                {Outputs: "Today is: 12/25/81" on Christmas}
                {     or: "Today is: 25/12/81" on Christmas}
             END.
```

### 10.3.20 Exit

A call on: PROCEDURE EXIT (EOT:BYTE);

is of the form:

EXIT(arg1);

where:

arg1 is an integer expression (0 to 255) which is used as the End of Task code, in the ensueing task termination, end-of-task message.

Action: EXIT terminates the execution of a Pascal task, with the specified return code of arg1.

```
An example:  VAR ERROR_CODE : BYTE;

             BEGIN  ...
               IF condition THEN ERROR_CODE := 3;
               ...
               IF ERROR_CODE <> 0 THEN EXIT(ERROR_CODE);
               EXIT(0);
             END.
```

## 10.4 USING THE SVC CAPABILITY

Perkin-Elmer Pascal provides the user the capability to use SVC calls in his Pascal program. The PASSVC support routine names are declared as EXTERN procedures with their appropriate parameter-list interfaces, and the SVC support routines, in the PASRTL.OBJ (see Section 10.2.5), are linked to the user program at task establishment time.

As with all EXTERN routines, the programmer is responsible for making the interface with which these procedures are called agree with the interface which they are prepared to accept. The interface they are prepared to accept is defined by the CONST/TYPE declarations, and parameter-lists in the PROCEDURE EXTERN declarations.

The programmer who uses these SVC procedures is expected to set up the required input field values in the variable serving as the SVC parameter-block, for those SVC routines requiring one in their interface, (e.g., SVC1, SVC5, SVC2PEEK, and SVC7), and to extract information that is returned in that parameter-block after the call, when desired.

Information on the OS/32 SVCs definitions are detailed in the OS/32 Programmer Reference Manual (PRM), Publication Number S29-613. The reader is assumed to be familiar with the OS/32 PRM.

The constant and type-definitions required prior to their use in the SVC external routine declarations are listed in Table 10-5. These definitions should be included in a type-definitions part of a block, prior to a variable-declarations part or the routine-declarations that reference them. The constant and type definitions may also be included prior to the PROGRAM or MODULE header of a compilation-unit.

The external procedure declarations that are required in the user source to establish the SVC routine-names and their interface are listed in Table 10-6. These EXTERN procedure declarations should be included in the routine-declarations part of a block. Note that they reference the type-definitions in Table 10-5.

The file provided with Pascal that contains samples of coding a variety of SVCs, written in Pascal source is SMPLSVCS.PAS. The user may also extract from this file (SMPLSVCS.PAS) the type and constant definitions and external routine declarations required for the entire set of supported SVCs, as they are depicted in Tables 10-5. and 10-6. Additional CONSTANT definitions are also available on this file which define key values with constant-identifiers for several pertinent fields in the record variables serving as SVC parameter-blocks.

## TABLE 10-5. PASCAL SVC SUPPORT TYPE-DEFINITIONS

```
TYPE CHAR2 = PACKED ARRAY [1..2] OF CHAR;
TYPE CHAR3 = PACKED ARRAY [1..3] OF CHAR;
TYPE CHAR8 = PACKED ARRAY [1..8] OF CHAR;
TYPE CHAR4 = PACKED ARRAY [1..4] OF CHAR;

TYPE LINE = ARRAY [1..132] OF CHAR;

{SVC1 PARAMETER BLOCK}

TYPE SVC1_BLOCK = RECORD
      SVC1_FUNC: BYTE;            {FUNCTION CODE}
      SVC1_LU: BYTE;              {LOGICAL UNIT NUMBER}
      SVC1_STAT: BYTE;            {DEV-INDEP STATUS}
      SVC1_DEV_STAT: BYTE;        {DEV-DEPENDENT STATUS}
      SVC1_BUFSTART: INTEGER;     {ADDRESS(BUFFER)}
      SVC1_BUFEND: INTEGER;  {ADDRESS(BUFFER)+SIZE(BUFFER)-1}
      SVC1_RANDOM_ADDR: INTEGER;{RANDOM ADDRESS FOR DASD}
      SVC1_XFER_LEN: INTEGER;     {TRANSFER LENGTH}
      SVC1_RESERVED: INTEGER;     {RESERVED FOR ITAM USE}
     END;

{SVC5 PARAMETER BLOCK}

TYPE SVC5_PARM = RECORD
       SVC5_OVNAME : CHAR8;
       SVC5_STAT :   BYTE;
       SVC5_OPT  :   BYTE;
       SVC5_LU   :   SHORTINTEGER;
     END;


{FILE DESCRIPTOR FOR SVC7 REQUESTS}

TYPE FD_TYPE = PACKED RECORD
      VOLN: CHAR4;                {VOLUME NAME}
      FN: CHAR8;                  {FILE NAME}
      EXTN: CHAR3;                {EXTENSION}
      ACCT: CHAR;                 {ACCOUNT CODE: P/S/G}
     END;

{SVC 7 PARAMETER BLOCK}

TYPE SVC7_BLOCK = RECORD
      SVC7_CMD: BYTE;             {COMMAND}
      SVC7_MOD: BYTE;             {MODIFIER/DEVICE TYPE}
      SVC7_STAT: BYTE;            {STATUS}
      SVC7_LU: BYTE;              {LOGICAL UNIT NUMBER}
      SVC7_KEYS: SHORTINTEGER;    {READ/WRITE KEYS}
      SVC7_RECLEN: SHORTINTEGER;{LOGICAL RECORD LENGTH}
      SVC7_FD: FD_TYPE;           {FILE DESCRIPTOR}
      SVC7_SIZE: INTEGER;         {FILE(/INDEX) SIZE}
     END;
```

TABLE 10-5. PASCAL SVC SUPPORT TYPE-DEFINITIONS (Continued)

```
CONST TASKQ_SLOT_COUNT = 4;
TYPE QSIZE_TY = 1..TASKQ_SLOT_COUNT;
TYPE TASKQ_TYPE = RECORD
       QSIZE: QSIZE_TY;
       FILL1, FILL2, FILL3: SHORTINTEGER;
       TASKQ_SLOTS: ARRAY[QSIZE_TY] OF INTEGER;
     END;

TYPE UDL_INDEX = 0..63;
TYPE PACK_OPTION = (USER_VOL, SYS_VOL, SPL_VOL, NO_DEFAULT);

TYPE PEEK_00_BLOCK = RECORD
       PEEK_OPT       : BYTE;
       PEEK_CODE      : BYTE;
       PEEK_NLU       : BYTE;
       PEEK_MPRI      : BYTE;
       PEEK_OSID      : CHAR8;
       PEEK_TASK_NAME : CHAR8;
       PEEK_CTSW      : INTEGER;
       PEEK_TOPT      : SHORTINTEGER;
       RESERVED       : SHORTINTEGER;
     END;

TYPE PEEK_01_BLOCK = PACKED RECORD
       PEEK_OPT       : BYTE;
       PEEK_CODE      : BYTE;
       RESERVED_1     : SHORTINTEGER;
       PEEK_OSID      : CHAR8;
       PEEK_OSUP      : CHAR2;
       PEEK_CPU       : SHORTINTEGER;
       PEEK_SOPT      : INTEGER;
       PEEK_UACT      : SHORTINTEGER;
       PEEK_GACT      : SHORTINTEGER;
       RESERVED_2     : INTEGER;
     END;

TYPE PEEK_02_BLOCK = PACKED RECORD
       PEEK_OPT       : BYTE;
       PEEK_CODE      : BYTE;
       RESERVED_3     : SHORTINTEGER;
       PEEK_OSID      : CHAR8;
       PEEK_LOAD_VOL  : CHAR4;
       PEEK_FILENAME  : CHAR8;
       PEEK_EXT       : CHAR3;
       PEEK_FILE_CLASS: CHAR;
     END;
```

## TABLE 10-6. EXTERN SVC DECLARATIONS TO CALL SVCs

```
PROCEDURE SVC1(VAR PARM:SVC1_BLOCK); EXTERN;

PROCEDURE SVC3(TERM_CODE : BYTE); EXTERN;

PROCEDURE SVC5(VAR PARM: SVC5_PARM); EXTERN;

PROCEDURE SVC7(VAR PARM:SVC7_BLOCK); EXTERN;

PROCEDURE SVC2PAUS; EXTERN;

PROCEDURE SVC2AFLT(ENABLE: BOOLEAN); EXTERN;

PROCEDURE SVC2FPTR; EXTERN;

PROCEDURE SVC2LOGM(MSG:LINE; LEN:INTEGER; IMAGE:BOOLEAN); EXTERN;

PROCEDURE SVC2FTIM(VAR TIME:INTEGER; VAR HHMMSS: CHAR8); EXTERN;

PROCEDURE SVC2FDAT(VAR MMDDYY:CHAR8); EXTERN;

PROCEDURE SVC2TODW(TOD : INTEGER);    EXTERN;

PROCEDURE SVC2INTW(INTVL : INTEGER);  EXTERN;

PROCEDURE SVC2PKNM(VAR VAL:INTEGER; BUF:LINE; VAR POSN:INTEGER;
                   OPT:INTEGER; VAR CC:INTEGER); EXTERN;

PROCEDURE SVC2PKFD(VAR FD:UNIV FD_TYPE; BUF:LINE;
                   VAR POSN:INTEGER; SKIP_BLANKS:BOOLEAN;
                   OPT: PACK_OPTION; VAR CC:INTEGER);  EXTERN;

PROCEDURE SVC2PEEK(VAR PARM:UNIV PEEK_CO_BLOCK); EXTERN;

PROCEDURE SVC2TMAD(INTVL: INTEGER; TASKQPARM: INTEGER;
                   ELAPSED: BOOLEAN; VAR CC: INTEGER); EXTERN;

PROCEDURE SVC2TMWT(INTVL: INTEGER;
                   ELAPSED: BOOLEAN; VAR CC: INTEGER); EXTERN;

PROCEDURE SVC2TMRP(ITEMCOUNT: SHORTINTEGER; ADDRESS: INTEGER;
                   ELAPSED: BOOLEAN; VAR CC: INTEGER); EXTERN;

PROCEDURE SVC2TMLF(VAR INTVL: INTEGER; TASKQPARM: INTEGER;
                   ELAPSED: BOOLEAN; VAR CC: INTEGER); EXTERN;

PROCEDURE SVC2TMCA(TASKQPARM: INTEGER;
                   ELAPSED: BOOLEAN; VAR CC: INTEGER); EXTERN;

PROCEDURE SVCINITQ(NSLOTS:QSIZE_TY;VAR TASKQ:TASKQ_TYPE); EXTERN;
PROCEDURE SVCTASKQ(VAR PARAM:INTEGER; WAIT:BOOLEAN); EXTERN;

PROCEDURE FROMUDL(I: UDL_INDEX; VAR VAL: UNIV INTEGER); EXTERN;
PROCEDURE TOUDL(I: UDL_INDEX; VAL: UNIV INTEGER); EXTERN;
```

The user has the capability to code SVCs by writing Pascal procedure-call statements, i.e., request SVCs be issued to the operating system to perform I/O to OS/32 files, or other services.

The Pascal procedure-call statements to invoke the SVCs are described below in Sections 10.4.1 to 10.4.24 and some examples are presented in Section 10.4.25.

### 10.4.1 SVC1

A call on: PROCEDURE SVC1(VAR PARM:SVC1_BLOCK); EXTERN;

is of the form:

SVC1(arg1);

where:

arg1 specifies a variable of record-type SVC1_BLOCK with required input field values set by the programmer; status is returned in the appropriate fields (SVC1_STAT and SVC1_DEV_STAT) of the record variable serving as a SVC parameter-block. If the requested function was "test and set" and the record was found to be locked, then the condition code returned by the OS is non-zero. In this case, the procedure SVC1 communicates this information to the caller by returning both both fields SVC1_STAT and SVC1_DEV_STAT equal to hexidecimal #7F.

Action: Issues a direct I/O data transfer or command function as an OS/32 SVC1 Input/Output Request. See Figure 10-1 for an example. Note that mixing SVC1 to a logical unit associated to a file which is also having Pascal READ and WRITEs performed on it as a Pascal named file may produce erroneous results.

### 10.4.2 SVC3

A call on: PROCEDURE SVC3(TERM_CODE : BYTE); EXTERN;

is of the form:

SVC3(arg1);

where:

arg1 is an integer expression that evaluates to a byte value which is to be used as the End of Task Code.

Action: Requests to end task execution, by calling either one of the two PSTERM routines, linked in from PASRTL.OBJ, depending on whether the program has been compiled to run (and is running) under OS/32 or RELIANCE.

### 10.4.3 SVC5

A call on: PROCEDURE SVC5(VAR PARM: SVC5_PARM); EXTERN;

is of the form:

SVC5(arg1);

where:

arg1 specifies a variable of record-type SVC5_PARM, with required input field values set by the programmer; error status is returned in the field SVC5_STAT of the variable.

The user sets the overlay name in SVC5_OVNAME; selects the option #01 (Load lu without positioning) or #04 (Load lu after rewind) in SVC5_OPT; and for TET overlays specifies an lu in SVC5_LU (not required for LINK overlays).

Action: Issues an OS/32 SVC 5 Fetch Overlay, using arg1 as the parameter-block; placing the task in a wait state until the overlay is loaded; with details dependent on whether the overlay was generated with TET or LINK.

### 10.4.4 SVC7

A call on: PROCEDURE SVC7(VAR PARM:SVC7_BLOCK); EXTERN;

is of the form:

SVC7(arg1);

where:

arg1 specifies a variable of record-type SVC7_BLOCK, with required input field values set by the programmer. Device independant status is returned in the field SVC7_STAT of the variable.

Note: Mixing SVC7 with Pascal I/O RESET and REWRITE on the same logical unit is not advised or must be done with extreme care.

Action: Issues an OS/32 SVC 7 File and Device Handling service request using arg1 as the SVC parameter-block. See Figure 10-2 for an example.

### 10.4.5 SVC2PAUS

A call on: PROCEDURE SVC2PAUS; EXTERN;

is of the form:

SVC2PAUS;

Action: Requests to pause task execution, by calling either one of the two PSPAUS routines, linked in from PASPTL.OBJ, depending on whether the program has been compiled to run (and is running) under OS/32 or RELIANCE. Task execution may be resumed upon the operator entering the OS CONTINUE command, under OS/32.

### 10.4.6 SVC2AFLT

A call on: PROCEDURE SVC2AFLT(ENABLE: BOOLEAN); EXTERN;

is of the form:

SVC2AFLT(arg1);

where:

arg1 is a boolean expression. If its value is:

TRUE enables the Arithmetic Fault Interrupt Bit
FALSE disables the Arithmetic Fault Interrupt Bit

Action: Issues an OS/32 SVC 2, Code 4: Set Status Option X'00' to either enable or disable the Arithmetic Fault Interrupt Bit; depending on the value of arg1.

### 10.4.7 SVC2FPTR

A call on:

PROCEDURE SVC2FPTR; EXTERN;

is of the form:

SVC2FPTR;

Action: Issues an OS/32 SVC 2, Code 5, Fetch Pointers; which copies the UTOP, CTOP, and UBOT in the task's TCB and stores them in the task's UDL. (This routine might be called after a CAL routine has used GET/RELEASE storage SVCs to modify UTOP.) Pascal's SVC support routine, FROMUDL, can be called to obtain the values of UTOP, UBOT, and CTOP in the UDL. Also see the routine TOUDL below.

### 10.4.8 SVC2LOGM

A call on:

```
PROCEDURE SVC2LOGM(MSG:LINE; LEN:INTEGER; IMAGE:BOOLEAN);
          EXTERN;
```

is of the form:

```
SVC2LOGM(arg1,arg2,arg3);
```

where:

arg1 specifies a string, (a variable array of characters, a named or literal string-constant), which contains the ASCII message to be logged.

arg2 is an expression evaluating to a positive integer length <= 132; i.e., the number of characters in the message to be logged.

arg3 is a boolean expression. If its value is:

TRUE causes the message to be logged in image mode.
FALSE causes the message to be logged in formatted mode.

Action: Issues an OS/32 SVC 2, Code 7, Log Message to the log device, using arg1 as the string-array of characters (the message) to be logged, arg2 as the length of the message, and to log in image mode if arg3 is TRUE; or in formatted mode if arg3 is FALSE. This routine should not be used in a RELIANCE environment.

### 10.4.9 SVC2FTIM

A call on:

```
PROCEDURE SVC2TIM(VAR TIME:INTEGER; VAR HHMMSS:CHAR8);EXTERN;
```

is of the form:

```
SVC2FTIM(arg1,arg2);
```

where:

arg1 specifies a variable of type INTEGER, which receives the time-of-day as a binary integer, in seconds from midnight.

arg2 specifies a variable of CHAR8 type, i.e., an eight-character string-array; which receives the time in ASCII format.

Action: Issues two forms of OS/32 SVC 2, Code 8, Interrogate Clock; i.e., the time-of-day is fetched in two formats: as a number of seconds from midnight returned in arg1; and as a character string 'hh:mm:ss' returned in arg2.

### 10.4.10 SVC2FDAT

A call on:

```
PROCEDURE SVC2FDAT(VAR MMDDYY:CHAR8); EXTERN;
```

is of the form:

```
SVC2FDAT(arg1);
```

where:

arg1 specifies a variable of CHAR8 type, i.e., an eight-character string-array; which receives the date in ASCII.

Action: Issues an OS/32 SVC 2, Code 9, Fetch Date; i.e., the calendar date is fetched from the OS, in the form 'mm/dd/yy' or 'dd/mm/yy' depending on the preset option in the operating system; and returned in arg1.

### 10.4.11 SVC2TODW

A call on:

```
PROCEDURE SVC2TODW(TOD : INTEGER); EXTERN;
```

is of the form:

```
SVC2TODW(arg1);
```

where:

arg1 is an integer expression which is the number of seconds from midnight; (e.g., the integer 36828 seconds from midnight means 10:13:48).

Action: Issues an OS/32 SVC 2, Code 10, Time of Day Wait; placing the task in a wait state until a specified time-of-day; where arg1 represents the time in seconds from midnight as the time-of-day to resume execution.

### 10.4.12 SVC2INTW

A call on:

```
PROCEDURE SVC2INTW(INTVL : INTEGER); EXTERN;
```

is of the form:

```
SVC2INTW(arg1);
```

where:

arg1 is an integer expression which is the number of milliseconds; (e.g., the integer 5000 milliseconds means wait 5 seconds).

Action: Issues an OS/32 SVC 2, Code 11, Interval Wait; placing the calling task in a wait state until arg1 as the interval to wait, given in milliseconds, expires.

### 10.4.13 SVC2PKNM

A call on:

```
PROCEDURE SVC2PKNM(VAR VAL:INTEGER; BUF:LINE; VAR POSN:INTEGER;
                   OPT:INTEGER; VAR CC:INTEGER); EXTERN;
```

is of the form:

```
SVC2PKNM(arg1,arg2,arg3,arg4,arg5);
```

where:

arg1 specifies a variable of type INTEGER, which receives the converted numeric.

arg2 specifies a string, (a variable array of characters, a named or literal string-constant), that contains the ASCII numeric.

arg3 specifies a variable of INTEGER type, containing the position in arg2 at which the ASCII numeric begins; arg3, upon return, will contain the position of the first character following the ASCII numeric; or in case of syntax error, the position of the character causing the error.

arg4 is an integer expression which selects the conversion option; and must evaluate to the hexidecimal integer values:

| | |
|---|---|
| #00 | to convert ASCII hexidecimal to binary |
| #40 | to convert ASCII hexidecimal to binary, and skip leading blanks |
| #80 | to convert ASCII decimal to binary |
| #C0 | to convert ASCII decimal to binary, skip leading blanks |

arg5 specifies a variable of type INTEGER, which receives the condition code:

| | |
|---|---|
| 0 | Normal termination |
| 1 | No numbers converted |
| 4 | Value of number to be converted > MAXINT |

Action: Issues an OS/32 SVC 2, Code 15, Pack ASCII numeric to binary; optionally skipping leading blanks; depending on the value of arg4.

### 10.4.14 SVC2PKFD

A call on:

```
PROCEDURE SVC2PKFD(VAR FD:UNIV FD_TYPE; BUF:LINE;
                  VAR POSN:INTEGER; SKIP_BLANKS:BOOLEAN;
                   OPT: PACK_OPTION; VAR CC:INTEGER); EXTERN;
```

is of the form:

```
SVC2PKFD(arg1,arg2,arg3,arg4,arg5,arg6);
```

where:

arg1 specifies a variable of type FD_TYPE which receives the packed file descriptor; and since FD_TYPE is preceded by UNIV in the parameter-list of SVC2PKFD arg1 may be any variable that occupies 16 consecutive bytes of storage, and is fullword aligned.

arg2 specifies a string, (a variable array of characters, a named or literal string-constant), which contains the unpacked file-descriptor. (See arg3).

arg3 specifies a variable of INTEGER type containing the position in arg2 at which the unpacked file descriptor begins; arg3, upon return, will contain the position of the first character in arg2 following the unpacked file descriptor, or in case of syntax error, the position of the character causing the error.

arg4 is a boolean expression. If its value is:

| | |
|---|---|
| TRUE | causes leading blanks to be skipped |
| FALSE | causes leading blanks not to be skipped |

arg5 is an expression of type PACK_OPTION. If its value is:

| | |
|---|---|
| USER_VOL | supply user-volume, if none in unpacked fd |
| SYS_VOL | supply system-volume, if none in unpacked fd |
| SPL_VOL | supply spool-volume, if none in unpacked fd |
| NO_DEFAULT | use volume already in packed fd, if none in unpacked fd |

arg6 specifies a variable of type INTEGER which receives the operating system condition code after executing the SVC. This condition code, when other than zero, may contain a combination of the CVGL bits set.

| | |
|---|---|
| 0000 | Normal termination |
| L bit | No volume name in unpacked fd |
| V bit | Syntax error |
| C bit | No extension in unpacked fd |

Action: Issues an OS/32 SVC 2, Code 16, Pack File Descriptor; packing the unpacked file descriptor beginning at position arg3 in arg2, optionally skipping blanks if arg4 is TRUE. Arg5 selects the volume specifier option of the SVC. The packed file descriptor is returned in arg1, the updated position is returned in arg3, and the resultant condition code is returned in arg6.

### 10.4.15 SVC2PEEK

A call on:

```
PROCEDURE SVC2PEEK(VAR PARM:UNIV PEEK_00_BLOCK);EXTERN;
```

is of the form:

```
SVC2PEEK(arg1);
```

where:

arg1 specifies a variable which is a SVC 2, Peek parameter-block This variable must be 7 consecutive fullwords (28 consecutive bytes), fullword aligned; (since PEEK_00_BLOCK is preceded by UNIV in the parameter-list of SVC2PEEK) or it may be a variable of one of the following types:

```
PEEK_00_BLOCK          {type for Option X'00' of SVC 2, Peek}
PEEK_01_BLOCK          {type for Option X'01' of SVC 2, Peek}
PEEK_02_BLOCK          {type for Option X'02' of SVC 2, Peek}
```

To set the fields of arg1,
of type PEEK_00_BLOCK, for an Option X'00' Peek:

```
WITH record-variable DO BEGIN
  PEEK_OPT := #00;
  PEEK_CODE := 19;    {or hexidecimal #13}
  RESERVED  := 0;
  END;

SVC2PEEK(record-variable);
  {On return these fields will contain: }
  PEEK_NLU        largest lu number available to task
  PEEK_MPRI       maximum priority at which task may execute
  PEEK_OSID       name of OS in ASCII as 8 characters
  PEEK_TASK_NAME  task name in ASCII as 8 characters
  PEEK_CTSW       current task status word, bits 0 to 31
  PEEK_TOPT       task options from TCB options field, bits 16-31
```

To set the fields of arg1,
of type PEEK_01_BLOCK, for an Option X'01' Peek:

```
WITH record-variable DO BEGIN
  PEEK_OPT := #01;
  PEEK_CODE := 19;     {or hexidecimal #13}
  RESERVED_1 := 0;
  RESERVED_2 := 0;
  END;

SVC2PEEK(record-variable);
  {On return these fields will contain: }
  PEEK_OSID        OS name in ASCII as 8 characters
  PEEK_OSUP        OS update level in ASCII as 2 characters
  PEEK_CPU         CPU model numbers
  PEEK_SOPT        system options
  PEEK_UACT        user account number from TCB
  PEEK_GACT        group account number from TCB
```

To set the fields of arg1,
of type PEEK_02_BLOCK, for an Option X'02' Peek:

```
WITH record-variable DO BEGIN
  PEEK_OPT := #02;
  PEEK_CODE := 19;     {or hexidecimal #13}
  RESERVED_3 := 0;
  END;

SVC2PEEK(record-variable);
  {On return these fields will contain: }
  PEEK_OSID        OS name in ASCII as 8 characters
  {The next four fields contain the file descriptor in ASCII
   from which the task was loaded. }
  PEEK_LOAD_VOL    volume name in ASCII as 4 characters
  PEEK_FILENAME    file name in ASCII as 8 characters
  PEEK_EXT         extension in ASCII as 3 characters
  PEEK_FILE_CLASS  account code in ASCII as 1 character
```

Action: Issues an OS/32 SVC 2, Code 19, Peek to obtain information from the operating system.

### 10.4.16 SVC2TMAD

A call on:

```
PROCEDURE SVC2TMAD(INTV: INTEGER; TASKQPARM: INTEGER;
                   ELAPSED: BOOLEAN; VAR CC:INTEGER); EXTERN;
```

is of the form:

```
SVC2TMAD(arg1,arg2,arg3,arg4);
```

where:

arg1 is an integer expression specifying an interval of time; or a time-of-day time; at which to schedule the addition to the task queue.

arg2 is an integer expression specifying the task queue parameter. Note: A task queue parameter may be an integer occupying bits 8 to 31 of a fullword (bits 0-7 become internally set to X'09' when added to a task queue); and should be uniquely distinguishable from other task queue parameters; e.g., 1,2,3,#AC4.

arg3 is a boolean expression. If its value is:

| | |
|---|---|
| TRUE | arg1 is an interval of time in milliseconds from now. |
| FALSE | arg1 is the time in seconds from midnight. |

arg4 specifies a variable of type INTEGER which receives the condition code after executing the SVC:

| | |
|---|---|
| 0 | Interval has started, normal termination. |
| 4 | Sufficient amount of system space unavailable. |

Action: Issues an OS/32 SVC 2, Code 23, Option X'0C' which schedules the addition of the arg2 task queue parameter to the task queue at a time which is determined by arg1 (arg1 means milliseconds from now if arg3 is TRUE or arg1 is the time in seconds from midnight if arg3 is FALSE). The condition code is returned in arg4. The task's subsequent statements continue to execute concurrently with the awaited scheduled addition to the task queue.

Note: Before executing this call, SVCINITQ must have been called to initialize the current TSW (Z bit set), allocate a task queue, and store its address in the UDL.

### 10.4.17 SVC2TMWT

A call on:

```
PROCEDURE SVC2TMWT(INTVL: INTEGER;
                   ELAPSED: BOOLEAN; VAR CC: INTEGER); EXTERN;
```

is of the form:

**SVC2TMWT(arg1,arg2,arg3);**

where:

arg1 is an integer expression specifying an interval of time; or a time-of-day time. The task is placed in a wait state, until the time specified elapses or occurs.

arg2 is a boolean expression. If its value is:

| | |
|---|---|
| TRUE | arg1 is an interval of time in milliseconds from now. |
| FALSE | arg1 is the time in seconds from midnight. |

arg3 specifies a variable of INTEGER type, which receives the condition code after executing the SVC:

| | |
|---|---|
| 0 | Interval has started, normal termination. |
| 4 | Sufficient amount of system space unavailable, no wait has occurred. |

Action: Issues an OS/32 SVC 2, Code 23, Option X'80' by which the calling task is placed in a time wait state until the specified time interval has elapsed, or the specified time-of-day occurs. No item is added to the calling task's queue, at the end of the elapsed time.

### 10.4.18 SVC2TMRP

A call on:

```
PROCEDURE SVC2TMRP(ITEMCOUNT: SHORTINTEGER; ADDRESS:INTEGER;
                   ELAPSED: BOOLEAN; VAR CC: INTEGER); EXTERN;
```

is of the form:

```
SVC2TMRP(arg1,arg2,arg3,arg4);
```

where:

arg1 is a shortinteger expression which is the number of items (task queue parameters) to be scheduled for addition to to the task queue. The entire group of scheduled queue additions are to be cyclically repeated within a specific time period.

arg2 is the address of an array defining the intervals or times and the associated task queue parameters; e.g., the first fullword of the array is a time specifier, and the second fullword of the array is a task queue parameter to be added to the task queue at the time specified in the first fullword of the array; and so on, in pairs. The number of items to be added is one for each pair of elements in the array; i.e., the number of items is the number of task queue parameters.

arg3 is a boolean expression. If its value is:

| | |
|---|---|
| TRUE | the intervals in arg2 are interpreted as milliseconds from now. |
| FALSE | the times in arg2 are interpreted as seconds from midnight. |

Note: The entire time period of a repetitive cycle, is that defined by the operating system. For interval timing, the entire cycle is the sum of intervals; for time-of-day timing, the entire cycle is the minimum range of the number of days delineated by the time-of-day specifications. That is, if all time-of-days occur on one day, they will begin again on the second day. If all time-of-days take two days to occur, and when the first time-of-day of the cycle has not already occurred on the third day, the cycle begins repeating on the third day; otherwise when the first time-of-day of the cycle has already passed by, the cycle begins repeating on the fourth day.

arg4 specifies a variable of type INTEGER, which receives the condition code after executing the SVC:

| | |
|---|---|
| 0 | Normal termination. |
| 4 | Sufficient amount of system space is unavailable, no interval is elapsing. |

Action: Issues an OS/32 SVC 2, Code 23, Option X'40' which repetitively schedules the addition of several (arg1) task queue parameters in the structure at address arg2, with the type of times determined by arg3, and a resulting condition code after executing the SVC is returned in arg4. The repetition recycles until the task terminates or cancels (via SVCTMCA) the SVC call.

Note: Before executing this call, SVCINITQ must have been called to initialize the current TSW (Z bit set), allocate a task queue, and store its address in the UDL.

### 10.4.19 SVC2TMLF

A call on:

```
PROCEDURE SVC2TMLF(VAR INTVL: INTEGER; TASKQPARM:INTEGER;
                   ELAPSED: BOOLEAN; VAR CC: INTEGER); EXTERN;
```

is of the form:

```
SVC2TMLF(arg1,arg2,arg3,arg4);
```

where:

arg1 specifies a variable of type INTEGER, which receives the remaining time left, before the elapse of a previously scheduled timer interval or time for a given task queue parameter, occurs.

arg2 is an integer expression specifying the task queue parameter being investigated.

arg3 is a boolean expression. If its value is:

| | |
|---|---|
| TRUE | find the remaining time, as an interval in milliseconds. |
| FALSE | find the remaining time, as a time-of-day in seconds from midnight. |

arg4 specifies a variable of type INTEGER, which receives a condition code after executing the SVC:

| | |
|---|---|
| 0 | Normal termination. |
| 4 | No interval is associated with the given task queue parameter. |

Action: Issues an SVC 2, Code 23, Option X'20' which finds the time remaining before the elapse of the interval or time-of-day associated with the arg2 task queue parameter previously established with Option X'00' (via SVC2TMAD) or with Option X'40' (via SVC2TMRP).

### 10.4.20 SVC2TMCA

A call on:

```
PROCEDURE SVC2TMCA(TASKQPARM: INTEGER;
                    ELAPSED: BOOLEAN; VAR CC:INTEGER); EXTERN;
```

```
SVC2TMCA(arg1,arg2,arg3);
```

where:

arg1 is an integer expression specifying the task queue parameter.

arg2 is a boolean expression. If its value is:

| | |
|---|---|
| TRUE | arg1 means the interval in milliseconds from now. |
| FALSE | arg1 means the time-of-day in seconds from midnight. |

arg3 specifies a variable of type INTEGER, which receives the condition code after executing the SVC:

| | |
|---|---|
| 0 | Normal termination. |
| 4 | No previous interval request exists that matches the task queue parameter provided. |

Action: Issues an OS/32 SVC 2, Code 23, Option X'10' to cancel a previously established timer management request; concerning this task queue parameter. All previous requests that match both the type of time and task queue parameter are cancelled. If the task queue parameter and its associated time is part of a periodic group (via SVC2TMRP), the entire time period is cancelled.

### 10.4.21 SVCINITQ

A call on:

```
PROCEDURE SVCINITQ(NSLOTS:QSIZE_TY;VAR TASKQ:TASKQ_TYPE);EXTERN;
```

is of the form:

```
SVCINITQ(arg1,arg2);
```

where:

arg1 is an integer expression specifying the number of slots (i.e., maximum number of task queue parameters to be held at one time) to establish in a circular list which is to serve as the Task Queue. The value of arg1 must lie in the subrange established by the type QSIZE_TY (predefined to be 1..TASKQ_SLOT_COUNT, where TASKQ_SLOT_COUNT is predefined to be 4). These can be redefined by the user for larger task queues (in which case arg1 could be > 4).

arg2 specifies a record variable of type TASKQ_TYPE which will be initialized as the circular list, with arg1 slots, to serve as the Task Queue.

Action: The address of arg2 is placed in the UDL as the Task Queue. Arg2 is initialized as a Perkin-Elmer circular list, with arg1 number of slots placed in its first halfword. The current TSW is changed to enable Task Queue entry traps by issueing an OS/32 SVC 9, Load TSW. (The TSW gets the logical sum ("or") of the old TSW with Y'0000DF20'). The Z bit enables the additions to the queue on time-out to occur. The Q bit is not set, so there is no Queue Service routine handling the additions to the queue as they occur on time-out of the elapses. The Pascal program can retrieve parameters from the queue with SVCTASKQ.

### 10.4.22 SVCTASKQ

A call on:

```
PROCEDURE SVCTASKQ(VAR PARAM:INTEGER; WAIT:BOOLEAN); EXTERN;
```

is of the form:

```
SVCTASKQ(arg1,arg2);
```

where:

arg1 specifies a variable of type INTEGER, which receives the top task queue parameter entry on the Task Queue; unless the Task Queue is empty (see Action below).

arg2 is a boolean expression. If its value is:

| | |
|---|---|
| TRUE | return a task queue parameter,<br>unless the Task Queue is empty;<br>if empty, enter TSW (Q bit set) wait,<br>until one arrives. |
| FALSE | return a task queue parameter, but<br>don't wait, if queue empty; return 0 in arg1. |

Action: Assuming SVCINITQ and other timer management requests have been previously called; this routine attempts to remove the top entry (a task queue parameter) from the Task Queue and return it in arg1.

If the user specified arg2 as FALSE, and the Task Queue is empty, arg1 will be returned as zero.

If the user specified arg2 as TRUE, this routine returns the top entry in the task queue; unless the task queue is empty. If the task queue is empty, this routine goes into a wait. The TSW gets the logical sum ("or") of the old TSW with Y'8800DF20'. This routine waits for an entry to be added to the Task Queue (by a previously issued timer management request). Upon its addition, this routine processes the new entry with a service routine within SVCTASKQ, and returns the new entry in arg1, reestablishing the current TSW (with no Wait or Q bit set, and the Z bit set - the TSW gets the logical sum ("or") of the old TSW with Y'0000DF20'. (Bits 16,17,19,20,21,22,23, and 26 of the TSW are set.) Refer to the OS/32 PRM on the TSW.

The user specifies a task queue parameter as an integer that fits in a fullword bits 8-31, in the timer-management requests, but when it is added to a Task Queue on time-out of an timer management request, the OS superimposes X'09' in bits 0-7 of the task queue parameter binary integer; (to distinguish it from other types of queue parameters).

SVCTASKQ returns the entry in the Task Queue without stripping the X'09' in bits 0-7 preceding the task queue parameter in bits 8-31.

### 10.4.23 FROMUDL

A call on:

```
PROCEDURE FROMUDL(I:UDL_INDEX; VAR VAL: UNIV INTEGER); EXTERN;
```

is of the form:

```
FROMUDL(arg1,arg2);
```

where:

arg1 is an integer expression which is within the range 0 to 63 and of compatible type to UDL_INDEX; (an index into the UDL).

arg2 specifies a variable of at least four bytes in length (as UNIV precedes the INTEGER parameter type) which receives the fullword at UDL[arg1] (thinking of the UDL as an ARRAY[UDL_INDEY] OF INTEGER).

Action: Given the value of arg1 as 0 to 63, returns the value of UDL[arg1] into arg2. For example: assuming V is a variable of type INTEGER;

```
     FROMUDL(0,V);     {returns CTOP into V}
     FROMUDL(1,V);     {returns UTOP into V}
     FROMUDL(2,V);     {returns UBOT into V}
     FROMUDL(8,V);     {returns UDL.EXT into V}
```

### 10.4.24 TOUDL

A call on:

```
PROCEDURE TOUDL(I: UDL_INDEX; VAL: UNIV INTEGER); EXTERN;
```

is of the form:

```
TOUDL(arg1,arg2);
```

where:

arg1 is an integer expression which is within the range 0 to 63 and compatible type to UDL_INDEX; (as an index into the UDL).

arg2 specifies a variable of least four bytes in length (as UNIV precedes the INTEGER parameter type) which is the value to be stored in the UDL at UDL[arg1], (thinking of the UDL as an ARRAY[UDL_INDEX] OF INTEGER).

Action: Replaces the fullword at UDL[arg1] with the value expressed by arg2. For example: assuming v is of type INTEGER, such as ADDRESS(V);

```
     TOUDL(0,v);       {replaces UDL[0] with the value v}
     TOUDL(1,v);       {replaces UDL[1] with the value v}
```

### 10.4.25 SVC Examples

Examples of utilizing the SVC capability in Pascal written programs, are presented in the following sample program fragments in Figures 10-1 for SVC1, 10-2 for SVC7, and SVC2PKFD; and Figure 10-3 for SVC2FTIM, SVC2FDAT, SVC2LOGM, and SVC2PAUS.

```
PROGRAM SVC1_EXAMPLE;

{The type-definition, referenced in the SVC1 parameter-list
 and VAR part, must first be declared in a TYPE declaration}

TYPE SVC1_BLOCK=RECORD
                  SVC1_FUNC:BYTE;
                  SVC1_LU:BYTE;
                  SVC1_STAT:BYTE;
                  SVC1_DEV_STAT:BYTE;
                  SVC1_BUFSTART:INTEGER;
                  SVC1_BUFEND:INTEGER;
                  SVC1_RANDOM_ADDR:INTEGER;
                  SVC1_XFER_LEN:INTEGER;
                  SVC1_RESERVED:INTEGER;
                 END;

{A variable is declared to serve as an SVC1 parameter-block}

VAR MY_BLOCK : SVC1_BLOCK;
    BUFFER   : ARRAY[1..46] OF CHAR;

{The external procedure-declaration defining the SVC1 interface
parameter-list and declaring the routine SVC1 as an EXTERN must
reside in the routine-declarations part of a block.}

PROCEDURE SVC1(VAR PARM:SVC1_BLOCK);EXTERN;

BEGIN

  BUFFER := 'Now is the time for all good men to come aid!!';

  WITH MY_BLOCK DO    {Assign values to the variable MY_BLOCK}
    BEGIN
      SVC1_FUNC:=#28;         {WRITE ASCII and WAIT}
      SVC1_LU:=3;             {to logical unit 3}
      SVC1_BUFSTART:=ADDRESS(BUFFER);
      SVC1_BUFEND:= ADDRESS(BUFFER) + SIZE(BUFFER) - 1 ;
    END;

  SVC1(MY_BLOCK);       {Writes BUFFER contents to lu 3}

  IF MY_BLOCK.SVC1_STAT <> 0 THEN ...{check independent status}

  IF MY_BLOCK.SVC1_DEV_STAT <> 0 THEN ...{check on device status}

  ...

END.
```

Figure 10-1. Programming an SVC 1 in Pascal

```
PROGRAM EXAMPLE_SVC2PKFD_AND_SVC7;


CONST

STRINGC = 'PRINTER.LST';          {unpacked file descriptor}

{The type-definitions referenced in the EXTERN declarations
 or VAR part must first be declared in a TYPE definitions part.}

TYPE CHAR3 = PACKED ARRAY [1..3] OF CHAR;
TYPE CHAR8 = PACKED ARRAY [1..8] OF CHAR;
TYPE CHAR4 = PACKED ARRAY [1..4] OF CHAR;

TYPE LINE = ARRAY[1..132] OF CHAR;

TYPE FD_TYPE = PACKED RECORD
       VOLN : CHAR4               {VOLUME NAME}
       FN : CHAR8                 {FILE NAME}
       EXTN : CHAR3;              {EXTENSION}
       ACCT : CHAR;               { P, G, S }
     END;

TYPE SVC7_BLOCK=RECORD
       SVC7_CMD:BYTE;             {COMMAND}
       SVC7_MOD:BYTE;             {MODIFIER/DEVICE TYPE}
       SVC7_STAT:BYTE;            {STATUS}
       SVC7_DEV_LU:BYTE;          {LOGICAL UNIT NUMBER}
       SVC7_KEYS:SHORTINTEGER;    {READ/WRITE KEYS}
       SVC7_RECLEN:SHORTINTEGER;  {LOGICAL RECORD LENGTH}
       SVC7_FD:FD_TYPE;           {FILE DESCRIPTOR}
       SVC7_SIZE: INTEGER;        {FILE/INDEX SIZE}
     END;

TYPE PACK_OPTION = (USER_VOL, SYS_VOL, SPL_VOL,NO_DEFAULT);

{A variable is declared to serve as an SVC7 parameter-block}

VAR OUR_BLOCK : SVC7_BLOCK;       {declares variable OUR_BLOCK}
    V       : INTEGER;            {variable for returning POSN}
    CCODE   : INTEGER;            {variable for returning cc code}
    STRINGV : ARRAY[1..11] OF CHAR;    {declares string variable}

{The external procedure-declarations are:}

PROCEDURE SVC7(VAR PARM:SVC7_BLOCK);EXTERN;
PROCEDURE SVC2PKFD(VAR FD:FD_TYPE; BUF:LINE;
                   VAR POSN:INTEGER; SKIP_BLANKS:BOOLEAN;
                   OPT: PACK_OPTION; VAR CC:INTEGER);  EXTERN;
```

Figure 10-2. Programming an SVC 7 in Pascal (Part a)

```
BEGIN

  V := 1;                 {Position in which string starts}

  STRINGV := STRINGC;            {Put constant string into variable}

  {Put packed file descriptor into SVC7_FD field with SVC2PKFD}

  SVC2PKFD(OUR_BLOCK.SVC7_FD,STRINGV,V,FALSE,USER_VOL,CCODE);

     {Assuming user volume is M300 and running under OS/32 MTM}
     {OUR_BLOCK.SVC7_FD contains the string 'M300PRINTER LSTP'}

  WITH OUR_BLOCK DO  {Assign values to the variable OUR_BLOCK}

    BEGIN
      SVC7_CMD:=#C0;             {ALLOCATE and ASSIGN}
      SVC7_MOD:=#E0;             {ERW and CONTIGUOUS}
      SVC7_STAT:=0;
      SVC7_LU:=8;                {Assignment to lu 8}
      SVC7_KEYS:=#0000;          {Unconditionally unprotected}
      SVC7_RECLEN:=132;          {logical record length}
      SVC7_SIZE := 16;           {Number of sectors}
    END;

  SVC7(OUR_BLOCK);                {Call the SVC7}

  {A 16 sector contiguous file named PRINTER.LST has been
   allocated on the user private account on volume M300
   (assuming the user volume is M300 and running under MTM)
   with Exclusive Read Write (ERW); with no protection keys;
   and the file named M300:PRINTER.LST/P is assigned to lu 8}

  IF OUR_BLOCK.SVC7_STAT <> 0 THEN ...{a check on status}

  ...

END.
```

Figure 10-2. Programming an SVC 7 in Pascal (Part b)

```
PROGRAM SVC2_EXAMPLES;

{Example constants: }
CONST STARTUP_MSG = 'This program started at:   ';

{The type-definitions referenced in the SVC2 parameter-lists
 or VAR part must first be declared in a TYPE definitions part.}

TYPE CHAR8 = PACKED ARRAY [1..8] OF CHAR;
TYPE LINE = ARRAY [1..132] OF CHAR;

{Variable declarations}

VAR BUFFER:LINE;
    I,J,K:INTEGER;
    ATIME, ADATE : CHAR8;

{The external procedure-declarations defining the SVC2 interfaces
and declaring the SVC2 routines as EXTERNs must reside in the
routine-declarations part of a block.}

PROCEDURE SVC2PAUS; EXTERN;
PROCEDURE SVC2LOGM(MSG:LINE; LEN:INTEGER; IMAGE:BOOLEAN); EXTERN;
PROCEDURE SVC2FTIM(VAR TIME:INTEGER; VAR HHMMSS: CHAR8); EXTERN;
PROCEDURE SVC2FDAT(VAR MMDDYY:CHAR8); EXTERN;

{For example, to log a message and then pause the user task:}

BEGIN

  FOR J := 1 TO 27  DO
      BUFFER[J] := STARTUP_MSG[J];
  I := 27;
  SVC2FTIM(K,ATIME);               {fetch the time}
  FOR J := 1 TO 8 DO BEGIN
    I := I + 1;
    BUFFER[I] := ATIME[J];         {move time into message buffer}
    END;
  I := I + 1;
  BUFFER[I]:= ' ';                 {space between time and date}
  SVC2FDAT(ADATE);                 {fetch today's date}
  FOR J := 1 TO 8 DO BEGIN
    I := I + 1;
    BUFFER[I] := ADATE[J];         {move date into message buffer}
    END;
  SVC2LOGM(BUFFER,I,FALSE);        {log the message of length I}
                                   {in formatted ASCII mode}
  SVC2PAUS;                        {pause the task}
  {After OS CONTINUE command, execution continues here}

  ...

END.
```

Figure 10-3. Programming some SVC 2's in Pascal

## 10.5 REGISTER USAGE IN THE EXECUTING PASCAL TASK

The machine-dependant compiler-generated object code for Pascal user programs, contains specific uses of the machine registers. All of the sixteen General Registers, R0 to R15, are usually in use in a running Pascal program. If REAL and SHORTREAL data-types are in use in the source program, the Double-precision and Single-precision Floating-point registers are in use in the object code.

Certain general registers are allocated specific run time system uses and must be preserved as such.

The general registers are allocated for the following uses:

| | |
|---|---|
| R0 | Contains the STACK LIMIT (SL), see Section 10.6.2 |
| R1 | Contains the GLOBAL BASE (GB), see Section 10.6.2 |
| R2 | Contains the LOCAL BASE (LB), see Section 10.6.2 |
| R3 to R13 | General usage and argument/parameter passing, see Section 10.7.1 |
| R14 | General usage |
| R15 | Linkage register between routine activations, see Section 10.6.4 and 10.7. |

If the Pascal REAL data-type is in use in the source program, operations with the Pascal REAL data-type involve machine-instructions using the eight Double-precision floating-point registers:

D0 to D14 are used for manipulating REAL data and particularly for passing argument data to REAL value-parameters of a called routine.

Programs using REALs should be established with the OS/32 LINK OPTION command selecting DFLOAT.

If the Perkin-Elmer Pascal SHORTREAL data-type is in use in the source program, operations on data of SHORTREAL type involve machine-instructions using the eight Single-precision floating-point registers:

F0 to F14 are used for manipulating SHORTREALs and particularly for passing argument data to SHORTREAL value-parameters of a called routine.

Programs using SHORTREALs should be task established using the OS/32 LINK OPTION command selecting the FLOAT option.

Programs using both SHORTREALs and REALs should be task established using the OS/32 LINK OPTION command selecting both the FLOAT and DFLOAT options.

## 10.6 MEMORY UTILIZATION

### 10.6.1 Internal Data Storage Representations

Every variable in the Pascal program, and every constant that is stored in memory, has a representation that is determined by its type. The method of representing a type does not depend on the kind of language entity the datum is; i.e., representing a type is not affected by whether the datum is a constant, global variable, local variable, or dynamically created variable. If a datum is part of a larger one, then its internal representation is not affected, except that the location of parts of a PACKED type are not aligned according to their normal alignment requirements.

The user may obtain the value of the size of any datum in Pascal code by calling on the function SIZE. SIZE returns as its function-value, the size in bytes of its argument; (see Section 3.5.8).

An example:

```
PROGRAM SIZES (OUTPUT);
VAR A:BYTE;B:BOOLEAN;C:CHAR;I:INTEGER;
    S:SHORTINTEGER; SR:SHORTREAL; R:REAL;
    RA : ARRAY[1..20] OF REAL;
BEGIN
  WRITELN('A OCCUPIES ',SIZE(A),' BYTES');
  WRITELN('B OCCUPIES ',SIZE(B),' BYTES');
  WRITELN('C OCCUPIES ',SIZE(C),' BYTES');
  WRITELN('I OCCUPIES ',SIZE(I),' BYTES');
  WRITELN('S OCCUPIES ',SIZE(S),' BYTES');
  WRITELN('SR OCCUPIES ',SIZE(SR),' BYTES');
  WRITELN('R OCCUPIES ',SIZE(R),' BYTES');
  WRITELN('RA OCCUPIES ',SIZE(RA),' BYTES');
  WRITELN('RA[1] OCCUPIES ',SIZE(RA[1]),' BYTES');
END.
```

The user may obtain the machine address of any datum in Pascal code by calling on the function ADDRESS. ADDRESS returns as its function-value, the machine address of its argument, as an INTEGER; (see Section 3.5.8).

However, to output the address in hexadecimal format, a conversion routine to convert the integer address (returned by ADDRESS) into ASCII hexadecimal format would have to be written.

The storage sizes and machine address alignment requirements of datum of the various Pascal data-types, measured in bytes, are listed in Table 10-7, below.

### TABLE 10-7. INTERNAL DATA REPRESENTATIONS

| Data Type | Storage Size(in bytes) | Alignment(by bytes) |
|---|---|---|
| CHAR | 1 | 1 |
| BOOLEAN | 2 | 2 |
| BYTE | 1 | 1 |
| SHORTINTEGER | 2 | 2 |
| INTEGER | 4 | 4 |
| SHORTREAL | 4 | 4 |
| REAL | 8 | 4 |
| Enumeration | 2 | 2 |
| Subrange | | |
| of CHAR | 1 | 1 |
| of SHORTINTEGER | 2 | 2 |
| of INTEGER | 4 | 4 |
| of Enumeration | 2 | 2 |
| Pointer-type | 4 | 4 |
| SET | 16 | 4 |
| FILE (non-text) | 80-byte FCB + component-type size | 4 |
| FILE (TEXT) | 80-byte FCB + 256-byte buffer | 4 |

TABLE 10-7. INTERNAL DATA REPRESENTATIONS (continued)

| | | |
|---|---|---|
| ARRAY element | component-type size, plus possible filler gaps, if structured or part of an unpacked ARRAY. | alignment of component-type, unless part of a PACKED array. |
| ARRAY | Product of the number of values in each index-type times size allotted to component-type; plus possible filler gaps. | 4 unless part of a PACKED structure. |
| PACKED ARRAY | Product of the number of values in each index-type times size allotted to component-type. | 4 unless part of a PACKED structure. |
| RECORD field | size of field-type plus possible filler gap if RECORD is unpacked. | alignment of field-type; unless part of a PACKED record. |
| RECORD | size of all fields plus any filler gaps including size of largest variant plus tag field, if any. | 4 unless part of a PACKED structure. |
| PACKED RECORD | size of all fields including size of largest variant plus tag field, if any. | 4 unless part of a PACKED structure. |

Byte alignment allows a datum to begin on any machine address boundary (i.e., the last hex digit of the address can be 0,1,2,3,4,5,6,7,8,9,A,B,C,D,E, or F).

Halfword alignment (2-byte alignment) allows a datum to begin on any machine address boundary which is a multiple of 2 (i.e., the last hex digit of the address can be 0, 2, 4, 6, 8, A, C, or E).

Fullword alignment (4-byte alignment) allows a datum to begin at any machine address boundary which is a multiple of 4 (i.e., the last hex digit of the address can be 0, 4, 8, or C).

Descriptions of the internal representations of each of the Pascal data-types follow.

### CHAR Type

A datum of the CHAR type occupies one byte of storage and its location is byte addressable. The internal representation within the byte of a datum of type CHAR is an ASCII untagged character code, whose value is in the range from X'00' to X'7F'.

### BYTE Type

A datum of the BYTE type occupies one byte of storage and its location is byte aligned, i.e., byte addressable. The values within a datum of the BYTE type is an unsigned integer, internally represented in the hexidecimal range X'00' to X'FF', corresponding decimally to the range from 0 to 255.

### SHORTINTEGER Type

A datum of the SHORTINTEGER type occupies one halfword (2 bytes) of storage; and its location is halfword aligned, when not part of a PACKED structure. The values of a SHORTINTEGER datum, is a signed integer, internally represented by the hexidecimal range X'8000' to X'FFFF' for -32768 to -1, X'0000' for zero, and the range X'0001' to X'7FFF' for +1 to +32768. The hardware binary representation of shortintegers is two's complement.

### INTEGER type

A datum of the INTEGER type, occupies a fullword (4 bytes) of storage and its location is fullword aligned, when not part of a PACKED structure. The values of an INTEGER datum, are internally represented by the hexidecimal range Y'800000000' to Y'FFFFFFFF' for -2147483648 to -1, Y'00000000' for zero, and the range Y'00000001' to Y'7FFFFFFF' for +1 to +214748347. The hardware binary representation of integers is two's complement.

### REAL and SHORTREAL Types

A datum of the REAL type occupies two fullwords (8 bytes) of storage; and its location is fullword aligned, when not part of a PACKED structure. A datum of the SHORTREAL type occupies one fullword (4 bytes) of storage; and its location is fullword aligned, when not part of a PACKED structure. The hardware representation of REAL and SHORTREAL numbers is excess 64 notation or base-16 floating point: one bit sign, 7 bits exponent, and 56 bits (for REAL) or 24 bits (for SHORTREAL) "mantissa", and the exponent is biassed +64.

### BOOLEAN Type

A datum of the BOOLEAN type occupies a halfword of storage and its location is halfword aligned, when not part of a PACKED structure. The values of a BOOLEAN datum are internally represented by either X'0000', a zero, for FALSE; or X'0001', a one, for TRUE.

### User-defined Enumeration type

A datum of a user-defined enumeration type occupies a halfword of storage (2 bytes); and is halfword aligned; when not part of a PACKED structure. The machine representation of the values of a user-defined enumeration type are internally represented by its ordinal value as a 2-byte integer, within the halfword. The possible values internally are X'0000' to X'007F' within the halfword, as the user-defined enumeration types are restricted to 128 members (an implementation-defined restriction).

### Subrange type

The representation of a subrange type is the same as that of its enclosing type. That is, a datum of the subrange type occupies the same amount of storage and inherits the alignment requirements as that of a datum of its enclosing type. For example, a subrange variable whose enclosing type is CHAR, occupies a byte of storage and its location is byte aligned; a subrange variable whose enclosing type is INTEGER and those integers are outside the shortinteger range occupy a fullword and their locations are fullword aligned. Subranges of INTEGER are represented as SHORTINTEGER, if their minimum and maximum values are in the range -32768..32767.

### Pointer type

A pointer-type occupies a fullword (4 bytes) of storage; and its location is fullword aligned, when not part of a PACKED structure. That is, the size of a pointer variable is not dependant on its associative target-type. The internal representation of the value of a pointer-type within the fullword is a machine address.

### Records

The alignment of any record is fullword, except if it is directly part of a PACKED structure. When part of a packed structure, its alignment may or may not be fullword aligned.

The fields of a record appear internally in the order in which they are declared in the record-type.

Each field of a record has its allotted size, depending of the field-type type declaration within the record-type definition. The sizes of fields of the simple types are listed in Table 10-7. The number of bytes allotted to a structure-typed field may be extended with filler gaps in an unpacked record.

In an unpacked record, the alignment of a field of simple type is its usual alignment, but that of a structure-typed field is fullword.

For each field of an unpacked record, its beginning offset from the starting location of the record, is the next available multiple of the alignment requirement of that field's field-type. Also, the number of bytes allotted to the field (its occupied space) is rounded up to be a multiple of the same alignment requirement such that there may be trailing filler gaps, in some cases.

For example a record field of type: ARRAY[1..6] of CHAR, has as alignment requirement of 4 (as it is an array in a unpacked record). If the array were the first field in a record not part of a higher packed structure (or following a field ending just prior to a fullword) no filler gaps would precede the array in the record; but if the array followed a field of BYTE type, occupying the first byte on a fullword alignment, three filler bytes would precede the array so that its alignment is a multiple of 4; and its allotted field size would be 8 bytes (also a multiple of 4); causing two extra trailing bytes to exist in the array. The three filler bytes preceding the array are part of the size of the record and not part of the size of the field array nor of its preceding BYTE type; just as the two filler bytes trailing the array are not part of the size of the array, but part of the record.

Thus there may be dead space, or "alignment gaps", in an unpacked record (caused by a field's alignment requirements in relation to its predecessor field, or by the structure-typed field's size being extended to occupy extra allotted space for subsequent compiler-generated code ease-of-access by fullwords).

For example, consider the record variable, RECVAR, below; to be of type RECTYPEA, which is defined as:

```
TYPE RECTYPEA = RECORD
                  B1 : BYTE;
                  A1 : ARRAY[1..6] OF CHAR;
                  S1 : SHORTINTEGER;
                END;
VAR RECVAR : RECTYPEA;
```

The size of RECVAR is 14 bytes and its alignment is fullword. B1 occupies the first byte of the record and is trailed by three filler bytes; A1 begins in the second fullword in the record, contains six characters and is trailed by two bytes; S1 begins in the fourth fullword and occupies its first two bytes.

When a record-type possesses variants, its usual size includes that of the largest variant. If a tag-field is specified, its size is the size of the tag-field type and its alignment is that of the tag-field type. If no tag-field is specified, no storage is allocated for a tag-field.

The user may direct the removal of filler gaps between fields on the first level of subdivision by declaring the record to be a PACKED RECORD; and may also direct the removal of filler gaps within fields by declaring structure-typed fields as PACKED types. (See PACKED types below). To achieve an entirely packed structure, when the structure contains other structures, both the entire structure type-definition and all of its structure-typed components must be declared as PACKED types. However, parts of a packed structure cannot be passed to variable parameters.

## Arrays

The alignment requirement of any array is fullword, except if it is directly part of a PACKED structure. When part of a packed structured, its alignment may or may not be fullword.

In an unpacked array, the alignment requirement of each element is the alignment requirement of the component-type of the array-type definition; (See Table 10-7). That is, the elements of an unpacked array have their alignment determined by the array's component-type, which may or may not be a PACKED structure itself.

Note that if the component-type is of a structured type, then each array element is fullword aligned, in an unpacked array. Additionally, the number of bytes alloted to each element is rounded up, if necessary, to be the next available multiple of the component-type alignment requirement. This means, there may be trailing filler gaps after and within each structure-typed element in an unpacked array.

The array consists of copies of the component-type (its allotted space); where the number of copies is the product of the number of values in each index-type, when there is more than one index-type; or when there is only one index-type, the number of values in that index-type. Therefore, the size of an array is the allotted space of the component-type times the number of copies. The allotted space of an element may be extended from its size as an isolated datum of the component-type, to safeguard the alignment requirements of its subsequent copy as an array-element.

For example, consider an array of records, such as the variable ARRAYREC below, to be an array of record-type RECTYPEA, where RECTYPEA is defined as:

```
     TYPE RECTYPEA = RECORD
                      B1:BYTE;
                      A1:ARRAY[1..6] OF CHAR;
                      S1:SHORTINTEGER;
                     END;
          ARAYTYPEA = ARRAY[1..3] OF RECTYPEA;
     VAR  ARRAYREC : ARAYTYPEA;
```

The array variable ARRAYREC consists of three copies of the record-type; i.e., although the record-type size is only 14 bytes, the allotted size to each record in an unpacked array is 16 bytes per element. Therefore, the size of the unpacked array is 3 * 16 = 48 bytes.

Thus, in an unpacked array, there may be an alignment gap after each structure-typed element.

Note that a multi-dimensioned unpacked array of some component-type, may also contain filler gaps, even if the component-type is not structured (such as CHAR and BYTE), since a multi-dimensioned array is considered to be an array of arrays.

The user may direct the removal of the filler gaps between elements of an array by declaring the array to be a PACKED ARRAY, and direct the removal of filler gaps within a structure-typed component-type by declaring the component-type to be PACKED. However, parts of a packed structure, can not be passed to variable parameters.

## PACKED types

When a structure is PACKED, then its elements or fields are treated as if they had one-byte alignments. That is, alignment gaps are squeezed out of the structure wherever they would normally exist in an unpacked version of the structure, between its elements or fields. The alignment of the whole packed structure is fullword.

Packing an array or record does not affect the internal structure of each of its elements or fields; but removes any filler gaps that may exist between the the elements or fields. If these elements or fields are themselves of unpacked structured types, displacements of their parts are the same as if they were not inside a larger packed type.

To pack the examples given above for records and arrays:

```
     TYPE RECTYPEB = PACKED RECORD
                       B1:BYTE;
                       A1:ARRAY[1..6] OF CHAR;
                       S2:SHORTINTEGER;
                     END;
          ARAYTYPEB = PACKED ARRAY[1..3] OF RECTYPEB;
     VAR  RECVAR2 : RECTYPEB;
          ARAYREC2 : ARRAY[1..3] OF RECTYPEB;
```

produces RECVAR2 to be a record variable, fullword aligned, occupying nine consecutive bytes. ARAYREC2, fullword aligned, contains three copies of RECTYPEB records, but occupying 3 * 12 = 36 bytes, as ARAYREC2 is not packed. Declaring ARAYREC3 as follows:

```
     VAR ARAYREC3 :  PACKED ARRAY[1..3] OF RECTYPEB;

     or

     VAR ARAYREC3 :  ARAYTYPEB;
```

produces ARAYREC3 to be a packed array variable, fullword aligned, containing three copies of packed records, the entire array occupying 3 * 9 = 27 consecutive bytes.

## Sets

A set is 16 bytes long (128 bits), aligned on a 4-byte boundary. A datum of the set type always occupies four fullwords of storage; and its location is fullword aligned, when not part of a PACKED structure. The internal representation of a set value is not dependant on the base member type of the set. A set may have up to 128 members. Each bit may be turned on to indicate that an element is a member in the set, or turned off to indicate that an element is not a member in the set. The leftmost bit indicates the member with ordinal number 0, and the rightmost bit indicates the member with ordinal number 127.

## Files

A file-variable occupies storage whose location is fullword aligned. Its size is a buffer size + 80 bytes for control information. For a non-TEXT file, the buffer size is the same as one datum of the component-type of the file, and this buffer is the Pascal file buffer variable, f^, associated with the file-variable, f. For a TEXT file, the buffer size corresponds to a logical record or line. The allotted size is 256 bytes. The Pascal file-buffer variable, f^, of a text file, f, is a particular byte in this 256-byte text buffer. One of the words, (FCB.TPTR), in the file control block (FCB) is the current address of the Pascal TEXT file-buffer variable, i.e., a pointer (f^) to one of the characters in the line.

The organization of the FCB control information (in CAL) is as follows.

```
FCB       STRUC
FCB.MW    DS     4                   Flags
FCB.TPTR  DS     4                   Text pointer, (next char)
FCB.CFSZ  DS     4                   Current file size
FCB.SVC1  DS     SVC1.               SVC_1 parameter block
FBC.SVC2  DS     8                   SVC_2 parameter block
FBC.SVC7  DS     SVC7.               SVC_7 parameter block
          DS     80 - *              Reserved (remainder of bytes)
          ENDS
```

The following flags are defined in FCB.MW

```
MW.RESET  EQU    Y'00000001'         The file may be read.
MW.REWRT  EQU    Y'00000010'         The file may be written.
MW.EOF    EQU    Y'00000100'         At end of file.
MW.EOLN   EQU    Y'00001000'         At end of line (TEXT).
```

Within the file-variable storage the eighty byte FCB precedes either the component-type buffer(for non-text files) or the 256 byte buffer (for TEXT files).

### Formal routines

A "formal routine" is a procedure or function that is declared as a parameter of another procedure or function. It is internally represented as two addresses, which are 4 bytes long and aligned on 4-byte boundaries. The first is a location in code, the beginning of an actual routine which has been directly or indirectly bound to the formal routine. The second is a data location. It is used by that actual routine as its "static link" to the non-global environment. Although these entities are specifiable in an internal routine parameter-list declaration, and actual routine names can be passed as an argument to the formal routine parameters, formal routine parameters are not allowed to be specified in a MODULE parameter-list, and actual routine names cannot be passed to modules.

### 10.6.2 Memory Management Overview (Initial State)

The starting execution address of a Pascal program is at the beginning of the body of the program. At this address is an entry point, whose name is the same as the program label (PROG label); which is taken from the Pascal program-name (truncated to 8 characters, if necessary).

Differing from Pascal R00, Pascal R01 and up compiled object code emits a load-transfer-address equated to the 8-character program label ENTRY (see Figure 10-4) to cause execution to begin directly where the main body is in the Code and Constants area depicted in that figure.

A Pascal program begins by calling P$INIT, the Pascal Initializer in PASRTL, passing to P$INIT the user-specified value of the MEMLIMIT compiler-option and the number of logical units reserved for external Pascal named files (in order to determine which lu may be used for internal Pascal named files, if any). At the return from P$INIT, the initial state of memory has been established. UTOP has been raised, as the result of a "Get Storage" SVC, to allot Workspace for stack variables and the heap. The registers called Stack Limit (SL), Local Base (LB), and Global Base (GB) have been initialized. Depending on the task options FLOAT and DFLOAT, the floating-point registers may be cleared. The start parameters, if any, have been moved to the top of the heap; (whereby a user call on START_PARMS, the Pascal Prefix RTL routine START_PA, can turn them into a valid heap item). Following the call on P$INIT is a call on P$ERR to enable the illegal instruction trap for Pascal's error handler.

Figure 10-4 depicts the state of the task's memory space after returning from P$INIT. In the diagram, there are ten areas.

In order of increasing address, they are:

- the User Dedicated Locations (UDL);
- other absolute code; (optional)
- code and constants;
- impure area; (optional)
- static data base for Pascal support;
- FORTRAN Static Communications Area (SCA); (optional)
- an RTL Scratch Pad;
- space for global variables;
- empty Workspace into which the stack and heap may expand;
- space between UTOP and CTOP, not used by Pascal code; if the user specified a non-default value for MEMLIMIT of less than 100%.

```
                    +-------------------+
                    |                   |<- CTOP
                    |                   |
                    |                   |
SL (R0), HEAP.TOP ->|___________________|<- UTOP after START <--
                    |                   |     ^                ^
                    |  Remaining Task   |     |                |
                    |  Workspace for    |     |                |
                    |   heap & stack    |     |                |
                    |                   | MEMLIMIT %           |
                    |-------------------| of (CTOP - GB)       |
                    |                   |     |                |
                    |    Global         |     |                |
                    |       variables   |     |                |
                    |                   |     |                |
 LB (R2), GB (R1) ->|___________________|____v                |
                    |                   |                      |
                    |       RTL         |                      |
                    |     Scratch       |                      |
                    |       Pad         |                      |
                    |                   |                      |
                    |-------------------|                      |
                    |                   |                      |
                    |     FORTRAN       |                      |
                    |       SCA         |                      |
                    |   ( optional )    |                      |
        (UDL.EXT) ->|___________________|                      |
                    |                   |                      |
                    |    Pascal RTL     |                      |
                    |  Static Data Area |                      |
                    |       SDA         |                      |
                    |                   |                      |
                    |___________________|<- UTOP after LOAD -->|
                    |                   |
                    |    Impure area    |
                    |   ( optional )    |
                    |                   |
                    |-------------------|
                    |                   |
 PROG label ENTRY ->|    Code &         |
                    |       constants   |
                    |                   |
       PROG label ->|___________________|
                    |                   |
                    |  Other absolute   |
                    |  code(optional)   |
                    |                   |
                    |-------------------|<- UBOT + X'100'
                    |                   |
                    |       UDL         |
                    |                   |
                    +-------------------+<- UBOT
```

**Figure 10-4. Initial Memory Map of Pascal Program Task**
(As a User Task in an OS/32 Environment)

The User Dedicated Locations (UDL) are the first X'100' bytes off UBOT, the bottom address of the established user task (under OS/32). The UDL internal structure is that defined by the operating system for user tasks. Under RELIANCE, the UDL and other absolute code depicted, ranges from UBOT to UBOT + X'1D00'.

Other absolute code is usually not present. It could be allocated for some special purpose system code such as the FORTRAN trap intercept mechanism or data used by the Reliance system. Additional space for the absolute code area, can be obtained during task establishment with the OS/32 LINK OPTION command option "ABS = n00", for example, to obtain X'n00' bytes, in place of the default X'100' bytes for the UDL.

The Code and Constants area contains the object code of the Pascal compiled program and its constants (which is generated as a pure code linkable object module processable by OS/32 LINK). If the compiled object program code has not been linked so as to be sharable, the Code and Constants area is not in a separate, shared segment; but rather follows the task's UDL (and/or optional additional ABSolute space) immediately. Within the Code and Constants area, the object code of procedure and function definitions come first, then the object code of the main program, followed by the defined values of the program constants. The constants are of higher addresses than those of the code; and follow the object of the main program. Following the constants area, are the object code of any run time support routines included by the selective editing of the PASRTL.OBJ Pascal Run Time Library file, which resolves the external references to the RTL. The Code and Constants area are in a separate, shared segment if the program has been linked so as to be sharable.

The Impure Area is present only if the task includes modules, written in languages other than Pascal, such as in CAL, which have impure data segments. If the program is not linked so as to be sharable, then the pure segments of these modules will alternate with the impure segments.

The Pascal RTL Static Data Area (SDA) contains variables needed by the Pascal run-time support system. The SDA does not reside in an impure segment and is located above the Impure Area (if any) and below the FORTRAN SCA (if any). It is placed in this area so as to only be accessed by the Pascal RTL. This enables the Pascal RTL to be re-entrant. The contents of the UDL at X'20', UDL.EXT, points to the first word after this area (see Figure 10-5). Neither UDL.EXT nor the contents of this SDA data base should be altered by externally written user routines. Briefly, the Pascal RTL SDA contains nine fullwords as outlined below:

## Pascal Static Data Area

| | |
|---|---|
| RING.HED | Used by Heap Management RTL routines |
| HEAP.TOP | Used by Heap Management RTL routines |
| CMP.FLAG | Zero for user tasks |
| CMP.PASS | Zero for user tasks |
| CMP.SLNO | Zero for user tasks |
| CMP.NAME | Zero for user tasks |
| STOR.BOT | P$TERM uses as bottom of storage space |
| MAX.LU | Maximum logical units |
| MIN.LU | Minimum logical units |

The FORTRAN Static Communications Area (SCA) is present if some module, or the main program, contains a declaration of a routine with the directive FORTRAN (and the routine is invoked). It is not present in a user Pascal system, if no linkage to FORTRAN is directed. This FORTRAN SCA area occupies a variable number of fullwords, dependent on the number of logical units in the user system. A location in the UDL, at X'20', called UDL.EXT, points to the beginning of this area. For the purpose of this area, see the documentation of the FORTRAN USER GUIDE, Publication Number 48-010F00R01. Also see Section 10.8 on the Pascal-FORTRAN Interface.

The RTL Scratch Pad is the area used for local storage by either the Pascal run-time support library routines (PASLIB, PASINIT, PAS.ERR, or PAS.REL groups) or the FORTRAN RTL. This area is reserved as X'600' bytes by the Pascal Initializer routine, P$INIT. Portions of this area are allocated for use in a decreasing stack. Some run time library routines temporarily decrease R1, and address this area with positive displacements off of the new R1. R1 points to the bottom of the area that is in use by the Pascal RTL. Upon exit from the Pascal RTL routines, R1 is reenstated to GB to point to the Global Variables.

Pointers to the Global Variables area, the top of the stack, and the (bottom) limit of the heap are in general registers:

| Register | Use |
|---|---|
| R0 | SL = Stack Limit (end of heap) |
| R1 | GB = Global Base |
| R2 | LB = Local Base |

The addresses in these registers are fullword aligned. Note that when any routine in the Pascal or FORTRAN run-time support is executing, GB is displaced to point to a location in the scratchpad; but when control returns from the run-time support to the main-line code, GB is brought back to the top of the scratchpad and the beginning of the global data area.

### 10.6.3 Memory Management Overview (Running State)

During execution of a Pascal program task, under OS/32, the memory map differs from Figure 10-4 in two respects: there may be local variables for one or several nested/recursive routine activations, and there may be dynamically allocated variables on the heap.

For each activation of a routine that has been entered and has not yet been exited, the local variables (and some compiler-generated variables) are contained in an "activation record" (see Section 10.6.4.). Activation records are added one at a time and deleted in reverse order; so they are kept on a stack. The Local Base in General Register R2 points to the beginning of the most recently created activation record.

Dynamically allocated variables that have been created by NEW (see Section 10.6.5 or 3.5.2), and not yet destroyed by DISPOSE occupy the space between UTOP and the target of the Stack Limit. There may be gaps in this space where variables have been DISPOSEd; the details are explained in Section 10.6.5. The Stack Limit points to the first fullword in the lowest item on the heap.

In Figure 10-5, it is assumed that UTOP has been changed to the value which it was given by P$INIT, based on the user-specified MEMLIMIT compiler-option, and the task establishment memory Workspace option, or the LOAD segment size increment.

If P$INIT cannot obtain the minimally required space to organize memory to run Pascal code, the user is notified with the message:

NOT ENOUGH SPACE TO RUN PASCAL

and user-specified adjustments to task memory allocations must be made. If the first check for the particular program's necessary Global Variables area space fails, the user is notified with the run time STACK OVERFLOW message displaying the line number of the program main body; and appropriate memory allocation adjustments must be made.

If the Pascal program is linked to external modules written in other languages, then these modules may call on the Operating System to get additional storage, above the UTOP established by P$INIT. The requested storage would be obtained in the area between UTOP and CTOP. However, the Pascal program never has access to memory above the initial value of UTOP, established by P$INIT.

```
                   +---------------------+
                   |                     | <- CTOP
                   |                     |
                   |                     |
         HEAP.TOP->|___________________._| <- UTOP
                   |______             | |
         RING.HED->| gaps |            | |
                   |______|            | |
                   |                   | |
                   |       Heap        | |
                   |                   | |
          SL (R0)->|___________________v_|
                   |                     |
                   |       unused        |
                   |      Workspace      |
                   |                     |
                   |-------------------^-|
                   |                   | |
                   |  Activation       | | See Fig. 10-6 for
                   |     record n      | |     more details
 Current LB  (R2)->|___________________|_|
           n       |                   | |
                   |  Activation       | |
                   |     record n-1    | |
                   |___________________|_|
                                       |
                            . . .      |
                                       |
                   |-------------------|-|
                   |  Activation       | |
                   |     record 1      | |
                   |. . . . . . . . . .|.|
Stack begins LB  ->|first old LBo= GB  | |
  ^            1   |-------------------^-|
  |                |                     |
  |                |     Global          |
  |                |                     |
  |                |        Variables    |
  |                |                     |
 LBo ---> GB (R1)->|---------------------|
          |        |   RTL Scratch Pad   |
          v        |                     |
R1 During RTL Use->|                     |

                            . . .

                   |---------------------|
                   |                     |
                   |        UDL          |
                   |                     |
                   +---------------------+<- UBOT
```

Figure 10-5. Memory Map During Execution of a Pascal Program Task
(As a User Task in an OS/32 Environment)

### 10.6.4 Stack Memory Management

Information on how the stack is managed internally is provided for those who wish to write an external CAL routine which will use the Pascal interface to external routines, and stack; or to understand the additional assembly-listing obtainable by compiling with the Pascal compiler option ASSEMBLY. The compiler-generated code of routine definitions and routine-invocations are reflected in that listing in an assembler level format.

An activation record is formed on the stack for each routine invocation being executed at the time of its execution. A procedure is activated when a procedure-call statement is executed. A function is activated when, during an expression evaluation, a function reference occurs. A user-written routine in CAL and externally linked to a Pascal compilation-unit(program or module) must observe the environment preservation techniques required to maintain the stack, avoid and/or warn of collision with the heap (if the called-routine uses/increases the stack), obtain any arguments passed to the routine's parameters, and adhere to the Pascal linkage conventions upon return. This desciption of the stack management is in effect for routine invocations in Pascal R01 and up. Also refer to Section 10.7 on Pascal Linkage Conventions for EXTERN routines.

An activation record of a routine contains, in order of increasing address:

- linkage data;
- space for actual arguments which are being passed to parameters, (if any), within available registers;
- actual arguments that have been passed to parameters, that cannot fit in registers; if any;
- local variables;
- compiler-generated "temporaries".

Figure 10-6 depicts the structure of an activation record on the stack for an internal procedure.

The linkage data of the new activation record is partially set up by the calling code. The new activation record is on top of the current stack of compiler-generated variables in the caller's activation record. The value of LB is stored in the "Old LB" field. If the routine being called requires a static link, (so as to access its own non-global environment), then the calling code finds its value and places that in the "Static Link" field. The calling code increases LB by the current height of its activation record, so that LB now points to the beginning of the new activation record. The calling code then passes control to the called routine using the instruction "BAL 15,Plabel"; where

Plabel is a compiler-generated label, such as P101, P102, P103, etc., both in the calling BAL instruction and on the beginning instruction entered in the called routine.

Here is a typical example of the emitted code, for a procedure invocation. Suppose that the routine definition being called is on the same lexical level as the routine whose body contains the calling code; then the static link of the latter will be the static link of the former. Let "sh" denote the current height of the activation record of the calling code, (see Figures 10-6 and 10-7) and Plabel is the label at the beginning of the routine being called. Then the emitted code looks like this:

```
          .
          .     Optionally pass arguments to parameters, if any.
          .
          ST    R2,sh(R2)           Save old LB in next activation
          L     R15,8(R2)           Get appropriate static link
          ST    R15,sh+8(R2)        and pass a copy.
   AI/AIS/AHI   R2,sh               Adjust LB to be New LB
          BAL   R15,Plabel          Pass control to routine
```

Compiler-generated calling sequences will vary from the above, in that the instructions to find the appropriate static link and save it (or zero) in "sh+8(R2)" may differ. Depending on how deeply nested the internal routine which is being called, the instructions which set static link vary, such as:

```
Calling a local routine from main body:     XR    R15,R15
                                            ST    R15,sh+8(R2)

Calling from an inner routine to an outer which is:

an outermost main program routine:          XR    R15,R15
                                            ST    R15,sh+8(R2)

its own local (non-nested) routine:         ST    R2,sh+8(R2)

an outer but nested routine (same level):   L     R15,8(R2)
                                            ST    R15,sh+8(R2)

an outer but nested routine (many levels):  L     R15,8(R2)
                                            L     R15,8(R15)
                                            L     R15,8(R15)
                                            ...
                                            L     R15,8(R15)
                                            ST    R15,sh+8(R2)

In calling an external routine;             XR    R15,R15
  EXTERN or FORTRAN, (from any level):      ST    R15,sh+8(R2)
```

In the routine being called, let the current line number in the Pascal source program be "Lineno", and let the size of the local activation's storage requirements be "lsh", as the local stack height of the new current activation record. Then the code at the beginning of the routine looks like this:

```
Lerrex   ERR     8,Lineno                 Error message exit
Plabel   ST      R15,4(R2)                Save return address
     CLI/CLHI    R0,lsh(R2)               Test for collision
         BCS     Lerrex
          .
          .      Optionally receive arguments, if any parameters.
          .
```

The called routine stores the contents of register 15 in the "Return address" field. It checks the R2 value of LB plus "lsh" against SL (in R0) to see if there is room for its local storage requirements (including any space for data it might store in the activation record of any routines it might call). Assuming there is enough room, it proceeds to receive arguments as described in Section 10.7.

```
                      |                     |
                      |---------------------|---^
                      |                     |   |
                      |     Temporaries     |   |
                      |                     |   |
                      |---------------------|   |
                      |                     |   |
                      |     Local           |   |
                      |       Variables     |   |
                      |                     |   |
                      |---------------------|  lsh (local stack
                      |      Parameters     |   |    height)
                      |       and/or        |   |
             + 12 ->  |  Parameters space   |   |
                      |---------------------|   |
              + 8 ->  |   Static Link or 0  |   |
                      |---------------------|   |
              + 4 ->  |    Return Address   |   |
                      |---------------------|   |
 (Current LB) + 0 ->  |     Old LB          |   |
                      |---------------------|---v
```

**Figure 10-6. Structure of an Activation Record (Internal Routine)**

The linkage data are three fullwords:

| Displacement <u>off LB Stack</u> | Meaning <u>of Contents</u> | Caller's <u>Responsibility</u> | Called's <u>Responsibility</u> |
|---|---|---|---|
| 0 | Old LB | Adjust & Store | Put in R2 on exit |
| 4 | For Return address | Send thru R15 | Save it to return |
| 8 | Static Link or 0 | Store it | Available for use |

The Old LB is the value of the Local Base for the calling routine. The calling sequence of an external routine, similar to the Pascal internal routine linkages, will store its LB as the Old LB on the stack in the first fullword of the activation record of the called routine (see Section 10.7).

The Return Address is the location in code where control should return after the current routine is finished. It is the responsibility of a called routine to save the return address given to it, via the "BAL R15,Plabel" instruction which has invoked it. Compiler-generated linkages stores R15 immediately upon entry into a called-routine. Just prior to the calling "BAL R15,Plabel", R2 has been adjusted to point to the beginning of the routine's activation record. Therefore, to maintain the stack, and have R15 available for other uses within the called routine, R15 may be stored at four off the new LB in R2. Following this, the called routine may reserve its local storage requirements on the stack by first checking if its needed allocations will not collide with the heap.

The Static Link is the address of an activation of the routine in which the present routine is nested, otherwise it zero. The current activation uses the Static Link to reach its non-global environment. If the routine is directly contained in a main program, then it does not really need a static link (so none is stored). The space is allocated, however, because this makes the structure of memory slightly more uniform. In this case, the fullword reserved for use as a Static Link is zero. Also, it simplifies the interface of formal routines.

Parameters

Space for each parameter is reserved in the activation record of the routine on the stack in the order in which the parameters are listed in the parameter-list of a routine-declaration. Space for a parameter is represented in the activation record by either the space required for the value or the address of the argument that has been passed. Whether it is a value or an address depends on the type of the parameter and whether it is a value or VAR parameter. Whether the value or address is either actually in the activation record or being passed in a register depends on the availability of the registers. If there are more parameters, than can be passed in registers, then they are passed on the stack in the space for parameters beyond the space reserved for those being passed in registers; but in their respective order as sequentially listed in a parameter-list.

The type is small if it naturally fits in a register. The small types are the standard types BYTE, SHORTINTEGER, INTEGER, SHORTREAL, REAL, CHAR, and BOOLEAN, and enumeration types, subrange types, and pointer types. When passed to value parameters, their values may be in registers (of a suitable kind); when passed to variable parameters, their addresses may be in the general registers.

A parameter is represented by value if it is passed to a value parameter and is of a small type or a SET type. If it is of another large type, or is a VAR parameter, then it is represented by address. A formal routine is passed as two addresses, as described in Section 10.6.1.

Among the other types, SET types are treated differently from ARRAYs, RECORDs, and FILEs. SETs are passed to value parameters on the stack and never in registers. ARRAYs, RECORDs, FILEs and SETs are passed to variable parameters by an address; and ARRAYs and RECORDs are passed to value parameters by an address of a copy. FILEs are passable to VAR parameters by address.

Details of how arguments are passed to parameters are given in Section 10.7.

### Local variables

The local variables of the activation correspond to the variables declared in the definition of the routine. They are allocated above the parameters, in the order of their declarations. The size and alignment of each local variable depends on its type, as described in Section 10.6.1.

### Compiler-generated temporaries

The compiler will store certain intermediate results in memory, above the local variables in a local activation record for a routine definition. These variables are created as needed, and are destroyed in reverse order to their creation; so this area is managed as a stack of elements of diverse sizes. The compiler always knows the height of the stack relative to the local base. This area is only generated for internal routines and is programmable in externally written CAL routines as needed for local storage on the stack.

### Function values

Most addresses off the local base have non-negative displacements. An exception is made for function values (within the called function). In Pascal, function values are always of the small types: CHAR, BYTE, BOOLEAN, SHORTINTEGER, INTEGER, SHORTREAL, REAL, and enumeration types, subrange types, and pointer types. To receive the function value, the caller (such as done by the compiler-generated code which invokes an external routine) allocates an 8-byte temporary area before the function is called. This space is reserved so that the caller may fetch the function value in that temporary area, upon return. Refer to Figure 10-8. Within the called function, the function value is located at -n relative to the local base of the current activation of the function. Here n is the length in bytes of the type of the function value. The result of this practice is that when the function returns to the caller, the function value is available on the top of the caller's activation record on the stack.

Within the called routine, when the routine is an external function written in CAL, for example, the user must store the function value result in this 8-byte temporary area by addressing the stack with a negative displacement off of his current LB in R2.

For example, depending on the function-value data type, prior to returning:

| Function-value Type | Passing Back the Function-value<br>(One of the following:) |
|---|---|
| CHAR, BYTE<br>or subrange thereof | STB Rx, -1(R2) |
| SHORTINTEGER, BOOLEAN,<br>enumeration-type<br>or subrange thereof | STH Rx, -2(R2) |
| INTEGER, or subrange thereof<br>or Pointer-type | ST Rx, -4(R2) |
| SHORTREAL | STE Fx, -4(R2) |
| REAL | STD Dx, -8(R2) |

assuming the function-value was developed in one of the respective registers Rx, Fx, or Dx (as is done in compiler-generated code within function-definitions to set the value of the function name).

After passing the function-value with one of the above mechanisms, into the space alloted for it by the calling sequence; the exiting code sequence is similar to that used for a procedure, e.g.,

```
L      R15,4(R2)        Fetch Return Address
L      R2,0(R2)         Reinstate Caller's LB
BR     R15              Return to Caller
```

Refer to Figure 10-7, for a depiction of the activation record on the stack for a function call.

```
                         |                     |
                         |---------------------|---^
                         |                     |   |
                         |     Temporaries     |   |
                         |                     |   |
                         |---------------------|   |
                         |                     |   |
                         |     Local           |   |
                         |       Variables     |   |
                         |                     |   |
                         |---------------------|  lsh (local stack
                         |     Parameters      |   |    height)
                         |       and/or        |   |
           + 12 ->       |  Parameters Space   |   |
                         |---------------------|   |
           + 8  ->       |  Static Link or 0   |   |
                         |---------------------|   |
           + 4  ->       |   Return Address    |   |
                         |---------------------|   |
 (Current LB + 0) ->     |     Old LB          |   v
                         |---------------------|----
                         |   Function value    |   ^
                         |                     |   |
 (Current LB - 8) ->     |- - - - - - - - - - -|   |
                         |                     |  sh (stack height)
                         |     Caller's        |   |
                         |  Activation Record  |   |
                         |                     |   |
                         |---------------------|---v
```

Figure 10-7. Structure of an Activation Record of a Function-call

Upon return to the caller, the receiving sequence fetches the function value from this 8-byte area, as though the value were right justified in it (note the depictions below and sequences).

| Function-value Type | Receiving Code<br>After the BAL R15,routine-name<br>(May be one of the following: ) | |
|---|---|---|
| CHAR, BYTE<br>or subrange thereof | LB | Rx,sh-1(R2) |
| SHORTINTEGER, BOOLEAN<br>enumeration type<br>or subrange thereof | LH | Rx,sh-2(R2) |
| INTEGER, or Pointer-type | L | Rx,sh-4(R2) |
| SHORTREAL | LE | Fx,sh-4(R2) |
| REAL | LD | Dx,sh-8(R2) |

The 8-byte area reserved in the caller's activation record, depending on the function-value type, is given in Figure 10-8.

```
 (Current LB + 0) ->   |       Old LB        |
                       |---------------------|
                       |                     |
                       |   Function value    |
                       |                     |
 (Current LB - 8) ->   |- - - - - - - - - - -|

CHAR, BYTE             |---------------------|
or subrange thereof    |               |char/|
                       |               | byte|
                       |---------------------|
                       |                     |
                       |                     |
 (Current LB - 8) ->   |---------------------|


SHORTINTEGER,BOOLEAN   |---------------------|
enumeration-type,      |          |<--value->|
or subrange thereof    |          |          |
                       |---------------------|
                       |                     |
                       |                     |
 (Current LB - 8) ->   |---------------------|


INTEGER,               |---------------------|
or subrange thereof,   |<------value-------->|
or Pointer-type        |                     |
                       |---------------------|
                       |                     |
                       |                     |
 (Current LB - 8) ->   |---------------------|



SHORTREAL              |---------------------|
                       |<.......value.......>|
                       |s|exp|<--fraction--->|
                       |---------------------|
                       |                     |
                       |                     |
 (Current LB - 8) ->   |---------------------|



REAL                   |---------------------|
                       |<-fractional digits->|
                       |                     |
                       |-|---|---------------|
                       | |   |               |
                       |s|exp|<-fractional-->|
 (Current LB - 8) ->   |---------------------|
```

Figure 10-8. Function-value Passed on Stack

### 10.6.5 Heap Memory Management

The internal structure of the heap is designed to allow the run time support of Pascal to implement NEW and DISPOSE efficiently, and re-use space where a variable has been DISPOSEd, while safeguarding the data from pointers with invalid values. To achieve these goals, each dynamically allocated variable has attached to it some surrounding words on the heap. These surrounding words are invisible to the Pascal program code but used by the Pascal run-time support. Some system variables which are used to manage the heap in the Pascal system environment, and which must not be destroyed by user-written CAL routines, are:

- the Stack Limit, which is maintained in R0;
- the Head-of-Ring, RING.HED in Figure 10-5, points to a link on the ring of free areas (gaps of reusable dead space) within the heap;
- the Top-of-Heap, HEAP.TOP in Figure 10-5, which points to word above the topmost word in the heap.

HEAP.TOP and RING.HED are maintained in the Pascal Static Data Area, see Figure 10-4.

For each dynamic variable created by NEW there is an item on the heap. Every item on the heap is fullword aligned and contains an integral number of fullwords. The item contains three fullwords of overhead. The rest of the item is the dynamically created variable as seen by the Pascal program. The heap management procedures do not do anything with the variable itself; they only use the surrounding overhead words. The Pascal program code does not have access to these internal words.

The heap contains valid data items and a "free ring" consisting of intervals of memory that had been allocated and then DISPOSEd of.

Each valid item on the heap begins with a leading word and a length word, and ends with a trailing word.

The leading and trailing words contain their own locations. The value of the leading word is used when accessing the target of a pointer, to check the validity of the pointer. The trailing word is used when an item is being DISPOSEd of, to check whether the adjacent data are on the free chain or not. The length word contains the total length of the item in bytes, counting the three words of overhead.

Each valid item contains at least four fullwords: the leading, length, and trailing words and at least one word containing data.

For example, see Figure 10-9.

```
             |--------------------|
location b   |    location b      |   trailing word
             |--------------------|
             \                    \
             /                    /
             \     Data Area      \   dynamic variable
             /                    /
             \                    \
             |--------------------|
             |    item-length     |   length word
             |--------------------|
location a   |    location a      |   leading word
             |--------------------|
```

Figure 10-9. A Dynamic Variable Item on the Heap (location a < b)

As the procedure NEW is called to create dynamic variables, they are allocated on the heap as valid items. The user writes Pascal code to define their data contents and their linkage to each other by setting pointer fields within the Data area, (see Chapter 5 on pointer-types). Refer to Figure 10-10.

```
                      |-------------------|
                      |   trailing word   |
                      |-------------------|
                      \                   \
                      /                   /
                      \       Data        \
                 ...../...o               /
                 .    \                   \
                 .    |-------------------|
                 .    |    item-length    |
                 .    |-------------------|
                 .    |   leading word    |
                 .    |-------------------|
|                .  |                       |                   |
|----------------.--|                       |-------------------|
|  trailing word .  |                       |   trailing word   |
|----------------.--|                       |-------------------|
\                .  \                       /                   /
/                .  /
\      Data      .  \                       /      Data         /
/                o../.........................>o
\                   \                       /                   /
|-------------------|                       |-------------------|
|   item-length     |                       |   item-length     |
|-------------------|                       |-------------------|
|   leading word    |                       |   leading word    |
|-------------------|                       |-------------------|
```

Figure 10-10. Sample Linked List on the Heap

When an item is DISPOSEd of, it is added to the ring of free areas. If it is adjacent to an area that is already free, then it is appended to that area; otherwise it becomes a new link in the ring.

Each free area contains three significant fullwords. The first word is a pointer to the beginning of the next free area on the ring. The second word is the length, in bytes, of the free area. The last fullword of the free area is a pointer to the head of the previous free area on the ring. A consequence of these rules is that when the run time procedures find either the first or the last word of a free area, they can find the other end and also the adjacent free areas on the ring. The actual value of the pointers stored in the first and last fullwords is biased -1 (a one byte negative differential) to differentiate the disposed areas from the occupied areas and to safeguard them.

Note that there is no requirement for the ring of free areas to be ordered by address. See Figure 10-11.

For example:

```
                        |-------------------|
|<----------------------|---last^ free-area |
|                       |-------------------|
|                       \                   \
|                       /                   /
|                       \   Reusable area   \
|                       /                   /
|                       \                   \
|                       |-------------------|
|                       |    area-length    |
|                       |-------------------|
|                     .>| next^ free-area o |<--^
|                     . |-----------------.-|   |
|                     . |                 . |   |
|                     .                   .     |
| |-----------------| .                   .   |-|---------------|
| | last^ free area _____________         .   | last^ free area |
| |-----------------| .          |        .   |-----------------|
| \                 \ .          |        .   /                 /
| /                 / .          |        .   \                 \
| \ Reusable area   \ .          |        .   / Reusable area   /
| /                 / .          |        .   \                 \
| \                 \ .          |        .   /                 /
| |-----------------| .          |        .   |-----------------|
| |   area-length   | .          |        .   |   area-length   |
| |-----------------| .          v________v__\|-----------------|
| |              o.....                      /|                 |
->| next^ free-area |<........................|o next^ free-area|
  |-----------------|                         |-----------------|
  |                 |                         |                 |
```

Figure 10-11. Ring of DISPOSEd Free Areas on the Heap.

When the procedure NEW(p) is invoked and executed, the ring of free areas is searched for one which is big enough to add a new item for p^. Heap management will use first-fit. If no free area is found, then space from the unused Workspace is obtained. If there is not enough space in the unused Workspace, then a run time error condition occurs. In this case, the run time error message HEAP OVERFLOW occurs. If there is space, NEW sets the values of the leading and trailing words of the newly created item, and the length word of the item's data size.

If the area remaining after the new item is at least 3 words long, then NEW sets it up as a new free area, spliced into the ring. If there is less than 3 words remaining, then the length of the new item is increased to absorb this remainder, and the two adjacent links are spliced together.

NEW sets the HEAD-of-RING to the successor of the item that was used. This has the effect that successive searches are evenly distributed around the ring.

When DISPOSE(p) is called, the words immediately before and after the item indicated by p^, to be destroyed, are inspected. The values of these words determine whether the present item is preceded or followed by a valid item or a free area.

If the present item is bordered on both sides by valid items, then this item is made into a free area and put in the ring of free areas.

If the present item is adjacent to the unused Workspace, then the unused Workspace is expanded into, to include this item. If, on the other side, there is another free area, then it is also added to the unused Workspace and removed from the free-area ring.

If the item to be DISPOSEd of, is adjacent to a free area on one side and valid item on the other, then it is annexed to the free area.

If the item to be DISPOSEd of, is between two free areas, the following free area is removed from the ring and, together with the present item, is annexed to the previous free area.

Note that the leading word of the original item should always be changed. If the item goes onto the ring, then the leading word is changed from containing its own position to a link in the ring. If the item is merged with a free area preceding it in memory, then the old leading word is wiped out.

The value of the function STACKSPACE is the current size of the unused Workspace.

The execution of a routine-invocation of the procedure MARK(m) creates a record item, big enough for a single integer, in the frontier of the heap; and returns that address in m.

The execution of a routine-invocation of the procedure RELEASE(m) where m was previously established in a MARK(m) call, effectively destroys the item on the heap for m, and all items which were created on the heap with lower addresses, by moving STACK LIMIT back to the end of the record item of m. RELEASE adjusts the chain of free areas, by removing any that that are now below the Stack Limit.

Note that a program which uses both MARK and RELEASE with DISPOSE may have items which were created after a use of MARK but have higher addresses, because they fit in gaps left by the action of DISPOSE. A RELEASE corresponding to that MARK will not destroy these items.

## 10.7 Pascal LINKAGE CONVENTIONS to Internal and EXTERN Routines

Programs, modules, procedures, and functions are the pieces from which Pascal software is constructed. How they are connected to one another, and what must happen as control enters or leaves one of them, are the concerns of this section, particularly for external Pascal MODULEs or CAL written routines declared as external from the main program with the EXTERN directive.

The Pascal Linkage Conventions described here apply to Pascal R01 and up compiled code; and differ from Pascal R00 (see Appendix P on the functional differences between Pascal R00 and R01).

These Pascal Linkage Conventions describing the passing of arguments to parameters, calling sequences, receiving sequences, and exits used within the routines apply to the following kinds of routines:

- internal Pascal PROCEDUREs and FUNCTIONs
- routines declared as external with the EXTERN directive:
    * externally compiled Pascal MODULEs
    * external CAL routines, using Pascal linkage conventions
    * external SVC Support routines
- Prefix routines, declared with procedure/function headings prior to the PROGRAM or MODULE header.

An operative description of stack memory management for internal Pascal PROCEDUREs and FUNCTIONs, and their calling/exit sequences, is detailed in Section 10.6.4. External FUNCTIONs written in CAL may pass back the function-value as also detailed in that section and not repeated here.

The SVC support routines are detailed in Section 10.2.5 and their use in Section 10.4. The Prefix support routines are detailed in Section 10.2.4 and their use in Section 10.3.

A call to an external MODULE written in Pascal, or to an external routine written in CAL (such as the SVC support routines), or user-written routines in CAL, when they are declared with the EXTERN directive, or a Prefix routine, contains an external reference in the object code, Otherwise, there is no difference from a call to an internal routine compiled in-line together with the calling code (a Pascal R01 enhancement). The code of the external module looks just like that of an ordinary internal routine, except that it has an entry label. The error handling mechanisms use the line number of the module source program, not related to the main program's line numbers. Also, an external routine written in CAL, to utilize Pascal's error handling mechanisms may define a similar illegal instruction described below as the ERR instruction.

The illegal instruction ERR is defined to be a user-specified CAL mnemonic, by equating the symbol ERR to X'8804', i.e., ERR EQU X'8804'. This defines ERR to be an RI1 formatted instruction, with an opcode of X'88'; a first-operand R1 field that may be between 0 and 16, but must be user restricted to the value 8 in order to obtain the STACK OVERFLOW Pascal error message; and a second-operand immediate value I2 field. This second-operand field may be any assembler expression yielding a halfword value whose value will be converted to its ASCII decimal representation in Pascal's run-time error message mechanism (in place of the usual Pascal source line number). Note that only the values 0 (for BREAKPOINT) and 1 through 9 (for Pascal runtime error messages) are handled by Pascal on trapping the ERR instruction; such that other first-operand values may produce untoward results.

The Pascal linkage conventions to routines declared in the Pascal code with the directive EXTERN presents to the called routine, the following state of registers:

| | |
|---|---|
| R0 | Contains the stack limit (to which stack may expand) |
| R1 | Contains Global Base(Not alterable without restoration) |
| R2 | Contains the Local Base (stack activation-record start) |
| R3-R12 | Contains parameters (if any) as per Section 10.7.1 |
| R13-R14 | Used as temporaries, so available for use |
| R15 | Contains address in Pascal code to return to |
| F14-F2 | Contains shortreal value parameters, if any |
| F0 | Available for use |
| D14-D2 | Contains real value parameters, if any |
| D0 | Available for use |

The state of the stack is as follows. The linkage data of the new activation record is partially set up by the calling code. The new activation record is on top of the current stack of compiler-generated variables in the caller's activation record. The calling code sequence stores its value of LB in the "Old LB" field, of the called routine's activation record. Zero is placed in the "Static Link" field. The calling code increases LB by the current height of its activation record, so that LB now points to the beginning of the new activation record, useable by the called routine. If a greater number or certain kind of parameter is being passed to the routine than can fit in available registers (or of a suitable kind of register) then they will have been passed on the stack as per Section 10.7.1. The calling code then passes control to the called routine using the instruction BAL 15,routine; i.e., General Register R15 contains the address to return to upon exit. The "Old LB" must also be restored, if R2 is used, prior exit.

Here is a typical example of the emitted code in the calling program at the place of invocation of an external module. Let "sh" denote the current height of the activation record of the calling code, and "routinex" is the ENTRY at the beginning of the routine being called.
Then the emitted calling sequence code for an EXTERN external routine, or MODULE, looks like this:

```
        .
        .      Load Registers with the arguments, if any in call
        .


        .
        .      Additionally pass arguments on stack, as necessary
        .
        ST     R2,sh(R2)              Save old LB
        XR     R15,R15                Zero
        ST     R15,sh+8(R2)           Store Zero in Static Link
        AHI    R2,sh                  Put new LB in R2
        BAL    R15,routinex           Pass control
```

In the module being called, let the current line number of the main statement forming the body of the MODULE in the Pascal source be "Lineno", and let the size of the local activation record be "lsh"; i.e., the amount of storage required for use by the module on the stack.

The value of "lsh" is determined so as to cover the storage requirements of all local data used by the routine to which it applies. This includes not only the activation record of the currently called module but also any data space needed for data that the called module is going to store in the activation record of another called routine (such as is done when more parameters must be passed to a routine than cannot fit in the number of available and suitable kind of registers).

Then the outline of the code of a Pascal MODULE looks like this (in assembler-level format) :

```
routinex PROG
         .
         .    Optional code for local/nested routines in module
         .
Lerrex   ERR   8,Lineno              Error message escape
P1       ST    R15,4(R2)             Save return address
         CLHI  R0,1sh(R2)            Test for collision with heap
         BCS   Lerrex
         .
         .     Store parameters passed in Registers onto stack
         .

         .
         .     Perform the operations of the MODULE body
         .

         L     R15,4(R2)             Restore Return Address
         L     R2,0(R2)              Restore old LB
         BR    R15                   Go back to calling code

         ENTRY routinex
routinex EQU   P1
         END
```

The called module stores the contents of general register R15 in the "Return address" field in the linkage data area of its activation record. It checks the R2 value of LB plus "lsh" against SL (in R0) to see if there is room for its parameters, local variables, and local temporary storage needs. This "lsh" includes not only the module's activation record but space for any data which the module knows it must store in the activation record of any routine which it is calling. Assuming there is enough room, the module proceeds to receive arguments passed to its parameters as described in Section 10.7.1

If there is not enough room on the stack to continue operating without colliding with the heap, the BCS Lerrex branch is taken. to an illegal instruction ERR 8,lineno. Pascal's run-time error handling mechanism, using the R1 field of the ERR instruction whose value is 8, will generate a STACK OVERFLOW run-time error message.

When a module is finished and ready to return control to the calling code, it must find the location to return to, and restore the environment of the calling code. It does this by the last three executable instructions shown in the example outline of a module above.

The compiler-generated object code to any routine declared external from the main program or another module with the directive EXTERN, allows the user to write external routines in CAL, if they adhere to the Pascal Linkage Conventions described above by returning to the address in R15, handle R2 appropriately in regards to the stack and further protect the environment of the Pascal calling code by not destroying R0 or R1; and processing the parameters passed in R3 to R12 and through R13; F14 downto F2 and through F0; D14 downto D2 and through D0; if any.

An example of a CAL routine, declared with EXTERN, will be presented in Section 10.7.1 following a description of how arguments are passed to parameters.

### 10.7.1 Passing arguments to parameters

In order to understand how the called routine receives its parameters, we must consider these things:

1. all of the arguments listed in a routine-invocation's argument-list are passed in ordered sequence in a one-to-one correspondence to all of the parameters listed in the parameter-list of the routine-declaration;

2. how the arguments are individually prepared to be passed;

3. what is actually passed from the calling code to the called routine;

4. where the argument is passed.

How, what, and where each individual argument is to be passed depends on the data-type of the parameter, and on whether the receiving parameter is a VAR parameter or a value parameter.

For the purposes of this analysis, the type is small if it naturally fits in a register. The small types are the standard types BYTE, SHORTINTEGER, INTEGER, SHORTREAL, REAL, CHAR, and BOOLEAN, and enumeration types, subrange types, and pointer types.

Preparation. For a value parameter, the argument is an expression; it must be evaluated by the calling code. For a VAR parameter, the argument is a selector; its address must be found by the calling code. For a formal routine, the argument is represented by two addresses; this is more fully explained in Section 10.6.1.

What is passed. For a VAR parameter of any type, the address of the argument is passed. For a value parameter of a small type or a SET type, the value of the argument is passed. For a value parameter of a structured type other than a SET, a copy of the argument is made in the activation record of the calling code, and the address of the copy is passed. For a formal routine, both addresses are passed.

Where it is passed. Note that except for value parameters of SET types, everything that is passed to the called routine can fit naturally in a register. An argument of a SET type is always copied by the calling code into the space that it will occupy in the new activation record. Other values or addresses are passed in registers, or, when the registers of the proper kind are all taken, in the locations that they will occupy in the new activation record.

The registers that are used to pass a routine's invocation arguments to a routine's parameters are as follows:

- for REAL values, (or compatibly typed values), being passed to REAL value parameters, the double-precision floating-point registers are used: from D14 down to D2 (for the first seven REAL value parameters); and D0 is used to pass the eighth and any subsequent REAL value parameters onto the stack in their alloted space. Note that REAL variable parameters are passed by address in either the general registers or on the stack, if the general registers R3 through R12 are occupied by previously listed arguments.

- for SHORTREAL values, (or compatibly typed values), being passed to SHORTREAL value parameters, the single-precision floating-point registers are used: from F14 down to F2 (for the first seven SHORTREAL value parameters); and F0 is used to pass the eighth and any subsequent SHORTREAL value parameters onto the stack in their allotted space. Note that SHORTREAL variable parameters are passed by address in either the general registers or on the stack, if the general registers R3 through R12 are occupied by previously listed arguments.

- for all other small values, and all addresses, the general purpose registers are used: from R3 to R12 (for the first ten such parameters); and R13 is used to pass the eleventh and any subsequent such parameters onto the stack in their allotted space. That is, value parameters of the following small types, are passed by value (the value of the actual argument expression) in an available general register (right-justified within the fullword register for values less than a fullword):

  BYTE
  SHORTINTEGER
  INTEGER
  CHAR
  BOOLEAN
  user-defined enumeration types
  subrange-type
  pointer-type

  The values of small types requiring less than a fullword may not necessarily be right-justified in a fullword when passed on the stack, as depending on its predecessor and successor's alignment requirements, it may not require a fullword on the stack (e.g., BYTE, CHAR, BOOLEAN, SHORTINTEGER, user-defined enumerations, or subranges thereof). Since arguments passed to structured-types for value parameters of the ARRAY and RECORD types, are passed by an address of a copy the actual argument, that address is passed in a general register or in a fullword on the stack.

All variable parameters, no matter what the parameter type: all small types or structured-types (including ARRAYs, RECORDs, FILEs, and SETs), are passed by address (the address of the actual argument variable-selector) either in a general register or in a fullword on the stack.

The following example Pascal program demonstrates some of the argument to parameter passing for an EXTERN routine; using a routine requiring nine shortreal value parameters, nine real value parameters, twelve integer value parameters, and three variable parameters in its call.

```
PROGRAM SHOWCAL(OUTPUT);

VAR GSR:SHORTREAL;GRL:REAL;GIT:INTEGER;

PROCEDURE ARGPASS (SR1,SR2,SR3,SR4,SR5,SR6,SR7,SR8,SR9:SHORTREAL;
                RL1,RL2,RL3,RL4,RL5,RL6,RL7,RL8,RL9:REAL;
      IT1,IT2,IT3,IT4,IT5,IT6,IT7,IT8,IT9,IT10,IT11,IT12:INTEGER;
          VAR SR:SHORTREAL;VAR RL:REAL;VAR IT:INTEGER);
          EXTERN;
BEGIN
  ARGPASS(0.1,0.2,0.3,0.4,0.5,0.6,0.7,0.8,0.9,
         1.0,2.0,3.0,4.0,5.0,6.0,7.0,8.0,9.0,
         1,2,3,4,5,6,7,8,9,10,11,12,GSR,GRL,GIT);

  WRITELN('SHORTREAL SUM = ',GSR);
  WRITELN('SUM OF REALS  = ',GRL);
  WRITELN('INTEGER SUM   = ',GIT);

END.
```

An external Pascal module such as the one below would be separately compilable and linkable to the object of the main program above.

```
MODULE ARGPASS (SR1,SR2,SR3,SR4,SR5,SR6,SR7,SR8,SR9:SHORTREAL;
                RL1,RL2,RL3,RL4,RL5,RL6,RL7,RL8,RL9:REAL;
      IT1,IT2,IT3,IT4,IT5,IT6,IT7,IT8,IT9,IT10,IT11,IT12:INTEGER;
VAR SR:SHORTREAL;VAR RL:REAL;VAR IT:INTEGER);
BEGIN
  SR := SR1+SR2+SR3+SR4+SR5+SR6+SR7+SR8+SR9;
  RL := RL1+RL2+RL3+RL4+RL5+RL6+RL7+RL8+RL9;
  IT := IT1+IT2+IT3+IT4+IT5+IT6+IT7+IT8+IT9+IT10+IT11+IT12;
END.
```

Executing an established Pascal task containing both of the above, produces on OUTPUT:

```
SHORTREAL SUM =  4.5000000E+00
SUM OF REALS  =  4.5000000000000011E+01
INTEGER SUM   =          78
```

The above example external routine ARGPASS can be written in CAL as follows: (Note the use of the CAL STRUC to outline the stack)

```
ARGPASS PROG
        PURE
        ENTRY ARGPASS
STACK   STRUC
* Linkage data area on stack
OLDLB   DSF  1
RETAD   DSF  1
SLINK   DSF  1
* Parameters space on stack for routine ARGPASS
SR1     DSF 1
SR2     DSF 1
SR3     DSF 1
SR4     DSF 1
SR5     DSF 1
SR6     DSF 1
SR7     DSF 1
SR8     DSF 1
SR9     DSF 1
RL1     DSF 2
RL2     DSF 2
RL3     DSF 2
RL4     DSF 2
RL5     DSF 2
RL6     DSF 2
RL7     DSF 2
RL8     DSF 2
RL9     DSF 2
IT1     DSF 1
IT2     DSF 1
IT3     DSF 1
IT4     DSF 1
IT5     DSF 1
IT6     DSF 1
IT7     DSF 1
IT8     DSF 1
IT9     DSF 1
IT10    DSF 1
IT11    DSF 1
IT12    DSF 1
SR      DSF 1
RL      DSF 1
IT      DSF 1
TEMPS   DS  400             Reserve additional space, as desired
        ENDS
ERR     EQU    X'8804'      Illegal instruction definition
LERREX  ERR    8,9999       STACK OVERFLOW Error message trigger
ARGPASS ST     15,RETAD(2)  Save return address
        CLI    0,STACK(2)   Room enough?
        BCS    LERREX

* Store register-passed parameters on stack or use as is.
* For example, prior to using the general registers, save the
* parameters they hold first:
```

```
        ST    3,IT1(2)       R3 contains 1st non-real parameter
        ST    4,IT2(2)
        ST    5,IT3(2)
        ST    6,IT4(2)
        ST    7,IT5(2)
        ST    8,IT6(2)
        ST    9,IT7(2)
        ST   10,IT8(2)
        ST   11,IT9(2)
        ST   12,IT10(2)

* IT11 and IT12 are already on the stack.
* R13 and R14 are available for use in EXTERN routines.

* Process the shortreals.

        SER   0,0
        AER   0,14           1st shortreal value in F14 (1st arg)
        AER   0,12           2nd shortreal value in F12 (2nd arg)
        AER   0,10           3rd shortreal value in F10 (3rd arg)
        AER   0,8            4th shortreal value in F8  (4th arg)
        AER   0,6            5th shortreal value in F6  (5th arg)
        AER   0,4            6th shortreal value in F4  (6th arg)
        AER   0,2            7th shortreal value in F2  (7th arg)
        AE    0,SR8(2)       8th shortreal val on stack(8th arg)
        AE    0,SR9(2)       9th shortreal val on stack(9th arg)
        L     3,SR(2)        Get address of VAR parm SR off stack
        STE   0,0(3)         Pass computed shortreal sum back;

* Process the reals.

        SDR   0,0
        ADR   0,14           First real value in D14  (10th arg)
        ADR   0,12           Second real value in D12 (11th arg)
        ADR   0,10           Third real value in D10  (12th arg)
        ADR   0,8            Fourth real value in D8  (13th arg)
        ADR   0,6            Fifth real value in D6   (14th arg)
        ADR   0,4            Sixth real value in D4   (15th arg)
        ADR   0,2            Seventh real value in D2 (16th arg)
        AD    0,RL8(2)       8th real value on stack  (17th arg)
        AD    0,RL9(2)       9th real value on stack  (18th arg)
        L     3,RL(2)        Get address of VAR parm RL off stack
        STD   0,0(3)         Pass computed real sum back;

* Process the integers.

*                            1st integer value passed in R3 but
        L     3,IT1(2)       R3 was saved so get IT1 from stack
        AR    3,4            2nd integer value in R4 (20th arg)
        AR    3,5            3rd integer value in R5 (21st arg)
        AR    3,6            4th integer value in R6 (22nd arg)
        AR    3,7            5th integer value in R7 (23rd arg)
        AR    3,8            6th integer value in R8 (24th arg)
        AR    3,9            7th integer value in R9 (25th arg)
        AR    3,10           8th integer value in R10 (26th arg)
        AR    3,11           9th integer value in R11 (27th arg)
```

```
         AR     3,12             10th integer value in R12(28th arg)
         A      3,IT11(2)        11th non-real on stack(29th arg)
         A      3,IT12(2)        12th non-real on stack(30th arg)
         L      4,IT(2)          Get address of VAR parm IT off stack
         ST     3,0(4)           Pass back sum of integers

*  Exit.

         L      15,RETAD(2)      Fetch return address
         L      2,OLDLB(2)       Restore LB
         BR     15               Return
         END
```

In the following example, assume a main program calls an external routine, passing eleven arguments to the routine's eleven parameters. In this case, each parameter is of structured type. The first argument is a string-array being passed to a value parameter. The second argument is a string-array being passed to a VAR variable parameter. The third and fourth arguments are unpacked records (containing an array) being passed to a value parameter and variable parameter, respectively. The fifth and sixth arguments are PACKED records being passed to a value parameter and VAR variable parameter, respectively. The seven and eighth arguments are unpacked ARRAYs being passed to a value and VAR variable parameter, respectively. The nineth and tenth arguments are PACKED ARRAYs being passed to a value and VAR variable parameter, respectively. The eleventh argument, of type STRING8, is included to depict how it will be passed on the stack, at a displacement in the activation record beyond the displacements of those arguments being passed to parameters in the general registers.

Assume the type-definitions of the Perkin-Elmer Prefix, and the type-definitions of Section 10.6.1 in the examples of unpacked/packed arrays and records are given.

Then the procedure's external declaration in the calling Pascal unit defines its interface, thusly:

```
PROCEDURE EXCAL(PARM1:STRING8; VAR PARM2:STRING8;
                PARM3:RECTYPEA; VAR PARM4:RECTYPEA;
                PARM5:RECTYPEB; VAR PARM6:RECTYPEB;
                PARM7:ARAYTYPEA; VAR PARM8:ARAYTYPEA;
                PARM9:ARAYTYPEB; VAR PARM10:ARAYTYPEB;
                PARM11:STRING8);
                EXTERN;
```

Then the external CAL routine may expect these parameters, and outline the stack, as follows:

```
EXCAL    PROG
         PURE
         ENTRY EXCAL
STACKA   STRUC
* Linkage data area on stack
OLDLB    DSF   1
RETAD    DSF   1
SLINK    DSF   1
* Parameters space on stack, reserved for 1st ten and actual 11th
* Displacements reserved for:
PARM1    DSF   1              address of string copy in R3
PARM2    DSF   1              address of string variable in R4
PARM3    DSF   1             address of unpacked record copy in R5
PARM4    DSF   1              address of unpacked record var in R6
PARM5    DSF   1              address of packed record copy in R7
PARM6    DSF   1              address of packed record var in R8
PARM7    DSF   1              address of unpacked array copy in R9
PARM8    DSF   1              address of unpacked array var in R10
PARM9    DSF   1              address of packed array copy in R11
PARM10   DSF   1              address of packed array var in R12
PARM11   DSF   1              address of string copy on stack
TEMPAREA DSF   50             Additional working storage on stack.
         ENDS
ERR      EQU   X'8804'        User-specified CAL mnemonic
LERREX   ERR   8,9998         STACK OVERFLOW message trigger
EXCAL    ST    15,RETAD(2)    Save return address to Pascal code
         CLI   0,STACKA(2)    Room enough on stack for storage?
         BCS   LERREX         Escape if not

* Fetch parameters and/or store in space on stack.

         ST    3,PARM1(2)      1st parameter
         ST    4,PARM2(2)      2nd parameter
         ST    5,PARM3(2)      3rd parameter
         ST    6,PARM4(2)      4th parameter
         ST    7,PARM5(2)      5th parameter
         ST    8,PARM6(2)      6th parameter
         ST    9,PARM7(2)      7th parameter
         ST    10,PARM8(2)     8th parameter
         ST    11,PARM9(2)     9th parameter
         ST    12,PARM10(2)    10th parameter

* Process parameters / perform routine operations.

         L     13,PARM11(2)    11th parameter actually on stack
         LB    14,0(13)        Fetch character from string value
         STB   14,0(4)         Store character to VAR string
 ...
* Exit.

         L     15,RETAD(2)    Fetch return address
         L     2,OLDLB(2)     Restore Local Base in R2
         BR    15             Return
         END
```

Additional structures may be outlined in the CAL code for ease of access to the contents of unpacked/packed structured type parameters. However, the Pascal rules for internal data storage and alignment requirements as discussed in Section 10.6.1 must be adhered to, particularly for arrays and records.

Another example to demonstrate an interface which mixes a variety of Pascal simple data-types being passed to value parameters (as all arguments passed to variable parameters are passed simply by an address) follows.

Assume an external procedure declaration is declared in the calling Pascal code, thusly:

```
PROCEDURE  SIMPVALS  (BYTEVALPARM:BYTE;
                      SINTVALPARM:SHORTINTEGER;
                      CHARVALPARM:CHAR;
                      INTVALPARM :INTEGER;
                      BOOLVALPARM:BOOLEAN;
                      REALVALPARM:REAL;
                      SRELVALPARM:SHORTREAL);
                      EXTERN;
```

Then an external CAL routine may expect these parameters, and outline the stack, as follows.

```
SIMPVALS PROG
         PURE
         ENTRY SIMPVALS
STACKB   STRUC
* Linkage data area on stack.
OLDLB    DSF   1
RETAD    DSF   1
SLINK    DSF   1
* Outline parameters space on stack.
BYTEPARM DS    1             1-byte for BYTE (byte aligned)
SINTPARM DSH   1             2-byte SHORTINTEGER(align halfword)
CHARPARM DS    1             1-byte for CHAR (byte aligned)
INTPARM  DSF   1             4-byte INTEGER (fullword aligned)
BOOLPARM DSH   1             2-byte BOOLEAN (halfword aligned)
REALPARM DSF   2             8-bytes for REAL (fullword aligned)
SRELPARM DSF   1             4-byte SHORTREAL(align fullword)
TEMPAREA DSF   20            Temporary work storage, as desired
         ENDS


* Although, because the parameters are few in number they will
* be passed in registers, when the registers of suitable kind
* are all filled with arguments, those passed to value-parameters
* of simple-types will passed with filler gaps on the stack
* as per Section 10.6.1 details internal data representations.
* Therefore, this example demonstrates how the CAL instructions
* to reserve displacements (DS, DSH, DSF) also provide filler
* gaps to obtain similarly displaced alignments.
```

```
* The stack parameters space could have also been outlined,
* with the user specifying where known filler gaps will exist.

STACKB   STRUC
* Linkage data area on stack.
OLDLB    DSF   1
RETAD    DSF   1
SLINK    DSF   1
* Outline parameters space on stack.
BYTEPARM DS    1
         DS    1                1-byte filler
SINTPARM DS    2
CHARPARM DS    1
         DS    3                3-bytes filler
INTPARM  DS    4
BOOLPARM DS    2
         DS    2                2-bytes filler
REALPARM DS    8
SRELPARM DS    4
TEMPAREA DSF   20               Additional displacements on stack
         ENDS

* Begin the routine.

ERR      EQU   X'8804'           User-specified mnemonic
LERREX   ERR   8,9997            STACK OVERFLOW message trigger
SIMPVALS ST    15,RETAD(2)       Save return address to Pascal code
         CLI   0,STACKB(2)       Room enough to work?
         BCS   LERREX            No, take an escape.

* Receive the parameters, and/or save on stack.

         STB   3,CHARPARM(2)     Save byte-value in R3 (1st arg)
         STH   4,SINTPARM(2)     Save shortinteger in R4 (2nd arg)
         STB   5,CHARPARM(2)     Save character in R5 (3rd arg)
         ST    6,INTPARM(2)      Save integer in R6(4th arg)
         STH   7,BOOLPARM(2)     Save boolean in R7(5th arg)
         STD   14,REALPARM(2)    Save real in D14(6th arg)
         STE   14,SRELPARM(2)    Save shortreal in F14(7th arg)

* Process the parameters / perform the routine's operations.

* Exit.

          L    15,RETAD(2)       Fetch return address
          L    2,OLDLB(2)        Restore Local Base in R2
          BR   15                Return
          END
```

## 10.8 Pascal INTERFACE TO ROUTINES DECLARED WITH THE FORTRAN DIRECTIVE

This section describes the Pascal-FORTRAN interface generated by the Pascal compiler and supported also by the Pascal Runtime Library to routines declared as external with the FORTRAN directive:

- external FORTRAN subprograms: subroutines and functions
- external FORTRAN subprograms using FORTRAN I/O
- external FORTRAN Runtime Library routines (no argument data-type checking)
- external CAL routines, using FORTRAN linkage conventions

Additionally discussed at the end of the section, is the case of an external CAL routine declared with EXTERN, which is to reference a FORTRAN routine, undeclared as FORTRAN in the Pascal code.

When a Pascal program calls on external subprograms that are written in FORTRAN, and the routine has been declared external with the FORTRAN directive, the Pascal program provides the FORTRAN subprogram with certain resources:

- a list of addresses of actual arguments, in the form that the FORTRAN subprogram expects; i.e., the address of this list;
- an RTL Scratchpad Space of approximately X'600' bytes, for the FORTRAN VII RTL(without argument checking); and a pointer in General Register R1, which points to the top of this area;
- a Static Communications Area; as needed for some FORTRAN I/O subprograms
- the highest available MAX_LU logical unit is not used by Pascal so that it is left free for FORTRAN's error handling mechanism.
- and an address to return to in the Pascal code.

The setup of the list of addresses is a cooperative effort shared by the Pascal compiler and RTL. The Pascal-FORTRAN interface provided is:

> General Register R14 contains zero if there is no argument-list in the routine-invocation of a FORTRAN subprogram. (previously declared with the Pascal "FORTRAN" directive.)
>
> or

General Register R14 contains the address of the list of arguments plus its high-order bit set.

> The number of arguments in the list to which R14 points, is the same number of arguments in the invocation (which is also the number of parameters in the routine-declaration).
>
> The arguments are passed in this list as addresses, with each address occupying one fullword, and the entire list begining on a fullword boundary. All arguments are passed as an address in this list to the external FORTRAN routine. Argument expressions are evaluated and a copy of the value is made. Arguments being passed to value parameters are sent in this list as an address of a copy of the value of the argument expression. Variable parameters have their actual address passed. This allows any changes made to a value parameter in the external routine not to be transmitted upon return to the caller. This also allows any change made to a variable parameter in the external routine to actually be reflected in the actual variable argument upon return to the caller.
>
> The last fullword of the address-list has its high-order bit set, to indicate the end of the list of arguments.

General Register R15 contains the return address in P$FORT of Pascal, for the FORTRAN subprogram to return to.

Pascal provides a FORTRAN interface, as documented in the FORTRAN VII Run Time Library Introduction and Overview Reference Manual, Publication Number S29-578R02, where it known as the "E-interface". This portion of S29-578 has been incorporated also in the FORTRAN VII User Manual, Publication Number 48-010. This interface is suitable for linkage to the FORTRAN VII RTL (without argument checking). Pascal does not provide, for example, the data class fields and type fields in the address-list of arguments, as would be used for the FORTRAN VII RTL (with argument checking). Also, the data class field (bits 1-3) in General Register R14 are not set to 5 (for a function) or 6 (for a subroutine) to differentiate the type of FORTRAN subprogram being linked to.

The RTL Scratchpad space is located before (at lower addresses than) the Global Variables area of the Pascal main program (see Figures 10-4 and 10-5). When a Pascal RTL routine is entered, R1 points to the top of the RTL Scratchpad area. Each Pascal RTL PASLIB routine gets space for its local variables by decreasing R1, and reinstates R1 prior to exiting. When a routine invocation in Pascal calls an externally declared FORTRAN routine, the Pascal compiler-generated code calls P$FORT in its RTL to set up the FORTRAN Interface. As any external FORTRAN routine, is called through Pascal's P$FORT, note that R1 has been decremented 12 bytes lower than GB for the local storage needed by P$FORT prior to linking to the external FORTRAN routine; such that X'600' - X'C' bytes are available on the common RTL Scratchpad for the FORTRAN routine to decrement R1 by.

If Pascal code contains a call to a routine which has been declared with the directive FORTRAN, then the alternate Pascal-FORTRAN interface version of P$INIT of the Pascal RTL will set aside space for the SCA and call the FORTRAN RTL routine .INITSCA to initialize the SCA.

An external routine written in CAL, using FORTRAN linkage conventions, and declared with the directive FORTRAN, also receives the above interface upon its routine invocation in the Pascal program. The Pascal-FORTRAN linkage to the FORTRAN VII RTL (with no argument checking) routines is an example of CAL written routines effectively using FORTRAN linkage conventions which can be linked to, by simply declaring them as external with the FORTRAN directive (see Chapter 3).

An external routine written in CAL, using Pascal linkage conventions, for EXTERN routines, is described above in Section 10.7.

It is possible to have external CAL routines, declared with the EXTERN directive, and also have them interface with FORTRAN routines. However, certain precautions must be taken in this case. Suppose the Pascal code calls an external routine written in CAL, which is declared in the Pascal code with the directive EXTERN. Then, if this external CAL routine calls FORTRAN, the need for SCA support would be invisible to the Pascal code generating interface system. What the programmer must do in this case is to put the following pseudo-op in the CAL code of any CAL routine using Pascal Linkage Conventions to the Pascal code but also refering to FORTRAN externals from itself, (and there is no other FORTRAN directive in the Pascal code). For example:

```
INCLD P$FORT
```

This forces an external reference to P$FORT to require resolution at task establishment linking time. Such a CAL routine should be included prior to the linking of the Pascal RTL file PASRTL.OBJ so the alternate P$INIT within P$FORT is resolved correctly for a Pascal-FORTRAN interface. Otherwise, the Pascal code, not seeing the need for FORTRAN interfacing will obtain resolution of its external references to P$INIT for a non-FORTRAN evironment.

When such a CAL EXTERN routine (calling FORTRAN) is linked prior to linking PASRTL.OBJ, the result is that the appropriate object code is linked in from the Pascal RTL (P$FORT). Also the alternate version of P$INIT to support the FORTRAN SCA is encorporated.

# APPENDIX A
# SAMPLE LISTINGS

```
PROGRAM NAME = PRIMES                                                   09:59:09  04/16/82  PAGE     1
  PASCAL R01-00      LICENSED RESTRICTED RIGHTS AS STATED IN  ************************

 PROG PRC SMT
 LINE LVL LVL

   1   0   0   !{PRIMES.PAS                                            03-248R00-00  }
   2   0   0   ! {Copyright Perkin-Elmer September 1980                              }
   3   0   0   !
   4   0   0   !{Test program for verification of compiler operation after     }
   5   0   0   !{unpackaging.                                                  }
   6   0   0   !
   7   0   0   !
   8   0   0   !PROGRAM PRIMES (LISTING);
   9   0   0   !
  10   0   0   !CONST LINELENGTH = 80;
  11   0   0   !
  12   0   0   !CONST LIMIT = 100000;
  13   0   0   !
  14   0   0   !CONST PAGE_SIZE   = 54;
  15   0   0   !
  16   0   0   !VAR { SCALARS }
  17   0   0   !
  18   0   0   !    LISTING    : TEXT;
  19   0   0   !    PROCESSING : BOOLEAN;
  20   0   0   !    P,I,VALUE  : INTEGER;
  21   0   0   !    PTR        : SHORTINTEGER;
  22   0   0   !    LINE_COUNT : INTEGER;
  23   0   0   !
  24   0   0   !    { ARRAYS }
  25   0   0   !
  26   0   0   !    NUMBER     : ARRAY [1..LIMIT] OF CHAR;
  27   0   0   !
  28   0   0   !PROCEDURE MARK_NON_PRIME;
  29   1   0   !BEGIN
  30   1   1   !  I := P;
  31   1   1   !  P := P + I;
  32   1   1   !  WHILE P <= LIMIT DO BEGIN
  33   1   2   !    NUMBER[P] := 'N';
  34   1   2   !    P := P+I;
  35   1   2   !    END;
  36   1   1   !END; { MARK_NON_PRIME }
  37   0   0   !
  38   0   0   !BEGIN { MAIN }
  39   0   1   !  REWRITE (LISTING);
  40   0   1   !  PAGE (LISTING);
  41   0   1   !  WRITELN (LISTING,'PRIMES:  PASCAL DEMONSTRATION PROGRAM  03-248R00-00');
  42   0   1   !  WRITELN (LISTING);
  43   0   1   !  WRITELN (LISTING,
  44   0   1   !   'THE FOLLOWING TABLE IS THE PRIME NUMBERS IN THE RANGE 1..100000');
  45   0   1   !  WRITELN (LISTING);
  46   0   1   !  FOR I := 2 TO LIMIT DO
  47   0   1   !    NUMBER[I] := 'Y';
  48   0   1   !  P := 2;
  49   0   1   !  MARK_NON_PRIME;
  50   0   1   !  PROCESSING := TRUE;
  51   0   1   !  WHILE PROCESSING DO BEGIN
  52   0   2   !    P := I;
  53   0   2   !    REPEAT
```

## APPENDIX A (Continued)
## SAMPLE LISTINGS

```
PROGRAM NAME = PRIMES                                                                       09:59:09  04/16/82  PAGE    2
   PASCAL R01-00      LICENSED RESTRICTED RIGHTS AS STATED IN  ************************

   54    0    2    !        P := P+1;
   55    0    2    !        UNTIL (P > LIMIT) OR (NUMBER [P] = 'Y');
   56    0    2    !      PROCESSING := NOT(P > LIMIT);
   57    0    2    !      IF P <= LIMIT THEN MARK_NON_PRIME;
   58    0    2    !      END;
   59    0    1    !    PTR := 1; { INITIALIZE POINTER INTO LINE }
   60    0    1    !    FOR P := 2 TO LIMIT DO
   61    0    1    !      IF (NUMBER[P] = 'Y') THEN BEGIN
   62    0    2    !        IF (PTR + 8) > LINELENGTH THEN BEGIN
   63    0    3    !          WRITELN (LISTING);
   64    0    3    !          LINE_COUNT := LINE_COUNT +1;
   65    0    3    !          IF LINE_COUNT = PAGE_SIZE THEN BEGIN
   66    0    4    !            PAGE ( LISTING );
   67    0    4    !            LINE_COUNT := 0;
   68    0    4    !            END;
   69    0    3    !          PTR := 1;
   70    0    3    !          END;
   71    0    2    !        WRITE (LISTING,P:8);
   72    0    2    !        PTR := PTR + 8;
   73    0    2    !        END;
   74    0    1    !      IF PTR > 1 THEN WRITELN (LISTING);
   75    0    1    !
   76    0    1    !END.
```

```
PROGRAM NAME = PRIMES                                                                        09:59:09  04/16/82  PAGE    3
   PASCAL R01-00      LICENSED RESTRICTED RIGHTS AS STATED IN  ************************

                                        CROSS REFERENCE LISTING


                                        INDEX TO MNEMONICS

                                        :# = CHANGE OF VALUE
                                        :L = LABEL DECLARATION
                                        :@ = LABEL REFERENCE
                                        :C = CONSTANT DECLARATION
                                        :T = TYPE DECLARATION
                                        :V = VAR DECLARATION
                                        :P = PROCEDURE NAME
                                        :F = FUNCTION NAME


BOOLEAN                     19
CHAR                        26
I                           20:V     30:#     31       34       46:#     47:#     52
INTEGER                     20       22
LIMIT                       12:C     26       32       46       55       56       57       60
LINELENGTH                  10:C     62
LINE_COUNT                  22:V     64:#     54       65       67:#
LISTING                     18:V     39       40       41       42       43       45       63       66       71       74
MARK_NON_PRIME              28:P     49       57
NUMBER                      26:V     33:#     47:#     55       61
P                           20:V     30       31:#     31       32       33:#     34:#     34       48:#     52:#     54:#     54
                            55       55       56       57       60:#     51       71
PAGE                        40       66
PAGE_SIZE                   14:C     65
PRIMES                       8
PROCESSING                  19:V     50:#     51       56:#
PTR                         21:V     59:#     62       69:#     72:#     72       74
REWRITE                     39
SHORTINTEGER                21
TEXT                        18
TRUE                        50
VALUE                       20:V
WRITE                       71
WRITELN                     41       42       43       45       53       74
```

```
PROGRAM NAME = PRIMES                                                          09:59:09  04/16/82  PAGE    4
   PASCAL R01-00      LICENSED RESTRICTED RIGHTS AS STATED IN  ***********************


PASCAL COMPILER INTERNAL STATISTICS


SUMMARY, PASS1, FILE LENGTH: 16 SECTORS
UNIQUE IDENTIFIERS   84

SUMMARY, PASS2, FILE LENGTH: 12 SECTORS

SUMMARY, PASS3, FILE LENGTH: 15 SECTORS , HEAP USED: 2768 BYTES
NOUNS USED  111  UPDATES USED    0

SUMMARY, PASS4, FILE LENGTH: 30 SECTORS , HEAP USED: 2840 BYTES

SUMMARY, PASS5, FILE LENGTH: 16 SECTORS

SUMMARY, PASS6, FILE LENGTH: 14 SECTORS , HEAP USED: 10512 BYTES
OPERATION CLASS (TOTAL/OPTIMIZED):
ARITHOPS      0      0  BOOLOPS      1      0  CMPRS        9      0
CONDJUMPS     8      0  INDXCHKS     4      0  FORS         2      0
MULDIVS       0      0  NEGNOTS      0      0  RANGECHKS    0      0
STDFUNCS      0      0  MODS         0      0  BUILDSETS    0      0
IMMEDS        5      4  BITIMMEDS    0      0  SETOPS       0      0
INDEXEXPR     0      0

SUMMARY, PASS7, FILE LENGTH: 22 SECTORS
REGISTERS ASSIGNED:
REG  GEN   REAL SREAL       REG  GEN
 0     0     0    0          1    0
 2     0     0    0          3   30
 4     3     0    0          5    0
 6     0     0    0          7    0
 8     0     0    0          9    0
10     0     0    0         11    0
12     0     0    0         13    0
14     0     0    0         15    0
PROCEDURE/FUNCTION CALLS     2

SUMMARY, PASS8, FILE LENGTH: 7 SECTORS , HEAP USED: 10700 BYTES
CODE BEFORE =   1054 BYTES  AFTER =         750 BYTES  LITS BEFORE =   116 BYTES  AFTER =          116 BYTES
DATA AREA LENGTH = 100384
PASS COUNTS, PHASE 1        4 PHASE 3          4 BLOCKS    2
OPTIMIZATIONS PERFORMED:
CROSS_LINK          0  COMBINE_LABELS     0  OPT_LAB          15
BRANCH_CHAIN        0  CHECK_COND1        0  CHECK_COND2       0
CHECK_COND3         0  OPT_BRANCH1        0  OPT_BRANCH2       0
OPT_BRANCH3         0  OPT_CODELAB        0  REMOVE_NO_COND    0
NULL_OP1            0  NULL_OP2           0  AFTER_RR          0
AFTER_RX            0  AFTER_LI           0  OPT_BR            0
CHAIN_BRANCH        0  TRY_SHORT_BR      10  BEFORE_LOAD       0
BEFORE_MULT         0  OPT_IMMEDIATES     0  AFTER_LA          0
```

```
PROGRAM NAME = PRIMES                                                  09:59:09  04/16/82  PAGE    5
   PASCAL R01-00      LICENSED RESTRICTED RIGHTS AS STATED IN  ************************

EXTERNAL FILE TABLE

LISTING   0


FILES USED FOR THIS COMPILATION

INPUT FILE:      M300:PRIMES.PAS/P

LISTING FILE:    M300:PRIMES.LST/P

OBJECT FILE:     M300:PRIMES.OBJ/P

ASSEMBLY LIST:   M300:PRIMES.ASM/P


START OPTIONS: LOG ASSEM SUM

                DEFAULT      FINAL  PASSED IN
                STATE        STATE     START

ASSEMBLY          OFF          ON        YES

BATCH             OFF          OFF       NO

BOUNDSCHECK       ON           ON        NO

CROSS REF         ON           ON        NO

LIST              ON           ON        NO

LOG               OFF                    YES

MAP               OFF          OFF       NO

OPTIMIZE          OFF                    NO

RANGECHECK        ON           ON        NO

SUMMARY           OFF          ON        YES

COMPILATION COMPLETE, NO ERRORS
CODE SIZE = 752 BYTES  CONST SIZE = 120 BYTES
PERKIN-ELMER OS/32 LINKAGE EDITOR 03-242 R00-01
 OPTION FLOAT,DFLOAT,ABS=100,WORK=(624,40000)
 INCLUDE PRIMES.OBJ
 LI PASRTL.OBJ
 LI M300:F7RTL.OBJ/S
 MAP PRIMES.LST,ADDRESS,XREF
 BUILD PRIMES
```

```
PERKIN-ELMER OS/32 LINKAGE EDITOR 03-242 R00-01         ESTABLISHMENT SUMMARY                              PAGE     1

 -- IMAGE LINKED AT 09:59:26 ON APRIL 16, 1982 --


FILE NAME: M300:PRIMES.TSK/P -- RECORDS:   17


UDOT:        0 -- UTOP:    FB8 -- CTOP:   15FE -- SIZE:     5.50 KB


TASK OPTIONS:

NXSVC1              NVFC                UTASK               AFPAUSE             FLOAT               NRESIDENT
NCONTROL            NCOMMUNICATE        SVCPAUSE            DFLOAT              ROLL                ACCOUNTING
NACPRIVILEGE        NDISC               NUNIVERSAL          KEYCHECK            NSEGMENTED

TEQSAVE=ALL  LU=15  SYSSPACE=3000  WORK=(624,40000)  ABSOLUTE=100  IOBLOCKS=1  PRIORITY=(128,128)  TSW=(0,170)


NODE MAP:

LEVEL NAME       LENGTH      PURE     IMPURE     COMMON     TABLES

  0   .ROOT         FB8         0        FB8          0          0
      (TOTALS)      FB8         0        FB8          0          0


VIRTUAL ADDRESS MAP:

FROM     TO        SEGMENT NAME        SIZE     ACCESS

000000  0015FF      (IMPURE)         5.50 KB
```

```
PERKIN-ELMER OS/32 LINKAGE EDITOR 03-242 R00-01          ADDRESS MAP                                     PAGE    1

 -- IMAGE LINKED AT 09:59:26 ON APRIL 16, 1982 --


NODE: .ROOT      - LEVEL:  0 - ADDRESS:      0 - SIZE:    FB8 - PARENT:

SYMBOL    --    ADDRESS         SYMBOL    --    ADDRESS         SYMBOL    --    ADDRESS         SYMBOL    --    ADDRESS

P-PRIMES        000100-P        E-PRIMES        000170-P        P-P$INIT        000468-P        E-P$INIT        000468-P
P-PAS.ERR       000598-P        E-P$ERR         000598-P        E-P$PAUS        000696-P        E-P$TERM        00069E-P
E-P$SEND        0006CC-P        P-PWRT.INT      0006F0-P        E-P$WRITSI      0006F0-P        E-P$WRITI       0006F0-P
E-P$WRITBY      0006F0-P        P-P$WRITS       0007A0-P        E-P$WRITS       0007A0-P        P-P$PAGE        0007F8-P
E-P$PAGE        0007F8-P        P-P$PURGE       000828-P        E-P$PURGE       000828-P        P-P$PUTT        000858-P
E-P$PUTT        000858-P        P-P$WRITLN      000890-P        E-P$WRITLN      000890-P        P-P$REWRIT      0008E8-P
E-P$REWRIT      0008E8-P        P-P$EFCB        000958-P        E-P$EFCB        000958-P        P-P$$REWD       000990-P
E-P$$REWD       000990-P        P-P$FCBERR      0009E0-P        E-P$FCBERR      0009E0-P        P-P$$SVC1       000A40-P
E-P$$SVC1       000A40-P        P-P$$SVC7       000AC0-P        E-P$$SVC7       000AC0-P        P-P$PUTERR      000CD0-P
E-P$PUTERR      000CD0-P        P-P$ERMES       000D58-P        E-P$ERMES       000D58-P
```

```
PERKIN-ELMER OS/32 LINKAGE EDITOR 03-242 R00-01         CROSS REFERENCE MAP                                          PAGE    1

 -- IMAGE LINKED AT 09:59:26 ON APRIL 16, 1982 --


SYMBOL       DEFINED        REFERENCED BY

E-P$$REWD    P$$REWD        P$REWRIT
E-P$$SVC1    P$$SVC1        P$$REWD    P$WRITLN
E-P$$SVC7    P$$SVC7        P$FCBERR
E-P$EFCB     P$EFCB         PRIMES
E-P$ERMES    P$ERMES        PAS.ERR
E-P$ERR      PAS.ERR        PRIMES
E-P$FCBERR   P$FCBERR       P$REWRIT
E-P$INIT     P$INIT         PRIMES
E-P$PAGE     P$PAGE         PRIMES
E-P$PAUS     PAS.ERR        P$$REWD    P$ERMES    P$PUTERR   P$REWRIT   P$WRITLN
E-P$PURGE    P$PURGE        P$PAGE     PRIMES
E-P$PUTERR   P$PUTERR       P$PUTT     P$WRITLN   PWRT.INT
E-P$PUTT     P$PUTT         P$PAGE     P$WRITS    PWRT.INT
E-P$REWRIT   P$REWRIT       PRIMES
E-P$SEND     PAS.ERR        P$$SVC1    P$$SVC7    P$ERMES    P$FCBERR   P$INIT     P$PUTERR
E-P$TERM     PAS.ERR        P$ERMES    P$INIT     P$PUTERR   PRIMES
E-P$WRITBY   PWRT.INT
E-P$WRITI    PWRT.INT       PRIMES
E-P$WRITLN   P$WRITLN       P$PAGE     P$PURGE    P$PUTT     PRIMES
E-P$WRITS    P$WRITS        PRIMES
E-P$WRITSI   PWRT.INT
E-PRIMES     PRIMES
```

```
                           PRIMES   PROG
                           * LINE 29
000000   8880 001D         L38      ERR   R8,29
000004   50F2 0004         P101     ST    R15,4(R2)
000008   C502 000C                  CLHI  R0,12(R2)
00000C   2086                       BCS   L38
                           * LINE 30
00000E   5831 0160                  L     R3,352(R1)
000012   5031 0164                  ST    R3,356(R1)
                           * LINE 31
000016   5831 0164                  L     R3,356(R1)
00001A   5131 0160                  AM    R3,352(R1)
                           * LINE 32
00001E   5831 0160         L2       L     R3,352(R1)
000022   F930 0001 86A0             CI    R3,100000
000028   4220 8036                  BP    L3
                           * LINE 33
00002C   5831 0160                  L     R3,352(R1)
000030   C930 0001                  CHI   R3,1
000034   2115                       BMS   L39
000036   F930 0001 86A1             CI    R3,100001
00003C   2113                       BMS   L40
00003E   8810 0021         L39      ERR   R1,33
000042   C840 004E         L40      LHI   R4,78
000046   C540 0080                  CLHI  R4,128
00004A   2183                       BCS   L42
00004C   8830 0021                  ERR   R3,33
000050   D241 4300 0173    L42      STB   R4,371(R1,R3)
                           * LINE 34
000056   5831 0164                  L     R3,356(R1)
00005A   5131 0160                  AM    R3,352(R1)
                           * LINE 35
00005E   4300 FFBC                  B     L2
                           * LINE 36
000062   58F2 0004         L3       L     R15,4(R2)
000066   5822 0000                  L     R2,0(R2)
00006A   030F                       BR    R15
                           * LINE 38
00006C   8880 0026         L43      ERR   R8,38
000070   C8E0 0064         P1       LHI   R14,100
000074   24D1                       LIS   R13,1
000076   41F0 4000 0000             BAL   R15,P102
00007C   41F0 4000 0000             BAL   R15,P103
000082   F502 0001 8820             CLI   R0,100384(R2)
000088   208E                       BCS   L43
00008A   E6E2 000C                  LA    R14,12(R2)
00008E   C8D0 0100                  LHI   R13,256
000092   24C0                       LIS   R12,0
000094   41F0 4000 0000             BAL   R15,P104
                           * LINE 39
00009A   E6E1 000C                  LA    R14,12(R1)
00009E   41F0 4000 0000             BAL   R15,P105
                           * LINE 40
0000A4   E6E1 000C                  LA    R14,12(R1)
0000A8   41F0 4000 0000             BAL   R15,P106
```

```
                                   * LINE 41
0000AE    F8D0 8000 0000                     LI      R13,-2147483648
0000B4    E6E1 000C                          LA      R14,12(R1)
0000B8    E6C0 8234                          LA      R12,752
0000BC    41F0 4000 0000                     BAL     R15,P107
0000C2    0033                               DC      51
0000C4    E6E1 000C                          LA      R14,12(R1)
0000C8    41F0 4000 0000                     BAL     R15,P108
                                   * LINE 42
0000CE    E6E1 000C                          LA      R14,12(R1)
0000D2    41F0 4000 00CA                     BAL     R15,P108
                                   * LINE 44
0000D8    F8D0 8000 0000                     LI      R13,-2147483648
0000DE    E6E1 000C                          LA      R14,12(R1)
0000E2    E6C0 823F                          LA      R12,804
0000E6    41F0 4000 003F                     BAL     R15,P107
0000EC    003F                               DC      63
0000EE    E6E1 000C                          LA      R14,12(R1)
0000F2    41F0 4000 00D4                     BAL     R15,P108
                                   * LINE 45
0000F8    E6E1 000C                          LA      R14,12(R1)
0000FC    41F0 4000 00F4                     BAL     R15,P108
                                   * LINE 46
000102    2432                               LIS     R3,2
000104    5031 0164                L4        ST      R3,356(R1)
                                   * LINE 47
000108    5831 0164                          L       R3,356(R1)
00010C    C930 0001                          CHI     R3,1
000110    2115                               BMS     L44
000112    F930 0001 86A1                     CI      R3,100001
000118    2113                               BMS     L45
00011A    8810 002F                L44       ERR     R1,47
00011E    C840 0059                L45       LHI     R4,89
000122    C540 0080                          CLHI    R4,128
000126    2183                               BCS     L47
000128    8830 002F                          ERR     R3,47
00012C    D241 4300 0173           L47       STB     R4,371(R1,R3)
000132    5831 0164                          L       R3,356(R1)
000136    2631                               AIS     R3,1
000138    F930 0001 86A0                     CI      R3,100000
00013E    4320 FFC2                          BNP     L4
                                   * LINE 48
000142    2432                               LIS     R3,2
000144    5031 0160                          ST      R3,352(R1)
                                   * LINE 49
000148    5022 4001 8814                     ST      R2,100372(R2)
00014E    5022 4001 881C                     ST      R2,100380(R2)
000154    FA20 0001 8814                     AI      R2,100372
00015A    41F0 FEA6                          BAL     R15,P101
                                   * LINE 50
00015E    2431                               LIS     R3,1
000160    C530 0002                          CLHI    R3,2
000164    2193                               BCS     L49
000166    8830 0032                          ERR     R3,50
00016A    4031 015C                L49       STH     R3,348(R1)
                                   * LINE 51
00016E    4831 015C                L6        LH      R3,348(R1)
000172    4330 8084                          BZ      L7
```

```
                                   * LINE 52
000176    5831 0164                          L      R3,356(R1)
00017A    5031 0160                          ST     R3,352(R1)
                                   * LINE 54
00017E    2431                     L8        LIS    R3,1
000180    5131 0160                          AM     R3,352(R1)
                                   * LINE 55
000184    5831 0160                          L      R3,352(R1)
000188    F930 0001 86A0                     CI     R3,100000
00018E    4220 8024                          BP     L20
000192    5831 0160                          L      R3,352(R1)
000196    C930 0001                          CHI    R3,1
00019A    2115                               BMS    L50
00019C    F930 0001 86A1                     CI     R3,100001
0001A2    2113                               BMS    L51
0001A4    8810 0037                L50       ERR    R1,55
0001A8    D331 4300 0173           L51       LB     R3,371(R1,R3)
0001AE    C930 0059                          CHI    R3,89
0001B2    4230 FFC8                          BNZ    L8
                                   * LINE 56
0001B6    2431                     L20       LIS    R3,1
0001B8    5841 0160                          L      R4,352(R1)
0001BC    F940 0001 86A0                     CI     R4,100000
0001C2    2322                               BNPS   L24
0001C4    2430                               LIS    R3,0
0001C6    C530 0002                L24       CLHI   R3,2
0001CA    2183                               BCS    L53
0001CC    8830 0038                          ERR    R3,56
0001D0    4031 015C                L53       STH    R3,348(R1)
                                   * LINE 57
0001D4    5831 0160                          L      R3,352(R1)
0001D8    F930 0001 86A0                     CI     R3,100000
0001DE    212C                               BPS    L9
0001E0    5022 4001 8814                     ST     R2,100372(R2)
0001E6    5022 4001 881C                     ST     R2,100380(R2)
0001EC    FA20 0001 8814                     AI     R2,100372
0001F2    41F0 FE0E                          BAL    R15,P101
                                   * LINE 58
0001F6    4300 FF74                L9        B      L6
                                   * LINE 59
0001FA    2431                     L7        LIS    R3,1
0001FC    C930 8000                          CHI    R3,-32768
000200    2115                               BMS    L54
000202    F930 0000 8000                     CI     R3,32768
000208    2113                               BMS    L55
00020A    8830 003B                L54       ERR    R3,59
00020E    4031 016C                L55       STH    R3,364(R1)
                                   * LINE 60
000212    2432                               LIS    R3,2
000214    5031 0160                L10       ST     R3,352(R1)
                                   * LINE 61
000218    5831 0160                          L      R3,352(R1)
00021C    C930 0001                          CHI    R3,1
000220    2115                               BMS    L56
000222    F930 0001 86A1                     CI     R3,100001
000228    2113                               BMS    L57
00022A    8810 003D                L56       ERR    R1,61
00022E    D331 4300 0173           L57       LB     R3,371(R1,R3)
000234    C930 0059                          CHI    R3,89
000238    4230 807C                          BNZ    L12
```

# APPENDIX A (Continued)
# SAMPLE LISTINGS

```
                              * LINE 62
00023C    4831 016C                   LH      R3,364(R1)
000240    2638                        AIS     R3,8
000242    C930 0050                   CHI     R3,80
000246    4320 8042                   BNP     L13
                              * LINE 63
00024A    E6E1 000C                   LA      R14,12(R1)
00024E    41F0 4000 00FE              BAL     R15,P108
                              * LINE 64
000254    2431                        LIS     R3,1
000256    5131 0170                   AM      R3,368(R1)
                              * LINE 65
00025A    5831 0170                   L       R3,368(R1)
00025E    C930 0036                   CHI     R3,54
000262    2139                        BNZS    L14
                              * LINE 66
000264    E6E1 000C                   LA      R14,12(R1)
000268    41F0 4000 00AA              BAL     R15,P106
                              * LINE 67
00026E    2430                        LIS     R3,0
000270    5031 0170                   ST      R3,368(R1)
                              * LINE 69
000274    2431                L14     LIS     R3,1
000276    C930 8000                   CHI     R3,-32768
00027A    2115                        BMS     L58
00027C    F930 0000 8000              CI      R3,32768
000282    2113                        BMS     L59
000284    8830 0045           L58     ERR     R3,69
000288    4031 016C           L59     STH     R3,364(R1)
                              * LINE 71
00028C    24D8                L13     LIS     R13,8
00028E    E6E1 000C                   LA      R14,12(R1)
000292    58C1 0160                   L       R12,352(R1)
000296    41F0 4000 0000              BAL     R15,P109
                              * LINE 72
00029C    4831 016C                   LH      R3,364(R1)
0002A0    2638                        AIS     R3,8
0002A2    C930 8000                   CHI     R3,-32768
0002A6    2115                        BMS     L60
0002A8    F930 0000 8000              CI      R3,32768
0002AE    2113                        BMS     L61
0002B0    8830 0048           L60     ERR     R3,72
0002B4    4031 016C           L61     STH     R3,364(R1)
                              * LINE 73
0002B8    5831 0160           L12     L       R3,352(R1)
0002BC    2631                        AIS     R3,1
0002BE    F930 0001 86A0              CI      R3,100000
0002C4    4320 FF4C                   BNP     L10
                              * LINE 74
0002C8    4831 016C                   LH      R3,364(R1)
0002CC    C930 0001                   CHI     R3,1
0002D0    2326                        BNPS    L15
0002D2    E6E1 000C                   LA      R14,12(R1)
0002D6    41F0 4000 0250              BAL     R15,P108
```

```
                                   * LINE 76
0002DC    E6E1 000C                L15       LA     R14,12(R1)
0002E0    41F0 4000 0000                     BAL    R15,P110
0002E6    24E0                               LIS    R14,0
0002E8    41F0 4000 0000                     BAL    R15,P111
0002F0                                       ALIGN  8
0002F0    5052 494D                          DCF    1347569997
0002F4    4553 3A20                          DCF    1163082272
0002F8    2050 4153                          DCF    542130515
0002FC    4341 4C20                          DCF    1128352800
000300    4445 4D4F                          DCF    1145392463
000304    4E53 5452                          DCF    1314083922
000308    4154 494F                          DCF    1096042831
00030C    4E20 5052                          DCF    1310740562
000310    4F47 5241                          DCF    1330074177
000314    4D20 2030                          DCF    1293951024
000318    3320 3234                          DCF    858599988
00031C    3852 3030                          DCF    944910384
000320    2D30 3020                          DCF    758132768
000324    5448 4520                          DCF    1414022432
000328    464F 4C4C                          DCF    1179601996
00032C    4F57 494E                          DCF    1331120462
000330    4720 5441                          DCF    1193301057
000334    424C 4520                          DCF    1112294688
000338    4953 2054                          DCF    1230184532
00033C    4845 2050                          DCF    1212489808
000340    5249 4D45                          DCF    1380535621
000344    204E 554D                          DCF    542004557
000348    4245 5253                          DCF    1111839315
00034C    2049 4E20                          DCF    541675040
000350    5448 4520                          DCF    1414022432
000354    5241 4E47                          DCF    1380011591
000358    4520 312E                          DCF    1159737646
00035C    2E31 3030                          DCF    774975536
000360    3030 3020                          DCF    808464416
000364    0000 0000                          DCF    0
                                             EXTRN  P$INIT
          0000 0078                P102      EQU    P$INIT
                                             EXTRN  P$ERR
          0000 007E                P103      EQU    P$ERR
                                             EXTRN  P$EFCB
          0000 0096                P104      EQU    P$EFCB
                                             EXTRN  P$REWRIT
          0000 00A0                P105      EQU    P$REWRIT
                                             EXTRN  P$PAGE
          0000 026A                P106      EQU    P$PAGE
                                             EXTRN  P$WRITS
          0000 00E8                P107      EQU    P$WRITS
                                             EXTRN  P$WRITLN
          0000 02D8                P108      EQU    P$WRITLN
                                             EXTRN  P$WRITI
          0000 0298                P109      EQU    P$WRITI
                                             EXTRN  P$PURGE
          0000 02E2                P110      EQU    P$PURGE
                                             EXTRN  P$TERM
          0000 02EA                P111      EQU    P$TERM
                                             ENTRY  PRIMES
          0000 0070                PRIMES    EQU    P1
000368    0000 0070                          END    P1
```

# APPENDIX B
# SOFTWARE CHANGE REQUEST SUBMISSION INFORMATION

If an unresovable problem should arise with the Pascal Compiler or the code it has compiled, the problem and the following materials should be directed to a Perkin-Elmer Software Change Request Coordinator:

- A compiled-program listing of the Pascal source with a cross-reference listing and an assembly listing;
- A hard copy of the source (card deck or tape), particularly if the program is large;
- A description of the problem, on an SCR form;
- Any run time data, if applicable and possible.
- For a compiler malfunction occurring during compilation of a user program, additionally supply the compiler failure error message listed to the log device. For example, compiler failure messages are of the form:

  ```
  PASS n LINE xxxxx, ADDR yyyyyy message.......
  COMPILING LINE zzzzz OF PROGRAM name....
  ```

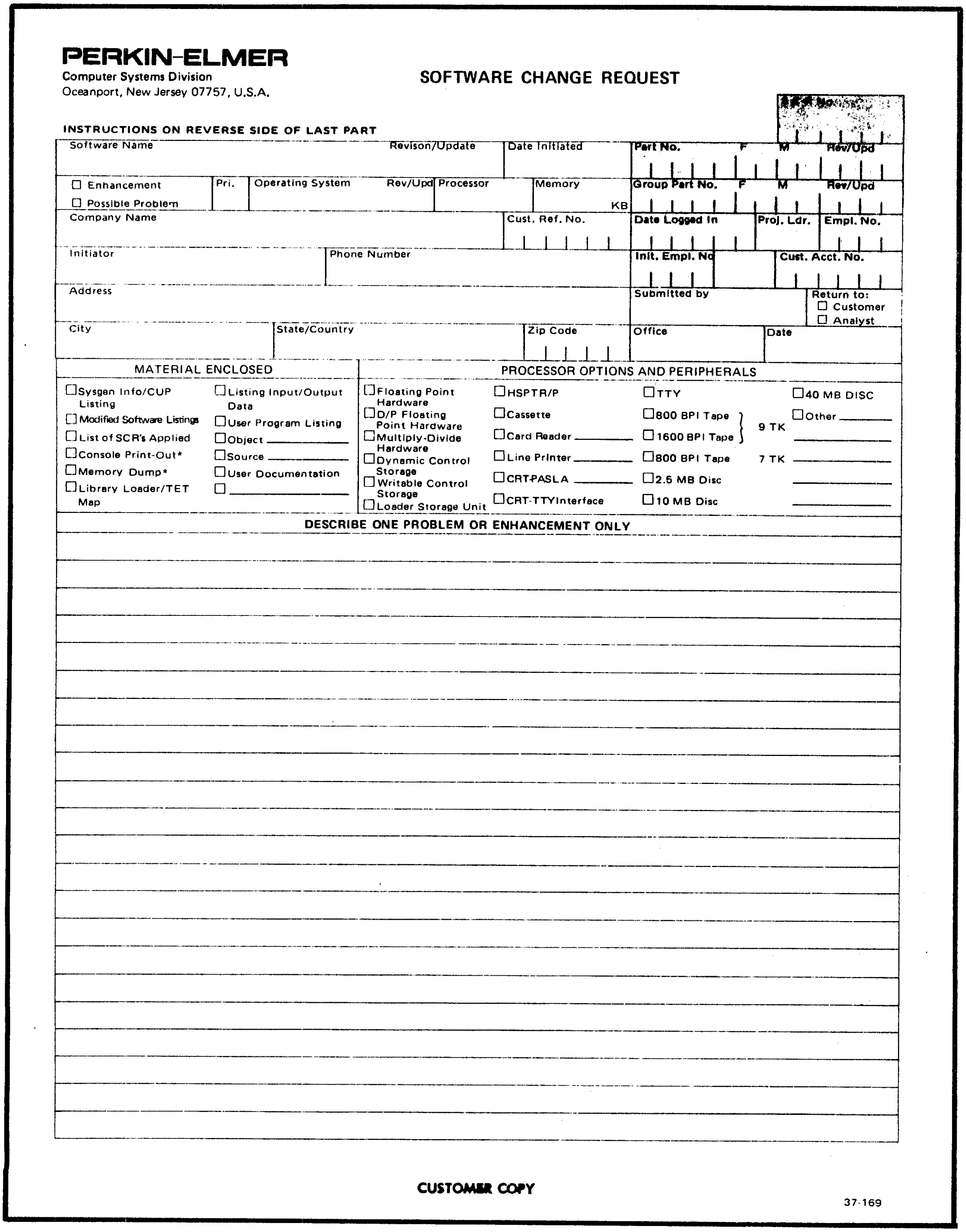
  when message is neither STACK OVERFLOW or HEAP OVERFLOW (which are problematic indicators of not enough memory allocated for the task); as detailed in Section 1.5.4.2 of Chapter 1 of this manual.

Failure to comply with the above requirements may render the problem unregeneratable.

When submitting an SCR, please label and identify clearly all enclosures additionally supplied with the SCR.

Enhancements to the compiler also can be requested through the use of an SCR.

A sample SCR form is included for your convenience.



**PERKIN-ELMER**
Computer Systems Division
Oceanport, New Jersey 07757, U.S.A.

## SOFTWARE CHANGE REQUEST

[illegible] No.

INSTRUCTIONS ON REVERSE SIDE OF LAST PART

| | | | | |
|---|---|---|---|---|
| Software Name | Revison/Update | Date Initiated | **Part No.** F M **Rev/Upd** | |
| ☐ Enhancement<br>☐ Possible Problem | Pri. | Operating System Rev/Upd | Processor | Memory KB | **Group Part No.** F M **Rev/Upd** |
| Company Name | | Cust. Ref. No. | **Date Logged In** | **Proj. Ldr.** | **Empl. No.** |
| Initiator | Phone Number | | **Init. Empl. No** | | **Cust. Acct. No.** |
| Address | | | Submitted by | Return to:<br>☐ Customer<br>☐ Analyst |
| City | State/Country | Zip Code | Office | Date |

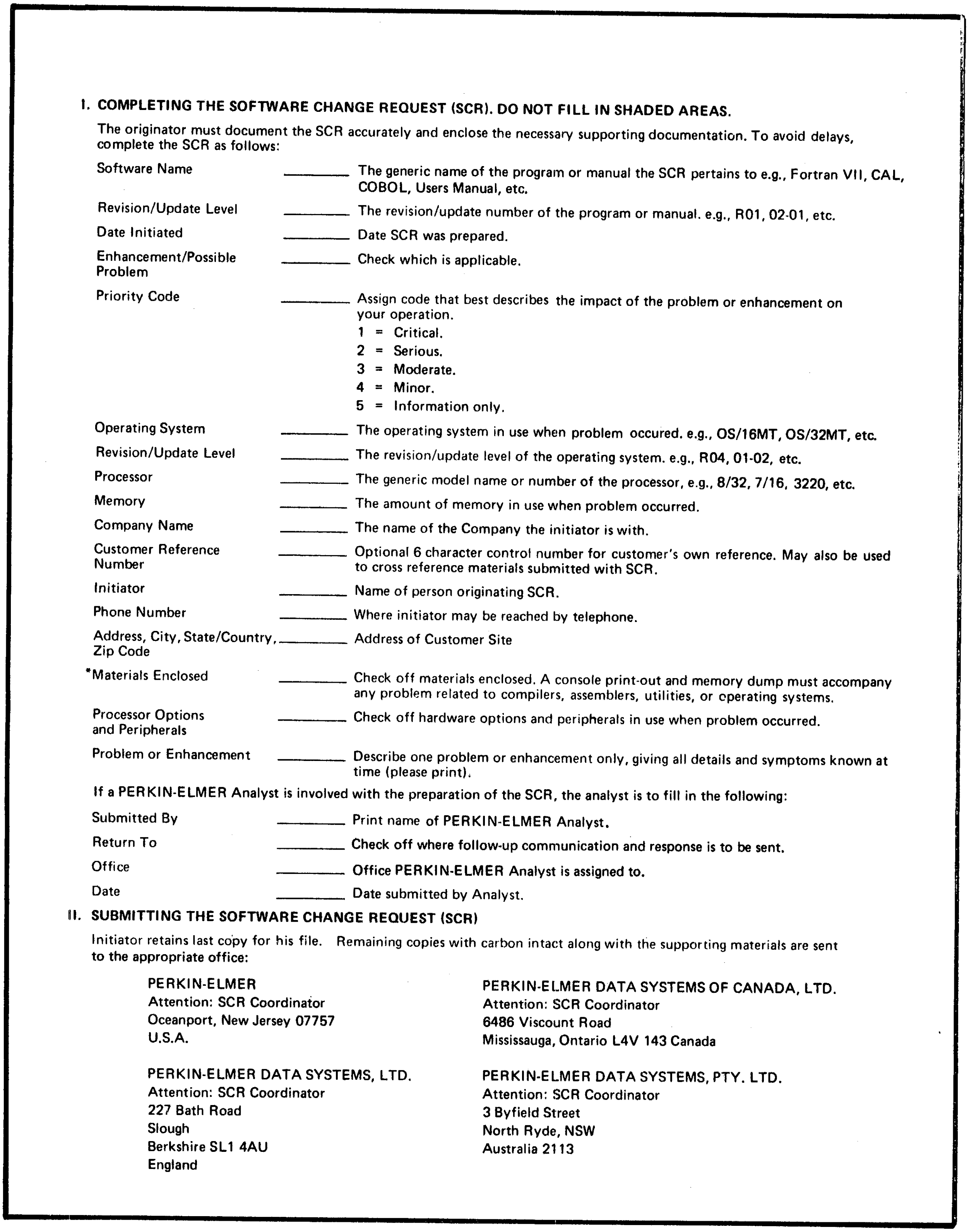
| MATERIAL ENCLOSED | | PROCESSOR OPTIONS AND PERIPHERALS | | | |
|---|---|---|---|---|---|
| ☐ Sysgen Info/CUP Listing | ☐ Listing Input/Output Data | ☐ Floating Point Hardware | ☐ HSPTR/P | ☐ TTY | ☐ 40 MB DISC |
| ☐ Modified Software Listings | ☐ User Program Listing | ☐ D/P Floating Point Hardware | ☐ Cassette | ☐ 800 BPI Tape } 9 TK | ☐ Other ______ |
| ☐ List of SCR's Applied | ☐ Object ______ | ☐ Multiply-Divide Hardware | ☐ Card Reader ______ | ☐ 1600 BPI Tape } 9 TK | ______ |
| ☐ Console Print-Out* | ☐ Source ______ | ☐ Dynamic Control Storage | ☐ Line Printer ______ | ☐ 800 BPI Tape 7 TK | ______ |
| ☐ Memory Dump* | ☐ User Documentation | ☐ Writable Control Storage | ☐ CRT-PASLA ______ | ☐ 2.5 MB Disc | ______ |
| ☐ Library Loader/TET Map | ☐ ______ | ☐ Loader Storage Unit | ☐ CRT-TTY Interface | ☐ 10 MB Disc | ______ |

**DESCRIBE ONE PROBLEM OR ENHANCEMENT ONLY**

**CUSTOMER COPY**

37-169

Front Page of SCR Form



## I. COMPLETING THE SOFTWARE CHANGE REQUEST (SCR). DO NOT FILL IN SHADED AREAS.

The originator must document the SCR accurately and enclose the necessary supporting documentation. To avoid delays, complete the SCR as follows:

| | |
|---|---|
| Software Name | The generic name of the program or manual the SCR pertains to e.g., Fortran VII, CAL, COBOL, Users Manual, etc. |
| Revision/Update Level | The revision/update number of the program or manual. e.g., R01, 02-01, etc. |
| Date Initiated | Date SCR was prepared. |
| Enhancement/Possible Problem | Check which is applicable. |
| Priority Code | Assign code that best describes the impact of the problem or enhancement on your operation.<br>1 = Critical.<br>2 = Serious.<br>3 = Moderate.<br>4 = Minor.<br>5 = Information only. |
| Operating System | The operating system in use when problem occured. e.g., OS/16MT, OS/32MT, etc. |
| Revision/Update Level | The revision/update level of the operating system. e.g., R04, 01-02, etc. |
| Processor | The generic model name or number of the processor, e.g., 8/32, 7/16, 3220, etc. |
| Memory | The amount of memory in use when problem occurred. |
| Company Name | The name of the Company the initiator is with. |
| Customer Reference Number | Optional 6 character control number for customer's own reference. May also be used to cross reference materials submitted with SCR. |
| Initiator | Name of person originating SCR. |
| Phone Number | Where initiator may be reached by telephone. |
| Address, City, State/Country, Zip Code | Address of Customer Site |
| *Materials Enclosed | Check off materials enclosed. A console print-out and memory dump must accompany any problem related to compilers, assemblers, utilities, or operating systems. |
| Processor Options and Peripherals | Check off hardware options and peripherals in use when problem occurred. |
| Problem or Enhancement | Describe one problem or enhancement only, giving all details and symptoms known at time (please print). |

If a PERKIN-ELMER Analyst is involved with the preparation of the SCR, the analyst is to fill in the following:

| | |
|---|---|
| Submitted By | Print name of PERKIN-ELMER Analyst. |
| Return To | Check off where follow-up communication and response is to be sent. |
| Office | Office PERKIN-ELMER Analyst is assigned to. |
| Date | Date submitted by Analyst. |

## II. SUBMITTING THE SOFTWARE CHANGE REQUEST (SCR)

Initiator retains last copy for his file. Remaining copies with carbon intact along with the supporting materials are sent to the appropriate office:

PERKIN-ELMER
Attention: SCR Coordinator
Oceanport, New Jersey 07757
U.S.A.

PERKIN-ELMER DATA SYSTEMS OF CANADA, LTD.
Attention: SCR Coordinator
6486 Viscount Road
Mississauga, Ontario L4V 143 Canada

PERKIN-ELMER DATA SYSTEMS, LTD.
Attention: SCR Coordinator
227 Bath Road
Slough
Berkshire SL1 4AU
England

PERKIN-ELMER DATA SYSTEMS, PTY. LTD.
Attention: SCR Coordinator
3 Byfield Street
North Ryde, NSW
Australia 2113

Back Page of SCR Form

# APPENDIX C
# PASCAL CSS's SUMMARIZED

The Pascal product contains the following CSS procedures:

- PASCAL.CSS to compile and link a Pascal program
- PASCOMP.CSS to compile a Pascal source program
- PASLINK.CSS to link a compiled Pascal program

The Pascal product also contains three similar CSS procedures for use from the OS/32 system console:

- PASCAL.CON to compile and link a Pascal program
- PASCOMP.CON to compile a Pascal source program
- PASLINK.CON to link a compiled Pascal program

The formats to invoke the CSS procedures are (where the asterisk indicates the OS/32 prompt):

**Format to use from a user terminal under OS/32 with MTM:**

```
*PASCAL sourcename,list,options,assemlist,memincr,worksize

*PASCOMP sourcename,list,options,assemlist,memincr

*PASLINK objectname,list,worksize
```

**Format to use from a system console under OS/32 MT:**

```
*PASCAL.CON sourcename,list,options,assemlist,memincr,worksize

*PASCOMP.CON sourcename,list,options,assemlist,memincr

*PASLINK.CON objectname,list,worksize
```

**Arguments:**

| | |
|---|---|
| sourcename | is the name of the file containing the Pascal source program to be compiled. An extension of .PAS is assumed; i.e., no extension should be given as part of the CSS argument; but the source file must exist with an extension of .PAS. |
| list | is the name of the file or device to which the source listing, cross reference, and linkage map are to be written. This argument is optional and defaults to PR:. |
| options | is a list of one or more compiler options separated by spaces. Refer to Section 1.5.2 of Chapter 1 for details on the compiler options. Options do not need to be specified to use the defaults, see Appendix E. |
| assemlist | is a file or device to which the assembly listing or map listing will be written if selected by appropriate options. This argument is optional and defaults to PR:. |
| memincr | is the memory segment size increment to be used to perform compilation or the additional memory space the compiler can use for stack and heap space to perform the compilation. This argument is optional and defaults to 64kb. |
| worksize | is a 1- to 6-digit hexadecimal number indicating the number of bytes of main storage to be added to the end of a task for its maximum workspace. This argument is optional and defaults to that value provided by Link. |
| objectname | is the file containing the object of the main compiled program to be linked. An extension of .OBJ is assumed; i.e., no extension should be supplied as part of the CSS argument, but the file must exist with an extension of .OBJ for PASLINK. |

Invoking the CSS procedures without specifying any arguments causes a series of messages to be logged which will help identify their necessary arguments.

The PASCAL and PASCOMP CSS's direct the object program to the file: sourcename.OBJ. The PASCAL CSS's direct the established task to the file: sourcename.TSK. The PASLINK CSS's direct the established task to the file: objectname.TSK.

# APPENDIX D
# Pascal COMPILER LOGICAL UNIT ASSIGNMENTS

| LU | USE | ASSIGNMENT | DATA FORMAT | LOGICAL RECORD LENGTH | MEDIA |
|---|---|---|---|---|---|
| 0 | Output: Log information | Always required | ASCII | 64 to 256 | device/ file |
| 1 | Input: User Pascal source program | Always required | ASCII | Up to 256 | device/ file |
| 2 | Output: List, cross, summary, batch statistics and program statistics | Always required for program statistics | ASCII | 132 | device/ file |
| 3 | Input/output: Temporary compiler scratch information | Always required | BINARY | 512 | file only |
| 4 | Input/output: Temporary compiler scratch information | Always required | BINARY | 512 | file only |
| 5 | Input/output: Temporary compiler scratch information | Controlled by compiler | ASCII | 256 | file only |
| 6 | Output: Assembly listing and Map listing | Optional Required for map and assembly | ASCII | 132 | device/ file |
| 7 | Output: Compiled object | Always required | BINARY | 126 | device/ file |
| 8 | Input: $INCLUDE file | Controlled by compiler | ASCII | Up to 256 | device/ file |

## APPENDIX D (Continued)
## Pascal COMPILER LOGICAL UNIT ASSIGNMENTS

The Pascal compiler can use any of the following logical units (lu):

- logical unit 0, log device or file
- logical unit 1, source input device or file
- logical unit 2, listing device or file
- logical unit 3, scratch file
- logical unit 4, scratch file
- logical unit 5, scratch file
- logical unit 6, map or assembly-listing device or file
- logical unit 7, object code output device or file
- logical unit 8, additional source input file for $INCLUDE

All necessary logical units, or those required for specified options, must be assigned to physical devices or files by the user. Logical unit 5, which is internally controlled by the compiler, and lu 8 which the compiler assigns from a user-specified $INCLUDE in-stream option are exceptions.

If not using the CSS procedures provided, the user must directly assign the logical units after loading the compiler and before executing it. Refer to Section 1.5.4 or Appendix C for CSS information.

The above is a listing of logical units, their use, when they are required, data formats, logical record length requirement, and allowable media.

# APPENDIX E
# PASCAL COMPILER OPTIONS

| OPTION-SPECIFIER | FUNCTION |
|---|---|
| ASSEMBLY | Assembly listing option prints out a listing on lu6, a disassembly of the compiled object program in an assembler-level format. |
| BATCH | Batch compilation option compiles batch or series of Pascal programs or modules from lu 1 until an end of file (EOF) or {$BEND} is encountered on the source file or device. |
| BEND | Batch end option signals end of batch to the compiler. Required only if BATCH is in effect and lu 1 is a non-random access device. |
| BOUNDSCHECK | Subrange bounds check option generates object code within the compiled program that will check for illegal out-of-bounds values assigned 'to variables with finite limits, such as variables of the subrange type with minimum and maximum limits defined. |
| CROSS | Cross reference option prints out a listing on lu 2 which is a cross reference of labels and identifiers used in the source program, relating place of definition and place of reference by source line number. |
| EJECT | Eject listing format control option causes a top-of-form (or page ejection) within the compiled-program listing output to lu 2. |
| HEAPMARK | The Heapmark option enables the compiler to recognize references to MARK and RELEASE in the user's source as predefined routines. |
| INCLUDE (fd,arg1, arg2) | Include additional source option includes, wherever {$INCLUDE (fd)} is specified in-stream, additional source from the file indicated by fd. Arg1 or arg2 may be LIST or NLIST, CROSS or NCROSS. |

| OPTION-SPECIFIER | FUNCTION |
|---|---|
| LIST | List option prints out the compiled program listing on lu 2. |
| LOG | Log option prints notifications (logs) on lu 0 notices of compiler operations, such as: current Pass number and the number of errors encountered, if any. |
| MAP | Map option prints out a listing or a map of the compiled program object, giving relative address displacements of statements and data. |
| MEMLIMIT=xx | Memory allocation option defines a percentage of taskspace for Pascal system workspace so the remainder of the task partition can be available, for example, for get-storage requests from external CAL written routines. |
| OPTIMIZE | Optimization option generates optimized object code so object program space and execution time are minimized. |
| RANGECHECK | Rangecheck option generates additional code within the compiled object program to check at run time for illegal out-of-range values used for subscripts, variant-tags, pointer values, and constant subrange parameters. |
| RELIANCE | The Reliance option causes the appropriate run-time error, task pausing, and task termination mechanisms to be generated in the compiler object code as is required in a Reliance environment, different from OS/32. |
| SUMMARY | Summary listing option prints out a listing on lu 6 of internal compiler statistics regarding table space, file sizes used, for this particular compilation-unit and lists register usage and the number and kind of any optimizations that were performed in the object code. |

The compile-time options default states, and abbreviated start or in-stream formats are listed below.

# APPENDIX E (Continued)
# PASCAL COMPILER OPTIONS

| COMPILER OPTION | DEFAULT | PLACEMENT | ABBREVIATED START OPTION | | ABBREVIATED IN-STREAM FORMAT | |
|---|---|---|---|---|---|---|
| | | | ON | OFF | ON | OFF |
| ASSEMBLY | OFF | START or in-stream | AS | NAS | {$AS} | {$NAS} |
| BATCH | OFF | START or in-stream | BA | --- | {$BA} | ----- |
| BEND | NULL | In-stream only | | | {$BEND} | ----- |
| BOUNDSCHECK | ON | START or in-stream | BO | NBO | {$BO} | {$NBO} |
| CROSS | ON | START or in-stream | CR | NCR | {$CR} | {$NCR} |
| EJECT | NULL | In-stream only | | | {$EJ} | ----- |
| HEAPMARK | OFF | START or in-stream | HE | NHE | {$HE} | {$NHE} |
| INCLUDE | NULL | In-stream only | | | {$IN(fd)} | ----- |
| LIST | ON | START or in-stream | LI | NLI | {$LI} | {$NLI} |
| LOG | OFF | START only | LO | NLO | | |
| MAP | OFF | START or in-stream | MA | NMA | {$MA} | {$NMA} |
| MEMLIMIT | 100% | START or in-stream | ME=xx | | {$ME=xx} | |
| OPTIMIZE | OFF | START only | OP | NOP | | |
| RANGECHECK | ON | START or in-stream | RA | NRA | {$RA} | {$NRA} |
| RELIANCE | OFF | START or in-stream | RE | NRE | {$RE} | {$NRE} |
| SUMMARY | OFF | START or in-stream | SU | NSU | {$SU} | {$NSU} |

An example of using the abbreviated option-specifiers as Pascal compiler start options in the OS/32 START Command is:

```
ST ,BA AS BO CR LI LO MA OP RA SU
```

An example of using the abbreviated option-specifiers as "options" in the invocation of the PASCAL.CSS to compile and link is:

```
PASCAL.CSS sourcename,list,BA LI CR OP NRA,,100,100
```

Specifying compiler options in the Pascal source is accomplished by coding option-specifier comments of the form:

Option-specifier comment

```
    |----> { ---> option-string ---> } ---->|
--->|                                       |--->
    |---> (* ---> option-string ---> *) --->|
```

where option-string is of the form:

Option-string

```
-------> $ ---> option-specifier ------->
   ^                               |
   |                               |
   |<-----------  ,  <-------------v
```

An option-specifier comment is a Pascal comment. The first nonblank character that follows the beginning comment delimiter is a dollar sign ($). Any comment so distinguished must only contain an option string; no intermixing of comment text and $option-specifiers are allowed.

An example of an option-string is:

```
$BATCH,$ASSEMBLY,$MAP,$LIST,$CROSS,$SUMMARY,$NRANGECHECK
```

Enclosed within the comment delimiters, the option-specifier comment would be:

```
{$BATCH,$ASSEMBLY,$MAP,$LIST,$CROSS,$SUMMARY,$NRANGECHECK}
```

Other examples of option-specifier comments are:

```
{$INCLUDE (VOLN:FILENAME.EXT,NLIST,NCROSS)}
{$EJECT}
(* $NBOUNDSCHECK *)
{$BEND}
```

# APPENDIX F
# PASCAL COMPILER END-OF-TASK CONDITIONS (EOT CODES)

| EOT CODE | MEANING |
|---|---|
| 0 | Normal termination. No compilation errors detected in either a single compilation or all of the compilations within a batch stream. |
| 1 | Illegal start options |
| 2 | Error detected in Passes 1 through 5 (syntax) |
| 3 | Error detected in Passes 6 through 9 (semantics) |
| 4 | Any error in processing one or more compilation units within a batch stream. |
| 5 | Error in locating or reading a {$INCLUDE fd} source file, fd. Abort. |

**NOTE**

Return codes 2 and 3 are applicable only when processing a single compilation unit; i.e., nonbatch operation. Return code 4 is applicable only when operating in batch mode. The appropriate return codes 0, 2, 3, or 5 for each individual compilation unit are found in the Batch Statistics Listing.

# APPENDIX G
# PASCAL ERROR MESSAGES

## COMPILE-TIME DIAGNOSTIC ERRORS DISPLAYED IN LISTINGS

The format of a compile-time diagnostic error message is:

Format:

```
****** LINE n, ERROR xyyy: message . . . . .
```

Parameters:

| | |
|---|---|
| n | is the offending source line number, |
| x | is the pass number that detected the error, |
| yyy | is the error code, and |
| message | is the error text. |

The possible error codes xyyy and messages are listed below:

| xyyy | message |
|---|---|
| 1001 | EOF FOUND IN COMMENT |
| 1002 | BAD NUMBER FORMAT: DIGIT REQUIRED |
| 1003 | NUMBER TOO LARGE |
| 1004 | EXPONENT MAGNITUDE TOO LARGE |
| 1005 | SYMBOL TABLE OVERFLOW |
| 1006 | INVALID CONTROL CHARACTER |
| 1007 | STRING OF LENGTH ZERO |
| 1008 | STRING TOO LONG |
| 1009 | BAD CHARACTER |
| 1010 | ILLEGAL IN-LINE OPTION |
| 2002 | DECLARATION SYNTAX |
| 2003 | CONSTANT DECLARATION SYNTAX |
| 2004 | TYPE DECLARATION SYNTAX |
| 2005 | TYPE SYNTAX |
| 2006 | ENUMERATION TYPE SYNTAX |
| 2007 | SUBRANGE TYPE SYNTAX |
| 2008 | SET TYPE SYNTAX |
| 2009 | ARRAY TYPE SYNTAX |

## COMPILE-TIME DIAGNOSTIC ERRORS DISPLAYED IN LISTINGS

| | |
|---|---|
| 2010 | RECORD TYPE SYNTAX |
| 2011 | 'PACKED' NOT ALLOWED HERE |
| 2012 | VAR DECLARATION SYNTAX |
| 2013 | ROUTINE SYNTAX |
| 2014 | PROCEDURE SYNTAX |
| 2015 | FUNCTION SYNTAX |
| 2016 | WITH STATEMENT SYNTAX |
| 2017 | PARAMETER SYNTAX |
| 2018 | BODY SYNTAX |
| 2019 | STATEMENT LIST SYNTAX |
| 2020 | STATEMENT SYNTAX |
| 2021 | ASSIGNMENT OR PROCEDURE CALL SYNTAX |
| 2022 | ARGUMENT LIST SYNTAX |
| 2023 | COMPOUND STATEMENT SYNTAX |
| 2024 | IF STATEMENT SYNTAX |
| 2025 | CASE STATEMENT SYNTAX |
| 2027 | WHILE STATEMENT SYNTAX |
| 2028 | REPEAT STATEMENT SYNTAX |
| 2029 | FOR STATEMENT SYNTAX |
| 2031 | EXPRESSION SYNTAX |
| 2032 | VARIABLE SYNTAX |
| 2033 | CONSTANT SYNTAX |
| 2035 | IMPROPERLY TERMINATED PROGRAM |
| 2037 | POINTER SYNTAX |
| 2038 | PROGRAM OR MODULE HEADER SYNTAX |
| 2040 | SYNTAX OF PREFIX ROUTINES |
| 2041 | PREFIX SYNTAX |
| 2042 | LABEL DECLARATION SYNTAX |
| 2043 | STATEMENT LABEL SYNTAX |
| 2044 | GOTO STATEMENT SYNTAX |
| 2045 | TOO MANY EXTERNAL FILES (LIMIT = 32) |
| 2046 | "INPUT" NOT SPECIFIED |
| 2047 | "OUTPUT" NOT SPECIFIED |

| | |
|---|---|
| 3002 | IDENTIFIER DECLARED TWICE |
| 3003 | TOO DEEPLY NESTED, FURTHER COMPILATION ABORTED |
| 3004 | INTERNAL OPERAND STACK OVERFLOW, FURTHER COMPILATION ABORTED |
| 3005 | SYMBOL TABLE OVERFLOW, FURTHER COMPILATION ABORTED |
| 3006 | UPDATE TABLE OVERFLOW, FURTHER COMPILATION ABORTED |
| 3007 | INVALID CONSTANT |
| 3009 | INVALID SUBRANGE BOUND |
| 3010 | SUBRANGE HIGH BOUND < LOW BOUND |
| 3011 | SUBRANGE TYPES INCOMPATIBLE |
| 3012 | MISSING ARGUMENT |
| 3013 | NOT A ROUTINE |
| 3014 | TOO MANY ARGUMENTS |
| 3015 | CASE LABEL VALUE OUT OF RANGE (0..127) |
| 3016 | ILLEGAL CASE LABEL TYPE |
| 3017 | DUPLICATE CASE LABEL |

## COMPILE-TIME DIAGNOSTIC ERRORS DISPLAYED IN LISTINGS

| | |
|---|---|
| 3018 | WITH VARIABLE NOT RECORD |
| 3020 | ROUTINE IS NOT A FUNCTION |
| 3021 | UNDECLARED OR INACCESSIBLE IDENTIFIER |
| 3022 | IDENTIFIER IS NOT OF PROPER CLASS |
| 3024 | NOT A RECORD; CANNOT SELECT FIELD |
| 3025 | SUBSCRIPTED VARIABLE NOT AN ARRAY |
| 3026 | CALL TO NON-PROCEDURE |
| 3027 | UNRESOLVED FORWARD TYPE REFERENCE |
| 3028 | REFERENCE MUST BE A POINTER VARIABLE |
| 3029 | UNRESOLVED 'FORWARD' DECLARATION |
| 3030 | VARIANTS TOO DEEPLY NESTED |
| 3031 | DUPLICATE VARIANT LABEL |
| 3032 | ILLEGAL VARIANT LABEL TYPE |
| 3033 | VARIANT LABEL OUT OF RANGE (0..31) |
| 3035 | NO TYPE ALLOWED ON 'FORWARD' FUNCTION |
| 3036 | NO FUNCTION TYPE SPECIFIED |
| 3037 | ARGUMENT LIST CANNOT BE REPEATED |
| 3038 | DUPLICATE LABEL DECLARATION |
| 3039 | UNRESOLVED LABEL DECLARATION |
| 3040 | UNDECLARED OR INACCESSIBLE LABEL |
| 3041 | DUPLICATE LABEL DEFINITION |
| 3042 | ARGUMENT OF MARK/RELEASE OF WRONG TYPE |

| | |
|---|---|
| 4002 | TOO MUCH DATA SPACE REQUIRED |
| 4007 | INVALID FUNCTION TYPE |
| 4009 | ENUMERATION CANNOT BE DEFINED IN RECORD |
| 4010 | TOO MANY VALUES FOR ENUMERATION (LIMIT = 128) |
| 4011 | INVALID INDEX TYPE |
| 4012 | INVALID SET MEMBER TYPE |
| 4021 | INTERNAL OPERAND STACK OVERFLOW, FURTHER COMPILATION ABORTED |
| 4022 | NESTING TOO DEEP |
| 4024 | ILLEGAL TAGFIELD TYPE |
| 4025 | ROUTINE FORWARD DECLARED TWICE |
| 4028 | STRUCTURES MAY NOT CONTAIN FILES |
| 4029 | IMPROPER ROUTINE ARGUMENT |
| 4030 | EXTERNAL ROUTINE CANNOT HAVE FORMAL ROUTINE PARAMETERS |
| 4031 | PACKED STRUCTURE'S COMPONENT NOT PASSABLE TO VAR PARAMETER |

| | |
|---|---|
| 5001 | OPERAND TYPE CONFLICT |
| 5002 | VARIABLE REQUIRED |
| 5003 | ILLEGAL ASSIGNMENT |
| 5005 | ILLEGAL 'FOR' VARIABLE |
| 5007 | MISSING/INVALID ARGUMENT IN I/O ROUTINE CALL |
| 5008 | IMPROPER USE OF FILE VARIABLE |
| 5009 | CONTROL VARIABLE USED IN OUTER 'FOR' |
| 5010 | CONTROL VARIABLE MUST BE LOCAL |
| 5011 | CONTROL VARIABLE CANNOT BE VAR ARGUMENT |
| 5012 | ASSIGNMENT TO CONTROL VARIABLE ILLEGAL |
| 5013 | FILE NOT DECLARED IN 'VAR' SECTION |
| 5014 | INDEX EXPRESSION IS OF WRONG TYPE |

## COMPILE-TIME DIAGNOSTIC ERRORS DISPLAYED IN LISTINGS

| | |
|---|---|
| 6000 | DIVISION BY 0 |
| 6001 | MOD RELATIVE TO 0 OR NEGATIVE NUMBER |
| 6002 | INTERNAL OPERAND STACK OVERFLOW |
| 6003 | CONSTANT OUT OF RANGE |
| 6004 | EXPRESSION VALUE TOO LARGE |
| 7001 | INTERNAL OPERAND STACK OVERFLOW |
| 7002 | THIS FEATURE UNIMPLEMENTED |
| 7003 | INTERNAL COMPILER ERROR |
| 7004 | 'D' FORMAT ILLEGAL HERE |

Receipt of any extraneous compile-time diagnostic error messages, not of the above documented numbers, xyyy, such as:

```
****** LINE n, ERROR xyyy: ERROR UNKNOWN
```

may be reported on an SCR form as detailed in Appendix B.

## COMPILER OPERATIONS MESSAGES

The following compiler operations messages are listed to logical unit 0, the compiler's log device/file.

PERKIN-ELMER PASCAL Rnn-uu

> The Pascal Compiler logs this message to identify itself and notify the user that compilation has started; where Rnn identifies the Pascal compiler revision level and uu identifies the update level.

INVALID OPTION(S)

> Compiler was started with invalid Pascal compiler start-option(s), and compilation cannot begin. The compiler aborts with END-OF-TASK CODE 1. Correct the options given to the compiler prior to restarting.

PASS n

> When the LOG compiler start-option has been selected, this message is listed to lu 0 for each pass of compiler operations beginning. where n is from 1 to 10. This message does not appear, if the LOG option was not specified.

nn ERRORS DETECTED IN PASS n

> When the LOG compiler start-option has been selected, this message is additionally listed when a number (nn) of errors are detected in Pass n, where n is the pass number from 1 to 10. This message does not appear, if the LOG option was not specified; or pass n detects no errors.

UNABLE TO OPEN FILE filename.ext
EITHER FILE DOES NOT EXIST OR IS INACCESSIBLE

> Compiler is attempting to perform a user-specified in-stream $INCLUDE (filename.ext) option, but cannot assign the file; and aborts the compile with an END-OF-TASK CODE 5. Check the $INCLUDE specification, or why the intended filename.ext appears not to exist or is inaccessible.

## COMPILER OPERATIONS MESSAGES

### INCLUDED FILE ATTEMPTED FROM NON RANDOM I/O DEVICE

A user-specified $INCLUDE in-stream option specified an argument file descriptor which is a non-random I/O device.

### COMPILATION ERRORS

Compile-time diagnostic errors were detected in the user Pascal program or module source code just compiled, the diagnostic error messages are displayed in the listing on the lu 2 list device/file, and the compiler terminates a single compilation-unit with an END OF TASK CODE= 2 or 3.

The diagnostic error messages are displayed in the compiled-program listing on the lu 2 list device/file, if LIST is on, and/or listed in a group in the program statistics at the end of the listing. If LIST is off, the group of diagnostic errors message are still available on lu 2.

In a batch job with COMPILATION ERRORS, the end-of-task code 2 or 3 is listed in the batch statistics listing for each compilation-unit containing errors, and the entire batch terminates with an END OF TASK CODE= 4.

Check the listings; correct the source; and recompile.

### xxxxxxxx-END OF TASK CODE= n

where xxxxxxxx is the identifier name of the system user/compiler-task ending, and n is the END OF TASK CODE. This system message occurs under OS/32 as the compiler terminates with an SVC 3, End of Task; giving an EOT code of n = 0,1,2,3,4, or 5 as detailed in Appendix F. An EOT of zero indicates a correct compile, a non-zero EOT indicates a problem.

The following message may occur when difficulty in performing I/O arises for the compiler.

## COMPILER OPERATIONS MESSAGES

I/O ERROR xxyy, LU= nnn

where xxyy is the non-zero hexidecimal OS/32 SVC 1 Error Status encountered on logical unit number, nnn. Examine the xxyy status, and lu 0 to 8 to determine the cause of I/O trouble, e.g., if lu = 1,5, or 8 the compiler cannot read source input. In an OS/32 environment, if operator intervention can correct the problem and continues with the OS CONTINUE command, the SVC 1 will be retried. Users not familiar with the xxyy SVC 1 Status halfword definitions may refer to the appropriate operating system manual on SVC 1.

As the compiler is executing as Pascal compiled-code itself, it too may encounter certain run time errors, described in this Appendix below under RUN-TIME ERRORS. If an unrecoverable error occurs in its own code the compiler generates the message:

PASS n LINE xxxxx, ADDR yyyyyy message....
COMPILING LINE zzzzz OF PROGRAM name....

The compiler issues this message prior to pausing, when on pass n, at its own line number xxxxx, and object address yyyyyy, is encountering a run time error; while compiling user source at its source line number zzzzz of user program-name or module-name "name....". If the message is STACK OVERFLOW or HEAP OVERFLOW, not enough memory has been allocated for the compiler to compile this program; and the user should reload with greater memory.

If the message is, during compile time:

INDEX RANGE ERROR
PARAM RANGE ERROR
VALUE RANGE ERROR
CASE LABEL ERROR
TRUNC RANGE ERROR
VARIANT TAG ERROR
POINTER ERROR

the compiler may be malfunctioning. Please report compiler malfunctions via an SCR as instructed in Appendix B.

## RUN-TIME ERRORS

Run time errors occuring during execution of user Pascal tasks are logged (via an SVC 2,log-message) to the console (user console in an MTM environment, system console in a stand alone OS/32 MT environment, or system journal in a RELIANCE environment); and are described below.

Some errors may allow continuing execution, after pausing for correction by operator intervention, others may require reloading with more or differently arranged memory allocations, relinking the task, or reprogramming and recompile, to correct the problem.

### NOT ENOUGH SPACE TO RUN PASCAL

This message occurs immediately after starting a user Pascal task, when the memory allocations available to the task are not even large enough for the basic internal workspace needed for the Pascal SDA, FORTRAN SCA, or the RTL Scratch Pad area. The user task is then terminated.

Reload or relink with more memory, and restart; or if MEMLIMIT was used, check the effect of the MEMLIMIT memory allocation option. If upon restarting, this message does not appear and the STACK OVERFLOW immediately does, or it occurs sometime thereafter, enough memory was added/arranged to accomodate the basic internal tables, but not enough for this particular user program's Global variables or stack data to be run. Reload or relink with greater memory, and restart.

When executing Pascal named file I/O (text file or non-text file), with RESET, REWRITE, READ, READLN, WRITE, WRITELN statements; the following runtime error messages may occur. Note that when the logical unit number, nnn, is an external Pascal named file; the position of the file-name in the PROGRAM header file-name-list determined its associated lu number. If the lu number, nnn, cannot possibly be an external file in the program concerned, an internal file-variable is of concern.

## RUN-TIME ERRORS

### NO LU AVAILABLE TO ASSIGN INTERNAL FILE

The attempt was made to allocate and assign a temporary file for an internal Pascal file-variable, but no logical unit number was available. Suggested responses follow. Relink the program to allow a larger number of logical units. Use fewer internal files, or rearrange the logic of the program to have fewer internal files in action at one time. Examine critically any recursive routines that contain internal files, as for each recursive activation another temporary file is assigned to an lu for each internal file-variable in the routine.

### READ ATTEMPTED ON A NON-RESET FILE, LU= nnn

A READ or READLN statement is causing an attempt to read from a file associated with logical unit, nnn, and the file-variable has not been properly RESET (i.e., not prepared to be in a read-only state). After this message the program is paused, and upon an attempt to continue with the OS CONTINUE command, goes to end-of-task. Check the program logic in the source, correct and recompile.

### READ ATTEMPTED PAST END-OF-FILE, LU= nnn

A READ or READLN statement is causing an attempt to read beyond an end-of-file condition, on the file-variable associated with lu nnn. After this message the program is paused, and upon an attempt to continue with the OS CONTINUE command, goes to end-of-task. Check the program logic in the source, correct and recompile. Prior to any READ or READLN attempt, the EOF condition on text file INPUT, or any user-specified Pascal filenames may be queried with the function EOF or EOF(file-variable-name), respectively.

### WRITE ATTEMPTED TO A NON-REWRITTEN FILE, LU= nnn

A WRITE or WRITELN statement is causing the attempt to write to a file-variable associated with logical unit nnn, and the file-variable has not been properly given a REWRITE; (i.e., the file has not been prepared to be in a write-only state). After this message the program is paused, and upon an attempt to continue with the OS CONTINUE command, goes to end-of-task. Check the program logic in the source, correct and recompile.

## RUN-TIME ERRORS

INVALID CHARACTER IN NUMERIC INPUT, LU= nnn

A READ or READLN statement is attempting to read an integer or real number as input data from a text file; and encountering an invalid character that does not form a required part of an integer or real number, mindful that leading blanks and end-of-lines are skipped prior to accessing the numbers for input.

If the device associated with lu nnn is an interactive terminal, and the program is continued with the OS CONTINUE command, the value zero will be returned as the number being read. If the device is not an interactive terminal, the program is paused following this message, and goes to end-of-task upon an attempt to CONTINUE.

Check either the intended type of arguments of the READ/READLN statements or the input data stream.

Additional system file error conditions may be detected while performing Pascal named file I/O, as follows.

I/O ERROR xxyy, LU= nnn

where xxyy is the non-zero hexidecimal OS/32 SVC 1 Error Status encountered on logical unit number, nnn, by an SVC 1 I/O being attempted.

After this message occurs the program is paused. Check the OS/32 SVC 1 status halfword as defined in the Operating System manual, and the file/device assigned to lu nnn; to determine the source of trouble.

In an OS/32 environment, the task is paused to allow operator intervention to correct the problem, and enter the OS/32 CONTINUE command to retry the SVC 1, and proceed.

## RUN-TIME ERRORS

ERROR IN INITIALIZING EXTERNAL FILE FOR READ/WRITE
SVC 7 ERROR, LU= nnn; FN= xxxxxxxxxxx, STATUS= yyyyyyyyyyyyyyyy

In performing a RESET or REWRITE, either automatically to the textfiles INPUT or OUTPUT or to user specified file-variables, the attempt was made to fetch-attributes on the file assigned to logical unit, nnn. If the attempt failed, this message occurs and the program is paused.

In an OS/32 environment, an operator entered CONTINUE command after the pause, causes the SVC 7 to be retried. Depending on the kind of SVC 7 ERROR, detailed below, correct the situation and continue or restart.

ERROR IN ASSIGNING INTERNAL FILE
SVC 7 ERROR, LU= nnn; FN= xxxxxxxxxxx, STATUS= yyyyyyyyyyyyyyyy

The attempt was made to allocate a temporary file on disc and assign it to the highest available logical unit, for an internal Pascal file-variable. If the attempt failed, this message occurs and the program is paused.

In an OS/32 environment, an operator entered CONTINUE command after the pause, causes the SVC 7 to be retried. Depending on the kind of SVC 7 ERROR, detailed below, correct the situation and continue or restart.

ERROR IN ATTEMPTING TC CLOSE INTERNAL FILE
SVC 7 ERROR, LU= nnn; FN= xxxxxxxxxxx, STATUS= yyyyyyyyyyyyyyyy

At exit from a block (program, module, procedure, or function) which contains an internal file-variable, the attempt is made to close the logical unit corresponding to that file.

If the attempt fails, then this message occurs and the program continues; i.e., the program is not paused.

## RUN-TIME ERRORS

The qualifier SVC 7 ERROR message of the above three messages identifies the logical unit number concerned, the function code attempted, and the error status encountered; and is of the form:

SVC 7 ERROR, LU= nnn; FN= xxxxxxxxxxx, STATUS= yyyyyyyyyyyyyyyy

where nnn is the logical unit upon which the SVC 7 was attempted, and the FN= xxxxxxxxxxx, is the function code attempted:

FN= ASSIGN
FN= CLOSE
FN= ALLOCATE
FN= RENAME
FN= REPROTECT
FN= DELETE
FN= CHANGE PRIV
FN= CHECK POINT
FN= FETCH ATTRB

and the STATUS= status-message encountered as an error may be:

STATUS= ILLEGAL FUNCTION
STATUS= ILLEGAL LU
STATUS= VOLUME
STATUS= NAME ERROR
STATUS= SIZE ERROR
STATUS= PROTECT ERROR
STATUS= PRIVILEGE ERROR
STATUS= BUFFER ERROR
STATUS= LU NOT ASSIGNED
STATUS= TYPE (DEVICE)
STATUS= FD SYNTAX ERROR
STATUS= SVC 6 DEVICE
STATUS= FILE IS /S OR /G
STATUS= I/O ERROR

## RUN-TIME ERRORS

Executing Pascal compiled-code enacts certain self-contained program logic and run-time data validation checks to detect exceptional circumstances that make it illogical or impossible for the program to continue executing. Subsequent to these run time error messages, the task is paused, and upon an attempt to continue with the OS CONTINUE command, the user task is terminated with END-OF-TASK, under OS/32. These run-time error messages may occur while executing Pascal compiled-code as follows, and is of the form:

Format:

```
LINE xxxxx, ADDR yyyyyy message...
```

Parameters:

| | |
|---|---|
| xxxxx | is indicating (when possible) the user's errant Pascal source line number in which the error was detectable, by the line's Pascal compiled-code; or xxxxx is zero, when the error was detectable by an RTL/support routine not having access to the user's line number. |
| yyyyyy | is the machine address in the compiled object code, of either the interupting ERR compiler generated instruction, near the detected error; or if line xxxxx is zero, yyyyy is the machine address in code which called the error detecting RTL routine; and |
| "message" | is one of the following:<br>INDEX RANGE ERROR<br>PARAM RANGE ERROR<br>VALUE RANGE ERROR<br>CASE LABEL ERROR<br>TRUNC RANGE ERROR<br>VARIANT TAG ERROR<br>POINTER ERROR<br>STACK OVERFLOW<br>HEAP OVERFLOW |

Each of these "messages" is detailed below.

## RUN-TIME ERRORS

### INDEX RANGE ERROR

Occurs when, in referencing an ARRAY component, the value of the index-expression is beyond the range of values defined by the index-type of the ARRAY.

Check either the index-expression used in the array-component-selector near line xxxxx, or once the array concerned is identifiable, check the array-type definition to investigate why the run-time value of the specified expression is not acceptable to the intended index-type of the array-type.

This index-range checking occurs only in code compiled under the compiler option RANGECHECK [by default or specification (RA)]; and is not checked for in code compiled under NRANGECHECK (NRA).

### PARAM RANGE ERROR

Occurs when executing a routine-invocation, which, in passing an expression to a value-parameter, contains a run-time value of the expression outside the subrange limits acceptable to the value-parameter.

Check either the expression's value in the routine invocation near line xxxxx, or once the routine in question is identifiable check in the routine-definition the intended value-parameter type to investigate why a run-time value of the coded expression is not acceptable to the intended value-parameter type; and/or reprogram for this condition.

This parameter-range checking occurs only in code compiled under the compiler option RANGECHECK [by default or specification (RA)]; and is not checked for in code compiled under NRANGECHECK (NRA).

## RUN-TIME ERRORS

### VALUE RANGE ERROR

Occurs when executing an assignment statement which is attempting to assign an expression value to a variable of subrange-type outside the range of values defined for the subrange-type. This message can also occur for other value ranges checked for under RANGECHECK.

Check the assignment statement near line xxxxx to determine how the expression is producing a value unacceptable to the subrange or other limits defined for the variable by its typing.

This value-range checking for subrange-types occurs only in code compiled under the compiler option BOUNDSCHECK (by default or specification); and is not checked for in code compiled under NBOUNDSCHECK (NBO). Other value-range checks are performed under the RANGECHECK option.

### CASE LABEL ERROR

Occurs while executing a CASE statement and the tag-field identifier (case-selector) contains a value that doesn't match a case-constant on one of the case-labeled-statements and there is no OTHERWISE clause [a Perkin-Elmer Pascal extension] to handle non-matching values.

Check the CASE statement near line xxxxx, to determine if and what value might be causing this error, what value has not been programmed for, or if the OTHERWISE clause can be used to handle it.

The detection of this error and its message cannot be turned off by a compiler option.

### TRUNC RANGE ERROR

Occurs when the integer part of a real number has a magnitude too large to fit in an integer. This error can occur if the functions TRUNC or ROUND are called and the integer part of their real-valued expression argument cannot fit into an intended integer. For example, INTEGERVARIABLE := TRUNC(REALNUMBER); and REALNUMBER > MAXINT.

The detection of this error and its error message cannot be turned off by a compiler option.

## RUN-TIME ERRORS

### VARIANT TAG ERROR

Occurs when, in specifying or accessing a field of a record, a record-field-selector is referring to a variant part of a record which is not consistent with the current value of the tag field or the value of the tag-field is not consistent with the variant referenced.

Check the source code near line xxxxx for the inconsistency.

This variant-tag-error checking occurs only in code compiled under the compiler option RANGECHECK [by default or specification (RA)]; and is not checked for in code compiled under NRANGECHECK (NRA).

### POINTER ERROR

Occurs when attempting to dereference a non-existing target through a pointer-variable, e.g., the pointer-variable contains the value NIL, or is otherwise undefined, i.e., the target has not yet been created with NEW, or has been DISPOSEd of already.

Check the program logic concerning dynamic variables, that pointers have been initialized correctly, and tests are made for NIL values prior to dereferencing targets.

This pointer-error checking occurs only in code compiled under the compiler option RANGECHECK [by default or specification (RA)]; and is not checked for in code compiled under NRANGECHECK (NRA).

## RUN-TIME ERRORS

### STACK OVERFLOW

Occurs while executing code from source line number xxxxx, and the program required for its routine activations and their data, more memory than is currently available. Both the stack and the heap share the task workspace, as modified by the MEMLIMIT compiler option.

If the line number is the main body source statement, and this message occurs immediately after starting the user task, not enough memory for Global Variables and main program temporaries is available. In the user task, re-allocate a larger memory segment size increment for workspace, re-load, and restart.

Otherwise occuring during program execution, not enough memory is available to accomodate the expanding stack, in the common task workspace used for both the heap and the stack, for routine-activations of the routine near line xxxxx. If stack overflow occurs within an RTL routine, such as the SVC support routines, the line number xxxxx may be zero.

The detection of this error and its message cannot be turned off by a compiler option.

### HEAP OVERFLOW

Occurs while executing, in order to store dynamic variables on the Heap, the Heap is about to collide with the stack and there is not enough memory to add another dynamic variable to the heap. The stack and the heap share the task workspace, as modified by the MEMLIMIT compiler option.

Check the possibilities of either incrementing the amount of memory, reprogramming dynamic structures with DISPOSE, or identifying why the heap is running out of space; e.g., whether repeated calls on NEW are correctly creating the intended dynamic variables, such as when NEW calls are embeded in recursive routines.

The detection of this error and its message cannot be turned off by a compiler option. As this error is detected by either the RTL routines, NEW or MARK, and not directly in Pascal compiled-code, no user line number is diplayed in the message, but rather the value zero.

# APPENDIX H
# THE ASCII CHARACTER SET

| ORDINAL NUMBER | CHARACTER | ASCII |
|---|---|---|
| 0 | nul | 00 |
| 1 | soh | 01 |
| 2 | stx | 02 |
| 3 | etx | 03 |
| 4 | eot | 04 |
| 5 | enq | 05 |
| 6 | ack | 06 |
| 7 | bel | 07 |
| 8 | bs | 08 |
| 9 | ht | 09 |
| 10 | lf | 0A |
| 11 | vt | 0B |
| 12 | ff | 0C |
| 13 | cr | 0D |
| 14 | so | 0E |
| 15 | si | 0F |
| 16 | dle | 10 |
| 17 | dc1 | 11 |
| 18 | dc2 | 12 |
| 19 | dc3 | 13 |
| 20 | dc4 | 14 |
| 21 | nak | 15 |
| 22 | syn | 16 |
| 23 | etb | 17 |
| 24 | can | 18 |
| 25 | em | 19 |
| 26 | sub | 1A |
| 27 | esc | 1B |
| 28 | fs | 1C |
| 29 | gs | 1D |
| 30 | rs | 1E |
| 31 | us | 1F |
| 32 |  | 20 |
| 33 | ! | 21 |
| 34 | " | 22 |
| 35 | # | 23 |
| 36 | $ | 24 |
| 37 | % | 25 |
| 38 | & | 26 |
| 39 | ' | 27 |
| 40 | ( | 28 |

| ORDINAL NUMBER | CHARACTER | ASCII |
|---|---|---|
| 64 | @ | 40 |
| 65 | A | 41 |
| 66 | B | 42 |
| 67 | C | 43 |
| 68 | D | 44 |
| 69 | E | 45 |
| 70 | F | 46 |
| 71 | G | 47 |
| 72 | H | 48 |
| 73 | I | 49 |
| 74 | J | 4A |
| 75 | K | 4B |
| 76 | L | 4C |
| 77 | M | 4D |
| 78 | N | 4E |
| 79 | O | 4F |
| 80 | P | 50 |
| 81 | Q | 51 |
| 82 | R | 52 |
| 83 | S | 53 |
| 84 | T | 54 |
| 85 | U | 55 |
| 86 | V | 56 |
| 87 | W | 57 |
| 88 | X | 58 |
| 89 | Y | 59 |
| 90 | Z | 5A |
| 91 | [ | 5B |
| 92 | \ | 5C |
| 93 | ] | 5D |
| 94 | ^ | 5E |
| 95 | _ | 5F |
| 96 | ` | 60 |
| 97 | a | 61 |
| 98 | b | 62 |
| 99 | c | 63 |
| 100 | d | 64 |
| 101 | e | 65 |
| 102 | f | 66 |
| 103 | g | 67 |
| 104 | h | 68 |

# APPENDIX H (Continued)
# THE ASCII CHARACTER SET

| ORDINAL NUMBER | CHARACTER | ASCII |
|---|---|---|
| 41 | ) | 29 |
| 42 | * | 2A |
| 43 | + | 2B |
| 44 | , | 2C |
| 45 | - | 2D |
| 46 | . | 2E |
| 47 | / | 2F |
| 48 | 0 | 30 |
| 49 | 1 | 31 |
| 50 | 2 | 32 |
| 51 | 3 | 33 |
| 52 | 4 | 34 |
| 53 | 5 | 35 |
| 54 | 6 | 36 |
| 55 | 7 | 37 |
| 56 | 8 | 38 |
| 57 | 9 | 39 |
| 58 | : | 3A |
| 59 | ; | 3B |
| 60 | < | 3C |
| 61 | = | 3D |
| 62 | > | 3E |
| 63 | ? | 3F |

| ORDINAL NUMBER | CHARACTER | ASCII |
|---|---|---|
| 105 | i | 69 |
| 106 | j | 6A |
| 107 | k | 6B |
| 108 | l | 6C |
| 109 | m | 6D |
| 110 | n | 6E |
| 111 | o | 6F |
| 112 | p | 70 |
| 113 | q | 71 |
| 114 | r | 72 |
| 115 | s | 73 |
| 116 | t | 74 |
| 117 | u | 75 |
| 118 | v | 76 |
| 119 | w | 77 |
| 120 | x | 78 |
| 121 | y | 79 |
| 122 | z | 7A |
| 123 | { | 7B |
| 124 | \| | 7C |
| 125 | } | 7D |
| 126 | ~ | 7E |
| 127 | del | 7F |

# APPENDIX I
# RESERVED WORDS AND PREDEFINED IDENTIFIERS

### RESERVED WORD SYMBOLS

AND
ARRAY
BEGIN
CASE
CONST
DIV
DO
DOWNTO
ELSE
END
FILE
FOR
FUNCTION
GOTO
IF
IN
LABEL
MOD
MODULE
NIL
NOT
OF
OR
OTHERWISE
PACKED
PROCEDURE
PROGRAM
RECORD
REPEAT
SET
THEN
TO
TYPE
UNIV
UNTIL
VAR
WHILE
WITH

### RESERVED WORD DIRECTIVES

EXTERN
FORTRAN
FORWARD

### PREDEFINED CONSTANT IDENTIFIERS

MAXINT
MAXSHORTINT
TRUE
FALSE

### PREDEFINED TYPE IDENTIFIERS

BOOLEAN
BYTE
CHAR
INTEGER
REAL
SHORTINTEGER
SHORTREAL
TEXT

## APPENDIX I (Continued)
## RESERVED WORDS AND PREDEFINED IDENTIFIERS

### PREDEFINED FUNCTIONS

ABS
ADDRESS
CHR
CONV
EOF
EOLN
LENG
LINENUMBER
ODD
ORD
PRED
ROUND
SHORTCONV
SHORTEN
SIZE
SQR
STACKSPACE
SUCC
TRUNC

### PREDEFINED PROCEDURES

DISPOSE
GET
MARK
NEW
PAGE
PUT
READ
READLN
RELEASE
RESET
REWRITE
WRITE
WRITELN

### PREDEFINED EXTERNAL TEXT FILE IDENTIFIERS

INPUT
OUTPUT

### NOTE

See Section 3.5.9 for additional mathmatical routines available from FORTRAN VII RTL for sine, cosine, arctangent, exponential, square root and natural logarithm.
Note that the identifiers "SIN", "COS", "ARCTAN", "LN", "EXP", and "SQRT" are not predefined.

See Section 10.3 (or Appendix N) for additional Prefix routines.

See Section 10.4 (or Appendix O) for additional SVC routines.

# APPENDIX J
# PASCAL SYNTAX GRAPHS

Context-free syntax is amended in some graphs to further clarify which kind of type, constant, expression, or identifier is semantically appropriate.

## Character

```
    |---> graphic-character --->|
--->|                           |--->
    |---> control-character --->|
```

## Graphic-Character

```
-------> special-character ------->
   |                           ^
   |---------> letter --------->|
   |---------> digit  --------->|
   V---------> space  --------->|
```

## Control-Character

```
-------> (: ----> digit ----> :) ------->
              ^            |
              |            |
              <------------V
```

## Digit

```
---------> 0 --------->
    |            ^
    |            |
    |----> 1 --->|
    |      .     |
    ~            ~
    ~      .     ~
    |            |
    V----> 9 --->|
```

# APPENDIX J
# PASCAL SYNTAX GRAPHS

### Letter

```
----------> A ---------->
     |            ^
     |----> B --->|
     |      .     |
     ~            ~
     ~      .     ~
     |            |
     |----> Z --->|
     |            |
     |----> a --->|
     |            |
     |----> b --->|
     |            |
     |      .     |
     ~            ~
     ~      .     ~
     |            |
     V----> z --->|
```

### Hexdigit

```
---------   0 ---------->
     |            ^
     |----  1 --->|
     |      .     |
     ~            ~
     ~      .     ~
     |            |
     |----> 9 --->|
     |            |
     |----> A --->|
     |            |
     |      .     ~
     ~            ~
     ~      .     |
     |            |
     V----> F ----|
     |            |
     |----> a --->|
     |            |
     |      .     ~
     ~            ~
     ~      .     |
     |            |
     V----> f ----|
```

# APPENDIX J
# PASCAL SYNTAX GRAPHS

## Special-Character

```
---------> ! --------->
    |            ^
    |            |
    |----> " --->|
    |            |
    |----> # --->|
    |            |
    |----> $ --->|
    |            |
    |----> % --->|
    |            |
    |----> & --->|
    |            |
    |----> ' --->|
    |            |
    |----> ( --->|
    |            |
    |----> ) --->|
    |            |
    |----> * --->|
    |            |
    |----> + --->|
    |            |
    |----> { --->|
    |            |
    |----> } --->|
    |            |
    |----> [ --->|
    |            |
    |----> ] --->|
    |            |
    |----> , --->|            ^----> > -------------->
    |            |            |            ^
    |----> - --->|            |----> ? --->|
    |            |            |            |
    |----> . --->|            |----> ^ --->|
    |            |            |            |
    |----> / --->|            |----> @ --->|
    |            |            |            |
    |----> : --->|            |----> _ --->|
    |            |            |            |
    |----> ; --->|            |----> | --->|
    |            |            |            |
    |----> < --->|            |----> ~ --->|
    |            |            |            |
    |----> = --->|            |----> \ --->|
    |                         |            |
    V------------------------>|----> ` --->|
```

Symbol

```
------->  special-symbol ------->
   |                         ^
   |--->  word-symbol   ---->|
   |                         |
   |----> identifier   ----->|
   |                         |
   V--> literal-constant  -->|
```

Separator

```
---------------------> space ------------------------>
    |                                           ^
    |                                           |
    |--------------> new line ----------------->|
    |                                           |
    |----> (* --> comment-sequence  --> *) ---->|
    |             not including *)              |
    |                                           |
    V----> { ---> comment-sequence  ---> } ---->|
                  not including }
```

Comment-sequence

```
 ------------------------------------->|
 ^                                     |
 |    |---> graphic-character  --->|   V
----->|                            |----->
   ^  |------>  new line   ------->| |
   |                                 |
   |<--------------------------------V
```

Identifier

```
-----> letter -------------------------->
                 ^                   |
                 |                   |
                 |<----- letter <----|
                 |                   |
                 |<----- digit  <----|
                 |                   |
                 |<-- underscore <---V
```

## Special-Symbols

```
---------->  + --------->
     |               ^
     |               |
     |---->  -  ---->|
     |               |
     |---->  *  ---->|
     |               |
     |---->  /  ---->|
     |               |
     |---->  &  ---->|
     |               |
     |---->  =  ---->|
     |               |
     |---->  <> ---->|
     |               |
     |---->  <  ---->|
     |               |
     |---->  >  ---->|
     |               |
     |---->  <= ---->|
     |               |
     |---->  >= ---->|
     |               |
     |---->  (  ---->|
     |               |
     |---->  )  ---->|
     |               |
     |---->  := ---->|
     |               |
     |---->  .  ---->|
     |               |
     |---->  ,  ---->|
     |               |
     |---->  ;  ---->|
     |               |
     |---->  .. ---->|
     |               |
     |---->  [  ---->|
     |               |
     |---->  ]  ---->|
     |               |
     |---->  ^  ---->|
     |               |
     V---->  @  ---->|
```

## Word-Symbols

```
--------->   AND   ---------->
   |                   ^
   |---->  ARRAY  ---->|
   |                   |
   |---->  BEGIN  ---->|
   |                   |
   |---->  CASE   ---->|
   |                   |
   |---->  CONST  ---->|
   |                   |
   |---->   DIV   ---->|
   |                   |
   |---->   DO    ---->|
   |                   |
   |----> DOWNTO  ---->|
   |                   |
   |---->  ELSE   ---->|              ^---->   OF   ------------>
   |                   |              |                   ^
   |---->  END    ---->|              |---->   OR   ----->|
   |                   |              |                   |
   |----> EXTERN  ---->|              |---->OTHERWISE---->|
   |                   |              |                   |
   |---->  FILE   ---->|              |----> PACKED ----->|
   |                   |              |                   |
   |---->  FOR    ---->|              |---->PROCEDURE---->|
   |                   |              |                   |
   |----> FORTRAN ---->|              |----> PROGRAM ---->|
   |                   |              |                   |
   |----> FORWARD ---->|              |----> RECORD  ---->|
   |                   |              |                   |
   |---->FUNCTION ---->|              |----> REPEAT  ---->|
   |                   |              |                   |
   |---->  GOTO   ---->|              |---->   SET   ---->|
   |                   |              |                   |
   |---->   IF    ---->|              |---->  THEN   ---->|
   |                   |              |                   |
   |---->   IN    ---->|              |---->   TO    ---->|
   |                   |              |                   |
   |---->  LABEL  ---->|              |---->  TYPE   ---->|
   |                   |              |                   |
   |---->  MOD    ---->|              |---->  UNIV   ---->|
   |                   |              |                   |
   |----> MODULE  ---->|              |----> UNTIL   ---->|
   |                   |              |                   |
   |---->  NIL    ---->|              |---->   VAR   ---->|
   |                   |              |                   |
   |---->  NOT    ---->|              |----> WHILE   ---->|
   |                                  |                   |
   |--------------------------------->|---->  WITH   ---->|
```

## Literal-Constant

```
     |-----------> TRUE ------------>|
     |                               |
     |-----------> FALSE ----------->|
     |                               |
     |-----------> NIL ------------->|
     |                               |
---->|------> unsigned integer ----->|---->
     |                               |
     |------> unsigned real number-->|
     |                               |
     |------> character literal ---->|
     |                               |
     |------> character string ----->|
```

## Constant

```
--------> constant-identifier --------->
    |                           ^
    |                           |
    |---> enumeration-constant --->|
    |                              |
    |---> real-constant  --------->|
    |                              |
    |---> string-constant  ------->|
    |                              |
    V---> pointer-constant ------->|
```

## Enumeration-Constant

```
--------> user-defined-enumeration -------->
    |        constant-identifier       ^
    |                                  |
    |-----> constant-identifier ----->|
    |                                 |
    |------> character-constant ----->|
    |                                 |
    |------> boolean-constant  ------>|
    |                                 |
    V------> integer-constant  ------>|
```

### Unsigned-Integer or Digits

```
-------->  digit ------->
     ^               |
     |               |
     |               |
     |<--------------V
```

### Integer-Constant (Signed)

```
            --> + -->
            ^       |
            |       |
            |       V
---------------------------> unsigned-integer ----------->
   |        |       ^                                ^
   |        |       |                                |
   |        V       |                                |
   |        --> - -->                                |
   |                                                 |
   V                                                 |
   -------------> # -------->   hexdigit   ---------->|
                       ^                       |
                       |                       |
                       |                       V
                       |<-----------------------
```

### Unsigned-Real-Number

```
             ------------------->|      --> + -->
             ^                   |      ^       |
             |                   |      |       |
             |                   V      |       V
---> digits ---> . --> digits -----> E ------------> digits --->
                               |        |       ^            ^
                               |        |       |            |
                               |        V       |            |
                               |        --> - -->            |
                               V                             |
                               ----------------------------->|
```

### Real-Literal Constant (Signed)

```
  --> + -->         ----------------->|     --> + -->
  ^       |         ^                 |     ^       |
  |       |         |                 |     |       |
  |       v         |                 v     |       v
----------- >digits---> . -->digits-----> E ---------->digits-->
  |       ^                         |       |       ^         ^
  |       |                         |       |       |         |
  v       |                         |       v       |         |
  --> - -->                         |       --> - -->         |
                                    v                         |
                                    ------------------------->|
```

### Character-Literal-Constant

```
-------> ' -------> character -------> ' ------->
```

### Character-String-Literal-Constant

```
-----> ' --------------> character -------------> ' ----->
          ^  |                            ^  |
          |  |                            |  |
          |  |                            |  |
          |  |-> (: -->  digits  --> :) ->|  |
          |                                  |
          |                                  |
          |<---------------------------------V
```

### Boolean-Constant-Identifier

```
---------> TRUE --------->
    |               ^
    |               |
    V---> FALSE --->|
```

### Pointer-Constant

```
---------> NIL --------->
```

### Program

```
 ---> prefix --->|
 ^               |
 |               V
-------------------> program-heading ---> block ---> .
```

### Program-Heading

```
                              ------------------->|
                              ^                   |
                              |                   V
---> PROGRAM ---> identifier ---> file-name-list ---> ; --->
```

### File-Name-List

```
---> ( ---> identifier ---> ) --->
        ^               |
        |               |
        |<---  ,  <-----V
```

### Prefix

```
  |<--- const definitions <---  |<- ; <-- procedure-heading <---
  |                          ^  |                              ^
  V                          |  V                              |
----------------------------------------------------------------->
  |                          ^  ^                              |
  |                          |  |                              |
  V---> type definitions --->|  |<- ; <-- function-heading <---V
```

### Block

```
---> declarations ---> body --->
```

### Declarations

```
------>label declarations----->|
   |                       ^   |
   |                       |   |
   V---------------------->|   |
                               |
                               |
   |<--------------------------V
   |
   |
   |  |<--const definitions<--^   ^-------------------->|
   |  |                       |   |                     |
   |  V                       |   |                     V
   V--------------------------------->var declarations----->|
      ^                       |                             |
      |                       |                             |
      |<--type definitions<---V                             |
                                                            |
                                                            |
                                 |<----------------------^  |
                                 |                       |  |
                                 V                       |  |
                               <---routine-declarations<----V
```

### Body

```
---> compound statement --->
```

### Label-Declarations

```
---> LABEL ---------> label -----> ; --->
                  ^            |
                  |            |
                  |<--- ,   <--V
```

### Label

```
-------> digits ------->
```

### Constant-Definitions

```
---> CONST ---> identifier ---> = ---> constant ---> ; --->
            ^                                            |
            |                                            |
            |<-------------------------------------------V
```

### Type-Definitions

```
---> TYPE ---> identifier ---> = ---> type ---> ; --->
           ^                                        |
           |                                        |
           |<---------------------------------------V
```

### VAR-Declarations

```
---> VAR ---> identifiers ---> : ---> type ---> ; --->
          ^                                         |
          |                                         |
          |<----------------------------------------V
```

### Identifiers

```
---> identifier --->
 ^                |
 |                |
 |<----  ,   <----V
```

## Type

```
----------> type-identifier --------->
      |                        ^
      |--> enumeration-type -->|
      |     (ordinal-types)    |
      |                        |
      |--------> REAL -------->|
      |                        |
      |-----> SHORTREAL ------>|
      |                        |
      |-----> array-type ----->|
      |                        |
      |-----> record-type ---->|
      |                        |
      |-------> set-type ----->|
      |                        |
      |-----> pointer-type --->|
      |                        |
      V------> file-type ----->|
```

## Enumeration-Type (ordinal-types)

```
------------------> CHAR ------------------>
   |                                   ^
   |-------------> BOOLEAN ----------->|
   |                                   |
   |--------------> BYTE ------------->|
   |                                   |
   |-----------> SHORTINTEGER -------->|
   |                                   |
   |-------------> INTEGER ----------->|
   |                                   |
   |-> user-defined-enumeration-type ->|
   |                                   |
   V-----------> subrange-type ------->|
```

## User-Defined Enumeration Type

```
---> ( ---> identifier ---> ) --->
        ^               |
        |               |
        |<----  ,  <----V
```

### Subrange-Type

```
---> constant --->  ..  ---> constant --->
```

### Array-Type

```
 ->PACKED->|
 ^         |
 |         V
------------->ARRAY--> [ -->index-type--> ] -->OF--->component-->
                          ^             |               type
                          |             |
                          |<---  , <----V
```

### Set-Type

```
---> SET ---> OF ---> base-member-type --->
```

### Pointer-Type

```
---> ^ -----> target-type ---->
```

### Target-Type

```
---> type-identifier --->
```

### File-Type

```
---> FILE ---> OF ---> component-type --->
```

### Index-type or Base-member-type

```
---> ordinal-type --->
```

### Component-type

```
---> type --->
```

## Record-Type

```
  --->PACKED--->|             --------------->|
  ^             |             ^               |
  |             V             |               V
--------------------> RECORD ---> field-list ---> END --->
```

## Field-List

```
 ---------------------->|                    ---> ; --->|
 ^                      |                    ^          |
 |                      V                    |          V
---> fixed-part ---> ; ---> variant-part ------------------->
                 |                        ^
                 |                        |
                 V----------------------->|
```

## Fixed-Part

```
---> identifiers ---> : ---> type --->
 ^                                 |
 |                                 |
 |<-------------  ;  <-------------V
```

## Variant-Part

```
            tag-field             type
--->CASE--->identifier--> : --->identifier-->OF--->variant--->
         |                   ^                  ^          |
         |                   |                  |          |
         V------------------>|                  |<--- ; <--V
```

## Variant

```
---> constant ---> : ---> ( ---> field-list ---> ) --->
 ^             |             |               ^
 |             |             |               |
 |<---  ,  <---V             V-------------->|
```

### Routine-Declarations

```
  |<--- ; <--- procedure <---^
  V                          |
--------------------------------->
  ^                          |
  |<--- ; <--- function  <---V
```

### Function

```
                                 |---> EXTERN  --->|
                                 |                 |
                                 |---> FORTRAN --->|
                                 |                 |
---> function-heading ---> ; --->|--->  block  --->|--->
                                 |                 |
                                 |---> FORWARD --->|
```

### Function-Heading

```
                      ^------------------------------------->|
                      |                                      |
                      |                                      V
->FUNCTION->identifier-->parameter list->:->type-identifier----->
```

### Parameter-List (Internal Procedures and Functions)

```
 ^------------------------------------------------------------>|
 |                                                             |
 |   /-------------> procedure-heading ------------------>|    |
 |   |-------------> function-heading ------------------->|    |
 |   |                                                    |    |
 |   | /--->|                   /---->|                   |    |
 |   | |    |                   |     |                   |    |
 |   | |    V                   |     V                   V    V
-->(---->VAR--->identifier-->:--->UNIV--->type-identifier-->)--->
    ^         ^           |                                |
    |         |<--- , <---V                                |
    |                                                      |
    |<----------------------- ; <--------------------------V
```

## Procedure

```
                                  |---> EXTERN  --->|
                                  |                 |
                                  |---> FORTRAN --->|
                                  |                 |
---> procedure-heading ---> ; --->|--->  block  --->|--->
                                  |                 |
                                  |---> FORWARD --->|
```

## Procedure-Heading

```
                                ^------------------>|
                                |                   |
                                |                   V
---> PROCEDURE ---> identifier ---> parameter-list -------->
```

## Module

```
 ---> prefix --->
 ^              |
 |              V
-----------------> module-heading ---> block ---> .
```

## Module-Heading

```
                                           ----------------------->|
                                           ^                       |
                            module-        |                       V
--->MODULE--->identifier--->parameter-list---> : ->type-identifier--->
                         |                                         ^
                         V                                         |
                         ----------------------------------------->|
```

## Module-Parameter-List

```
 ^---------------------------------------------------------->|
 |                                                            |
 |   ^----->|                  ^------>|                      |
 |   |      |                  |       |                      |
 |   |      V                  |       V                      V
-->(--->VAR--->identifiers->:--->UNIV--->type-identifier--->)--->
    ^                                                   |
    |                                                   |
    |<--------------------- ; <-------------------------V
```

## Procedure-call Statement

```
---> identifier ----------------------------->
                   |                        ^
                   |                        |
                   V---> argument-list --->|
```

## Argument-List

```
  ^-------------------------------------------------------->|
  |                                                          |
  |             |---> variable-selector --->|                |
  |             |                           |                |
  |             |---> procedure-argument -->|                |
  |             |                           |                V
-----> ( ----->|                            |-----> ) ----->
           ^    |                           |   |
           |    |---> function-argument --->|   |
           |    |                           |   |
           |    |-------> expression ------>|   |
           |                                    |
           |<-------------- , <-----------------V
```

## Procedure-Argument

```
---> procedure-identifier --->
```

## Function-Argument

```
---> function-identifier --->
```

## Variable-Selector (General)

```
--------> identifier -------------->
   |                           ^
   |----> array-component ---->|
   |                           |
   |----> record-field  ------>|
   |                           |
   |----> pointer-target  ---->|
   |                           |
   V---->  file-buffer    ---->|
```

## Variable-Selector (Summarized in Detail)

```
---> variable-identifier ---------------------------------------->
 |                        ^ ^ |            index-            |
 |                        | | |--> [ --> expression --> ] -->|
 V--> field-identifier -->| | |       ^             |        |
                            | |       |<---- , <----V        |
                            | |                              |
                            | |                              |
                            | |--> . --> field-identifier -->|
                            | |                              |
                            | V--> ^ ----------------------->|
                            |                                |
                            |                                |
                            |<-------------------------------V
```

## Index-expression

```
---> expression --->
```

## Array-Component (Selector)

```
---> variable-selector ---> [ ---> index-expression ---> ] --->
                               ^                     |
                               |                     |
                               |<------  ,  <--------V
```

## Record-Field (Selector)

```
 ------------------------------->|
 ^                               |
 |                               v
---> variable-selector ----> . ---> field-identifier ---->
                         ^                              |
                         |                              |
                         |<-----------------------------V
```

## Pointer-Target (Selector)

```
---> variable-selector -----> ^ ----->
```

## File-Buffer (Selector)

```
---->identifier----> ^ --->
```

## Expression

```
---> simple-expression ---------------------------------->
                        |   |  |   |  |  |   |       ^
                        |   |  |   |  |  |   |       |

                        =  <>  <  <=  >  >=  IN   simple-
                                                 expression
                        |   |  |   |  |  |   |       |
                        |   |  |   |  |  |   |       |
                        V   V  V   V  V  V   V       |
                        ---------------------------->|
```

## Simple-Expression

```
  ^--> + -->|
  |         |
  |         V
---------------> term ------------------>
  |         ^            ^   |  |   |
  |         |            |   |  |   |
  V--> - -->|            |   |  |   |

                       term  +  -  OR

                         |   |  |   |
                         |   |  |   |
                         |   V  V   V
                         |<----------
```

## Term

```
---> factor ---------------------------------->
              ^     |   |   |    |    |
              |     |   |   |    |    |

            factor  *   /  DIV  MOD  AND

              |     |   |   |    |    |
              |     |   |   |    |    |
              |     V   V   V    V    V
              |<-----------------------
```

## Factor

```
----------> unsigned constant ------------>
    |                                ^
    |                                |
    |---->  variable-selector  ----->|
    |                                |
    |------>   function-call   ----->|
    |                                |
    |---> ( --> expression --> ) --->|
    |                                |
    |------> NOT ----> factor ------>|
    |                                |
    V------>  set-constructor  ----->|
```

## Set-Constructor

```
          ----------------------------------->|
          ^                                   |
          |                                   V
--->  [  -------> expression..expression -------> ]  --->
            ^ |                           ^ |
            | |                           | |
            | V-------> expression ------>| |
            |                               |
            |<------------- , <-------------V
```

## Function-Call

```
---> identifier ---------------------------->
                  |                       ^
                  |                       |
                  V---> argument-list --->|
```

Statement

```
---> label ---> : --->|
 |                 ^  |
 |                 |  |
 V---------------->|  |
                      |
|<--------------------V
|
V--------------------------------------->
    |                            ^
    |---> compound-statement --->|
    |                            |
    |-----> case-statement ----->|
    |                            |
    |-----> for-statement ------>|
    |                            |
    |------> if-statement ------>|
    |                            |
    |----> while-statement ----->|
    |                            |
    |----> repeat-statement ---->|
    |                            |
    |-----> with-statement ----->|
    |                            |
    |--> assignment-statement -->|
    |                            |
    |-----> procedure-call ----->|
    |                            |
    V-----> goto-statement ----->|
```

Compound-Statement

```
---> BEGIN -------> statement -------> END --->
               ^                |
               |                |
               |<----- ; <------V
```

GOTO-Statement

```
---> GOTO -----> label ----->
```

### CASE-Statement

```
--->CASE--->expression--->OF--->labeled statements--->END--->
```

### Labeled-Statements

```
  ----->  OTHERWISE  ------------->|              ---> ; -->|
  ^                                |              ^         |
  |                                V              |         V
----->enumeration constant----> : --->statement--------------->
 ^ ^                       |                    |
 | |                       |                    |
 | |<--------- , <---------V                    |
 |                                              |
 |<---------------------- ; <-------------------V
```

### FOR-Statement

```
--->FOR--->identifier---> := --->expression1----->TO------>|
                                             |          ^  |
                                             |          |  |
                                             V->DOWNTO->|  |
                                                           |
                    <---statement<---DO<---expression2<----V
```

### IF-Statement

```
--->IF-->expression-->THEN-->statement--->ELSE-->statement---->
                                       |                    ^
                                       |                    |
                                       V------------------->|
```

### WHILE-Statement

```
---> WHILE ---> expression ---> DO ---> statement --->
```

### REPEAT-Statement

```
---> REPEAT -----> statement -----> UNTIL ---> expression --->
              ^                |
              |                |
              |<----- ; <------V
```

### WITH-Statement

```
---> WITH ---> variable-selector ---> DO ---> statement --->
           ^                              |
           |                              |
           |<------------- , <------------V
```

### Assignment-Statement

```
---->  variable-selector  -------> := ---> expression --->
 |                             ^
 |                             |
 V---> function-identifier --->|
```

### Procedure-call Statement

```
---> identifier ---------------------------->
                  |                       ^
                  |                       |
                  V---> argument-list --->|
```

### Write-Parameter

```
                                      ^-------------------->|
                                      |                     |
                          field-width |         frac-digits V
 --->expression---> : ---> expression---> : ---> expression--->
                |                                            ^
                |                                            |
                V------------------------------------------->|
```

### READ procedure-call Statement

```
---> READ ---> Read-parameter-list --->
```

### Read-Parameter-List
(for non text files)

```
 ---> ( ---> file-variable ---> , ---> variable-selector ---> ) --->
                                  ^                       |
                                  |                       |
                                  |<-------- , <---------V
```

### Read-Parameter-List
(for text files)

```
---> ( ------------------------------>variable-selector---> ) --->
        |                         ^ ^                   |
        |                         | |                   |
        V-->file-variable---> , ->| |<------- , <-------V
```

### READLN procedure-call Statement

```
---> READLN ---------------------------->
             |                       ^
             |        Readln-        |
             V---> parameter-list --->
```

### READLN-Parameter-List

```
                         -------------------------------->
                         ^                               |
                         |                               v
---> ( --->file-variable---> , ------>variable-selector------> ) --->
        |                       ^ ^                    |
        |                       | |                    |
        v----------------------> <--------- , <-------v
```

### WRITE procedure-call Statement

```
---> WRITE ---> Write-parameter-list --->
```

### Write-Parameter-List
(for non text files)

```
---> ( ---> file-variable ---> , ---> expression ---> ) --->
                                  ^               |
                                  |               |
                                  |<------ , <----V
```

### Write-Parameter-List
(for text files)

```
---> ( ------------------------------>write-parameter----> ) --->
        |                         ^ ^                 |
        |                         | |                 |
        v--->file-variable-> ,--->| |<------ , <------v
```

### WRITELN procedure-call Statement

```
--->WRITELN------------------------------->
              |                        ^
              |                        |
              |         Writeln-       |
              V---> parameter-list --->|
```

### Writeln-Parameter-List

```
                          -------------------------------->
                          ^                               |
                          |                               v
---> ( ---> file-variable ---> ,-----> write-parameter -----> ) --->
        |                       ^ ^                    |
        |                       | |                    |
        v---------------------->| <-------- , <--------v
```

# APPENDIX K
# EXTENSIONS AND EXCEPTIONS TO "STANDARD" PASCAL

Presently, at this writing, there exists no recognized definition of a standard Pascal language. There are however two defacto language specifications upon which the Perkin-Elmer definition of the language is based. These are the aforementioned _Pascal User Manual and Report_ by Jensen and Wirth and _A Draft Proposal for Pascal_ by A.M. Addyman, called the British standard, and as published in SIGPLAN NOTICES, Volume 15, Number 4, April 1980; is a proposed ISO standard Pascal, currently under review/revision.

When a direct conflict is apparent between the two documents, the Addyman proposal prevailed. Only one exception to the Jensen and Wirth report is notable, namely the specification of procedural and functional formal parameters where Perkin-Elmer Pascal R01 (and up) follows the Addyman proposal on this feature.

There are a few exceptions and several extensions in Perkin-Elmer Pascal to the Addyman proposed standard of the language.

Briefly summarized, features of "standard" Pascal which are not supported:

1. Two routines: PACK and UNPACK, are not provided.
2. Variant-tag arguments to NEW and DISPOSE, are not allowed.
3. The construct "conformant-array-schema" as an alternative to type-identifier within variable-parameter-specifications is not provided.
4. Introducing a user-defined enumeration type definition, instead of its type-identifier, in a record-type definition is not allowed; i.e., a previously declared "user-defined enumeration type" type-identifier must be used in declaring fields to be of a user-defined enumeration type.
5. REWRITE does not destroy any information externally existing from previous WRITE/WRITELNs, on its file argument, at the time of the call; the file is merely rewound and EOF(f) becomes true. If the previously written information is not overwritten, it still exists.
6. The mathematical functions for sine, cosine, arctangent, exponential, and natural logarithm (standard Pascal's SIN, COS, ARCTAN, EXP, SQRT, and LN) do not have predefined identifiers, but are provided via external FORTRAN function declarations within the Pascal compilation-unit referencing them, with linkage to FORTRAN RTL math routines. They have, especially ARCTAN and LN, have slightly different names. See #6.6.6.2 below.

In anticipation of the requirement, for a Pascal implementation to be accompanied by a document separately describing any features acceptable to the Pascal "processor" (compiler) which are not specified in the standard (currently limited to Paragraph 6, entitled REQUIREMENTS, in the Addyman proposed standard); this appendix summarizes both exceptions and extensions to that proposed standard. This appendix includes as "limitations" below, certain restrictions that are imposed on the size or complexity of a program and as permitted by Section 1.2(a) of the draft standard, such limitations are not considered, herein, exceptions to the standard; e.g. the maximum limit of user-defined enumeration type values is 128.

The exceptions and extensions are listed by the associated paragraph numbers of the Addyman proposed standard. The reader is directed into the Pascal Language Reference Manual Chapters 2 through 9 for further details, where appropriate.

Note that we present the language syntax in the notation of syntax graphs, which may differ slightly in nomenclature from the Backus-Naur form used in the standard. However, no divergeance in meaning from the proposed standard is intended nor implied, unless it is explicitly mentioned in this document.

Perkin-Elmer Pascal contains the following exceptions, limitations, and extensions to the Addyman proposed standard:

# 6. REQUIREMENTS

## 6.1 Lexical Tokens.

### 6.1.1 General.

In addition to letter and digit, we define hexdigit characters:

```
hexdigits:  0 1 2 3 4 5 6 7 8 9 A B C D E F
                                a b c d e f
```

which when a digit-sequence of hexdigits is preceded by the pound sign (#) forms a literal integer constant (see Numbers below).

### 6.1.2 Special-symbols.

We additionally define, as special-symbols:

| | | |
|---|---|---|
| & | Ampersand | alternative symbol for logical "AND" |

and additionally the special-characters:

| | | |
|---|---|---|
| # | Pound sign | hexadecimal integer beginning delimiter |
| _ | Underscore | may be a character in an identifier (after the first letter of the identifier: e.g., COMPUTE_ITEM56_totals_AND_Print) |

We additionally define as reserved word-symbols:

UNIV
MODULE
FORTRAN
EXTERN
OTHERWISE

We additionally treat the standard directive FORWARD as a reserved word-symbol.

### 6.1.3 Identifiers.

Identifiers may also serve to denote module names.

In this implementation, identifiers may be of up to 140 characters in length.

We permit the use of the underscore character "_" as an other than first character of an identifier.

We additionally provide the predefined constant-identifier MAXSHORTINT to identify the value of +32767.

### 6.1.4 Directives.

In addition to providing the required directive, FORWARD, we provide the directives EXTERN and FORTRAN (for external linkage);

as allowed by 6.1.4 and referenceable as directives as stated in 6.6.1 and 6.6.2 of the standard.

6.1.5 Numbers.

We additionally allow a hexadecimal notation (#hexdigits) to denote literal integer constants.

6.1.6 Labels.

We extend the range of statement labels, i.e., the labels declared in a LABEL declaration, used to prefix labeled-statements for the purpose of a "GOTO label" statement. A statement label may be an unsigned integer in the closed interval 0 to the value of MAXINT (up to ten digits in the digit-sequence of a label); whereas the Addyman proposal restricts label to the closed interval 0 to 9999 (up to four digits in the digit-sequence).

6.1.7 Character-strings.

We additionally define and permit the construct control-character within a literal character-string as a method to represent the non-printable characters in the ASCII character set. A control-character is of the form (:nn:) where nn is the ordinal value between 0 and 127, thereby giving the control-characters with ordinal values 0 to 31, and 127 a method of representation as characters within a literal character string; e.g. 'MESSAGE(:13:)' is a literal 8-character string whose last character is the carriage-return; and as a single character string is considered of CHAR type, the string '(:07:)' is a single character, the bell. Note that the construct control-character is not defined syntactically as a form of character-literal but as noted above a single character literal string is of CHAR type, not a structured array of characters; such that the single character literal string serves as a character-literal. See Chapter 3, Section 3.2.2 for control-character definition and the last paragraph in Section 3.3.4 for an example of explicitly writing strings meant to be output as the literal characters "(:nnn:)" and not meant to be a control-character.

6.1.8 Token Separators.

We allow one level of nested comments, by restricting and requiring the pairing of either the comment delimiters { and }, or the comment delimiters (* and *) to enclose the text of a comment. We do not allow the nesting of comments using the same pair of delimiters. We do not recognize the intermixed forms (* comment } nor { comment *). The Addyman proposal define { and (* as equivalent, and likewise } and *) as equivalent. Within the proposed standard, the left and right braces to define comments as separators in 6.1.8 have alternatives defined in the NOTE of 6.11, such that any intermixing of (* and { to begin comments and *) and } to end comments is allowed, disallowing nested comments.

### 6.2 Blocks and scope.
### 6.2.1 Block.

We extend the standard to allow several, and the intermixing of, constant-declaration-parts and type-declaration-parts; to occur in a declarations part; e.g., prior to the variable-declaration-part in a block.

### 6.2.2 Scope.
### 6.2.2.1 - 6.2.2.11

### 6.3 Constant-definitions.

### 6.4 Type-definitions.
### 6.4.1 General.
### 6.4.2 Simple-types.
### 6.4.2.1 General.
### 6.4.2.2 Standard simple-types.

We additionally define three additional predefined simple-types, with the predefined type-identifiers:

BYTE

SHORTINTEGER

SHORTREAL

See Chapter 5 for details. SHORTINTEGER type occupies 2 bytes and is implemented as a subrange of INTEGER values, BYTE type occupies 1 byte and is implemented as a subrange of SHORTINTEGER values, enabling some error checking for out-of-range values under appropriate compiler options. See Chapter 1 for details. BYTE and SHORTINTEGER data have discrete scalar values. SHORTREAL data have non-discrete scalar values. The range of values for BYTE type data is 0 to 255 inclusively; the range of values for SHORTINTEGER type data is -32768 to +32767, inclusively. SHORTREAL is implemented with the same range of values as type REAL, but with less precision and half the internal storage requirements; i.e., SHORTREAL is a single-precision floating-point number internally occupying 4 bytes requiring fullword alignment; whereas REAL is a double-precision floating-point number internally occupying 8 bytes requiring fullword alignment.

### 6.4.2.3 Enumerated-types.

We deviate from standard in not allowing a user-defined enumeration type to be introduced in a record-type definition, when declaring field types.

We impose a limitation of the number of identifiers listed in a user-defined enumeration type to be no more than 128.

### 6.4.2.4 Subrange-types.
### 6.4.3 Structured-types.

### 6.4.3.1 General.

The optional token "PACKED" prior to the keywords SET OF, or FILE OF, is ignored, and has no further "packing" effect on set-types nor file-types. Therefore, the keyword token "PACKED" is not presented in the syntax graphs of set-type and file-type, as they are in array-type and record-type. To conform to standard, though, the compiler simply ignores the presence of "PACKED" in the syntax of set-type and file-type. PACKED array-types and PACKED record-types are implemented as allowed by the standard.

### 6.4.3.2 Array-types.

We extend the definition of a string-type which may be denoted by: as an array-type declared as ARRAY[type1] OF CHAR, where type1 is a subrange-type-identifier of subrange-type m..n; or a string-type may be denoted by: ARRAY[m..n] OF CHAR where in either form: m <= n and m..n is a subrange-type with lower bound m and upper bound n. The Addyman proposal further restricts this definition by requiring that type1 is a subrange-type where m as the lower bound is 1; we do not. The Addyman proposal further requires the token "PACKED" prior to ARRAY[Type1] OF CHAR to designate a string-type; we do not.

### 6.4.3.3 Record-types.

We impose the limitation that the tag-field type of a variant-selector must have all its ordinal values in the range 0..31. We also limit the depth of nesting of variants to be no more than 16.

### 6.4.3.4 Set-types.

We impose a limitation on set-types by requiring the base-member-type to have its ordinal values in the range 0..127.

### 6.4.3.5 File-types.
### 6.4.4 Pointer-types.

### 6.4.5 Compatible types. (See Chapter 6, section 6.2 on Type Compatibility).

We adhere to the standard on type-compatibility for those types specified in the standard where the full definition of type-compatibility requires the joint analysis of both 6.4.5 and the preceding paragraph 6.4.1 on General (Type-definitions).

As we provide additional predefined type-identifiers (BYTE, SHORTINTEGER, SHORTREAL) type-compatibility is expanded to cover their inter-relationships amongst other types, etc.

### 6.4.6 Assignment-compatibility. (See Chapter 6, Section 6.2, on assignment-compatibility.)

Assignment compatibility rules are expanded to cover our additional types and their inter-relationships amongst other types.

6.4.7 Example
6.5 Declarations and denotations of variables.
6.5.1 Variable-declarations.
6.5.2 Entire-variables.
6.5.3 Component-variables.
6.5.3.1 General.
6.5.3.2 Indexed-variables.
6.5.3.3 Field-designators.
6.5.4 Referenced-variables.
6.5.5 Buffer-variables.
6.6 Procedure and function declarations.

We impose a limitation on the depth of nesting of procedures and functions to be no more than 16, counting the outermost program level as 1, an outermost routine may contain 14 nested levels of routines.

6.6.1 Procedure-declarations.
6.6.2 Function-declarations.
6.6.3 Parameters.
6.6.3.1 General.

Here, we adhere to the Addyman proposed standard for declaration of procedural and functional formal parameters, and hence differ (in a positive way) from the original Jensen and Wirth definition.

However, in this same paragraph, for variable-parameter-specifications we do not provide the construct "conformant-array-schema" as an alternative to type-identifier in parameter-lists.

6.6.3.2 Value parameters.
6.6.3.3 Variable parameters.

The paragraphs pertaining to "conformant-array-schema", as this is not provided in our implementation, is not applicable. This "conformant-array-schema" essentially allows adjustably dimensioned arrays to be passed to routines. We do not support any other proposed variation of this feature either.

6.6.3.6 Parameter-list congruity.

(b) Again, "conformant-array-schema" rules are not applicable.

We additionally provide a reserved word, UNIV, which partially defeats the compile-time verification of type-compatibility between parameter and argument data. UNIV only requires the internal storage size of both the parameter and associated argument to be the same.

6.6.4 Standard procedures and functions.

6.6.4.1 General.
6.6.5 Standard procedures.
6.6.5.1 General.
6.6.5.2 File handling procedures.

We limit the implementation of the standard procedure REWRITE in that a call to that routine does not destroy any information existing in the file argument at the time of the call; the file is merely rewound, although EOF(f) is set to true. The usual definition in standard Pascal of REWRITE implies that not only does EOF(f) become true but that the specified file is made empty.

6.6.5.3 Dynamic allocation procedures.

Pascal R00 provided NEW, MARK, and RELEASE, but not DISPOSE.

Pascal R01 provides both NEW and DISPOSE, but not their argument specification variant-tag forms: NEW(p,t1,...,tn) nor DISPOSE(p,t1,...,tn). Records with variants, when created by NEW, occupy enough space to accomodate the largest variant. We additionally provide MARK and RELEASE routines to programs compiled under the HEAPMARK compiler option.

6.6.5.4 Transfer procedures.

We do not provide the array restructuring transfer procedures:

PACK
UNPACK

6.6.6 Standard functions
6.6.6.1 General

We additionally provide miscellaneous functions:

ADDRESS
SIZE
LINENUMBER
STACKSPACE

6.6.6.2 Arithmetic functions.

ABS and SQR are provided as standard predefined functions, but SIN, COS, EXP, LN, SQRT, and ARCTAN are provided with slightly different names and require an external FORTRAN function declaration in the Pascal compilation-unit for each routine to be referenced as, either:

SIN,COS,EXP,ALOG,SQRT,ATAN, respectively for SHORTREAL result;

or:

DSIN,DCOS,DEXP,DLOG,DSQRT,DATAN, respectively for REAL result.

See Chapter 3, Sections 3.5.3 and 3.5.9.

### 6.6.6.3 Transfer functions.

In addition to TRUNC and ROUND, we define four additional predefined standard functions for explicit type conversion of data:

SHORTEN
LENG
CONV
SHORTCONV

### 6.6.6.4 Ordinal functions.

We extend the function ORD by allowing it to accept pointers as arguments.

### 6.6.6.5 Boolean functions.

### 6.7 Expressions.
### 6.7.1 General.
### 6.7.2 Operators.
### 6.7.2.1 General.
### 6.7.2.2 Arithmetic operators.
### 6.7.2.3 Boolean operators.
### 6.7.2.4 Set operators.
### 6.7.2.5 Relational operators.

Comparisons can be made between data of certain types, either "identically" typed structures, or "assignment-compatible" scalars, or a variety of string structures, and a brief generalized summary of those that are valid is listed in Table 6-7. Structure comparisons are non-standard Pascal, and in this implementation of Pascal, structured comparisons (arrays and records) are performed binarily on a byte by byte basis, regardless of alignment gaps imbedded in their internal storage; such that they may only be useful in comparing structures without alignment gaps. Also see Section 10.6.1 on Internal Data Storage Requirements.

### 6.7.3 Function designators.

### 6.8 Statements.
### 6.8.1 General.
### 6.8.2 Simple-statements.
### 6.8.2.1 General.
### 6.8.2.2 Assignment-statements.
### 6.8.2.3 Procedure-statements.
### 6.8.2.4 Goto-statements.
### 6.8.3 Structured-statements.
### 6.8.3.1 General.
### 6.8.3.2 Compound-statements.

Clarification: This paragraph may imply by its definition of "statement-sequence" and example that no semi-colon may exist between the last statement of a statement-sequence and the word END; but may be subject to interpretation. Perkin-Elmer Pascal effectively allows this semi-colon at the end of a "statement-sequence" in both a compound-statement and repeat-statement, considering the last semi-colon as introducing an empty-statement between the last presented statement of a statement-sequence within BEGIN and END, or within REPEAT and UNTIL. However, we adhere to the aforementioned interpretation of the standard, if that last semi-colon at the end of a statement-sequence is not present; as Perkin-Elmer Pascal does not "require" it to be present.

6.8.3.3 Conditional-statements.
6.8.3.4 If-statements.
6.8.3.5 Case-statements.

We extend the syntax of the CASE statement to permit an OTHERWISE clause which indicates its statement is to be executed if no match of the case-selector expression and any of the possible explicit case-label constants is made. There is no colon required nor allowed between the keyword OTHERWISE and its following statement, as follows the other case-label constants.

We impose a limitation on the CASE statement by requiring that each case-label constant have its ordinal value in the range 0..127.

6.8.3.6 Repetitive-statements.
6.8.3.7 Repeat-statements.

See paragraph number 6.8.3.7 above, for semi-colon allowed (but not required) at the end of a statement-sequence.

6.8.3.8 While-statements.
6.8.3.9 For-statements.
6.8.3.10 With-statements.

6.9 Input and Output.
6.9.1 General.
6.9.2 The procedure READ.
6.9.3 The procedure READLN.
6.9.4 The procedure WRITE.
6.9.4.1 Multiple parameters.
6.9.4.2 Write-parameters.
6.9.4.3 CHAR-type.
6.9.4.4 Integer-type.

(also applies to our BYTE and SHORTINTEGER types)

6.9.4.5 Real-type.

(also applies to our SHORTREAL type)

6.9.4.5.1 The floating-point representation.
6.9.4.5.2 The fixed-point representation.
6.9.4.6 Boolean-type.

The string 'TRUE ', not the string 'TRUE', is output right-justified in a field of total-width (default 5 positions) positions for Boolean-typed expressions in write-parameters to text file fields.

6.9.4.7 String-types.
6.9.5 The procedure WRITELN.
6.9.6 The procedure PAGE.
6.10 Programs.

We permit a "prefix" to a program or module where the prefix contains declarations of constants, types, and routines which become globally visible to the compilation-unit being compiled.

We provide a Perkin-Elmer predefined Prefix, (see Appendix N), which extends the Pascal language to those compilation-units compiled with it, to obtain OS/32 file-handling and operating system services.

6.11 Hardware representation.

Extension to NOTE of 6.11 given above in 6.1.8.

Additional Feature: external routines:

In addition to the standard topics outlined above, we provide a separately compilable unit known as MODULE which may be linked to the main program or other modules. We provide directives to declare within a compilation-unit an unlimited number of routines (PROCEDUREs or FUNCTIONs) as external and thereby specify their calling sequence through their parameter-lists and choice of either the EXTERN or FORTRAN directive. That is, separately compiled Pascal modules, FORTRAN, or CAL written routines are linkable to a Pascal main program or module. See Chapter 9 for syntactical rules and Chapter 10 for internal linkage conventions.

We allow the source for a compilation to be taken from one or several additional source files, by the $INCLUDE compiler-option.

# APPENDIX L
# PASCAL RUN TIME LIBRARY (PASRTL.OBJ)

The arrangement of separate object programs in the Pascal RTL R01 is listed below in their order of appearance on PASRTL.OBJ, and the external references of each program.

FORTRAN Interface routine:

| | |
|---|---|
| P$FORT | Contains an alternate version of P$INIT |
| P$INIT | weak entry; alternate version<br>Calls .INITSCA, P$SEND, P$TERM |

Initialization routine and error message routine:

| | |
|---|---|
| P$INIT | strong entry; principal version<br>Calls P$SEND, P$TERM |
| P$ERMES | Calls P$SEND, P$PAUS, P$TERM |

Error handling routine group under OS:

| | |
|---|---|
| PAS.ERR | Object program containing these routines: |
| P$ERR | |
| PASERROR | Calls P$ERMES, P$SEND, P$PAUS<br>(Internal routine, referenced by P$ERR but the symbol PASERROR does not appear in LINK MAP) |
| P$PAUS | weak entry |
| P$TERM | weak entry |
| P$SEND | weak entry |

Error handling routine group under Reliance:

| | |
|---|---|
| PAS.REL | Object program containing these routines: |
| P$PAUS | weak entry<br>Calls RPAUSE in the FORTRAN/RELIANCE Interface |
| P$TERM | weak entry<br>Calls RPROG in the FORTRAN/RELIANCE Interface |
| P$SEND | weak entry<br>Calls RPAUSE in the FORTRAN/RELIANCE Interface |

Prefix support routines:

| | |
|---|---|
| OPEN | Calls P$ERMES |
| CLOSE | Calls P$ERMES |
| ALLOCATE | Calls P$ERMES |
| RENAME | Calls P$ERMES |
| REPROTEC | Calls P$ERMES |
| DELETE | Calls P$ERMES |
| CHANGE_A | Calls P$ERMES |
| CHECKPOI | Calls P$ERMES |
| FETCH_AT | Calls P$ERMES |
| REWIND | Calls P$IOFUN, P$ERMES |
| WRITE_FI | Calls P$IOFUN, P$ERMES |
| BACK_REC | Calls P$IOFUN, P$ERMES |
| BACK_FIL | Calls P$IOFUN, P$ERMES |
| FORWD_RE | Calls P$IOFUN, P$ERMES |
| FORWD_FI | Calls P$IOFUN, P$ERMES |
| BREAKPOI | Calls P$ERMES |
| START_PA | Calls P$ERMES |
| TIME | Calls P$ERMES |
| DATE | Calls P$ERMES |
| EXIT | Calls P$ERMES, P$TERM |
| P$IOFUN | |

SVC support routines:

| | |
|---|---|
| SVC1 | Calls P$ERMES |
| SVC3 | Calls P$ERMES, P$TERM |
| SVC5 | Calls P$ERMES |
| SVC7 | Calls P$ERMES |
| SVC2PAUS | Calls P$ERMES, P$PAUS |
| SVC2AFLT | Calls P$ERMES |
| SVC2FPTR | Calls P$ERMES |
| SVC2LOGM | Calls P$ERMES |
| SVC2FTIM | Calls P$ERMES |
| SVC2FDAT | Calls P$ERMES |
| SVC2TODW | Calls P$ERMES |
| SVC2INTW | Calls P$ERMES |
| SVC2PKNM | Calls P$ERMES |
| SVC2PKFD | Calls P$ERMES |
| SVC2PEEK | Calls P$ERMES |
| SVC2TMAD | Calls P$ERMES |
| SVC2TMWT | Calls P$ERMES |
| SVC2TMRP | Calls P$ERMES |
| SVC2TMLF | Calls P$ERMES |
| SVC2TMCA | Calls P$ERMES |
| SVCINITQ | Calls P$ERMES |
| SVCTASKQ | Calls P$ERMES |
| FROMUDL | Calls P$ERMES |
| TOUDL | Calls P$ERMES |

## APPENDIX L (Continued)
## PASCAL RUN TIME LIBRARY (PASRTL.OBJ)

Heap management routines:

| | |
|---|---|
| P$NEW | Calls P$ERMES, P$$REMV |
| P$DISP | Calls P$ERMES, P$$REMV |
| P$MARK | Calls P$ERMES |
| P$REL | Calls P$ERMES, P$$REMV |
| P$SPAC | |
| P$$REMV | |

Routines for Manipulation of structured variables:

P$STCPY
P$STCMP0
P$STCMP1
P$STCMP2
P$STCMP3
P$FILCPY

Set operations routines:

P$SCOMP
P$SAND
P$SOR
P$SDIF

Routines for Input, non-text:

| | |
|---|---|
| P$READ | Calls P$GET, P$GETER1 |
| P$RESET | Calls P$GET, P$$REWD, P$FCBERR, P$PAUS |
| P$GET | Calls P$$SVC1, P$GETER1, P$GETER2, P$PAUS |

Routines for Input, text:

| | |
|---|---|
| P$RESETT | Calls P$GETT, P$PURGE, P$$REWD, P$FCBERR, P$PAUS |
| P$READBY | Calls P$$RDINT |
| P$READSI | Calls P$$RDINT |
| P$READI | Calls P$$RDINT |
| P$READSR | Calls P$GETT, .ATOF, P$PAUS, P$TERM, P$NUMERR |
| P$READR | Calls P$GETT, .ATOD, P$PAUS, P$TERM, P$NUMERR |
| P$READCH | Calls P$GETT |
| P$$RDINT | Calls P$GETT, P$GETER1, P$NUMERR, P$TERM, P$PAUS |
| P$GETT | Calls P$READLN, P$GETER1 |
| P$READLN | Calls P$$SVC1, P$GETER1, P$GETER2, P$PAUS |

# APPENDIX L (Continued)
# PASCAL RUN TIME LIBRARY (PASRTL.OBJ)

Routines for Output, non-text:

| | |
|---|---|
| P$WRITE | Calls P$PUT, P$PUTERR |
| P$PUT | Calls P$$SVC1, P$PUTERR, P$PAUS |

Routines for Output, text:

| | |
|---|---|
| PWRT.INT | Calls P$PUTT, P$PUTERR<br>This program has the following entry points:<br>P$WRITBY, P$WRITSI, P$WRITI |
| P$WRITSR | Calls P$WRITLN, P$PUTERR, .FTOA |
| P$WRITR | Calls P$PUTERR, .DTOA, P$WRITLN |
| P$WRITCH | Calls P$PUTT, P$PUTERR |
| P$WRITB | Calls P$WRITS |
| P$WRITS | Calls P$PUTT |
| P$PAGE | Calls P$PURGE, P$PUTT, P$WRITLN |
| P$PURGE | Calls P$WRITLN |
| P$PUTT | Calls P$PUTERR, P$WRITLN |
| P$WRITLN | Calls P$$SVC1, P$PUTERR, P$PAUS |

I/O programs common to text and ordinary files:

| | |
|---|---|
| P$REWRIT | Calls P$$REWD, P$FCBERR, P$PAUS |
| P$IFCB | Calls P$EFCB, P$$SVC7, P$PAUS, P$SEND, P$TERM |
| P$EFCB | |
| P$CLOSE | Calls P$$SVC7, P$SEND |
| P$$REWD | Calls P$$SVC1, P$PAUS |

I/O error servicing routines:

| | |
|---|---|
| P$FCBERR | Calls P$$SVC7, P$SEND |
| P$$SVC1 | Calls P$SEND |
| P$$SVC7 | Calls P$SEND |
| P$GETER1 | Calls P$PAUS, P$SEND, P$TERM |
| P$GETER2 | Calls P$PAUS, P$SEND, P$TERM |
| P$PUTERR | Calls P$PAUS, P$SEND, P$TERM |
| P$NUMERR | Calls P$SEND |

Duplicates on PASRTL.OBJ of object programs from FORTRAN VII RTL:

| | |
|---|---|
| .ATOD | |
| .DTOA | |
| .ATOF | |
| .FTOA | |
| .INITSCA | Calls .CPLUB |
| .CPLUB | |

# APPENDIX M
# PASCAL-RELIANCE ENVIRONMENT INFORMATION

Pascal R01 supports features which make it possible to write applications programs in Pascal to run in the environment of RELIANCE II. This appendix describes how a RELIANCE application program can be written in Pascal and prepared for running.

## M1. Documentation

For an overview of the RELIANCE II system, the programmer should read the introductory documentation in these manuals:

- Reliance Overview, Publication Number 29-718
- Integrated Transaction Controller (ITC) Introduction, Publication Number 29-710
- Data Management System/32 (DMS/32) Introduction, Publication Number 29-714.

A RELIANCE application program written in Pascal should make use of the FORTRAN interfaces to the RELIANCE system. These interfaces are described in the following manuals:

- Integrated Transaction Controller (ITC) FORTRAN Programmers Reference Manual, Publication Number 48-044
- Data Management System/32 (DMS/32) FORTRAN Programmers Reference Manual, Publication Number 48-045.

In this appendix, these two manuals will be referred to as the "interface manuals."

Other manuals which describe the RELIANCE system include:

- Reliance Sample Application, Publication Number 29-713
- Integrated Transaction Controller (ITC) Systems Programming and Operations Manual, Publication Number 29-712
- Data Mangement System/32 (DMS/32) Systems Programming and Operations Manual, Publication Number 29-717
- Reliance Operator's Reference Summary, Publication Number 29-722.

## M2. Restrictions

In the RELIANCE system, it is assumed that each application program is short and simple. In particular, each program deals with at most one message from a terminal and one message to the terminal. Communication with the terminal should be done only through the ITC system. The Pascal applications program should make calls to FORTRAN external programs which in turn call the FORTRAN - RELIANCE interface. The details of setting this up are described in part M3 of this appendix.

For communication with a shared data base, the DMS software should be used. Pascal applications programs should make calls to FORTRAN external programs which in turn call the FORTRAN - DMS interface. Again, the details are described in part M3.

It is possible for the applications program to do I/O through files that are not managed by the RELIANCE system. Certain restrictions are imposed. Programs running under ITC must not use logical units 4 through 10 inclusive. If a program does use I/O of its own, then its logical units must be closed before the program terminates.

The normal way to terminate a RELIANCE application program is a call to RPROG or RPUTMSG. In Pascal, these calls should be explicitly programmed. If the flow of control reaches the end of a Pascal applications program, then a call to RPROG will be made, with the effect of displaying the screen-form "SYSTEM".

Abnormal termination of a Pascal application program occurs in any condition where, running in an OS/32 environment, a program would log an error message. In a RELIANCE environment, the error message is written to the journal. The application program is terminated at once. An attribute of the screen form belonging to the program tells the system whether failure of this transaction could have corrupted the data base. If it could, then upon abnormal termination the system is "quiesced."

The RTL routine SVC2PAUS has the same effect as an abnormal termination of the RELIANCE application program.

## M3. Techniques for calling the FORTRAN - RELIANCE interface.

The arguments which are accepted by the routines in the FORTRAN - RELIANCE interface are described in the "interface manuals" (see M1). These routines should not be called directly from Pascal programs, because they take advantage of special properties of the FORTRAN interface. To make use of these routines, the programmer can write "connector" subprograms in FORTRAN. The Pascal code calls the connectors, and the connectors call the RELIANCE software.

In calling the ITC system, Pascal experiences difficulty with what, in FORTRAN, are called CHARACTER data. Under FORTRAN VII, a CHARACTER parameter is passed as two parameters, first the array of characters and then the length. To accomodate this, the Pascal code must pass both arguments explicitly.

Here is an example of how a FORTRAN "connector" can be used to tailor the FORTRAN - RELIANCE interface to a specific application. As described in the "interface manual," the routine RGETVAR has five parameters; in FORTRAN, it is invoked thus:

```
      CALL RGETVAR( name,elementnumber,area,length,result)
```

The expected types of the parameters are

name: CHARACTER*12
elementnumber: INTEGER*2
area: CHARACTER(*)
length: INTEGER*4
result: Array of three elements, each INTEGER*2

The parameter length is actually the length of area; it must be passed explicitly to RGETVAR.

In writing the connector, let us suppose that we want to receive a 24-byte data field. The Pascal program may contain the following declarations and invocation:

```
TYPE
  CHAR12 = ARRAY [1..12] OF CHAR;
  CHAR24 = ARRAY [1..24] OF CHAR;
  RESULT_TYPE = ARRAY [1..3] OF SHORTINTEGER;

...

VAR
  FIELD_A: CHAR24;
  RESULT: RESULT_TYPE;

...

PROCEDURE CGETVAR(NAME:CHAR12; NAME_LENGTH: INTEGER;
                ELEMENT_NUMBER: SHORTINTEGER;
            VAR AREA: CHAR24; AREA_LENGTH: INTEGER;
            VAR RESULT: RESULT_TYPE);
        FORTRAN;
```

```
...

  CGETVAR('FIELDA      ', 12, 7, FIELD_A, 24, RESULT);
```

Note that the string passed to NAME should be 12 bytes long and padded with blanks.

The connector program may be written as follows:

```
      SUBROUTINE CGETVAR(NAME, ELNUM, AREA, RESULT)
      CHARACTER NAME*12, AREA*24
      INTEGER*2 RESULT(3), ELNUM
      CALL RGETVAR(NAME, ELNUM, AREA, LEN(AREA), RESULT)
      RETURN
      END
```

In comparing the Pascal version of the CGETVAR with the FORTRAN version, we note that the FORTRAN version appears to have fewer parameters. In fact, the parameter NAME in the FORTRAN code corresponds to NAME and NAME_LENGTH in the Pascal declaration, and the parameter AREA in the FORTRAN code corresponds to AREA and AREA_LENGTH in the Pascal. In the call to RGETVAR, the fourth argument is the length of AREA, as found by the intrinsic function LEN.

The same technique may be used to connect Pascal programs to the FORTRAN - DMS interface. For example, the subprogram RENDTRAN may be invoked thus:

CALL RENDTRAN( <u>reply</u>)

or

CALL RENDTRAN( <u>reply</u>, <u>noios</u>).

The obligatory parameter <u>reply</u> and the optional parameter <u>noios</u> may be INTEGER*2 or INTEGER*4.

For the connector, let us suppose that both parameters are desired, and both are to be INTEGER*4. Then the Pascal declaration may be

```
  PROCEDURE CENDTRAN(VAR REPLY, NOIOS: INTEGER);
            FORTRAN;
```

and the FORTRAN code for the connector may be

```
      SUBROUTINE CENDTRAN(REPLY, NOIOS)
      INTEGER*4 REPLY, NOIOS
      CALL RENDTRAN(REPLY, NOIOS)
      RETURN
      END
```

## M4. Preparing to run a RELIANCE application program

1. *Compilation*. In compiling the Pascal code for a RELIANCE application program, be sure to set the compile-time option RELIANCE, either in the start command parameter or in a comment in the code of the program.

2. *Linkage*. The "interface manuals" (see section M1) describe linkage commands that must be used to give the program access to the RELIANCE system. For a Pascal program, the Pascal RTL library, PASRTL.OBJ, should be linked in with a LIB command, or included as a shared segment. Note that an absolute storage area of X'1D00' bytes should be reserved.

3. *If the applications program uses floating-point numbers, then the ITC monitor must have been linked with the options FLOAT and DFLOAT*.

# APPENDIX N
# Perkin-Elmer PASCAL PREFIX LANGUAGE EXTENSIONS

The extended Pascal Language features provided by those routines declared in the Perkin-Elmer Pascal Prefix allow the user to code Pascal procedure-call statements to:

- Open a file or device
- Close a file or device
- Allocate a file
- Rename a file
- Reprotect a file
- Delete a file
- Change access privileges to a file or device
- Checkpoint a file or device
- Fetch the attributes of an assigned file or device
- Write a file mark
- Rewind a file
- Backspace a record
- Backspace to a file mark
- Forward space a record
- Forward space to a file mark
- Breakpoint
- Obtain Start Parameters
- Obtain the time
- Obtain the date
- Exit the program with a specified return code

The Prefix source declarations are listed collectively below and in Table 10-4 of Chapter 10, and are available on the file PREFIX.PAS.

The user may use the $INCLUDE command just prior to the PROGRAM or MODULE header to easily include the Prefix source prior to any compilation-unit(see Chapter 1). For example:

```
{$INCLUDE (voln:PREFIX.PAS)}  or  {$INCLUDE (voln:PREFIX.PAS)}
PROGRAM name(file-name-list);     MODULE name(module-param-list);
```

The object routines to support the Perkin-Elmer Prefix are contained in the Pascal Runtime Library, on the file PASRTL.OBJ, which is linked to at task establishment time.

See Section 10.3 in Chapter 10 for complete details on using the individual Prefix routines.

## Perkin-Elmer Pascal PREFIX LANGUAGE EXTENSIONS

```
{PREFIX.PAS}
CONST     CR = '(:13:)'; FF = '(:12:)';
TYPE      LUNIT = 0..255;
          IDENTIFIER = ARRAY [1..19] OF CHAR;
          FILE_TYPE = (CONTIGUOUS,INDEXED);
          ACCESS_PRIVILEGE = (SRO,ERO,SWO,EWO,SRW,SREW,ERSW,ERW);
          ATTRIBUTE_BLOCK = PACKED RECORD
                              DEVICE_CODE: SHORTINTEGER;
                              PRECL: SHORTINTEGER;
                              VOLUME_NAME: ARRAY [1..4] OF CHAR;
                              FILE_NAME: ARRAY [1..8] OF CHAR;
                              EXTENSION: ARRAY [1..3] OF CHAR;
                              FILE_CLASS: CHAR;
                              FILE_SIZE: INTEGER
                              END;
          STRING8 = ARRAY [1..8] OF CHAR;
          PARM_POINTER = ^PARM_STRING;
          PARM_STRING = ARRAY [1..132] OF CHAR;

PROCEDURE OPEN (LU:LUNIT; ID:IDENTIFIER; AP:ACCESS_PRIVILEGE;
                KEYS:SHORTINTEGER; VAR STATUS:BYTE);
PROCEDURE CLOSE (LU:LUNIT; VAR STATUS:BYTE);
PROCEDURE ALLOCATE (FT:FILE_TYPE; ID:IDENTIFIER;
                    KEYS:SHORTINTEGER;
                    SIZE,DATA_BLOCK,INDEX_BLOCK:INTEGER;
                    VAR STATUS:BYTE);
PROCEDURE RENAME (LU:LUNIT; ID:IDENTIFIER; VAR STATUS:BYTE);
PROCEDURE REPROTECT (LU:LUNIT; KEYS:SHORTINTEGER;
                     VAR STATUS:BYTE);
PROCEDURE DELETE (ID:IDENTIFIER; KEYS:SHORTINTEGER;
                  VAR STATUS:BYTE);
PROCEDURE CHANGE_ACCESS_PRIVILEGE (LU:LUNIT;
                                   AP:ACCESS_PRIVILEGE;
                                   VAR STATUS:BYTE);
PROCEDURE CHECKPOINT (LU:LUNIT; VAR STATUS:BYTE);
PROCEDURE FETCH_ATTRIBUTES (LU:LUNIT;
                            VAR BLOCK:ATTRIBUTE_BLOCK;
                            VAR STATUS:BYTE);

PROCEDURE REWIND          (LU:LUNIT; VAR STATUS:SHORTINTEGER);
PROCEDURE WRITE_FILE_MARK (LU:LUNIT; VAR STATUS:SHORTINTEGER);
PROCEDURE BACK_RECORD     (LU:LUNIT; VAR STATUS:SHORTINTEGER);
PROCEDURE BACK_FILE_MARK  (LU:LUNIT; VAR STATUS:SHORTINTEGER);
PROCEDURE FORWD_RECORD    (LU:LUNIT; VAR STATUS:SHORTINTEGER);
PROCEDURE FORWD_FILE_MARK (LU:LUNIT; VAR STATUS:SHORTINTEGER);

PROCEDURE BREAKPOINT (LN:INTEGER);
PROCEDURE START_PARMS (VAR PTR:PARM_POINTER);
PROCEDURE TIME (VAR BUFR:STRING8);
PROCEDURE DATE (VAR BUFR:STRING8);
PROCEDURE EXIT (EOT:BYTE);
```

# APPENDIX O
# PASCAL SVC SUPPORT

Perkin-Elmer Pascal provides the user with the capability to perform a basic set of operating system Supervisor Calls (SVCs) by coding Pascal procedure-call statements and utilizing predefined source interfaces for the SVC support routines, which are:

| Pascal SVC Routine Name and Support Routine Entry | OS/32 SVC Supported or System Service Request |
|---|---|
| SVC1 | SVC 1 Input/Output Request |
| SVC3 | Ends Task Execution via P$TERM |
| SVC5 | SVC 5 Fetch Overlay |
| SVC7 | SVC 7 File Handling Services |
| SVC2PAUS | Pauses Task Execution via P$PAUS |
| SVC2AFLT | SVC 2, Code 4: Set Status |
| SVC2FPTR | SVC 2, Code 5: Fetch Pointers |
| SVC2LOGM | SVC 2, Code 7: Log Message Option X'00' or X'80' |
| SVC2FTIM | SVC 2, Code 8: Interrogate Clock |
| SVC2FDAT | SVC 2, Code 9: Fetch Date |
| SVC2TODW | SVC 2, Code 10: Time of Day Wait |
| SVC2INTW | SVC 2, Code 11: Interval Wait |
| SVC2PKNM | SVC 2, Code 15: Pack ASCII numeric to binary |
| SVC2PKFD | SVC 2, Code 16: Pack File Descriptor |
| SVC2PEEK | SVC 2, Code 19: Peek |
| SVC2TMAD | SVC 2, Code 23, Option X'00' |
| SVC2TMWT | SVC 2, Code 23, Option X'80' |
| SVC2TMRP | SVC 2, Code 23, Option X'40' |
| SVC2TMLF | SVC 2, Code 23, Option X'20' |
| SVC2TMCA | SVC 2, Code 23, Option X'10' |
| SVCINITQ | Initialize Task Queue, set TSW Z bit |
| SVCTASKQ | Fetch a task queue parameter from the Task Queue |
| FROMUDL | Access UDL |
| TOUDL | Modify UDL |

These routines are contained in the Pascal Run Time Library, on the file PASRTL.OBJ, which is linked to at task establishment time.

The required predefined source interfaces are listed below, and details on invoking the SVCs are given in Section 10.4 of Chapter 10.

```
{PASCAL SVC SUPPORT CONSTANT AND TYPE-DEFINITIONS}

TYPE CHAR2 = PACKED ARRAY [1..2] OF CHAR;
TYPE CHAR3 = PACKED ARRAY [1..3] OF CHAR;
TYPE CHAR8 = PACKED ARRAY [1..8] OF CHAR;
TYPE CHAR4 = PACKED ARRAY [1..4] OF CHAR;

TYPE LINE = ARRAY [1..132] OF CHAR;

{SVC1 PARAMETER BLOCK}

TYPE SVC1_BLOCK = RECORD
      SVC1_FUNC: BYTE;            {FUNCTION CODE}
      SVC1_LU: BYTE;              {LOGICAL UNIT NUMBER}
      SVC1_STAT: BYTE;            {DEV-INDEP STATUS}
      SVC1_DEV_STAT: BYTE;        {DEV-DEPENDENT STATUS}
      SVC1_BUFSTART: INTEGER;     {ADDRESS(BUFFER)}
      SVC1_BUFEND: INTEGER;  {ADDRESS(BUFFER)+SIZE(BUFFER)-1}
      SVC1_RANDOM_ADDR: INTEGER;{RANDOM ADDRESS FOR DASD}
      SVC1_XFER_LEN: INTEGER;     {TRANSFER LENGTH}
      SVC1_RESERVED: INTEGER;     {RESERVED FOR ITAM USE}
     END;

{SVC5 PARAMETER BLOCK}

TYPE SVC5_PARM = RECORD
       SVC5_OVNAME : CHAR8;
       SVC5_STAT  :   BYTE;
       SVC5_OPT   :   BYTE;
       SVC5_LU    :   SHORTINTEGER;
     END;

{FILE DESCRIPTOR FOR SVC7 REQUESTS}

TYPE FD_TYPE = PACKED RECORD
      VOLN: CHAR4;                {VOLUME NAME}
      FN: CHAR8;                  {FILE NAME}
      EXTN: CHAR3;                {EXTENSION}
      ACCT: CHAR;                 {ACCOUNT CODE: P/S/G}
     END;

{SVC 7 PARAMETER BLOCK}

TYPE SVC7_BLOCK = RECORD
      SVC7_CMD: BYTE;             {COMMAND}
      SVC7_MOD: BYTE;             {MODIFIER/DEVICE TYPE}
      SVC7_STAT: BYTE;            {STATUS}
      SVC7_LU: BYTE;              {LOGICAL UNIT NUMBER}
      SVC7_KEYS: SHORTINTEGER;    {READ/WRITE KEYS}
      SVC7_RECLEN: SHORTINTEGER;{LOGICAL RECORD LENGTH}
      SVC7_FD: FD_TYPE;           {FILE DESCRIPTOR}
      SVC7_SIZE: INTEGER;         {FILE(/INDEX) SIZE}
     END;
```

## PASCAL SVC SUPPORT

```
{PASCAL SVC SUPPORT CONSTANT AND TYPE-DEFINITIONS}

CONST TASKQ_SLOT_COUNT = 4;
TYPE QSIZE_TY = 1..TASKQ_SLOT_COUNT;
TYPE TASKQ_TYPE = RECORD
       QSIZE: QSIZE_TY;
       FILL1, FILL2, FILL3: SHORTINTEGER;
       TASKQ_SLOTS: ARRAY[QSIZE_TY] OF INTEGER;
     END;

TYPE UDL_INDEX = 0..63;
TYPE PACK_OPTION = (USER_VOL, SYS_VOL, SPL_VOL, NO_DEFAULT);

TYPE PEEK_00_BLOCK = RECORD
       PEEK_OPT       : BYTE;
       PEEK_CODE      : BYTE;
       PEEK_NLU       : BYTE;
       PEEK_MPRI      : BYTE;
       PEEK_OSID      : CHAR8;
       PEEK_TASK_NAME : CHAR8;
       PEEK_CTSW      : INTEGER;
       PEEK_TOPT      : SHORTINTEGER;
       RESERVED       : SHORTINTEGER;
     END;

TYPE PEEK_01_BLOCK = PACKED RECORD
       PEEK_OPT       : BYTE;
       PEEK_CODE      : BYTE;
       RESERVED_1     : SHORTINTEGER;
       PEEK_OSID      : CHAR8;
       PEEK_OSUP      : CHAR2;
       PEEK_CPU       : SHORTINTEGER;
       PEEK_SOPT      : INTEGER;
       PEEK_UACT      : SHORTINTEGER;
       PEEK_GACT      : SHORTINTEGER;
       RESERVED_2     : INTEGER;
     END;

TYPE PEEK_02_BLOCK = PACKED RECORD
       PEEK_OPT       : BYTE;
       PEEK_CODE      : BYTE;
       RESERVED_3     : SHORTINTEGER;
       PEEK_OSID      : CHAR8;
       PEEK_LOAD_VOL  : CHAR4;
       PEEK_FILENAME  : CHAR8;
       PEEK_EXT       : CHAR3;
       PEEK_FILE_CLASS: CHAR;
     END;
```

# APPENDIX O (Continued)
## PASCAL SVC SUPPORT

```
PROCEDURE SVC1(VAR PARM:SVC1_BLOCK); EXTERN;

PROCEDURE SVC3(TERM_CODE : BYTE); EXTERN;

PROCEDURE SVC5(VAR PARM: SVC5_PARM); EXTERN;

PROCEDURE SVC7(VAR PARM:SVC7_BLOCK); EXTERN;

PROCEDURE SVC2PAUS; EXTERN;

PROCEDURE SVC2AFLT(ENABLE: BOOLEAN); EXTERN;

PROCEDURE SVC2FPTR; EXTERN;

PROCEDURE SVC2LOGM(MSG:LINE; LEN:INTEGER; IMAGE:BOOLEAN); EXTERN;

PROCEDURE SVC2FTIM(VAR TIME:INTEGER; VAR HHMMSS: CHAR8); EXTERN;

PROCEDURE SVC2FDAT(VAR MMDDYY:CHAR8); EXTERN;

PROCEDURE SVC2TODW(TOD : INTEGER);    EXTERN;

PROCEDURE SVC2INTW(INTVL : INTEGER);  EXTERN;

PROCEDURE SVC2PKNM(VAR VAL:INTEGER; BUF:LINE; VAR POSN:INTEGER;
                   OPT:INTEGER; VAR CC:INTEGER); EXTERN;

PROCEDURE SVC2PKFD(VAR FD:UNIV FD_TYPE; BUF:LINE;
                   VAR POSN:INTEGER; SKIP_BLANKS:BOOLEAN;
                   OPT: PACK_OPTION; VAR CC:INTEGER);  EXTERN;

PROCEDURE SVC2PEEK(VAR PARM:UNIV PEEK_00_BLOCK); EXTERN;

PROCEDURE SVC2TMAD(INTVL: INTEGER; TASKQPARM: INTEGER;
                   ELAPSED: BOOLEAN; VAR CC: INTEGER); EXTERN;

PROCEDURE SVC2TMWT(INTVL: INTEGER;
                   ELAPSED: BOOLEAN; VAR CC: INTEGER); EXTERN;

PROCEDURE SVC2TMRP(ITEMCOUNT: SHORTINTEGER; ADDRESS: INTEGER;
                   ELAPSED: BOOLEAN; VAR CC: INTEGER); EXTERN;

PROCEDURE SVC2TMLF(VAR INTVL: INTEGER; TASKQPARM: INTEGER;
                   ELAPSED: BOOLEAN; VAR CC: INTEGER); EXTERN;

PROCEDURE SCV2TMCA(TASKQPARM: INTEGER;
                   ELAPSED: BOOLEAN; VAR CC: INTEGER); EXTERN;

PROCEDURE SVCINITQ(NSLOTS:QSIZE_TY;VAR TASKQ:TASKQ_TYPE); EXTERN;
PROCEDURE SVCTASKQ(VAR PARAM:INTEGER; WAIT:BOOLEAN); EXTERN;

PROCEDURE FROMUDL(I: UDL_INDEX; VAR VAL: UNIV INTEGER); EXTERN;
PROCEDURE TOUDL(I: UDL_INDEX; VAL: UNIV INTEGER); EXTERN;
```

# APPENDIX P
# PASCAL R01 FUNCTIONAL DIFFERENCES FROM R00

## 1. Language changes

Procedure and function names are accepted as parameters of other procedures and functions. The syntax is in accord with the forthcoming ANSI/ISO standard.

The PACKED attribute affects the storage of data in arrays and records to which it is applied.

Subrange-types and set-types no longer achieve identity of type any differently than other types. Identity of type, as required when passing variables to variable-parameters, is enforced.

## 2. New predefined routines

DISPOSE is introduced. When a dynamically allocated variable is DISPOSEd of, the space it occupied becomes available for later use.

STACKSPACE is introduced, so that a program can find out how much space is available.

The standard function ORD accepts pointer variables as arguments.

The procedure MARK actually creates a valid heap item.

## 3. Management of source code

The BATCH and BEND options allow several compilation units to be compiled together.

The INCLUDE option allows source to be taken from several files.

## 4. Interfaces

The interface for calling EXTERN routines has been simplified. (NOTE: This may affect external routines which have been written in assembly language.)

The interface of "prefix" routines has been made identical with that of EXTERN routines. In particular, prefix routines no longer need to be in a fixed order. New prefix routines can be added without changing any tables in the run time library.

Compilation units compiled with Pascal R01 should not be linked with compilation units compiled with Pascal R00. Therefore, in a system of a main program and several modules, previously compiled with Pascal R00, if one part is recompiled then the others should be recompiled as well.

## 5. Run time library

The run time library for R01 has been made modular, so that each task need only contain the routines that it actually calls. Code for the run time library is pure. All position dependencies have been removed.

The PASSVC package is included in the run-time library and has been changed by the addition of several routines to take the place of one routine having multiple uses. See Chapter 10 for details.

In the predefined Prefix, the names of certain routines have been changed so as to make each name unique within the first eight characters. Only one file, PREFIX.PAS, containing Prefix source declarations without R00's descriptive comments, is packaged. See Chapter 10 or Appendix L for details.

When a routine defined with the directive FORTRAN is called, the run time library prepares the task's memory space to be fully compatible with the FORTRAN run time system.

Compilation under the option "RELIANCE", produces compiled code and run time support compatible with the RELIANCE system.

Outputting REAL/SHOPTREAL values to textfile fields have had their external floating-point representations' default total field widths changed. Pascal R01 P$WRITR/P$WRITSR default total field widths for REALs to 24 positions, and SHORTREALs to 14 positions; unlike Pascal R00 which defaulted both REAL and SHORTREAL to 8 positions; allowing output fields up to a maximum field width of only 20 max for REALs or 14 max for SHORTREALs whenever a field-width > max was specified to produce this type of format.

## 6. New compiler options

BATCH and BEND are supported, as is INCLUDE (see Section 3, above).

RELIANCE renders a program compatible with the RELIANCE system (see Section 5, above).

HEAPMARK is required to use the predefined routines MARK and RELEASE.

BOUNDSCHECK, MEMLIMIT, EJECT, and LOG are introduced.

The compiler start options may be given to the R01 compiler via the OS START command, or through the Pascal CSS's in small letters, such as "ba as su ma log", not only in capital letters, such as was required in Pascal R00, e.g. "AS NCR SU MA NRA".

## 7. Listings

The source listing now incorporates line numbers showing a line's position in a batch compilation and on an included file.

The assembly listing includes PROG, ENTRY, and EXTRN instructions.

## 8. Pascal CSS's

The Pascal R01 CSS's default minimim workspace to X'624' to allow X'600' (or 1536 bytes) for the RTL Scratchpad and X'24' (or 36 bytes) for the Pascal SDA minimum additional workspace in the task being established to start running Pascal R01 compiled-code.

Also changed, is the LIBRARY command to link to PASRTL.OBJ as the Pascal RTL is all on one file, and only those routines requiring external reference resolution are obtained for the task.

# INDEX

## PUBLICATION COMMENT FORM

We try to make our publications easy to understand and free of errors. Our users are an integral source of information for improving future revisions. Please use this postage paid form to send us comments, corrections, suggestions, ect.

1. Publication number________________________________________

2. Title of publication________________________________________

3. Describe, providing page numbers, any technical errors you found. Attach additional sheet if neccessary.

   __________________________________________________

   __________________________________________________

   __________________________________________________

4. Was the publication easy to understand? If not, why?

   __________________________________________________

5. Were illustrations adequate? ______________________________

   __________________________________________________

   __________________________________________________

6. What additions or deletions would you suggest? ____________________

   __________________________________________________

   __________________________________________________

7. Other comments: ________________________________________

   __________________________________________________

   __________________________________________________

From ______________________________ Date ______________________

Position/Title ________________________

Company _____________________________

Address ______________________________

______________________________

______________________________

STAPLE STAPLE

FOLD FOLD

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

**BUSINESS REPLY MAIL**

FIRST CLASS PERMIT NO. 22 OCEANPORT, N.J.

POSTAGE WILL BE PAID BY ADDRESSEE

**PERKIN-ELMER**

**Data Systems Group**
106 Apple Street
Tinton Falls, NJ 07724

**ATTN:**
**TECHNICAL SYSTEMS PUBLICATIONS DEPT.**

FOLD FOLD

STAPLE STAPLE