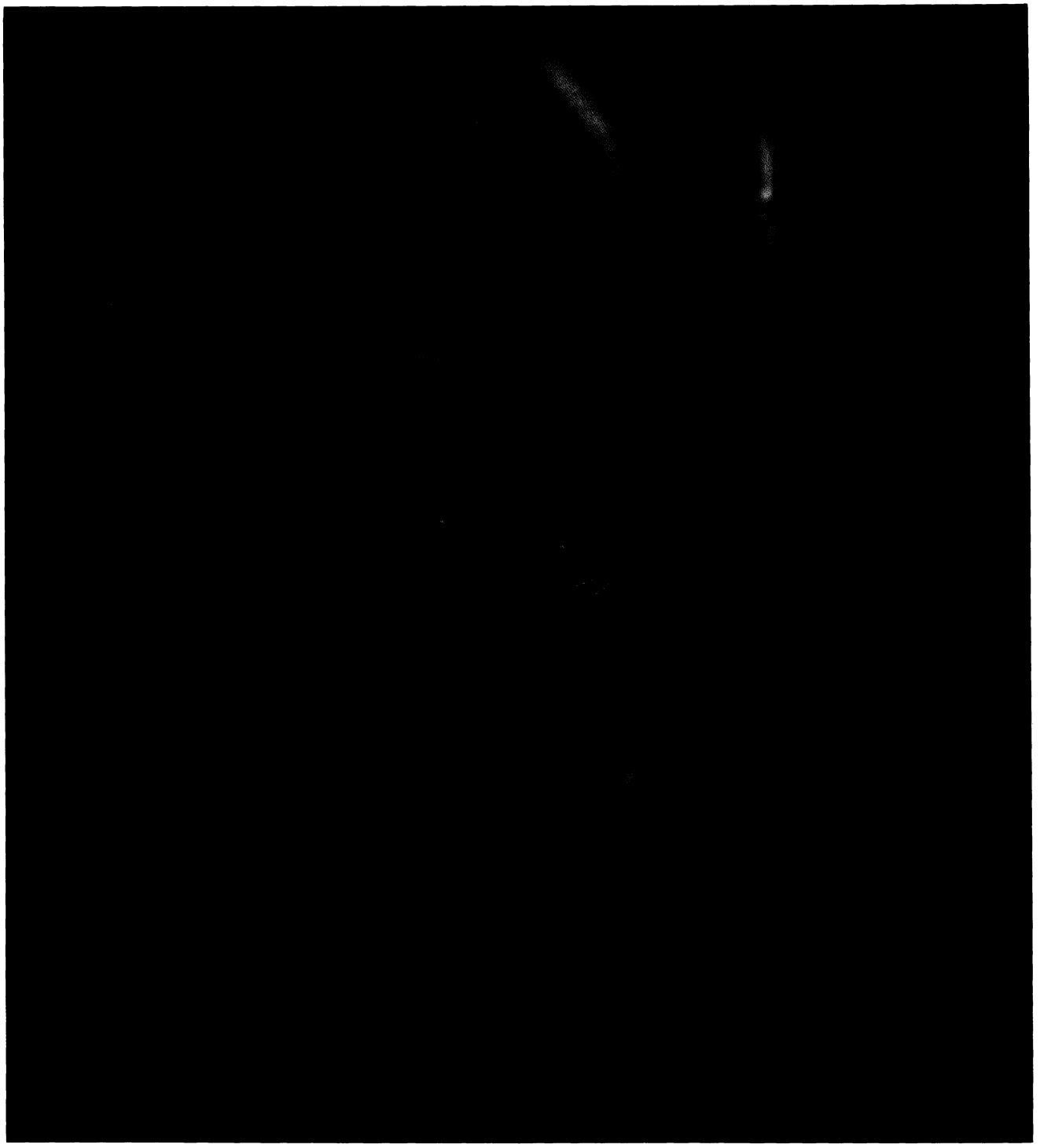
SP/Pascal

Programmer's Reference



DataGeneral

NOTICE

Data General Corporation (DGC) has prepared this document for use by DGC personnel, customers, and prospective customers. The information contained herein shall not be reproduced in whole or in part without DGC's prior written approval.

DGC reserves the right to make changes in specifications and other information contained in this document without prior notice, and the reader should in all cases consult DGC to determine whether any such changes have been made.

THE TERMS AND CONDITIONS GOVERNING THE SALE OF DGC HARDWARE PRODUCTS AND THE LICENSING OF DGC SOFTWARE CONSIST SOLELY OF THOSE SET FORTH IN THE WRITTEN CONTRACTS BETWEEN DGC AND ITS CUSTOMERS. NO REPRESENTATION OR OTHER AFFIRMATION OF FACT CONTAINED IN THIS DOCUMENT INCLUDING BUT NOT LIMITED TO STATEMENTS REGARDING CAPACITY, RESPONSE-TIME PERFORMANCE, SUITABILITY FOR USE OR PERFORMANCE OF PRODUCTS DESCRIBED HEREIN SHALL BE DEEMED TO BE A WARRANTY BY DGC FOR ANY PURPOSE, OR GIVE RISE TO ANY LIABILITY OF DGC WHATSOEVER.

IN NO EVENT SHALL DGC BE LIABLE FOR ANY INCIDENTAL, INDIRECT, SPECIAL OR CONSEQUENTIAL DAMAGES WHATSOEVER (INCLUDING BUT NOT LIMITED TO LOST PROFITS) ARISING OUT OF OR RELATED TO THIS DOCUMENT OR THE INFORMATION CONTAINED IN IT, EVEN IF DGC HAS BEEN ADVISED, KNEW OR SHOULD HAVE KNOWN OF THE POSSIBILITY OF SUCH DAMAGES.

Ordering No. 069-400203

Rev. 00, June 1982

Preface

This manual documents an extended Pascal for system programmers. SP/Pascal has all of the features of MP/Pascal as well as extensions to implement the MP/AOS and AOS operating systems. Chapter 1 provides an overview of the entire language.

Chapters 2 through 5 describe the language conventions and syntax.

Chapters 6 and 7 describe SP/Pascal routines and I/O, followed by the rules on program structure in Chapter 8.

Chapters 9 and 10 describe predefined and external routines, respectively.

Chapter 11 covers the new SP/Pascal exception handling feature. Chapter 12 reviews the multitasking features.

Chapter 13 describes the CLI commands and macros that invoke the compiler and Binder, and that execute your programs.

A series of appendices provide information on error messages, interfacing to other languages, using the system call translator under AOS, comparisons with other Data General versions of Pascal, and other useful reference information.

An Index follows the appendices.

In addition, the following forms appear at the back of the book:

DG Offices: list of all Data General facilities world-wide.

How to Order Technical Publications: provides the addresses and phone numbers of agencies from which order forms and manuals can be obtained.

Technical Products Publications Comment Form: invites you to assist DGC in improving future publications by evaluating this manual.

Users' Group Membership Form: brings DGC software users together, with group meetings and publications, to exchange ideas, applications, problems, and solutions.

Related Manuals

The following manuals also belong to the series of books published on the MP/AOS operating system.

MP/AOS Concepts and Facilities (DGC No. 069-400200) provides a concise but thorough introduction to the MP/AOS operating system for users who want to assess the system's advantages.

MP/AOS System Programmer's Reference (DGC No. 093-400051) documents MP/AOS system structure and provides a complete dictionary of system calls and library routines.

MP/AOS Command Line Interpreter (CLI) (DGC No.069-400201) describes the interactive CLI program, the user's primary interface to the MP/AOS system. A command dictionary provides command descriptions, formats, and examples.

Loading MP/AOS (DGC No. 069-400207) describes how to install MP/AOS software on ECLIPSE-line computers and how to load tailored systems.

MP/AOS System Generation and Related Utilities (DGC No. 069-400206) describes the generation of an MP/AOS system tailored to specific applications. It also describes the following utilities, including sample dialogues as appropriate:

- SYSGEN, the interactive system generation utility;
- DINIT, the disk initializer;
- FIXUP, the disk repair utility;
- SPOOLER, which controls line printer operations;
- ELOG (error logger), the utility for interpreting the system log file.

MP/AOS Debugger and Performance Monitoring Utilities (DGC No. 069-400205) describes the following utilities, providing a dictionary of debugger commands and sample dialogues as appropriate:

- FLIT, the process debugger;
- PROFILE, which measures execution-time performance;
- OPM, the process monitor that displays current system resource allocation and status.

MP/AOS Macroassembler, Binder, and Library Utilities (DGC 069-400210) documents the MP/AOS macroassembler and binder as well as the library file editor (LED) and system cross-reference analyzer (SCAN). It includes programming examples and a dictionary of assembler pseudo-ops.

MP/AOS Advanced Program Development Utilities (DGC 069-400208) describes the following utilities:

- Text control system (TCS), a method for managing different versions of a single file;
- BUILD, which creates a new version of a file from existing files, thus minimizing effort and errors in program development;
- FIND, which locates occurrences of strings in text files.

MP/AOS SPEED Text Editor (DGC No. 069-400202) documents the features of SPEED, the MP/AOS character-oriented text editor.

MP/AOS SLATE Text Editor (DGC 069-400209) documents the features of SLATE, a screen- and line-oriented text editor.

MP/AOS File Utilities (DGC No. 069-400204) describes the following utility programs, providing sample dialogues for each:

- FEDIT, a file editor that permits modification of system files, including program and data files;
- FDISP, which can display the address and data contents of a file or compare two files, displaying the parts that differ;
- SCMP, which can compare two source programs line by line;
- MOVE, which allows the transfer of files among directories;
- AOSMIC, which allows manipulation of MP/AOS and MP/OS disks and files on an AOS system;
- FOXFIRE, which permits the transfer of files among MP/OS, MP/AOS, and AOS systems over asynchronous communication lines.

Books on three additional programming languages supported by MP/AOS have previously been published as part of the bookset for the MP/OS operating system:

MP/Pascal Programmer's Reference (DGC No. 069-400031) documents for system programmers a Pascal-based language targeted for the MP/OS operating system.

MP/FORTRAN IV Programmer's Reference (DGC No. 069-400033) documents for system programmers a language based on ANSI 1966 standard FORTRAN with extensions.

MP/Basic Programmer's Reference (DGC No. 069-400032) documents for new users a programming language based on ANSI standard Basic with extensions.

MP/OS

For information on Microproducts and a bibliography of documentation on the Microproducts line, see *Introduction to Microproducts* (DGC No. 014-000685).

For information on cross development between MP/OS and MP/AOS, see *MP/OS System Programmer's Reference* (DGC No. 093-400001).

Conventions and Abbreviations

Throughout this manual we use the following conventions to illustrate instruction formats:

COMMAND	Uppercase letters in THIS typeface indicate an instruction mnemonic. You type an instruction mnemonic exactly as it appears.
argument	Lowercase italic letters represent a command's argument. You must replace this symbol with the exact code for the argument you need.
[optional]	Brackets denote an optional argument (Command switches appear in this format as well.) If you use this argument or switch, do not type the brackets into your command line: they only set off the choice.
CTRL-	Depress and hold the Control key while you press the character following CTRL-.
arg1| *arg2*	Denotes either *arg1* or *arg2*.
`EXAMPLE LINE`	Uppercase letters in `THIS TYPEFACE` are used for programming examples.
RESPONSE	If the program can respond to the command in the example the response is shown in uppercase letters in ***THIS TYPEFACE***.

Contents

4. Expressions

5. Program Statements

6. SP/Pascal Routines

7. Input/Output

8. SP/Pascal Program Structure

9. Predefined Routines

10. External Routines Supplied by DGC

Figures

Tables

Introduction

This manual describes SP/Pascal, a Pascal compiler that runs under the MP/AOS operating system on Data General's ECLIPSE computers. SP/Pascal combines the elegant structure of the original Pascal language with a number of enhancements that greatly improve its usefulness in advanced "real-world" applications.

History of Pascal

The Pascal language was created by Dr. Niklaus Wirth at the *Institut fur Informatik* in Switzerland. Wirth named the language for Blaise Pascal (1623 - 1662), the French philosopher, mathematician, and inventor of the adding machine. First published in 1971, Pascal was quickly accepted in both the academic and business worlds. More than a decade later, it is still widely regarded as the state of the art in general purpose programming languages. It is available on everything from 8-bit microprocessors to huge mainframes, including most Data General computers.

Pascal is conceptually based on ALGOL 60, from which it obtains features such as block structuring, explicit declaration of all variables, and efficient control statements such as IF-THEN-ELSE and REPEAT-UNTIL. Pascal expands on ALGOL in ways that are harmonious with the basic structure, not tacked on haphazardly. Its most important improvements are its data structuring and input/output facilities.

Language Features

Data General's SP/Pascal language embodies a number of extensions to standard Pascal. These extensions make the language more powerful and easier to use. Some of the most important features are listed below.

- A program may be divided into a number of source files which can be compiled separately. This is especially important when several people are working concurrently on a large program.
- An exception handling feature allows all types of error conditions to be trapped by the program and handled in an orderly manner. The EXCEPTION keyword defines a statement, or series of statements, that are executed whenever an error occurs within a specified part of the program. The built-in ERROR_CODE function allows the program to find the specific cause of an error.
- A built-in STRING data type provides for efficient text processing.
- SP/Pascal routines can easily call, and be called by, routines written in other languages, including assembler language.
- A RECAST facility allows the programmer to bypass Pascal's normal strict type-checking of data; for example, to do arithmetic with pointers (memory addresses) by recasting them as integers.

Advanced Support

In addition to the above, SP/Pascal has a number of special features that give the programmer detailed control over the program's use of memory and other system resources. The SP/Pascal programmer has access to virtually all the features of the ECLIPSE hardware and the MP/AOS operating system. In many cases, this power eliminates the need to write programs in assembler language, since programs can have the run-time efficiency of assembler as well as the "programmer-time" efficiency of Pascal. In fact, most MP/AOS

system utilities are written in SP/Pascal. Some key features are listed below.

- SP/Pascal supports multitasking, to simplify programs that must "do several things at once."
- A ZREL keyword allows data to be placed in page zero of the ECLIPSE memory. (Page zero can often be accessed more efficiently than other memory.)
- A WHOLE data type supports unsigned arithmetic on 16-bit words.
- An OVERLAY facility allows the creation of programs that are too large to fit into main memory.
- Built-in functions are provided for allocating and releasing blocks of memory at run time.
- A built-in SYSTEM routine gives the program direct access to all functions of the MP/AOS operating system.

Compiler Features

The SP/Pascal compiler is a state-of-the-art program that generates very efficient machine code. It performs many types of optimization, and exploits the entire ECLIPSE instruction set in order to make the compiled program small and fast. The generated code is in the form of a relocatable object module, which is then processed by the MP/AOS Binder to produce an executable program. The Binder can combine the program with other routines that are written either in SP/Pascal or other Data General languages.

In addition to the object module, the compiler can produce a listing of the source program, with messages to indicate any errors that are detected. Optionally, the compiler can also produce a listing of the machine instructions that comprise the generated code. This can be very useful when debugging a program at the instruction level.

Operating Environment

The SP/Pascal compiler will run on any ECLIPSE line computer that supports the MP/AOS operating system. The computer must contain the floating-point and character instruction set options.

Since the MP/AOS operating system is a functional subset of AOS, SP/Pascal programs can be developed and run under AOS with the MP/AOS system call translator. A program can be moved from one system to the other with no modifications, although it must be re-bound with the AOS or MP/AOS Binder.

MP/Pascal, which runs on Data General's MP/series computers, is a functional subset of SP/Pascal. MP/Pascal programs can be compiled by SP/Pascal, and transported to OZMOS or AOS systems, with little or no modification.

Lexical Structure

This chapter introduces the basic elements of the SP/Pascal language. It describes the character set and the basic symbols that make up an SP/Pascal program.

Character Set

All SP/Pascal statements must be made up from the following standard and special characters. Characters not specified here are illegal and are not accepted by the compiler, except as part of a character or string constant.

Standard Character Set

The standard ASCII character set for SP/Pascal includes the following alphanumeric characters:

A - Z, a - z, 0 - 9, $, ?, _ (underscore)

In general, the compiler treats upper-case and lower-case characters as equivalent; i.e., WHILE, while, and While are equivalent. There is only one exception to this case indifference: all characters in string literals represent themselves. Appendix A lists the complete ASCII character set.

Special Symbols

Special symbols within the character set serve as delimiters and operators. The special SP/Pascal symbols are listed below. Their uses are explained and illustrated in the chapters that follow.

+	≪	(	@
—	(*	)	^
*	*)	[	Blank
/	:=	]	Tab
=	.	{	Form Feed
<>	..	}	Carriage Return
<	, (comma)	≫	New-Line
>	;	%	
<= or =<	:		
>= or =>	' (quote)		

Table 2.1 Special MP/Pascal symbols

Reserved Words

Certain *reserved* words in the language have fixed meanings and can be used only as defined. For instance, you cannot use reserved words as identifiers. The reserved words for SP/Pascal are as follows:

AND	END	MOD	REPEAT
ARRAY	*ERETURN	*MODULE	*RETURN
*ASSEMBLY	*EXCEPTION	NIL	SET
BEGIN	*EXITLOOP	NOT	THEN
*BIT	*EXTERNAL	OF	TO
CASE	FILE	OR	TYPE
*CLRE	FOR	*OTHERWISE	UNTIL
CONST	FORWARD	*OVERLAY	VAR
DIV	FUNCTION	PACKED	WHILE
DO	IF	PROCEDURE	WITH
DOWNTO	IN	PROGRAM	*ZREL
ELSE	*INCLUDE	*RECAST	
*ENTRY		RECORD	

Table 2.2 SP/Pascal reserved words

**These words are DGC extensions to standard Pascal.*

Identifiers

An identifier is a name that denotes either a variable, a type, a procedure, a function, a program, a module or a constant. An identifier must be unique within its scope of validity.

There are two kinds of identifiers:

- predefined identifiers that have a specific definition in the language, but that, unlike reserved words, can be assigned a different meaning by the user;
- user-defined identifiers that you specify and that are specific to your program.

Predefined Identifiers

The predefined identifiers name routines, constants, files, and data types. If you wish, you can redefine any identifier as something specific to your application. Some examples of predefined identifiers:

```
ARCTAN, EOLN, FALSE, INTEGER, READ, TEXT
```

User-Defined Identifiers

Identifiers can be composed of upper-case or lower-case alphabetics, digits, underscores (_), question marks (?) and dollar signs ($). (Note that the question-mark character is used extensively as a character in system identifiers; to prevent conflict, avoid using it in a user-defined identifier.)

An identifier must begin with an alphabetic; decimal digits can be used but not as the first character. Imbedded blanks are not permitted in identifiers. The compiler treats the characters ?, $, and _ (underscore) as letters. For example, SEMI_TOTAL, A123456, and SETUP$CALL could be used as legal identifiers.

An SP/Pascal identifier can be up to 135 characters long. The first ten characters of externally visible routine, module, and external variable names and the first five characters of external assembly language routine names must be unique. (The SP/Pascal compiler's L option can be used to specify truncation of an externally visible identifier. Refer to Chapter 13, "Operating Procedures.")

Delimiters

Identifiers, reserved words and arithmetic constants in a program must be separated by one or more delimiters. Legal delimiters include the Blank, Tab, Form Feed, Carriage Return and New Line characters. All delimiters are equivalent, except within string literals where all characters represent themselves. A comment is equivalent to a single delimiter.

Comments

The SP/Pascal compiler provides for explanatory text about any statement in the program. A comment can contain any legal ASCII characters except the comment delimiter. Comments in a program can be of any length.

The SP/Pascal compiler accepts comments that are pair-delimited, either by curly braces ({ }) or by asterisks within parentheses ((**)), as follows:

{*comment*}, or

(**comment***)

A comment delimited by curly braces can be nested within a comment delimited by asterisks and parentheses, and *vice versa*. But you cannot nest braces within braces or asterisks within asterisks. For example:

Legal nesting: (* { } *)

Illegal nesting: { { } }

SP/Pascal introduces the percent sign (%) as a single comment delimiter. The percent sign indicates that the rest of the line is a comment. A comment delimited by a percent sign can follow code on a line or can stand alone on a line. For example, both of the following two lines are legal.

```
IF STATUS <> 0 THEN RETURN; %Check for success
% Generate summary report after file analysis.
```

NOTE: *A comment can also be used as a compiler directive instructing the compiler with various options from within the source program. This option is described with the other compiler options in Chapter 13.*

Data Declarations

This chapter describes the methods for declaring data in an SP/Pascal program. SP/Pascal provides a variety of ways to define constants, data types, and variables. It provides some features, such as array and record constants, that are a significant enhancement over the standard Pascal language.

Any identifier that is not a reserved or predefined word must be declared in the program's *declaration section* before the identifier can be used in the program. The declaration section of a typical program includes constant, variable, and type declarations. Additionally, user-written routines (procedures and functions) must be defined in the declaration section before they can be called in the executable portion of the program. (User-defined routines are described in Chapter 6.) This section defines the various language conventions and rules for SP/Pascal declarations.

Any number of declarations can appear in any order. For example, you can alternate several variable, constant, and type declarations. However, when declarations of the same kind are grouped together, the program is easier to read and understand, and documentation comments can be reduced.

SP/Pascal permits any declaration to occur in several places in a source program, provided that they are all equivalent. This feature is a convenience when assembling a number of source files into a single large program. For more information on this feature, see "Benign Redefinition" at the end of this chapter.

Constant Declarations

A constant declaration introduces an identifier and assigns to it a known value that cannot be redefined. The SP/Pascal syntax allows scalar expressions to be used in constant definitions. However, such expressions must be *compile-time evaluable;* i.e., all operands used in the expression must be defined before use. (Compile-time evaluation of expressions involving real constants is not supported.)

Additionally, SP/Pascal allows structured constants for use as constants or literals.

The form of a constant declaration is:

```
CONST
    identifier = expression;
    [...identifier = expression;]
```

The value for a constant can be numeric (integer, whole, real, double_real), non-numeric (character, string, Boolean, pointer), or a user-defined or structured value or expression. Following are some examples of constant declarations.

```
CONST
LETTER = 'A';
MINUS1 = -1;
FACTOR = SUM/12;
 BLANK = ' ';
    PI = 3.14159262;
  MASK = 0FFFFR16;
```

Once you declare it, you can use the constant identifier to denote the named literal value in your program. (The only limitation is the scope of the identifier when you do not declare it as global to the entire program. Scope is detailed in Chapter 6.) The value of the identifier cannot be changed throughout the program.

Numeric Constants

SP/Pascal numeric constants can represent integer and real numbers. Integers can be either signed or unsigned (WHOLE). Real numbers can be either single or double precision.

NOTE: *The underscore character may be used for clarity in numeric constants: for example, you can write one million as 1_000_000, rather than 1000000. The underscore has no effect on the value of a numeric constant.*

Integer Constants

Integer constants represent signed values ranging from -32,768 to +32,767. Decimal points and commas are illegal in integer numbers; the plus sign is optional; the minus sign is required for negative numbers. The values for integer constants can be entered in radices of 2 through 16. The decimal radix is the default.

An integer constant is written as a signed string of decimal digits. The following are valid integer constants:

```
+3256        017632R8
-1           111_0000R2
+40          MAXINT
```

The predefined or standard constant MAXINT denotes the system's largest available integer constant. You need not declare it before using it. For SP/Pascal, MAXINT is defined as +32,767.

Whole Constants

Whole constants represent unsigned values in a range from 0 to 65,535. The whole type is provided to allow the generation of unsigned arithmetic and comparison operations. Specific rules for using it are detailed further on, under "Predefined Simple Data Types".

Binary Constants

Binary constants can be used in place of integers, whole constants, or literals. The range of binary constants is the same as the range of integers and wholes. Binary constants consist of one to 16 binary digits followed by R2 or r2. For example, 1011_1111_0010R2.

Octal Constants

Octal constants can be used in place of integers, whole constants, or literals. The range of octal constants is the same as the range of integers and wholes. Octal constants consist of one to six octal digits followed by R8 or r8. For example, 17777R8.

Hexadecimal Constants

Hexadecimal constants can be used in place of integers, whole constants, or literals. The range of hexadecimal constants is the same as the range of integers and wholes. Hexadecimal constants consist of one to four hexadecimal digits followed by R16 or r16. For example, 12BR16.

Note that the hexadecimal digits are 0 through 9 and A through F. The first character of a hexadecimal constant must be numeric. For example, FFFFR16 is an identifier; 0FFFFR16 is a numeric literal.

Real Constants

An SP/Pascal real number can assume absolute values between approximately zero and 7.2 x 10^{75}; they are precise to approximately seven decimal places. The smallest positive real value is approximately 5.4 x 10^{-79}. Double-precision numbers (double_real type) have the same ranges, but have a greater precision (to 15 decimal places).

A real number must contain either a decimal point, an exponent, or both. Optionally, the number can be preceded by a sign. A large real number can be entered in scientific (E) notation as a floating-point number. For example, the following are legal real numbers:

```
1.500          77.876503
0.003          123_456.0
3.0            8E4
65E-4
```

E notation can be used with an exponent field directly after the last digit. The exponent field, denoted by E or e, multiplies the number to its left by the power of 10 specified by the integer to the right of the E. The exponent field is an optionally signed integer.

Real number syntax is one of the following:

[+|−]*d[d...d].d[d...d]*

[+|−]*d[d...d][.d[d...d]*]E[+|−]*d[d...d]*

Note that, if the E notation is not used, the decimal point (.) is required. The point must be both preceded and followed by digits. The point is optional if E notation is used.

All real constants are converted to double-precision values by the compiler. However, in arithmetic operations, real constants are coerced to the precision of the other operands in the expression to allow single-precision operations to use real constants. Compile-time evaluation of expressions involving real constants is not supported.

Double_Real Constants

SP/Pascal introduces a new predefined numeric data type, double_real, for double-precision (four-word) quantities. The double_real value range is the same as that for real (0 to 7.2 x 10^{75}); however, a double_real value is precise to 15 rather than 7 decimal places. The rules for writing double_real constants are the same as for real constants.

Non-Numeric Constants

Non-numeric constants permitted by SP/Pascal include character, string, Boolean, and pointer constants. These are described in the following subsections. (SP/Pascal structured constants are detailed under the subhead, "Structured Data Types," further on in this chapter.)

Character Constants

A character constant must be enclosed in single quotes ('). The character can be any single printing character (plus blank) in the ASCII-7 character set. By enclosing the relevant octal code in angle brackets, you can include carriage control and other non-printing characters as character constants. For example, '<12>' represents New Line. Within character constants, two consecutive single quotes are used to represent the single quote character; for example, '''' represents a character constant with a value of the single quote. With angle-bracket notation, you can generate any eight-bit character.

The following examples illustrate character constants:

```
CONST
  ZERO = '0';
  QUOTE = '''';
  NEW_LINE = '<12>';
  ODD_PARITY_NL = '<212>';
```

When the characters between the angle brackets are not octal digits, or when the converted value of the octal digit sequence exceeds the maximum character value (377R8), the entire sequence of characters between the angle bracket characters (including the angle brackets themselves) is treated as a string literal. Each of the illegal character constants in the following actually would be defined in SP/Pascal as a string constant.

```
CONST
  ODD_PARITY_FF = '<200R8 + 14Ri>'; {illegal: expressions not allowed}
  NINE = '<9>';  {illegal: non-octal digit}
  TOO_BIG = '<400>'; {illegal: outside range of CHAR}
  RUBOUT = 177R8;
  DELETE = '<RUBOUT>'; {illegal: symbolic values not allowed}
  NOT_EQUAL = '< >'; {illegal: no octal value supplied}
```

String Constants

You represent string constants by enclosing a group of characters in single quotes. All characters within a string constant are significant. To insert special characters (the single quote and non-printing characters), use the method described under "Character Constants," above.

The maximum length of a string constant is 132 characters. String constants containing only a single character are considered to be of type CHAR rather than type string.

The compiler generates all string constants so that they terminate with at least one null character; the null character is not counted in the length of the string. Examples of string and character constants follow.

```
CONST
  SAMPLE = 'YELLOW';
  MESSAGE = 'IT''S TOO LATE!<12>';
  LIGHT = 'GO';
  TAB = '<11>';
  EMPTY = '';
```

NOTE: *For benign redefinition, string literals are equivalent when they are textually identical. Case is significant in string and character literals.*

Boolean Constants

The values *true* and *false* are predefined Boolean constants. For example,

```
CONST
  TESTING= FALSE;
  DEBUGGING = FALSE;
  PRODUCTION = NOT (TESTING OR DEBUGGING);
```

Pointer Constant

The reserved word NIL is the only pointer constant. For example,

```
CONST
  EMPTY_LIST = NIL;
```

Type Declarations

In SP/Pascal, *every variable in a program must be of one specific type that is either predefined or user-defined.* The type determines the set of allowable values a variable of that type can have and the operations that can be performed on that variable.

Much of the checking for type consistency is done during compilation. Programming errors, such as assigning a Boolean value to an integer variable, are detected by the compiler. Early error detection is one of the major advantages of the language's emphasis on typing.

In general, legal data type categories are:

1. Simple types (basic scalar values that cannot be subdivided),
2. Structured types (complex values made up of grouped simple types)
3. Dynamic pointer types.

Each of these categories is described further in the following paragraphs. Details of each specific data type are provided further on in this section; for information on using references to structured data types in expressions, refer to Chapter 4, "Expressions". The use of the SP/Pascal extended capability, the bit qualifier, is described in the section called "User-Controlled Data Storage" at the end of this chapter.

Data Types

There are six predefined simple data types: *INTEGER, REAL, CHAR, BOOLEAN* (all standard Pascal) along with *WHOLE* and *DOUBLE_REAL,* (SP extensions). Each simple data type represents an ordered set of values. For example, integer type represents the whole numbers in sequential order from −32768 to +32767 so that 2 is less than 3, and so forth; character type denotes the ASCII character set in its standard sequence so that F comes after E. (See Appendix A for ASCII order.)

Two kinds of user-defined simple data types, *enumerated* and *subrange,* allow you to set up your own ordered sets of values. You define an enumerated data type by listing the elements in the set in their own order; for example, type RACE and its enumerated list (WIN, PLACE, SHOW). A subrange is a contiguous subset of an existing simple data type (either predefined or user-defined enumeration). You define a subrange data type by listing the low and high limits of the subset, such as a subrange data type GRADES as character constants 'A'..'F'. Details of these types are provided further on in this chapter.

In SP/Pascal, there are five predefined structured data types: *ARRAY, RECORD, SET, FILE* (standard Pascal data types) and *STRING,* (an SP extension). Each structured data type is made up of simple data values, such as an array of real numbers or a string of single ASCII characters.

The pointer data type permits dynamic allocation of storage for variables whose size and lifetime are not known at compile-time (a linked list, for example). Details on pointer handling are provided farther on in this section.

Declaring a Data Type

There is a type declaration that allows you to associate an identifier with a type. The format for declaring an identifier's type follows.

```
TYPE
   identifier = type;
   [...identifier = type;]
```

You must specify a type for all variables declared in the program. The type is provided along with the variable in the declaration section. For example:

```
VAR SEMITOTAL, TOTAL: INTEGER;
```

When this technique is used with simple predefined types, you do not need a separate type declaration. Also, the type for constants need not be declared; the compiler determines the type from the constant declaration.

Predefined Simple Data Types

SP/Pascal offers six predefined simple (scalar) data types: *integer, whole, character, Boolean, real* and *double_real.* Further details for some of these data types appear in the previous section on constants.

Integer

Integer variables can assume whole number values ranging from −32,768 to +32,767.

Whole

Whole values are full-word unsigned quantities in the range of 0...65535. The whole type allows the generation of unsigned arithmetic and comparison operations.

Character (CHAR)

CHAR variables assume the value of any single character in the ASCII character set (ordinal values from zero to 255). (Refer to Appendix A for the ASCII character set.)

Character values 128 through 255 are not assigned printable graphics. However, this extension has meaning when applied to character sets representing other languages. This feature is supported to comply with ASCII-8 and with various foreign character sets (e.g., Japanese), whose characters occupy a full byte.

Boolean

Boolean variables denote logical values of either FALSE or TRUE. The identifiers are predefined Boolean constants with FALSE defined as less than TRUE.

Real

Real variables are 32-bit single-precision quantities that can assume signed numeric values with absolute values between zero and approximately 7.2 x 10^{+75}. The smallest positive real value is approximately 5.4 x 10^{-79}. Real numbers are precise to within approximately seven decimal digits.

Double_Real

Double_real variables are 64-bit double-precision quantities that can assume the same signed numeric values as single-precision real. Double_real numbers are precise to within approximately 15 decimal digits.

User-Defined Simple Data Types

SP/Pascal allows you to define your own simple data types in several different ways. The simplest of these declaration forms is

identifier = previously_defined_type_identifier

The previously-defined type identifier can be a predefined one or another user-defined one. You can define two different simple data types: *enumeration* and *subrange*.

Enumeration Data Type

This data type declares an ordered set of values by specifying all of the identifiers denoting those values. An enumeration type can contain up to 256 elements. The ordering of these elements is determined by the left-to-right order in which they are listed in the type declaration. If X precedes Y, then X is less than Y. The ordinal value of the leftmost element is zero.

For example, you could define an enumeration data type called WEEK and could order its days as follows:

```
TYPE
  WEEK = (MONDAY, TUESDAY, WEDNESDAY,
          THURSDAY, FRIDAY, SATURDAY, SUNDAY);
```

Variables of type WEEK would be allowed to take on only the above values.

NOTE: *Enumeration types must contain more than one enumeration identifier; single identifier enumeration lists conflict with a parenthesized constant expression. For example,*

```
TYPE E = (ONE); % illegal
TYPE S = (ONE)..10; % legal
TYPE P = (ONE, TWO); % legal
```

Subrange

Users can define types that are subranges of another defined simple type, such as integer, CHAR or enumeration. To declare a subrange data type, provide two constants to define the boundaries of the range. The constants are separated by the 'to' symbol, which is written as two periods with no intervening delimiters (..). For example, once WEEK is defined as an enumeration type, you can declare a subrange of WEEK, called WORKWEEK, as follows:

```
TYPE
  WORKWEEK = MONDAY .. FRIDAY
```

The next example illustrates how you could declare subranges of type integer and CHAR:

```
TYPE
  RANK = 1 .. 10
  LETTER = 'A' .. 'M';
```

You can use signed constants in declaring your subrange. For example:

```
CONST
    X = 10;
TYPE
    RANGE = -X .. +X;
```

NOTE: *You can declare subranges only of Boolean, CHAR, enumeration, and integer.*

All numeric subranges in SP/Pascal are defined as signed integer subrange types. There are no subranges of type whole. This definition implies that the minimum value in a subrange must be less than or equal to the maximum value. The upper bound of all subranges must be less than or equal to MAXINT (<=32767).

Anonymous Types

The identifiers defined in a type declaration section may be used in other declarations to define the component types of structures, the types of variables, and the types of routine parameters. This technique associates the type name and its properties with its usage in the other definitions. However, in structured type and variable definitions (described below), you are not required to use a previously declared type. Instead, you can choose to define a new data type in one of the declarations. For example:

```
TYPE
  R = RECORD
    F : 0..255
  END;
VAR
  X,Y : ARRAY [1..] OF REAL;
```

In these declarations, field F and variables X and Y are said to be anonymous types because there is no type name associated with them. Only the properties of the type are defined. In general, variables of anonymous structured types are not compatible with each other even if their actual structure is the same. Exceptions to this rule are variables that are declared in the same definition, such as X and Y above, and variables whose type is either string or packed array of CHAR. For more information on these rules, see the section on type compatibility in the next chapter.

Because the compatibility rules are special for anonymous types, you should be careful when using them. For example, no variable of an anonymous type (except string) can be passed as a VAR parameter.

Structured Data Types

Structured data types consist of data elements arranged in a composite group. For example, a string is composed of elements of the simple data type CHAR. More complex structures are possible, such as a group made up of a number of individual arrays of real numeric values. Details on referring to structured data items are provided in Chapter 4, "Expressions".

Five varieties of structured data types are permitted in SP/Pascal: *array, record, set, file* (all standard Pascal) and *string,* (SP extension). In SP/Pascal four of the structured types, array, record, set, and string can be used to define structured constants.

Text is a predefined data type used for input/output files of character data. Variables of type text, known as *text files,* contain elements of type CHAR grouped into variable length lines. Text files are described under "File Declarations" in this chapter, and in Chapter 7, Input/Output.

Array Structured Data Type

An array consists of a fixed number of ordered components. Each component is called an *element* of the array. All array elements are of the same type, the *element type.* You refer to a specific element by an *array_index* or *subscript.* There is a unique element of the array for every allowable value of the *index_type.*

The form of the array type declaration is

ARRAY [*index_type*] OF *element_type*

Element_type specifies the type of each element in the array. The *index_type* (which must be enclosed by square brackets) gives the type and range of the array subscripts. The type for the array index can be integer, whole, character, Boolean, enumeration or a subrange of any of these. For example:

```
TYPE
   A = ARRAY [CHAR] OF BOOLEAN;
   MY_ARRAY = ARRAY [1..10] OF REAL;
   Y = 1..3;
   Z = ARRAY [Y] OF DOUBLE_REAL;
```

The above example defines the type *MY_ARRAY* as a one-dimensional array of real variables with subscripts ranging from 1 to 10, inclusive. These subscripts are a subrange of the predefined integer type. Subrange values must be a valid scalar range with the minimum less than or equal to the maximum.

Since there are no restrictions on *element-type,* multidimensional arrays can be formed by nesting array declarations, as follows:

```
TYPE
   TWO_DIM = ARRAY [ 1..6 ]  OF MY_ARRAY;
   TWO_DIM2 = ARRAY [ 1..6 ] OF ARRAY [ 1..5 ] OF INTEGER;
   Q = ARRAY [BOOLEAN,CHAR] OF (MON,TUE,WED);
```

A convenient abbreviation can be used to declare *TWO_DIM2*, e.g.,

```
TYPE
  TWO_DIM2 = ARRAY [ 1..6, 1..5 ] OF INTEGER;
```

This abbreviation is exactly equivalent to the more verbose form used in the first example.

NOTE: *It is easier to debug an SP/Pascal program with the process debugger when arrays are defined as starting with element zero rather than 1, such as*

```
TWO_DIM = ARRAY [ 0..5 ] of MY_ARRAY;
```

In some instances, the code generated for references to an array element is slightly more efficient if the low bound is zero.

Array Constants

SP/Pascal allows the definition of array constants in any context where an expression of a structured type is allowed. The syntax for structured array constants is:

array-name [constant-expression-list]

Arrays must have all components assigned values in the constant definition; there are no default values. Structured constants can contain other structured constants as subcomponents. For example,

```
TYPE
   AR = ARRAY [1..4] OF BOOLEAN;

CONST
   AR_CON = AR [TRUE,TRUE,FALSE,TRUE];
```

To avoid restrictions on the storage mapping for arrays, array constants can only be vectors. Therefore, in order to define a multi-dimensional array constant, each dimension in the array type definition must be specified by a separate type definition. For example,

```
TYPE
   A = ARRAY [1..3] OF INTEGER;
   B = ARRAY [1..3] OF A;
   C = ARRAY [1..3,1..3] OF INTEGER;

CONST
   D = B[ A[1,1,1], A[2,2,2], A[3,3,3], ];  {a valid definition}
   E = C[ 1,1,1,2,2,2,3,3,3 ]; {diagnosed by the compiler as an error}
```

Record Structured Data Type

Record components can be of different types. Each component is called a *field* of the record and is identified by a *field identifier.* A record-type declaration has the following form:

RECORD
[...field-definition ;]
END

Field-definition looks like this:

identifier [...,identifier] **:** *type*

For example:

```
TYPE
   D1 = RECORD
                HI, LO : INTEGER;
        END;

   D2 = RECORD
                HI : INTEGER;
                LO : INTEGER;
        END;
```

Both of these examples declare a record type having two integer fields called HI and LO. D1 and D2 have the same representation, but denote separate types in the program.

The type of a field in a record can be another record. This permits record definitions to be nested inside each other. In SP/Pascal, records can be nested to a maximum of 15 levels. Note that the nested record can include field names that are identical to the field names in an enclosing record. However, within a single record definition, all the field names must be unique. For example,

```
TYPE
   NESTED_REC = RECORD
                   LEVEL:WHOLE;
                   STRUCT:RECORD
                             LEVEL:0..10;
                             NAME:STRING 10;
                             RESPONSE:(YES,NO,MAYBE);
                          END;
                   VALID:YES..NO;
                END;
```

It is possible to declare a record type that allows variables of that type to differ in the number and/or type(s) of their fields. Record types with this characteristic are known as *variant* records. The form of the variant record definition follows.

RECORD
    [...field-definition ;]
    CASE *[identifier :] type* OF
        variant [...;variant]
END

A *variant* looks like this:

variant-label [...,variable-label] :
(*[...field-definition ;]* **)**

The identifier following CASE is known as a *tag field* identifier. The type of a tag field must have ordinal values that fall only in the range of 0 to 127. Boolean and enumerations with 128 or fewer elements satisfy this requirement. Otherwise, the type must be a subrange (of integer, character, enumerated, or Boolean).

Each *variant-label* in the list must be of the same type as the type of the tag field. The values of the case constants should exhaust the type. There should be a defined constant for each possible value of the tag field. Otherwise, the compiler issues a warning. For example,

the following type declaration section could appear in a program that processes simple personnel records:

```
TYPE
    PERSON_INFO = RECORD
            .
            .
            .
            CASE MARRIED : BOOLEAN OF
                TRUE : (SPOUSE : ARRAY [ 1..9 ] OF CHAR;
                          CHILDREN : 0..15 );
                FALSE : ( );              % Note empty variant
    END;
```

In the preceding example, the number and types of fields following the tag field in the record depend on the determination of the current value of the tag field identifier MARRIED. When that value is true, the tag field is followed by two fields, namely, SPOUSE and CHILDREN; when the value is false, the tag field is followed by an empty field list, denoted by ( ). Thus, records for married people contain two fields not present in records for unmarried personnel.

The subrange notation X..Y is permitted in a record definition for specifying a list of variant tags. This notation is a shorthand way for referring to a list. For example, 'A'..'C' is equivalent to specifying 'A','B','C' in a tag constant.

Untagged variant records are also supported. You can, for example, have the following declaration section:

```
VAR
   S : RECORD
        F : INTEGER;
        CASE BOOLEAN OF
               TRUE : (G:INTEGER);
               FALSE: (P:REAL);
   END;
```

With the variable S declared in this manner, there is no tag field. Your program may reference the integer variable S.G, or the real variable S.P. This provides a way to bypass SP/Pascal's normal type-checking, similar to the RECAST facility (described in Chapter 6).

Since the tag field is not stored with the record, references to fields in different variants cannot be checked for consistency with the tag value. As a result, untagged variants are inherently unsafe. You should use care when selecting this particular record structure.

Variants can be nested within other variants in a manner similar to the way records nest within each other. However, in SP/Pascal, variants can be nested only to a maximum of five levels. It is possible to mix tagged and untagged variants at different levels of nesting. For example:

```
TYPE
 SCALAR_TYPES = (DISCRETE, REALS);
 DISCRETE_TYPES = (INTEGERS,WHOLES,BOOLEANS,CHARS,SUBRANGE,ENUMERATION);

 SCALAR_REC = RECORD
      SCALAR_NAME:STRING 80;
      CASE SCALAR_KIND:SCALAR_TYPES OF
         DISCRETE:(USER_DEFINED:BOOLEAN;
                   CASE DISCRETE_KIND:DISCRETE_TYPES OF
              INTEGERS,WHOLES:(SIGNED:BOOLEAN);
              BOOLEANS,CHARS: ();
              SUBRANGE:(SUBRANGE_KIND:DISCRETE_TYPES;
                        CASE DISCRETE_TYPES OF
                   INTEGERS:   (IMIN,IMAX:INTEGER);
                   WHOLES:     (WMIN,WMAX:WHOLE);
                   BOOLEANS:   (BMIN,BMAX:BOOLEAN);
                   CHARS:      (CMIN,CMAX:CHAR);
                   SUBRANGE:   ();
                   ENUMERATION:(EMIN,EMAX:WHOLE));
              ENUMERATION:(ENUM_SIZE: 2..256;
                           ENUM_NAMES:ARRAY[0..255] OF STRING 80));
         REALS:(PRECISION:(SINGLE,DOUBLE));
              END;
```

Record Constants

SP/Pascal allows the definition of record constants in any context where an expression of a structured type is allowed. The syntax for record constants is

record-type-name (constant-expression-list)

Records must have all components assigned values in the constant definition; there are no default values. The order of the constants in the *constant-expression-list* must match the order of declaration of the fields in a record. Tag field values must be specified, even for untagged variants.

Structured constants can contain other structured constants as subcomponents. For example,

```
TYPE
   RC = RECORD
           F:INTEGER;
          CASE B:BOOLEAN OF
        TRUE: (G:REAL);
        FALSE: ();
        END;
CONST
  RC_CON = RC (3, FALSE);
  VC_CON = RC (3, TRUE, 1.22);
```

Set Structured Data Type

A set is a structured type that represents a group of values of the base type. The form of a set declaration is

SET OF *base-type*

Base-type is a scalar data type with elements whose ordinal values are in the range of 0 to 255.

Set elements are not treated as individually stored elements like record fields or array elements; the elements making up a set are part of a type collection of related elements, such as ALPHABET representing a set of the characters A through Z.

A variable of set type can adopt values that are in the powerset of the base type (set of all possible subsets). The maximum set cardinality is 256; set sizes are packed to the nearest word boundary. The empty set, which has no elements, (denoted by []) is considered to be an element of every set type.

For example,

```
TYPE
  DECIMAL_DIGITS = SET OF 0..9;

TYPE
  CHARSET = SET OF CHAR;
  ASCII7 = SET OF '<0>'..'<177>';
  SPECTRUM = SET OF (RED,ORANGE,YELLOW,GREEN,BLUE,INDIGO,VIOLET);
```

Set values are specified by enclosing the list of set elements in square brackets. Within these brackets, elements can be listed individually or by using subrange notation. Subrange notation is permissible as a shorthand way of denoting a set of values. For example, once the preceding sets are declared, you can declare variables and assign them set values as follows:

```
VAR
   TOPHALF,LASTHALF:DECIMAL_DIGITS;
   PRIMARIES:SPECTRUM;
   ALPHANUMERICS:CHARSET;

   TOPHALF:= [0..4];
   LASTHALF:= [5..9];
   PRIMARIES:= [RED,YELLOW,BLUE];
   ALPHANUMERICS:=  ['0'..'9','A'..'Z','a'..'z'];
```

Sets are allocated the minimum number of words necessary to store the values in the conceptual subrange from 0 to a maximum ordinal value of the base type. All sets are implemented as bitstrings with an origin of 0. Even if you do not start a set at 0, the space for element 0 still is allocated. For example, Page 29

```
TYPE
   S1 = SET OF 0..10;  {a one-word set}
   S2 = SET OF 20..30;  {a two-word set}
   S3 = SET OF CHAR;  {a 16-word set}
   S4 = SET OF '0'..'9';  {a four-word set}
```

Set Constants

SP/Pascal allows the definition of set constants in any context where an expression of a set type is allowed. The syntax for structured set constants is:

set-name [constant-expression-list]

The subrange member designator (*x..y*) can be used in set constant definitions. For example,

```
TYPE
   DECIMAL_DIGITS = SET OF 0 .. 9;
   COLOR = (RED, ORANGE, YELLOW, GREEN, BLUE, VIOLET);
   SPECTRUM = SET OF COLOR;
   CHARSET = SET OF CHAR;

CONST
   TOPHALF = DECIMAL_DIGITS[0..4];
   LASTHALF = DECIMAL_DIGITS[5..9];
   PRIMARIES = SPECTRUM[RED,YELLOW,BLUE];
   ALPHANUMERICS = CHARSET['0'..'9','A'..'Z','a'..'z'];
```

String Structured Data Type

String is a structured data type representing a sequence of characters of varying length (bounded by the maximum length). You define a string type as follows:

string_type = STRING *unsigned_integer*

The *unsigned_integer* specifies the maximum number of characters for that string variable. The *unsigned_integer* must be greater than zero and cannot exceed 32,767.

The current length of a string variable can change. (The string length can assume values from zero to the maximum length specified for that string type.) For example:

```
TYPE
  LINE = STRING 80 ;
```

defines a type, LINE. Variables of this type LINE can assume string values with lengths ranging from 0 to 80 characters.

Pointer Types

Often it is necessary to deal with data structures whose memory space requirement cannot be determined at compile-time. The pointer facility allows you to handle varying amounts of data in a structured manner. With the pointer facility, you are not restricted to the stack-based allocation schemes of a block-structured language. The pointer facility allows for dynamic memory allocation. You handle this dynamic storage using variables of *pointer-type*. A pointer is declared in one of the following two ways:

pointer-type = ^*resolution-type*

pointer-type = @*resolution-type*

The up-arrow or at-sign (^ or @) indicates that the declared type identifier is a pointer type. Variables of this type can be used only to refer to elements of the *resolution-type,* which can be any valid type. Often the resolution type itself contains pointers; this allows linked data structures to be generated.

The declaration of a pointer type can precede the declaration of its resolution type; that is, you can declare a pointer to a record before you declare the record. This is the exception to the general rule that every identifier must be declared before it is used. For example:

```
TYPE
 POINTER = ^LIST;
 LIST =  RECORD
         NEXT : POINTER;
         INFO: INTEGER;
         CLASS: (FIRST, SECOND, THIRD);
 END;
```

Managing dynamic storage of pointer variables is done with the NEW and DISPOSE procedures (or the MARK and RELEASE procedures) described in Chapter 9.

Pointer types that are defined before the declaration of the resolution type are referred to as *unresolved pointer types*. Once the resolution type becomes defined, all unresolved pointer types that are dependent on this type become resolved. All pointer types must be resolved before they can be used in any variable or routine declarations.

For example, in the following sequence of declarations, the definition of SOME_RECORD must precede the variable declaration for I.

```
TYPE PTR = @SOME_RECORD;
  .
  . {any number of CONST or TYPE declarations}
  .
VAR I:INTEGER;
```

Variable Declarations

Variable declarations associate an identifier with data of a specific type. The format of a variable declaration section is as follows:

VAR
identifier [...,identifier] : *type*;
[...identifier [...,identifier] : *type]*;

The *identifier* can be any valid SP/Pascal identifier. The *type* can be any of the predefined types or any type defined in previous type declarations. Use the word VAR once and follow it with the variable definitions. For example:

```
VAR
  X,Y : INTEGER;
  LETTER : CHAR;
  GRADES : TEXT;
```

Also, you can declare a separate type in the VAR declaration. For example:

```
VAR
  Y: ARRAY [1..50] OF BOOLEAN;
     {defines "Y" and also specifies its type}
```

Variables declared in this manner are of anonymous type (see earlier section on anonymous types.)

File Declarations

The *file* structured data type contains a sequence of elements of identical type. File elements are not all accessible at the same time; just a single element, a *window,* is available. A file type declaration follows:

identifier = FILE OF *type*

where *identifier* is the name of the file type, and *type* is one of the legal simple or structured data types.

Text is a predefined data type specifically for files. The text file type allows formatted output of the standard data types. A text file contains elements of type CHAR in units called lines. The lengths of these lines can vary. Text file lines are separated by a special delimiting character (New Line, Carriage Return, Form Feed or Null).

Declare a text file as follows:

VAR *textfile*: TEXT

Textfile is the name of the file variable used for I/O operations in your program. The statements specific to I/O operations are described in Chapter 7, "Input/Output".

SP/Pascal has two predefined text files provides as default files for I/O operations. These are the standard INPUT and OUTPUT files (declared with the program heading). For users familiar with the assembler I/O conventions, INPUT and OUTPUT correspond to ?INCH and ?OUCH. Since INPUT and OUTPUT files are defined as entry variables, they can be referred to from other program modules.

User-Controlled Data Storage

SP/Pascal introduces bit-level data storage for some data types. This feature allows you to specify your own storage packing and alignment. The following sections provide details on the SP/Pascal bit qualifier and its use in controlling the storage of data.

Bit Qualifier

The SP/Pascal bit qualifier provides extended capabilities for user-controlled storage allocation. This qualifier can be used to specify the absolute number of bits that a subrange, enumeration, character, or Boolean type representation is to occupy. Scalar types with storage requirements specified with the bit qualifier can modify the storage alignment only for array and record structures. Except for these structures, the bit qualifier does not affect storage allocation.

In SP/Pascal the reserved word PACKED has no effect on storage allocation. Instead, the bit qualifier provides type-independent field widths, direct control of alignment requirements, and both packed and unpacked fields in the same record.

The bit qualifier does not affect the range of the type it modifies, and the number of bits cannot be greater than 16. The following values are the minimum acceptable for the various types that can be bit-qualified:

Boolean	One single bit
Character	Eight bits

Enumerations	Number of bits required to represent the ordinal value of the last enumeration identifier.
Subranges	Number of bits required to represent the maximum value in the subrange. Only positive subranges can be bit-qualified.

When the bit qualifier is greater than the minimum number of bits required to represent the type, then the values denoted by the type are right-justified in the entire field. Bit-aligned components of variables are referred to with the same notation as for non-aligned components. The SP/Pascal compiler generates the masking and shifting instructions for the component access.

SP/Pascal provides two new predefined compile-time functions, BITSIZE and BYTESIZE, to retrieve the number of bits or bytes defined for a type. Refer to Chapter 9 for details of these predefined functions.

Storage Allocation and Alignment

The basic alignment rule for all data structures in SP/Pascal is that data structures begin and end on word (16-bit) boundaries. In an array structure, elements are aligned on a bit, byte, or word boundary. If the bitsize is one, then the array is a bit vector; if the bitsize is in a range from two to eight bits (or the element type is a non-bit-qualified character type) the components are stored in bytes; otherwise, the array elements occupy an integral number of words.

Generally, non-bit-qualified arrays of CHAR are packed implicitly into bytes. However, you can specify a word-aligned array of CHAR with the bit qualifier. For example,

```
A1 = ARRAY [1..10] OF BOOLEAN BIT 1; {a bit vector}
A2 = ARRAY [1..10] OF 0..255 BIT 8; {a byte array}
A3 = ARRAY [1..10] OF 1..7 BIT 3; {another byte array}
A4 = ARRAY [1..10] OF CHAR; {an implicit byte array of CHAR}
A5 = ARRAY [1..10] OF CHAR BIT 8; {an explicit byte array of CHAR}
A6 = ARRAY [1..10] OF CHAR BIT 16; {a word-aligned array of CHAR}
A7 = ARRAY [1..10] OF BOOLEAN; {a word-aligned array}
```

In record definitions, it is possible to mix both bit-aligned and word-aligned fields. All fields are allocated in the order they appear in the record declaration. In a list of field names, the allocation is from left to right. Bit fields in a single word are filled from left to right starting with bit 0. Space is skipped only to prevent a field from crossing a word boundary.

Two other restrictions apply to variant records. The tag field in a tagged variant always occupies a full word, and the variant part always starts on a word boundary. If any field in the record is not completely filled, the compiler issues a warning message.

NOTE: *Bits are numbered from zero to 15 with bit 0 the leftmost and most significant and bit 15 the rightmost and least significant.*

The following examples illustrate field placement and padding. The first example is a record definition for a single-word structure with the format of an ECLIPSE I/O instruction:

```
TYPE
IO_INST = record
        IO_CODE : 0..7 bit 3;
        AC : (IO_AC0, IO_AC1, IO_AC2, IO_ac3) bit 2;
        OP_CODE : (DIA, DIB, DIC, DOA, DOB, DOC, NIO, SKP) BIT 3;
        FLAGS: (NO_FLAGS, S_FLAG, C_FLAG, P_FLAG) BIT 2;
        DEV_CODE : 0..63 BIT 6;
END;
```

The next example is a type declaration for type R with a two-word size. Field F occupies bits 0..3 of the first word; field G occupies bits 4..7 of the first word; field H occupies bits 0..7 of the second word. The second and fourth bytes in R are inaccessible fillers. Because H is a structured type, it is aligned on a word boundary in R.

```
TYPE
     FOO = RECORD B:BOOLEAN BIT 8 END;
     R = RECORD
             F,G: 0..7 BIT 4;
             H: FOO;
         END;
```

The last example declares type VREC with a size of four words. Field C1 occupies the entire first word. The character values in this field are to be right-justified (stored in bits 8..15 of the word). Field C2 occupies bits 0..7 of the second word. Field T, the variant tag, is allocated in word 3, not in the second byte of word two. Each of the record variants occupies a single word.

```
TYPE
     TAGS = (ONE, TWO, THREE, FOUR) BIT 8;
     VREC = RECORD
             C1: CHAR;
             C2: CHAR BIT 8;
             CASE T:TAGS OF
                  ONE: (I:INTEGER);
                  TWO: (W:WHOLE);
                  THREE,FOUR: (X:0..10 BIT 8;
                               Y:TAGS);
                  END;
```

Note that packing of data has an effect on the compatibility of different data types. For more information, see the section on "Compatibility Rules" in the next chapter.

Benign Redefinition

In order to simplify the writing of modular programs, SP/Pascal permits any identifier to be declared more than once, *as long as all the declarations are equivalent.* This applies to declarations for types, constants, and variables. Note that the declarations do not have to be exactly identical, as long as they create the same internal structure. For example, the following pairs of declarations are equivalent:

```
TYPE
   S = SET OF 1 .. 10;
   R = RECORD
       F, G: REAL
       END;

TYPE
  S = SET OF 1 .. (2 * 5);
  R = RECORD
      F: REAL;
      G: REAL
      END;
```

On the other hand, the following definitions are not equivalent to the ones just given:

```
TYPE
  RANGE = 1 .. 10;
  S = SET OF RANGE; {not equivalent because of named base type}

  R = RECORD
      G, F: REAL {not equivalent because field names are in wrong order}
      END;
```

Expressions

This chapter describes the forms of expressions and variable references in SP/Pascal programs. SP/Pascal supports the familiar arithmetic and logical operators, plus standard Pascal operators such as IN (for sets) and "." (dot, for records). It also contains some Data General extensions of standard Pascal, such as the "<< >>" form for referencing pieces of strings. This chapter also contains information on compatibility of different data types.

An expression is a user-specified combination of variables and constants with SP/Pascal operators and functions. An expression is made up of expression operands (names of variables, constants, and literals) combined with expression operators (arithmetic, relational, Boolean, set) and function designators, such as ROUND, SQRT, SIN). Some examples of expressions:

```
COUNTERA + 1
TEXTIN > LINEMAX
STATCHECK OR (TIMER3 AND CLOCKMSK)
TALLY MOD 4
TAGVALUE IN SETOFTAGS
ERRWD < > -1
SIN(TOTAL/PI)
```

The compiler resolves an expression to a specific data value or address. Any variable or constant used as an operand in an expression must be declared prior to its use; for example, you must declare the variables A and B before using C:= A+2*B in a program. If an operand does not have a previously-assigned value, the expression evaluation produces indeterminate results at run-time.

Operands

The following paragraphs provide further information about data types so you can use operands of these data types in SP/Pascal expressions. For more information, see the preceding chapter.

Constants

A constant reference is the name of the constant. Literal values and elements of enumerated types are constants. SP/Pascal permits the use of scalar expressions in all contexts that allow a constant. Such expressions must be compile-time evaluable and all operands must be defined before they are used. *Exception:* The constant following the name STRING in a type declaration must be either an unsigned integer or the name of a previously declared constant.

Set, array, and record constants are allowed in any context where an expression of a structured type is permitted. However, operands in compile-time expressions cannot be components of structured constants. For example,

```
CONST
  M = RC_CON + D[1,1]; {flagged as a compiler error}
```

Qualification of structured constants is identical to qualifications of variables of the same type.

All unary and binary operations defined on operands of a scalar type are permissible in compile-time expressions. In addition certain function invocations can be used if their actual arguments are

compile-time constants. These functions are ABS, CHR, LENGTH, MAXLENGTH, ODD, ORD, PRED, SQR, and SUCC. The predefined functions BITSIZE AND BYTESIZE, whose arguments are type names, may also be used in constant expressions. For example,

```
CONST
   C = 3 + 12 * 2;
   S = 'abcdefg';
   D = ODD(Length(s));
TYPE
   T = ARRAY [1..C*2] OF INTEGER;
```

Referring to Array Variables

An array consists of a fixed number of components of the same type. Each component is called an *element* of the array. You refer to a specific array element using a *subscript list* along with an *array identifier.*

For example, with MY_ARRAY declared as a one-dimensional array of real variables with subscripts ranging from 1 to 10, inclusive, you could refer to an element of MY_ARRAY by following the array name with the element number in square brackets. For example:

```
MY_ARRAY [8]
```

refers to the real variable element stored in position 8 of MY_ARRAY.

If an element of an array is itself an array (an array of arrays), as in

```
VAR
  X : ARRAY [ 1..3 ] OF ARRAY [ BOOLEAN ] OF CHAR;
```

then a valid array reference could be

```
X [ 2 ] [ TRUE ]
```

This can be abbreviated to

```
X [ 2, TRUE ]
```

In other words, the sequence "] [" in an array reference can be replaced by a single comma.

Referring to Fields of Records

To address a single record field, the reference takes the form:

record-name.field-name

You use the same form to refer to fields in the variant section of a record, with one additional requirement: the current value of the *tag field* identifier must be identical to one of the constants identifying the *field-name* in the record definition.

Assume, for example, that you declare a record, CITY, as follows:

```
TYPE
  LOCATION = (IN_STATE, OUT_OF_STATE, FOREIGN);

   CITY = RECORD
          NAME : STRING 40;
          CASE TAG: LOCATION OF
                 IN_STATE:  (DISTANCE:INTEGER);
                 OUT_OF_STATE: (STATE: STRING 20;
                        TIME_ZONE: STRING 10);
                 FOREIGN:  (COUNTRY: STRING 40;
                        PROVINCE: STRING 40);
          END;
```

To assign the time zone for an OUT_OF_STATE city in this record, you must first execute

```
CITY.TAG:= OUT_OF_STATE;
```

and then assign the field value:

```
CITY.TIME_ZONE:= 'CENTRAL';
```

Before the above assignment to CITY.TIME_ZONE is made, the current value of the tag field identifier is checked to see that it corresponds to OUT_OF_STATE. If there is no tag field, its current value is not checked; it is then up to the program to ensure that it has the correct variant. (The WITH statement, described in the next chapter, provides another method of referring to record fields.)

NOTE: *To guarantee the initial values of certain data types (such as the strings in the preceding example) when a value is assigned to the tag field of a variant record, SP/Pascal sets to zero the storage occupied by the remainder of the variant portion in the record.*

Referring to Strings and Substrings

When you use string variables in expressions, remember that all string variables are initialized as null strings. You can refer to a string in two ways:

1. Refer to a whole string by using the name of the string variable, or
2. Refer to a portion of a whole string (a *substring)* using the name of the whole string and a character range in the form:

   stringname≪*expr*$_1$:*expr*$_2$ ≫

Expr$_1$ indicates the starting point of the referred-to substring, that is, the number of the character position within the named string for the substring's first character. (The first character in a string is numbered 1.) *Expr*$_2$ is the length (total character count) of the referred-to substring. Substrings are variables of type string.

For example, once you define a string constant, TELEGRAPHIC, you can refer to the substring GRAPH as follows:

```
CONST
  S = 'TELEGRAPHIC';
  .
  .
  .
  WRITE (S <<5 : 5>>); {writes ''GRAPH''}
```

NOTE: *A run-time error occurs if* $expr_1$ *is less than or equal to 0, or* $expr_2$ *is less than 0, or* $expr_1 + expr_2 - 1$ *is greater than the string's current length.*

You can refer to a single character within a string (yielding data of type CHAR) with

stringname ≪*expr*≫

where *expr* is an integer expression denoting the character's position in the string.

NOTE: *Expr must be greater than 0 and less than or equal to the string's current length.*

In SP/Pascal, values of type CHAR are converted implicitly to type STRING1 in all expression constants. This coercion means that CHAR values can be assigned to string variables, passed as string value parameters, and used as operands in string expressions.

Referring to Pointer Variables

Pointer variables are initialized to the predefined constant NIL, which does not point to any variable. You control the allocation of pointer values dynamically, using the predefined procedure NEW (described in Chapter 9). To refer to a dynamically allocated variable, use either of the following forms:

pointer-variable @
pointer-variable ^

Each form accesses the current value pointed to by that pointer. If the type of the value referred to by the pointer is a record, array or string, further qualification of the value is permitted.

Consider, for example, the following program fragment:

```
TYPE
 T = RECORD
              X : STRING 20;
                ...
       END;
VAR
 PTR : @ T;
```

Assuming that you properly allocate T, using NEW, and that PTR refers to it, and further assuming that T is properly initialized, then the following is an example of a valid reference.

```
PTR @ . X ≪ 1:3 ≫
```

NOTE: *A run-time error occurs if an attempt is made to refer to a dynamic variable whose value is NIL.*

It is important to distinguish between pointer variables and the instances referred to. The variable PTR in the preceding example is a pointer variable. At different times during program execution it is possible for PTR to point to several different instances of records of type T.

Set Literals

Sets are defined as having values in the powerset of the base-type. The empty set, denoted by [], is a subset of every set. A set literal with a single element is denoted by *[expression],* where the expression is within the allowable range of the base type.

You write literals with multiple elements, using a simple extension of this notation. For example, [1, 2+3, 6] represents a set with elements 1, 5 and 6. The compiler takes the type of a set literal from the types of its components.

When all the elements in the set literal are constants, it is more efficient to use a set constant (refer to "Structured Data Types"). For example,

```
TYPE
  T = SET OF 0..10;
```

allows you to replace the set literal in the preceding example with T[1, 2+3, 6];.

Subrange notation is permissible in denoting set values. If, for example, you declare a type MONTHS with a set variable SKIMONTHS, you can assign the value of SKIMONTHS as follows:

```
TYPE
  MONTHS = (JA, FE, MR, AP, MY, JE, JL, AG, SE)
VAR
  SKIMONTHS :  SET OF MONTHS
  SKIMONTHS := [ JA ..  AP ]
```

NOTE: *If the ordinal value of the first range element is greater than the ordinal value of the last element, the set is an empty set. For example, a set such as [9..3] is empty.*

Function Designators

A function designator consists of the function name followed by a parenthesized comma list of actual arguments (if the function requires arguments). Functions and their argument lists are described in Chapter 6.

Operators

Four classes of operators can be used in expressions to evaluate the expression's operands. They are arithmetic, Boolean, relational, and set, all detailed in the following paragraphs.

Arithmetic Operators

In SP/Pascal, predefined functions are treated like operators. The precision of the returned result depends on the actual argument. If the argument is a double-precision value, then the predefined function returns a double-precision result. If the argument is a single-precision value (or constant), then the result is single-precision. The floating-point operations for calculating the predefined result use the same precision as that of the returned value.

Arithmetic operators act on operands that are compatible with real, double_real, integer, or whole types.

The +, −, and * operators act on both real and integer operands. The / operator acts on real operands. The DIV and MOD operators act only on integer types. For type whole, unique operations are the unsigned multiplication and division (DIV) operations, along with the unsigned comparisons.

For real expressions, when one operand is of type real and the other operand is of type double_real, the REAL operand is converted automatically to a double-precision value by zero-extending the single-precision value.

In SP/Pascal, binary operations on operands of type whole are performed without overflow checking. For mixed-mode operations with one operand of type whole and the other of type integer, the whole operand is implicitly coerced to an integer with no range checking. A scalar operand can be coerced to a whole operand by using the type name as a function.

You can convert integers to real numbers and vice versa, using the predefined mathematical functions TRUNC, FLOAT and ROUND, described in Chapter 9. No arithmetic operations can be performed on arrays and structures, but you can operate on scalar elements of these structured types.

Unary Operations

The + and − operators act on real or integer operands as shown below.

+ Identity

− Sign inversion

Binary Operations

Binary operators perform as follows:

+ Addition

− Subtraction

* Multiplication

/ Division

DIV Division (always returns integer or whole result)

MOD Integer modulus (remainder)

The DIV operator performs division on integer type compatible operands. The operand to the left of the operator is divided by the operand to the right of the operator. The result is truncated so that an integer value is obtained. For example, 23 DIV 5 evaluates to 4.

NOTE: *Division by zero yields an error, and division by a negative number is defined and produces the mathematically correct result.*

The MOD operator yields the modulus that results from an integer division. MOD always returns a result in the range *0..modulus−1* for positive moduli and in the range *modulus+1..0* for negative moduli. For example, 5 MOD −3 evaluates to −1 and 12 MOD 8 evaluates to 4.

Boolean Operators

The Boolean operators act exclusively on operands compatible with Boolean types; the result is always Boolean. Boolean operators are

AND Performs the logical AND of two Boolean operands. The result of A AND B is true if both the value of A and the value of B are true. Otherwise, the result is false.

OR Performs the logical inclusive OR of two Boolean operands. The result of A OR B is true if either the value of A or the value of B is true, or the values of both are true.

NOT Complements the Boolean value of a single Boolean operand. The result of NOT A is true if A is false and the result is FALSE if A is true.

Relational Operators

Relational operators act on compatible operands of any type except the file type (or a record containing a field of file type) by evaluating a relationship between expressions. These operators always yield a Boolean result, — i.e., true if a specified condition is met and false if it is not. Relational operators consist of:

=	Equal to
<	Less than
>	Greater than
<>	Not equal
>= or =>	Greater than or equal to
<= or =<	Less than or equal to

Any two operands of compatible type can be compared for equality and inequality. You also can use all the comparison operators with structured and simple types, except type file and records or arrays containing elements of type file. Note, however, that comparisons between simple types are signed, but comparisons between structures are unsigned. However, type whole provides for unsigned comparison operations.

String Comparisons

String comparisons are performed by means of a left-to-right, character-by-character comparison based on the ASCII collating sequence. To be considered equal, two strings must be of the same length and contain an identical sequence of characters. The strings WARP and WOOF, for example, are considered unequal, WOOF being greater than WARP. If the two strings are equal up to the length of the shorter string, the longer string is greater. For example, 'ABCD' is less than 'ABCDE'.

Set Comparisons

When applied to set operands, the operators <= and >= work as follows:

The expression SETOFTOYS <= SETOFCUBES is true if SETOFTOYS is a subset of SETOFCUBES.

The expression SETOFTOYS >= SETOFCUBES is true if SETOFCUBES is a subset of SETOFTOYS.

Set Operators

The set operators are

+	Union
*	Intersection
−	Set difference
IN	Set inclusion

The first three operators listed (+, *, −) require set data for both operands and return a set as a result. Both operand sets must be of the same base type.

IN is used to determine if the left operand, an expression compatible with the base type of the right-hand set, is a member of the associated set. IN yields a Boolean result. The expression PEACHES IN [SPINACH,PEACHES,ZUCCHINI] would yield true.

Operator Precedence

There are four levels of operator precedence for operations in SP/Pascal:

Highest	NOT						
	*	/	DIV	MOD	AND		
	+	−	OR				
Lowest	=	<>	<	<=	>	>=	IN

The compiler evaluates operators of different precedence in the order shown above. Operators at the same level of precedence are evaluated from left to right. Any expression in parentheses, however, is evaluated first, regardless of the operators preceding or following the parentheses. Within parentheses, the normal rules of precedence apply.

Compatibility Rules

Each expression has a type as well as a resolved value. The operands (variables, constants, and functions) that appear in an expression must be compatible with the expression's operators (arithmetic, relational, or Boolean). Type compatibility determines if an operation is applicable to its operand(s).

Compatibility of Two Operands

Two operand types are compatible in an assignment statement if any one of the following conditions is true:

- They are identical in type; that is, they are defined by the same type definition. Note that the compiler performs transitive closure on all type definitions. For example, if you define X as an INTEGER type and then define Y as type X, then X and Y are identical types.
- One is a simple type, and the other is a subrange of the same type; or both are subranges of the same simple type.
- They both are set types whose base type is the same simple type.
- Both are set types, and at least one is the null set.
- Both are STRING types.
- One is a packed array of CHAR, and the other is a string literal or another packed array of CHAR of the same length.
- One is a real or double_real type, and the other is an integer or whole type.

Some sample assignment statements:

```
DBL.HI := RANDOM(4);
COUNTER := -24;
TALLY := CURRENTSUM;
ADJUSTMENT := (PERCENTILE*GROSS)/AGENCYFEECODE;
PTR@.X<<I>> := PTR@.X <<I+1>>;
A [34] := 23;
```

Whole and Integer Types

Special assignment rules apply to variables of whole and integer type. A whole variable can only be a positive integer value. An integer variable can only be assigned to a whole value that is less than 32768. (Refer to the compiler option W in Chapter 13.)

Binary operations on variables of type whole are unsigned and are performed without overflow checking. For mixed-mode operations (one operand is of type whole and the other of type integer) the whole operand is coerced implicitly to an integer with no range checking.

A simple operand can be coerced to a whole operand by using the type name as a function. (The semantics are analogous to the ORD function for integers.) For example:

```
W: WHOLE(i);
```

Unique operations on values of type whole are the unsigned comparisons and the unsigned multiplication and division operations.

In constant definitions, signed values are defined to be of type integer, while unsigned values are of type whole. Constants in the range of 0..32767 are treated as either integer or whole, depending on context. For example:

```
CONST A =  -32768;  (an integer constant)
      B =   65535;  (a whole constant)
      C = +65535;  (causes an error at compile-time)
      D =       1;  (either an integer or a whole constant)
```

Packed Structures

Packed components of structures cannot be passed as VAR parameters. A packed component is any component that is not word-aligned.

A bit-qualified type is compatible with a non-bit-qualified type of the same name. That is, the bit qualifier does not create a new type. This rule is essential for determining the compatibility of packed components. For example,

```
TYPE
    T1 = CHAR BIT 16;
    T2 = CHAR BIT 10;
    T3 = 0..7;
    T4 = RECORD
            C1: CHAR BIT 16;
            C2 :T1;
            C3: T2;
            F1: 0..7 BIT 16;
            F2: T3 BIT 16;
         END;

VAR
    X:T1;
    Y:ARRAY[1..3] OF T2;
    Z:T4;

PROCEDURE P (VAR CH:CHAR; VAR RANGE:T3);
```

In this example, types T1, T2, and CHAR are all compatible. Variables X, Y[1], Y[2], Y[3], Z.C1 and Z.C2 can be passed to procedure P as argument CH. Variable Z.C3, a packed component, cannot be passed. Only variable Z.F2 can be passed to the procedure as argument range; variable Z.F1 is not compatible with type T3. The compatibility rules can be best understood by examining all types with the bit-qualifiers removed.

Program Statements

This chapter describes the various forms of SP/Pascal program statements. In addition to the standard Pascal control and data-structuring statements, SP/Pascal provides some powerful extensions, such as EXITLOOP and EXCEPTION.

This chapter describes SP/Pascal program *statements.* Generally, statement categories are: *assignment statements* that initialize or set a specific value for a variable; *compound statements* that represent a collection of statements to be executed in sequential order; and *flow of control* statements that direct changes in the order of statement execution (including iteration and recursion). Tables 5.1 and 5.2 provide a summary of the statements in each category, complete with examples.

Category	Statements	Examples
Assignment	... := ...	TOTCOUNT:= SEMI+STOCK; COUNTER:=0; I:= 1;

Compound	BEGIN...END	**An executable section**
		BEGIN
		WHILE NOT EOF DO
		BEGIN
		READLN (INGOTNO, WTGRMS);
		WRITELN('BAR',NUMBER:9,'-',PRICE:8,'US');
		END;
		END.
		
		A routine declaration
		FUNCTION PRICE (WTGRMS:REAL):REAL;
		CONST OZPRICE = 567.989;
		BEGIN
		PRICE:= WTGRMS*OZPRICE;
		END;
		
		Exception-handling compound statement
		BEGIN
		.
		.
		.
		EXCEPTION
		.
		.
		.
		END;

Table 5.1 SP/Pascal statement categories: assignment and compound

Category	Statements	Examples
Flow of Control	IF	IF EOF THEN RETURN ELSE CHECKSTATUS;
	CASE	CASE STATUS OF 0: NEXTREAD; 1: ERRORHAN; OTHERWISE WRITELN (STATCOD:6, 'BUG TEST HERE'); END;
	WHILE	WHILE TEMP < HOTPT DO RAISEHEAT;
	REPEAT	REPEAT UPDATE UNTIL EOF;
	FOR	FOR PAYROLLNO:= 200 TO 1000 DO STATECALC;
	WITH	WITH STUDENT[N] DO FINALGRADE;
	EXITLOOP	IF TOTAL > 6000 THEN EXITLOOP;
	RETURN	CASE CHANREAD OF 2: TNK2INPUT; 3: TNK3INPUT; 4: TNK4INPUT; OTHERWISE RETURN; END;
	ERETURN	IF STATUS <> 0 THEN ERETURN(STATUS);

Table 5.2 SP/Pascal statement categories: flow of control

NOTE: *Program statements, like declarations, are separated by semicolons, which are not, however, part of statements. The language also has a null statement, so consecutive semicolons are allowed.*

Assignment Statement

An assignment statement sets a variable to a specific value. The value assigned to the variable can be a literal value, a predefined constant, another variable, or an expression. Assignment statements also are used to assign a result to a function.

variable := *value*

This assignment statement sets the value of the variable on the left of the := sign to the value on the right; := is the assignment operator. The variable and the value expression must be compatible (as described shortly).

In an assignment statement, a double-precision value is truncated when it is stored into a single-precision variable. Single-precision values are zero-extended when stored into double-precision variables.

```
CONST
  PI = 3.14159;
  TWO = 2.0;
VAR
  A,B:REAL;
  X,Y:DOUBLE_REAL;
```

```
A:= B + TWO;          {single-precision add}
A:= B + X;            {double-precision add}
X:= Y + PI;           {double-precision add}
```

You can assign a full string or a substring to a string variable or substring. You can also assign a character to a single subscripted substring. For example:

```
WORDS<<3>> := 'R';
```

If the variable to the left of the := is a full string, then its current length is adjusted to equal that of the string expression on the right. An error occurs if this adjustment causes the current length to exceed the maximum declared length of the string. If the left side is a substring, then the length of the right side must be exactly the same length as the left side. For example:

```
S ≪I≫ := 'A';
S ≪1:2≫ := 'AB';
S := 'ABCD';
S := '1';
```

Assignment Compatibility

A value of type T2 is assignment compatible with a variable of type T1 if any of the following conditions is true:

- T1 and T2 are identical types and neither is a file type nor a structured type containing a component that is a file type.
- T1 is a real or double_real type and T2 is an integer or whole type.
- T1 and T2 are compatible simple types and the value of type T2 is in the subrange bounds of type T1.
- Both T1 and T2 are real or double_real type.
- T1 and T2 are compatible set types and all the members of the value of type T2 are in the subrange bounds of the set base type of type T1.
- Both T1 and T2 are string types.
- T1 is a packed array of CHAR and the value of type T2 is a string literal or another packed array of CHAR of the same length.

Compound Statement

The compound statement specifies that its component statements are to be executed in the same sequence as they are written. The reserved words, BEGIN and END, act as statement brackets. The entire executable section of a program is a compound statement, as is the executable section of a declaration for a procedure or function. Compound statements can appear in the same contexts as simple statements.

An example of the use of a compound statement follows:

```
IF INDX > 24 THEN
        BEGIN
              I := I + 1;
              J := J DIV 3;
        END
ELSE
        BEGIN
              WHILE S «I» < > 'A' DO
              I := PRED (I);
             S «I» := 'A';
        END;
```

EXCEPTION

SP/Pascal introduces the extended compound statement that contains an *exception block*. The exception block is a sequence of statements to which control is transferred whenever an error condition occurs. An exception block begins with the EXCEPTION keyword. The syntax follows:

BEGIN *statement-sequence-1*
EXCEPTION *statement-sequence-2*
END

Normal execution of an exception block only executes the statements of *statement-sequence-1*. If no error (signal) occurs during the execution of *statement-sequence-1*, the statements of *statement-sequence-2* are not executed. Control flows to the statement following the exception block.

If an error is signalled anywhere during the execution of *statement-sequence-1*, the remainder of the statements in that sequence are not executed; instead *statement-sequence-2* is executed. For details on exception-handling, refer to Chapter 11.

An example of a compound statement with an exception block:

```
BEGIN
        .
        .
        .
  NEW(PTR);
EXCEPTION
     IF ERROR_CODE = HEAP_OVERFLOW
      THEN DONE:=TRUE
     ELSE ERETURN(ERROR_CODE);
END;
```

WITH Statement

By using the WITH statement, you can expand the scope of a particular record and refer to its components by their component names only. Once a record is identified, the field name(s) alone are sufficient for processing. The format of the WITH statement:

WITH *reference ...[,reference]* DO *statement*

In the *statement* following DO, the components of each record specified by *reference* can be addressed by their component names only.

For example,

```
VAR
    A,B: RECORD
         HERE: INTEGER;
         FEAR: CHAR;
    END;
   .
   .
   .
WITH B DO
       BEGIN  .
              .
              .
              HERE:=-1000;
              FEAR:='A';
              .
              .
              .
         END;
            .
            .
            .
```

The statement

WITH $reference_1$,$reference_2$ DO

is equivalent to

WITH $reference_1$ DO
    WITH $reference_2$ DO

In SP/Pascal, the *named-with* extension allows you to define an identifier that is associated with a particular record variable of the WITH statement or to perform a recast operation in the statement section and avoid the overhead of a procedure invocation. A *reference* can be declared as follows:

identifer1 [: identifier2] = variable-reference

Identifier1, called the *with-name,* is a variable with a scope local to the WITH statement. The type of the *with-name* is determined by the presence of the optional *identifier2.* If specified, *identifier2* dictates the type for the with-name, and the *variable-reference* is allowed to be recast to this type. This enables you to change a data type temporarily without using the recast feature. (The recast facility is detailed in Chapter 6.)

If *identifer 2* is omitted, the with-name is assigned the derived type of the *variable-reference.* In this case, the with-name becomes a shorthand notation for the variable-reference in the WITH statement.

One useful application of this form is to allow some, but not all, of the fields of one record to be copied to another record of the same type without having to respecify each reference fully. For example:

```
WITH A = P@.REC_FIELD, B = P1@.REC_FIELD DO BEGIN
     A.F1:= B.F2;
     A.F2:= B.F1;
END;
```

This type of notation makes a program shorter, and may also allow the compiler to generate more efficient code.

The following example illustrates the use of the recast operation in a WITH statement.

```
TYPE
      A1 = ARRAY[1..N] OF 0..255 BIT 8;
      A2 = ARRAY[1..BYTESIZE(A1) DIV 2] OF INTEGER;

VAR
      X:A1;

PROCEDURE P;
      .
      .
      .
      WITH Y : A2 = X DO BEGIN {references to Y are treated as
                                accesses to a word array}
      .
      .
      .
  END;
  .
  .
  .
END;  {p}
```

Flow of Control

SP/Pascal provides structured statements that establish a pattern of program statement execution that is not sequential. Control of the program flow varies, depending on the statement. Changes in program flow can be affected by: conditional statements (an IF or a CASE statement), iterative statements (the WHILE, REPEAT and FOR statements), or by the unconditional statements EXITLOOP, RETURN, and ERETURN.

Additionally, program flow can transfer to other procedures. Procedure calls are described at the end of this chapter; further information about user-defined and external procedures is provided in Chapters 6 through 10. SP/Pascal extensions provide exception handling as another level of flow-control in the language. The exception block and ERETURN statement are described in this chapter, and exception handling is detailed in Chapter 11.

IF Statement

The IF statement executes a statement only when its accompanying conditional expression is true. (Note that the *expression* must yield a result of type Boolean.) If *expression* is false, either the statement is not executed or the *statement* following the optional ELSE keyword is executed.

The two forms of the IF statement are:

IF *expression* THEN *statement*

IF *expression* THEN *statement* ELSE *statement*

For example:

```
IF A>B THEN WRITELN ('Enter another order') ELSE WRITELN ('Please wait')
```

Without ELSE WRITELN ..., the program would execute the next sequential statement when A is equal to or less than B.

NOTE: *The syntax of the language does not allow a semicolon immediately before the ELSE clause.*

CASE Statement

The CASE statement provides a test-and-branch mechanism and selects one statement from a group of statements for execution based on the value of a selector expression. When the CASE statement executes, the current value of the expression is matched to one of the specified case-constants; then the executable statement corresponding to that case-constant executes.

CASE *expression* OF
    case_constant(s):statement;
    [...case_constant(s):statement;]
[OTHERWISE *statement*]
END;

The CASE statement includes an expression called the *selector* and a list of statements. One or more case-constants, which are values compatible with the type of *expression,* precede each statement. Case constants can be of type CHAR or Boolean; they can be integer or whole, constants, or items of an enumeration.

The subrange notation, X..Y, is permitted in CASE statements for specifying a list of case labels. When this notation is used, the maximum value must be greater than or equal to the minimum value. The compatibility rules for the subrange bounds are the same as those for a single-value label. For example,

```
VAR
    I:INTEGER;
BEGIN
        CASE I OF
  0: N:=1;
  1..10: N:=N DIV I;
        END;
END;
```

The X..Y notation is a shorthand method for the following list of values:

```
X,
SUCC(X),
SUCC(SUCC(X)),
      .
      .
      .
Y
```

If the expression does not correspond to any of the case-constants, the statement(s), if any, in the OTHERWISE clause are executed. When there is no corresponding case-constant and the OTHERWISE clause is absent, an error occurs at run-time.

For example,

```
CASE I OF
   0,1 : J := 2;
     2 : J := 3;
     4 : J := 5;
OTHERWISE J := 10;
END;
```

In this example, if the selector expression, I, has the value 0, 1, 2 or 4, the corresponding statement is executed. Then control passes to the statement following the CASE statement. If I is not 0, 1, 2 or 4, the statement in the OTHERWISE clause is executed.

NOTE: *The matching value is referred to as a case-constant and not a statement label; statement labels are not permitted.*

WHILE Statement

The iterative WHILE statement evaluates a Boolean expression and, if the result is true, executes the accompanying statement, repeating these two steps until the test condition is no longer true. When the condition becomes false, the iteration stops and program flow goes on to the next sequential statement. The format for the WHILE statement is

WHILE *expression* DO *statement*

Expression is evaluated before each execution of *statement*. If the value of *expression* is false when first tested, *statement* is never executed. (To execute first and then test, use the REPEAT statement.)

An example of the WHILE statement is

```
WHILE S≪I≫='A' DO
  I=I+1;
```

REPEAT Statement

The iterative REPEAT...UNTIL statement executes its accompanying statement(s) and then evaluates a Boolean expression, repeating these two steps until the condition is no longer false. When the UNTIL condition becomes true, the program flow proceeds to the next sequential statement. The format of the REPEAT statement is

REPEAT *statement_list* UNTIL *expression*

Statement_list is executed at least once. The Boolean expression is evaluated after each execution of the statements. (To test before executing the statement, use the WHILE statement.) Note that REPEAT...UNTIL may enclose more than one statement, unlike WHILE...DO which applies to a single statement.

An example of the REPEAT statement is

```
REPEAT

J:=J+I MOD 10;
I:=I DIV 10;
UNTIL I=0;
```

FOR Statement

The iterative FOR statement repeatedly executes its controlled statement once for each value of the control variable in the span of values assigned it. With each iteration, the control variable changes automatically by one; you do not need a separate statement to adjust the value of the control variable. There are two FOR-loop formats: one for an increasing control variable, and one for a decreasing variable. The two formats for the FOR statement are:

FOR *variable* := exp_1 TO exp_2
    DO *statement*

FOR *variable* := exp_1 DOWNTO exp_2
    DO *statement*

Upon execution, exp_1 and exp_2 (the initial and final values of the control variable) are calculated. If you specify TO and if exp_1 is less than or equal to exp_2, the *variable* is assigned a sequence of consecutive values of increasing magnitude, beginning with exp_1 and ending with exp_2. If you specify DOWNTO and if exp_1 is greater than or equal to exp_2, the *variable* is assigned a sequence of consecutive values of decreasing magnitude, beginning with exp_1 and ending with exp_2. The loop *statement* is executed once for each value of the control variable.

You should remember the following:

- The initial and final values are calculated once, rather than each time through the loop.
- It is possible for the loop statement never to execute, depending on the initial and final values of the variable.
- When the loop is not executed, the control variable is not assigned.
- *Variable*, exp_1, and exp_2 must be of compatible simple data types (character, integer, whole, Boolean, subrange, enumeration, but not real or double_real).

Examples of the FOR statement are :

```
FOR COLOR1 := BLUE DOWNTO RED DO
BEGIN
  A [COLOR1] := SUCC (A [COLOR1]);
  K := K + 1
  END;

  FOR I := 1 TO 10
     DO J := J+B[I];
```

Note that the first example presupposes a type declaration in which the ordinal value of BLUE is larger than the ordinal value of RED.

EXITLOOP Statement

An EXITLOOP statement must be used only inside a FOR, REPEAT or WHILE loop. It unconditionally transfers control to the statement following the immediately enclosing loop. For example,

```
.
.
.
FOR I:= 1 TO LAST DO BEGIN
        .
        .
        .
    WHILE A[I] > 10 DO BEGIN
                .
                .
                .
            EXITLOOP;       {transfers control to J:=2 statement}

    END; {while}
    J:=2;
        .
        .
        .

    IF B[I] = A[I] THEN EXITLOOP; {transfers control to statement
                                   after END statement, K:=0}
        .
        .
        .
END;  {for}
K:=0;
```

RETURN Statement

A RETURN statement unconditionally transfers control from the current procedure or function. Control is transferred as shown in Table 5.3. Returning from program level terminates the program normally and returns control to the operating system.

From	To
Procedure	Statement following statement invoking the procedure
Function	Expression in which function was invoked
Main program	Program which executed this program (usually CLI)

Table 5.3 RETURN control transfer

ERETURN Statement

An ERETURN statement unconditionally terminates the current procedure or function and initiates a search of the currently executing compound statement in all prevously activated routines for an associated exception handler. (The routines are searched in the reverse of their called order.) If no exception handler is located, a default exception handler is executed as part of the program or task termination.

The format of the ERETURN statement is

ERETURN(expression);

ERETURN requires an integer valued expression. The expression can be the reserved word ERROR_CODE. For example

```
ERETURN(ERROR_CODE);

ERETURN(0);
```

For details on the use of ERETURN with exception handling, refer to Chapter 11.

Routine Invocations

A routine is a procedure or function activated by a *routine call.* A *procedure call* invokes the procedure and specifies any arguments to be passed to the procedure; a function call also invokes the function and passes any required arguments; however, a function call appears only as part of an expression, since the function returns a value.

A procedure call is a statement. The format is as follows:

procedure name [(argument..,argument)];

A function call has the same syntax as a procedure call, but is used only as part of an expression.

When the routine is a user-defined procedure or function, the declaration for the routine must be specified somewhere in the program. Predefined routines do not require declarations. Routines and their declarations are described in Chapter 6.

Examples of routine calls are:

```
NEWTON(X,Y);                    {a user-defined procedure, NEWTON}
WRITE ('Name -  ', LASNAM: 12); {a predefined procedure, WRITE}
A:= Z/MYMEAN(SCORES);           {a user-defined function, MYMEAN}
DAILYDOUB = 8.44*RANDOM;        {a predefined function, RANDOM}
```

SP/Pascal Routines

This chapter describes the process for declaring and implementing user-written routines in an SP/Pascal program. Declared only once in the source code, a routine can be invoked repeatedly in the executable portion of the program. Sections further on in the chapter describe parameter-passing for routines and the scope (range of association) for identifiers in routines.

Kinds of Routines

SP/Pascal supports two different kinds of routines. A routine can be either a *procedure* or a *function*; both contain a *block* of declarations and statements that accomplish a specific operation. Routines can be application-specific, user-defined subprograms defined in the declaration section of the program. Alternatively, there are a number of predefined routines (both standard Pascal routines and SP-specific routines) that can be used in a program without a routine declaration.

Chapter 9 details some of the predefined routines that are available. Chapter 10 describes the external routines provided by Data General Corporation with each system in a library of external routine declarations and commonly used definitions. (These can be specified in the program as *include* files; then the routines can be used in a program. Details of the include and overlay facilities are provided in Chapter 8. Information in Chapter 8 introduces the various methods for calling these routines and for calling assembly-language routines in a program. Appendix C details SP/Pascal's internal routine initialization and calling sequences.)

Other available predefined routines are described elsewhere in this manual. The predefined routines that handle input and output are described in Chapter 7. Routines specific to multitasking programs are found in Chapter 12; rules for exception-handling routines are described in Chapter 11.

Procedures

A procedure is a routine that contains a *block* or group of SP/Pascal declarations and statements that accomplish a specific algorithm. A procedure is defined once in the program with a *procedure declaration* and then can be invoked any number of times using a *procedure call.*

A procedure declaration introduces an identifier (the procedure name) which is used to call the procedure. The declaration has the following form:

PROCEDURE *identifier [formal-parameter-list];*
    block ;

The *block* is a compound statement that is executed when the procedure is invoked. (Parameter lists are detailed further on in this chapter.)

The following example is a procedure declaration for a procedure, NEXTCHAR, which could be called from a program whenever a new input character is to be read:

```
CONST
      END_OF_MEDIUM = '<031>';
PROCEDURE NEXTCHAR (VAR CH:CHAR);
   BEGIN
      IF EOF THEN CH:=END_OF_MEDIUM
      ELSE READ(CH);
   END;
```

Each time the program invokes this procedure, the statement section of the program needs only a single statement procedure call such as:

```
WHILE NEXTCHAR(C)< > END__OF__MEDIUM DO
  APPEND(STRING,C);
```

Functions

A function, like a procedure, is specified once in the program. A function performs its operation and returns a value. Functions are used as constituents of expressions. A function declaration, like a procedure declaration, introduces the function name used to invoke the routine. Additionally, a function declaration indicates the type of result the function returns.

A function declaration has the following form:

FUNCTION *identifier [formal-parameter-list]*: *type-identifier* ;
    block ;

The optional *formal-parameter-list* provides any data that is needed by the function to perform its operation. The *type-identifier* specifies the type for the result returned by the function.

NOTE: *A function can return values of any type except type file or structured types containing elements of type file.*

The following example is a function declaration for a function, BEANCOUNT, which could be called from a program whenever its calculation is required:

```
FUNCTION BEANCOUNT (AREA:INTEGER):INTEGER;
  CONST
     BEANS_PER_AREA = 12;
   BEGIN
     BEANCOUNT:= BEANS_PER_AREA * AREA;
   END;
```

Each time the program calls this function, the statement section of the program need only use the *function designator* in an expression, such as:

```
TOTAL = SEMITOT*BEANCOUNT(5) DIV 12;
```

Each time the function is invoked by a function designator, the block is executed. Within the block, at least one statement must assign a value to the function identifier. This value is the *function result*, which must be of the same type as the function designator.

Scope of Routines

Both procedures and functions can have a block containing declarations. The routine's declarations can be any number of constant, type, variable or routine declarations occurring in any order. (Nested routine declarations, however, must not exceed the limit of eight levels.)

A block creates a *scope,* i.e., a range of visibility for the identifiers declared within the block. Specifically, the scope of an identifier is limited only to the block in which it is declared and to any routines nested within that block.

Depending on where the declaration appears in the program, the scope of a variable or constant can be either *global* or *local.* Identifiers declared within a particular block are said to be *local* to that block. Identifiers declared in the outermost block, i.e., the main program, are described as *global,* because their scope encompasses all of the program's inner blocks.

For example, a program containing two independent procedures could be partially declared as follows:

```
PROGRAM P;
      VAR
         A : CHAR;
      .
      .
      .
      PROCEDURE ONE;
         VAR
               D : INTEGER;
         .
         .
         .
         BEGIN
         END; {ONE}
      PROCEDURE TWO;
         CONST
               S = 'ABC';
         .
         .
         .
         BEGIN
         END; {TWO}
```

NOTE: *The formal parameter names are within the scope of the routine block.*

Variable A is *global*, that is, accessible to all routines within the program block. Identifier D is *local* to procedure ONE and cannot be referred to by procedure TWO or by the main program; similarly, identifier S is *local* to procedure TWO. (If procedure TWO were nested within procedure ONE, the scope of variable D would extend then to the nested routine: D could be accessed by procedure TWO. At the same time, identifier S still would remain *local* to procedure TWO.)

Local identifiers can have the same name as identifiers in a containing scope, but, in that event, the local definition takes precedence. In other words, if two identifiers in a scope have the same name, and one of the identifiers is declared in an inner scope, a reference to the identifier from the inner scope refers to the local identifier, not the global identifier. Of course, names must be unique within any scope.

In the following declaration section, identifier X is declared in both the program and procedure blocks:

```
PROGRAM Z;
VAR
     X : INTEGER;
.
.
.
PROCEDURE CHARCOUNT;
VAR
     X : CHAR;
```

Since the procedure has redefined the identifier X, any operations with X within the procedure now relate only to its local definition as a variable of type CHAR. The global identifier X no longer is accessible to the procedure.

When a routine is invoked, most of its local variables have arbitrary values: local pointer variables are set to NIL and local string variables are set to the null string. Local file variables are initialized to the closed state. These local variables are allocated on the stack, so any time a procedure or function invokes itself *recursively*, each invocation uses a different set of local variables. Additionally, routines are reentrant, so that the same routine can be used by more than one task. (Refer to Chapter 12 for details on multitasking.)

Routine Parameters

The formal parameter list in the declarations for both procedure and function headers is the same; it takes the following form:

[(*formal-element* [...;*formal-element*])]

where *formal-element* is

*[*VAR] *identifier-list:[*RECAST *]] type-identifier*

and *identifier-list* is an identifier or a list of identifiers separated by commas.

Using Parameters

Each identifier in the identifier list is referred to as a *formal parameter.* Each formal parameter acts like a locally-declared identifier within the block of the routine. If the formal parameter is preceded by the keyword VAR, then it is known as a *VAR* parameter or a *reference* parameter and each identifier acts like a locally declared variable. Identifiers in VAR parameters can be modified by statements within the routine block; also, they can be passed to other VAR parameters.

A formal parameter list not preceded by the keyword VAR is known as a *value* parameter. These routine value identifiers are treated as local constants: *they cannot be modified within the block; therefore, they cannot be used to pass values out of the procedure.*

Note that, in SP/Pascal, when an actual parameter of some structured type (record, array, file, or string) is passed as a value parameter, a copy of the actual parameter is not constructed.

NOTE: *Standard Pascal specifies that value parameters are copied. SP/Pascal does not copy value parameters for efficiency reasons. To mimic the effect of the standard Pascal value parameter, declare a local variable and assign the parameter to it. Perform the copy only when you must do it.*

The syntax of both the *procedure statement* and the *function designator* specifies an optional *actual parameter list.* The actual parameter lists of both types of routines have the same form:

(*expression [* **,** *...expression]* **)**

The number of expressions, called arguments, in the actual parameter list must equal the number of identifiers in the corresponding formal parameter list to avoid a compilation error. Thus, if a routine has no formal parameter list, its invocation must not have an actual parameter list.

The type of the expression or variable in the actual parameter list must be compatible with the type of its corresponding formal parameter, as defined below.

Examples

The following example shows the declaration of the function EXPON to perform exponentiation, given a power and an integer.

```
TYPE
  POSINT = 0..MAXINT;

FUNCTION EXPON (INT: INTEGER; POWER: POSINT): INTEGER;
  VAR
        I   : POSINT;
        OUT : INTEGER;
  BEGIN
        OUT := 1;
        FOR I := 1 TO POWER DO
        OUT := OUT * INT;
        EXPON := OUT;
  END;
```

A statement using this function could be

```
X := Y * EXPON ( Z, 3 );
```

The following is a sample procedure to perform exponentiation:

```
PROCEDURE EXPON ( VAR OUTCOME :  INTEGER;
                      INT : INTEGER;
                      POWER : POSINT );
  VAR
        I : POSINT;
        OUT : INTEGER;
  BEGIN
        OUT := 1;
        FOR I := 1 TO POWER DO
        OUT := OUT * INT;
        OUTCOME := OUT;
  END;
```

A possible procedure call would be:

```
EXPON ( X, Z, 3 );
```

The following program fragment illustrates nested routines: (Routine declarations can be nested to a maximum depth of eight levels.)

```
PROGRAM NEST(INPUT,OUTPUT);
 PROCEDURE HEADS;
       CONST
              FF = '<014>';
              TAB = '<11>';
              M = 'MONTH';
              D = 'DAY';
              T = 'TEMPERATURE';
       PROCEDURE LINE;
              CONST
                     LINE1  = '____';
                     LINE2  = '________';
       BEGIN

              WRITELN(OUTPUT,LINE1,TAB,TAB,LINE1,TAB,TAB,LINE2);

       END; {LINE}
 BEGIN {HEADS}
        WRITELN(OUTPUT,FF);
        WRITELN(OUTPUT,M,TAB,TAB,D,TAB,TAB,T);
        LINE;
 END; {HEADS}
BEGIN
 HEADS;
END. {NEST}
```

Parameter List Compatibility

The compatibility rules for all types, except string, are identical:

- *Actual parameters that correspond to value formal parameters must be "assignment compatible" with those parameters.*
- An actual parameter that corresponds to a VAR formal parameter must have the same type.
- The actual parameter that corresponds to a VAR formal parameter must be assignable; you cannot pass a constant or a literal as a VAR parameter.
- You cannot pass an element of a byte-aligned array of CHAR nor a single character from a string to a VAR parameter.
- Variant tag fields and packed (non-word-aligned) components of structures cannot be passed as VAR parameters.
- A FOR loop index variable may not be passed as a VAR parameter.

For example, using the following declarations:

```
VAR
    I: 1..10;
    S: STRING 10;
    A: ARRAY [1..10] OF CHAR;
    B: ARRAY [1..10] OF CHAR BIT 16;
    R: RECORD
        F:CHAR;
        G,H: CHAR BIT 8;
             CASE TAG:CHAR OF
        'A'..'Z': (K:CHAR);
        END;

 PROCEDURE P(VAR ARG:CHAR);
```

The following calls to procedure P are illegal:

```
P( S<<I>>);    %Character in a string

P( A[I] );     %Character in a byte-aligned array

P( R.G);       %Character in a byte-aligned record field

P( R.TAG);     %Record tag field
```

The following calls to procedure P are legal:

```
P( B[I] );   %Character in a word-aligned array

P( R.F );    %Character in a word-aligned record field

P( R.K );    %Character in a word-aligned variant field
```

Any string expression can be passed as a string parameter. If the string actual parameter is a substring and is passed to a procedure requiring a corresponding VAR formal parameter, assignment to the formal parameter cannot alter the current length. Any attempt to do so causes a run-time error.

Recasting Routine Parameters

Occasionally, it can be advantageous to override the strong type-checking features in the language. For example, you might want to treat a set as an array of integers in order to print the set as a sequence of octal numbers. You can override the type-checking with the recast option in the routine interface. The inclusion of the keyword RECAST as a prefix to a type identifier in the formal

parameter list allows the actual parameter, whatever its type, to be treated as the formal parameter type for the duration of the routine. For further details on the recast option (particularly for recasting file types), refer to Chapter 7, "Input/Output". For another way to override strong typing, see the WITH (named with extension) statement described in Chapter 5. There are two restrictions on this facility:

1. The actual and formal parameters must occupy the same amount of storage.
2. String pieces cannot be passed as recast parameters.

For example:

```
TYPE
  DOUBLE =
         RECORD
                HI, LO : INTEGER;
         END;
  DBLARRAY = ARRAY [ 1..2 ] OF INTEGER;

VAR
  A : DBLARRAY;
  B : DOUBLE;

PROCEDURE CVTDBL
      ( VAR X :  RECAST DOUBLE; Y :  DOUBLE );
  BEGIN
         X := Y;
  END;
```

Note that, due to strong typing, the assignment A:=B normally is illegal. However, the recast facility allows the assignment to be made using the procedure call CVTDBL (A,B).

Routine Qualifiers

Standard Pascal's program structure requires that all routines be defined in a single program unit, with each routine declared before it is used. However, SP/Pascal permits more modular and flexible programs as detailed in Chapter 8, "Program Structure".

SP/Pascal provides extensions to allow forward reference to routines within the program (the *forward* qualifier); SP/Pascal also provides for separate compilation (or assembly) with the *entry, external,* EXTERNAL CLRE, and *external assembly* qualifiers. The *include* and *overlay* facilities also can be used to expand the modularity of programs. Chapter 5 describes each of these program qualifiers. The SP/Pascal routine calling-sequence details are in Appendix C.

Input/Output

SP/Pascal programs can perform input/output operations using predefined procedures and functions with arguments of type file. The standard Pascal file type, text, allows formatted output of the standard data types. Input also is facilitated, since text files provide for automatic conversion from character data to, for example, real or integer values. This chapter describes Pascal files and explains the predefined procedures used for I/O operations. Table 7.1 summarizes these procedures.

Name	Operation
CLOSE	Closes a file and releases the file buffer space
EOF	Detects end-of-file using the current file position
EOLN	Detects end-of-line using the current character position in a line (text files only)
FGPOS	Retrieves the current file position for random-access read
FILEAPPEND	Opens a file and positions pointer after last data element so additional write operations append data to end of existing file
FSPOS	Sets the file position for random-access write
PAGE	Writes a Form Feed to the text file and terminates the print line
READ	Performs sequential read according to file type and input argument
READLN	Performs sequential read and positions file pointer at the beginning of the next line after the read (text files only)
RESET	Opens an existing file and positions file pointer at beginning of file in preparation for reading
REWRITE	Opens a file and clears the contents in preparation for generating a new file
WRITE	Performs sequential write according to file type and input argument
WRITELN	Performs sequential write according to file type and input argument and outputs a New-Line character (text files only)

Table 7.1 Input/output procedures

File Formats

SP/Pascal data is stored in two different kinds of files: *text files* and *binary files.* Data in text files is represented by sequences of ASCII characters. For example, the integer −32 would be stored in a text file as the three consecutive characters −, 3, and 2. Data in binary (non-text) files always is stored in the same form as its internal representation. In a binary file, −32 would be stored as a single-word item with a bit pattern equal to the binary value −32.

Other points of contrast between the two files are:

1. Text files can be printed or displayed at a console while binary files typically are non-printable.
2. Text files require conversion of numeric data on input and output while binary files do not require conversion.

SP/Pascal provides a third kind of file format, called *variable-length record format* for use in special applications. This format is defined for files with components of type string.

I/O Extensions

To increase the performance of I/O operations in a program, SP/Pascal provides *buffering.* When you open a file, you can select a buffer size by specifying the number of components of the file type that are to be transferred during each system call. (This extension is intended primarily for non-interactive files.) Buffering can be specified with the three file-opening routines, RESET, REWRITE, and FILEAPPEND described in this chapter.

An SP/Pascal-defined ZREL location called P?STD allows you to select run-time checking for certain non-standard capabilities. For I/O operations, these capabilities are reading and writing to the same open file, an extended syntax for real numbers during input conversion, and the value returned for the end-of-line character in text files. The SP/Pascal run-time routines test the value of P?STD to control detection of these extended capabilities. The value in P?STD, by default, is initialized to zero to permit these extensions; however, the value can be redefined at bind time. (A value of 1 indicates that you prefer use of these extensions to be flagged as errors at run-time. The /STD switch to the SP/Pascal Binder macro is described in Chapter 13.)

Files

This section describes file types and details the predefined procedures used to perform input/output using variables of the file type.

Predefined Text Files

SP/Pascal provides two predefined text files for I/O: INPUT and OUTPUT (For assembly language users, these correspond to the channels ?INCH and ?OUCH. See *MP/AOS System Programmer's Reference,* (DGC No. 093-400051.))

The INPUT and OUTPUT files are defined automatically in the language as *entry* variables. Therefore, they can be referred to by other modules. To use these standard files in a program, it is only necessary to declare them in the program heading as in one of the following examples:

```
PROGRAM TESTA(INPUT,OUTPUT);
PROGRAM TESTB(INPUT);
PROGRAM TESTC(OUTPUT);
```

Declaring Files

A file-type declaration for the structured type, FILE has the following format:

identifier = FILE OF *type*

Identifier is the name of the file type, and *type* is one of the types described in Chapter 3.

NOTE: *Type* cannot be FILE, or a structure containing a file type. That is, FILE OF FILE is explicity disallowed.

Examples of valid file declarations follow:

```
TYPE
 INTFILE = FILE OF INTEGER;
 RECFILE = FILE OF RECORD
              HI, LO : INTEGER
           END;

 VAR
   XFILE : INTFILE;
   YFILE : FILE OF STRING 80;
```

Text Files

Text is a standard Pascal predeclared file type. INPUT and OUTPUT are files of this type. A text file contains elements of type CHAR in units called lines. Text files correspond to data-sensitive files in the MP/AOS system. That is, each line is terminated by a delimiter character called the end-of-line character. The default delimiter characters are Null, New Line, Form Feed, and Carriage Return. Using end-of-line characters as line terminators allows the length of a line in a text file to vary.

To declare a text file, write

VAR *file* : TEXT

File is the name of one of the file variables you use in your program for I/O operations.

NOTE: *I/O operations on files of text are extended in SP/Pascal to allow the writing of byte-aligned or packed arrays of characters (type CHAR).*

String Files

The SP/Pascal file of type string has an ANSI- and AOS-compatible variable-length record format. In this format, the first four bytes are used for an ASCII length field, followed by the actual characters. The maximum number of characters in a variable-length record is 9995 as a result of this format. For example:

```
TYPE
  VAR F = FILE OF STRING 80;
  FF:F;
```

FF's elements are strings whose lengths may vary from 1 to 80 characters.

File Variables

For SP/Pascal the definition and use of files is extended so that all file variables have a word size that is specified by the predefined value parameter FRCSZ. (Refer to the assembly language parameters specific to SP/Pascal listed in Appendix E and supplied in the file SYSPASCAL.SR.) When a file is opened with either the RESET, REWRITE, or the FILEAPPEND procedure, space for the file buffer is allocated on the heap (see Chapter 9, "Dynamic Variable Pointers,"

for a description of the heap). This SP/Pascal extension conserves storage for unopened files and permits the actual size of the file buffer to be determined at run-time, an important feature for multi-element I/O operations.

Multi-Element I/O Operations

SP/Pascal permits you to control I/O operations by specifying the number of file elements to be allocated in the in-memory file buffer associated with a file variable. The RESET, REWRITE, and FILEAPPEND procedures are extended to accept an (optional) third parameter that allows you to provide an integer value representing the element count for the buffer.

NOTE: *You must supply a filename (the second parameter) in order to use this extension.*

When the third parameter is specified, all system-level I/O operations are done in units of the element count multiplied by the byte representation size of the element type. The rules used to determine the byte size of a file component are the same as those for determining the size of an array component. (Refer to the end of Chapter 3.) This facility allows a convenient mechanism for multi-element I/O and removes any line limit for text files. It reduces the number of system calls required to process a file, and can improve performance dramatically.

Both text and non-text (binary I/O) files can use this feature. For text files, when the third parameter is omitted, all reads and writes are done as data-sensitive operations, a line at a time. Otherwise, dynamic I/O is performed. The element count feature is transparent when you are using non-interactive files. However, for interactive files, an element count must be used with caution, since it could affect the timing of the I/O. (For an example of this facility, refer to the detailed description of the RESET procedure in the following section.)

I/O Procedures

Before a file can be read or generated, it must be opened. RESET opens a file for reading, REWRITE opens a file for writing, and FILEAPPEND opens a file for writing new data to the end of file. SP/Pascal extensions to the I/O procedures provide CLOSE for closing an open file, as well as, for random-access, FGPOS for retrieving the current file position and FSPOS for setting the file position.

RESET

RESET initializes a file for reading and positions the file pointer at the beginning of the file (the first data item position). The first READ after a RESET accesses the first element in the file. If the file is not empty, EOF is set to false. You must reset all input files except for INPUT before reading from them. The format for the RESET procedure is

RESET *(file [,external-pathname[,components]])*

File is a file variable. The *file* parameter must be assignable; it cannot be a value parameter in the current routine. *External-pathname* is an optional string parameter naming a file or device to be reset. *Components* is an optional parameter for file buffering.

When you do not specify *external-pathname,* the file associated with *file* already must be open. SP/Pascal sets the file position to the beginning of the file. If the file is closed, the procedure causes an error at run-time when P?STD is set. Without P?STD set, the RESET procedure creates a temporary scratch file for output.

For example:

```
VAR F : TEXT;

BEGIN
RESET(F,'ROUTES');
```

Here ROUTES is the external pathname of a file open for reading and associated with file variable F.

Components specifies the number of file components to be allocated in the in-memory file buffer associated with this file variable. This SP/Pascal extended parameter permits you to specify the number of file components to be transferred during each system call. The default value is the number of bytes for a single file component.

For example, consider the following files and calls to the RESET procedure:

```
VAR
   F:TEXT;
   G:FILE OF INTEGER;
        .
        .
        .
RESET(F,'@TTI');        %Call Number 1
        .
        .
        .
RESET(F,'DATA1',512);   %Call Number 2
        .
        .
        .
RESET(G);               %Call Number 3
        .
        .
        .
RESET(G,'DATA2',256);   %Call Number 4
```

In Call Number 1, the text file F is opened on the external file named '@TTI'. Since no third argument is specified, reading from the file is performed with data-sensitive operations. In Call Number 2, F is opened on the external file 'DATA1,' but this time an element count is given. The read operations on this file are done in units of 512 bytes. SP/Pascal correctly processes READLN and EOLN operations on this file using the standard default data-sensitive delimiters: Null, Form Feed, New-line, and Carriage Return.

Call Number 3 opens the non-text file G with no optional parameters. In this case, the variable G must already be open or an error occurs at run-time. The I/O operations are performed in units of two bytes. In Call Number 4, G is opened on the external file 'DATA2' with an element count of 256. The I/O operations are executed in units of 512 (BYTESIZE(INTEGER) * 256) bytes.

SP/Pascal enhances random I/O operations and file updating by permitting optional write operations to be performed on a file opened for input with RESET, according to a global flag P?STD. If P?STD is left set at its zero default value at bind time, both reading and writing are permitted for the file. When P?STD is non-zero, writing to the RESET-opened file causes a run-time error.

REWRITE

REWRITE initializes a file for writing by creating a new file, opening it, and setting EOF to true. The file pointer is positioned at the beginning of the file (the first data item). The syntax is

REWRITE *(file [,external-pathname[,components]])*

File is a file variable. The *file* parameter must be assignable and cannot be a value parameter in the current routine. *External-pathname* is an optional string parameter identifying a file or device to be opened for writing. *Components* is the optional file buffering parameter. You must use the REWRITE command on any output files (except for OUTPUT) before you write. For example:

```
VAR F : TEXT;

BEGIN
  REWRITE(F,'LISTS');
          .
          .
          .
```

When you do not provide an *external-pathname* in the REWRITE command, the *file* variable is closed if it is open, deleted if it is a temporary file, and a temporary scratch file is opened for output. For example:

```
REWRITE(F);
```

Once the temporary scratch file is generated, you can read its contents using the RESET statement without an *external-pathname*, for example:

```
RESET(F);
```

NOTE: *All temporary files are deleted when the program terminates.*

Components specifies the number of file components to be allocated in the in-memory file buffer associated with this file variable. This SP/Pascal extended parameter permits you to specify the number of file components to be transferred during each system call. The default value is the number of bytes for a single file component.

SP/Pascal enhances random I/O operations and file updating by permitting optional read operations to be performed on a file opened for output with the REWRITE command, according to a global flag P?STD. If P?STD is set to zero, both reading and writing are permitted for the file. (The default for P?STD is zero.) When P?STD is set to a non-zero value with the /STD switch at bind time, writing to the file opened with the RESET command causes a run-time error.

FILEAPPEND

This statement can be used in place of REWRITE to append data following the last element in the file. FILEAPPEND opens the file or device specified by *external-pathname*. If the file does not exist, it is created. In the case of a pre-existing file, the file pointer is set immediately after the end of the former data, and EOF is set to true. The syntax is

FILEAPPEND *(file, external-pathname[,components])*

File can be a file of any type. The *file* parameter must be assignable and cannot be a value parameter in the current routine. *External-pathname* is a string parameter identifying a file or device to be opened for appending data. Notice that *external-pathname* is a required argument to this routine. *Components* is the optional file buffering parameter.

For example, to update a file LISTS created in the REWRITE statement:

```
BEGIN
    FILEAPPEND(F,'LISTS')
```

It is also possible to create a new file. In that case, the file pointer points to the beginning of the new file. For example:

```
BEGIN
    FILEAPPEND(F,'NEWF')
```

where NEWF is the *external-pathname* of a new file or device.

Components specifies the number of file components to be allocated in the in-memory file buffer associated with this file variable. This SP/Pascal extended parameter permits you to specify the number of file components to be transferred during each system call. The default value is the number of bytes for a single file component.

SP/Pascal enhances random I/O operations and file updating by permitting optional read operations to be performed on a file opened for output with FILEAPPEND, according to a global flag P?STD. If P?STD is set to zero, both reading and writing are permitted for the file. (The default for P?STD is zero.) When P?STD is non-zero, writing to the RESET-opened file causes a run-time error.

CLOSE

CLOSE closes a specified file and releases the file buffer space in the heap. The syntax is

CLOSE *(file)*;

File is a file variable. The *file* parameter must be assignable and cannot be a value parameter in the current routine. For example:

```
        VAR F : TEXT;

    BEGIN
      .
      .
      .
  CLOSE(F);
```

If the file specified with the CLOSE command is not open, a run-time error occurs.

File-Positioning (Random-Access)

Two predefined procedures are provided in SP/Pascal for file positioning for random access, FGPOS, for retrieving the current file position, and FSPOS, for setting the current file position.

FGPOS

FGPOS returns the current file pointer position, a 32-bit number pointing to the next byte in the file to be read or written. FGPOS is the equivalent to the system call ?GPOS. The syntax is

FGPOS *(file,position)*

File is a file variable. The *file* parameter must be assignable; it cannot be a value parameter in the current routine.

Position represents a 32-bit unsigned quantity. *Position* must be an assignable parameter with a size equal to two words.

For an example of FGPOS, refer to the following paragraphs on FSPOS.

FSPOS

FSPOS allows you to set the file pointer position to the value specified. FSPOS is the equivalent to the system call ?SPOS. The syntax is

FSPOS *(file,position)*

File is a file variable. *Position* represents a 32-bit unsigned quantity: the position, after the call, of the next byte to be read or written.

The following program illustrates the use of FGPOS and FSPOS to perform random-access operations on a file of records. The records in the file simulate a list, which uses the file position as links. Remember that after a read or write operation, the file position is advanced to the next sequential record.

```
PROGRAM DISK_LIST;
TYPE
        POSITION = RECORD HI,LO:INTEGER END;
        DISK_REC = RECORD
                        FORWARD_LINK,BACKWARD_LINK:POSITION;
                        DATA:STRING 80;
                        END;

CONST
        NIL_LINK = POSITION(-1,-1);
        FIRST_REC = POSITION (0,0);

VAR
        CURR_REC,NEW_REC:DISK_REC;
        DATA_BASE:FILE OF DISK_REC;
        CURR_POSITION,NEXT_FREE_POSITION:POSITION;

PROCEDURE APPEND_RECORD;
BEGIN
        NEW_REC.FORWARD_LINK:= CURR_REC.FORWARD_LINK;
        NEW_REC.BACKWARD_LINK:= CURR_POSITION;
        CURR_REC.FORWARD_LINK:= NEXT_FREE_POSITION;
        FSPOS(DATA_BASE,CURR_POSITION);
        WRITE(DATA_BASE,CURR_REC);
        FSPOS(DATA_BASE,NEXT_FREE_POSITION);
        WRITE(DATA_BASE,NEW_REC);
        FGPOS(DATA_BASE,NEXT_FREE_POSITION);
END;

BEGIN
        FILEAPPEND(DATA_BASE,'TEST');
        FGPOS(DATA_BASE,NEXT_FREE_POSITION);
        CURR_POSITION:= FIRST_REC;
        IF NEXT_FREE_POSITION = FIRST_REC THEN BEGIN
```

```
                WITH CURR_REC DO BEGIN
                        FORWARD_LINK:= NIL_LINK;
                        BACKWARD_LINK:= MIL_LINK;
                        DATA:= 'INITIAL DATA';
                END;
                WRITE (DATA_BASE,CURR_REC);
                FGPOS(DATA_BASE,NEXT_FREE_POSITION);
END
ELSE BEGIN
                FSPOS(DATA_BASE,FIRST_REC);
                READ(DATA_BASE,CURR_REC);
        END
        NEW_REC.DATA:= 'EXAMPLE DATA';
        APPEND_RECORD;
END.
```

Text File Position-Testing

The two standard I/O functions, EOF and EOLN are used to test for and identify the end of a file (EOF) and the end of the current line (EOLN) within a text file. These functions can be used to test the file position during reading.

EOF

The syntax for EOF is

EOF *(file)*

EOF returns the Boolean constant true when the end of the *file* is reached. Otherwise, the function returns the Boolean constant false. EOF can be used on both text files and binary files. If *file* is omitted, EOF refers to the standard file INPUT.

EOLN

The syntax for EOLN is

EOLN *(textfile)*

EOLN returns the Boolean constant true if the end of the current line in *textfile* is reached, that is, if the next character to be read is one of the legal line delimiters (Carriage Return, New-line, Form Feed, or Null). Otherwise, the function returns the Boolean constant, false. If *textfile* is omitted, EOLN refers to the standard file INPUT.

NOTE: *This function applies only to files of type TEXT.*

I/O Procedures (Text Files)

There are five procedures for performing I/O operations on text files: READ, READLN, WRITE, WRITELN, and PAGE.

Note that default file references are supported in each of these procedures. When you omit the file variable in an output reference such as WRITE or WRITELN, the standard file OUTPUT is used.

Similarly , if you omit a file variable in an input reference such as READ, READLN, EOLN or EOF, the standard file INPUT is used. If you intend to use INPUT and OUTPUT files, either by specific reference or by default, you must declare them in your program heading.

READ (Text Files Only)

The syntax for the READ procedure is

READ *([file,] var$_1$[...,var$_n$])*

The READ routine sequentially reads in data from *file* (or from the standard text file INPUT if *file* is omitted). After the read, the file pointer points after the last item read. *Var$_1$* through *var$_n$* are variables. For text files, allowable types for these variables are real, character, string, boolean, integer, whole or double_real, or subranges of integer, character or boolean.

Data is associated with a variable according to its position in the file. That is, the first data value in the file is associated with the first variable, the second value with the second variable, and so on.

When reading integers or reals, data is scanned until a character that cannot be part of a valid numeric constant of that type is encountered. Following this scan, the characters are used to build a signed number whose value is then assigned to the variable in the READ statement. If the sequence of characters does not form a signed number, or if the value assigned to the variable is not assignment-compatible with the type of variable, the result is a run-time error.

For error processing on conversion, use the appropriate DG-supplied routines (ST2SI and ST2DI) described in Chapter 10. When using the procedures, you should read numeric values into a string and then convert them.

To read reals and double-reals, if P?STD is zero (default), the syntax of reals is extended to include the syntax of integers, *e.g.*, 1.0 and 1 are *both* valid. The syntax is restricted to standard Pascal style if P?STD is non-zero. You may also use exception handling to recover from conversion errors (see Chapter 8).

The format for entering reals or integers is the same as the legal format described in Chapter 3 for integer, whole, real and double_real constants. Decimal values are the only values permitted; the alternate radix notation is not supported.

Reading a character simply assigns the next character in the text file to the character variable. This could cause EOLN (*textfile*) to become true, in which case the character returned is the line terminator.

In SP/Pascal you can choose to have a space character returned as the end-of-line character, rather than having the actual delimiter character in the file (New-line, Carriage Return, Null or Form Feed) returned when EOLN is true. When P?STD is non-zero, the space character is used; otherwise, the usual delimiting characters are used.

String variables are flexible. In SP/Pascal, string variables do not overflow when read, and do not produce an error. In SP/Pascal, when the number of characters remaining in the current line exceeds the declared maximum length of the string, the string is filled only to its maximum length, permitting you to perform some simple input formatting. If the number of characters remaining in the input line is less than or equal to the maximum length of the string, then the remaining characters are assigned to the string, and its length is adjusted.

Reading strings from a textfile is *data sensitive,* that is to say, the remainder of the current line and the delimiter are assigned to the string. If EOLN *(textfile)* is true, only the delimiter is assigned to the string.

When reading Booleans from a text file, the system reads one character (upper-case or lower-case) at a time and attempts to match the largest substring corresponding to either true or false.

The following illustrates a READ procedure. Note that, as an example of the default file parameter, *file* is omitted in the output reference WRITE.

```
          PROGRAM IOTEST(INPUT,OUTPUT);

VAR
      F : TEXT;
      LINE, STARS :  STRING 80;

BEGIN

      RESET(F,'PROSE');
WHILE NOT EOF(F) DO


BEGIN
      READ(F,LINE);
      WRITE(LINE);
      END;
END.
```

READLN (Text Files Only)

The syntax for READLN is

READLN *[([textfile,] [var][...,var])]*

READLN reads in data from a *textfile* in the same way as READ except that after the procedure call is executed, the pointer is positioned at the beginning of the next line. (Before READLN can be used, the file must be opened with the RESET procedure.) If there is no next line (EOF (*textfile*) is true), then any further request to read that file causes an error.

Examples

```
READLN(F);
```

This statement causes the pointer to skip to the next line in the file. Assume a textfile contains the following characters:

```
54   123   19
4     12   157
349   10   45
```

If variables I and J are declared as integers, the statement

```
READLN(F,I)
```

assigns I the value 54. The two remaining numbers on the line are ignored and the pointer is set at the beginning of the second line. If the next operation is

```
READLN(F,J)
```

J is assigned the value 4, and the pointer is set at the beginning of the third line. If VAR is not specified, the pointer skips to the beginning of the next line.

NOTE: *This procedure applies to text files only. Non-text files can be read only with the READ procedure.*

WRITE (Text Files Only)

The syntax for WRITE is

WRITE *([file,]item[...,item$_n$])*

Once a file is opened with REWRITE or with FILEAPPEND, WRITE can be used to write the value(s) of the *item(s)* to the specified *file* or to the standard file OUTPUT when you omit the optional *file* argument. The value(s) of the *item(s)* are written in the same order as expressed in the actual argument list. *Items* can be expressions of type character, Boolean, real, string, integer, whole, double_real, or subranges of integer, character, whole, or Boolean.

Items of type CHAR or string are written without conversion, but expressions of other allowable data types are converted to strings; Boolean values are written as either true or false; real values are written in scientific notation; integer or subranges of integer values are written in signed decimal notation.

The values of *items* are printed with a default *field width* (number of columns), which depends on the data type.

You can specify an explicit field width to override the default with the following format:

WRITE *(file, x:y)*;

File is the name of the file; *x* is the name of the item which can be of type integer, Boolean, whole, double_real, character, string, or real; *y* is the maximum field width to be assigned to the item.

If the size of *x* is less than *y* (that is, if the number of characters required to represent *x* is less than the maximum field width assigned to it), *x* is preceded by a number of blanks equal to the difference in width between the size of *x* and *y*. If *y* is equal to zero, and the type of *x* is either CHAR or string, then nothing is printed. If P?STD is non-zero, then setting *y* equal to zero causes an error. If P?STD is zero, or if the type of *x* is other than CHAR or string, then setting *y* to zero or less causes an error.

To write real numbers in fixed-point notation, use the format

WRITE *([textfile,] x:y:z)*;

File is the name of the file; *x* is the item; *y* is the maximum field width; and *z* is the number of digits to appear following the decimal point. Again, if *y* is greater than the width of *x*, *x* is preceded by a number of blanks equal to the difference between *x* and *y*. If *y* is less than or equal to zero, a run-time error occurs.

When you do not specify a field width, SP/Pascal uses the default values shown in Table 7.2.

Type	Default Field Width (No. of Characters)
INTEGER	Numeric representation and a sign (if negative)
CHAR	1
BOOLEAN	5
REAL	12 (*(sign)x.yyyyyE(sign)ZZ*)
STRING	Current string length
WHOLE	Numeric representation (no sign)
DOUBLE_REAL	20 (*(sign)x.y*^{13} E *(sign ZZ)*)

Table 7.2 Default values for WRITE field width

Because WRITE generates a continuous print line, you must specify a termination point for the line. You can do this with WRITELN, which terminates the current line and emits the New-line character. The PAGE procedure can also be used to terminate a line.

NOTE: *You must REWRITE or FILEAPPEND all files except OUTPUT before writing to them.*

Example 1

```
      PROGRAM IOTEST(INPUT,OUTPUT);
VAR
       F :  TEXT;
       LINE,STARS :  STRING 80;
BEGIN
       .
       .
       .
       REWRITE(F,'PROSE');
       WHILE (LINE < > STARS) DO BEGIN
       READ(INPUT,LINE);
       WRITE(F,LINE);
       END;
END.
```

Example 2

```
    PROGRAM FIELDWIDTH(INPUT,OUTPUT);
    VAR
     F :  TEXT;
     NUM,COUNT,M :  INTEGER;
          BEGIN

     COUNT := 0;
     READ(NUM);
     REPEAT
     M := NUM * 10;
     WRITELN(NUM:5,M:10);
     COUNT := SUCC(COUNT)
     NUM := M;
     UNTIL(COUNT = 4);
          END.
```

Note that *file* is omitted from the input and output statements.

The output of this program looks as follows:

```
    1        10
   10       100
  100      1000
 1000     10000
```

WRITELN (Text Files Only)

The syntax for WRITELN is

WRITELN *[([textfile,] [item][...,item$_n$])]*

WRITELN terminates a print line. You can use WRITELN as a line terminator immediately after a WRITE statement, or you can use it instead of a WRITE statement as in the following two equivalent examples:

```
WRITE (X, A + B); WRITELN(X);

WRITELN (X, A + B);
```

X is a text file and A and B are variables of integer or real type. When you do not specify a text file, the standard file OUTPUT is used.

NOTE: *WRITELN is permitted only on files of type text. Non-text files must use the WRITE procedure.*

You can specify field widths with WRITELN. See WRITE for details.

The following program illustrates one method of prompting a user for input. The sequence of I/O statements must be executed in the order shown. First, the prompt message is displayed. Second, a test for end-of-file should be performed. Finally, the actual data is read.

```
PROGRAM DISPLAY_CHAR(INPUT,OUTPUT);
CONST EXITED = FALSE;
VAR CH:CHAR;
BEGIN
        REPEAT
                WRITE ('TYPE A CHARACTER : ');
                IF EOF(INPUT) THEN EXITLOOP;
                READLN(CH);
                WRITELN('THE DECIMAL VALUE IS ',ORD(CH));
        UNTIL EXITED;
END.
```

The following program demonstrates simple loops to read a file until the end-of-file condition occurs and to read a line of text until the end-of-line occurs.

```
PROGRAM REMOVE_TABS(INPUT,OUTPUT);

CONST
        TAB_SIZE = 8;
        TAB= '<11>';
        BLANK = ' ';
        NULL = '<0>';

 VAR INFILE,OUTFILE:TEXT;
    NEXTCHAR:CHAR;
    COLUMN,NUM_BLANKS,ITERATION:WHOLE;
    FILE_NAME:STRING 136;

BEGIN
    WRITE('Name of input file: ');
    IF NOT EOF THEN READLN(FILE_NAME) ELSE RETURN;
    FILE_NAME<< LENGTH(FILE_NAME) >>:= NULL;
    RESET(INFILE,FILE_NAME,512);
    WRITE('Name of output file: ');
    IF NOT EOF THEN READLN(FILE_NAME) ELSE RETURN;
    FILE_NAME<< LENGTH(FILE_NAME) >>:= NULL;
    REWRITE(OUTFILE,FKILE_NAME,512);
    WHILE NOT EOF(INFILE) DO BEGIN
        COLUMN:= 1;
        WHILE NOT EOLN(INFILE) DO BEGIN
                READ(INFILE,NEXTCHAR);
                IF NEXTCHAR = TAB THEN BEGIN
                        NUM_BLANKS:=TAB_SIZE - (COLUMN MOD TAB_SIZE);
                        FOR ITERATION:= 1 TO NUM_BLANKS DO
                                WRITE(OUTFILE,BLANK);
                        COLUMN:= COLUMN + NUM_BLANKS;
                        NEXTCHAR:= BLANK;
                END;
                WRITE(OUTFILE,NEXTCHAR);
                COLUMN:= SUCC(COLUMN);
        END;  {not eoln}
        READLN(INFILE); {next line}
        WRITELN(OUTFILE);
END.  {not eof}
```

PAGE (Text Files Only)

The syntax for PAGE is

PAGE *(textfile)*

PAGE writes a Form Feed to the text file and terminates the print line.

I/O Procedures (Non-Text Files)

Two procedures perform I/O operations on non-text files: READ and WRITE. Non-text files include files with binary data and files of strings.

READ

The syntax for a READ procedure is

READ *(file, var_1[...,var_n])*

READ sequentially reads in data items from the file named by *file* and assigns those items to the actual arguments in the parameter list (var_1,...,var_n in the above format). Each argument must be of a type *identical* to that of the file. The elements of the files are transferred directly: no data conversion is performed.

WRITE

The syntax for a WRITE procedure is

WRITE *(file, $item_1$[...,$item_n$])*

WRITE sequentially writes the value(s) of the item(s) to *file*. The *items* are expressions of the file type. *No* data conversion is allowed and none is performed. As with any other file except OUTPUT, you must REWRITE or FILEAPPEND the file before writing to it.

Files of Type String

SP/Pascal treats files of type string in a special way. These files are composed of variable-length records. Each record consists of an integer four-byte count as its first field, followed by the actual data bytes. The maximum byte count is specified by the length of the string in the file type definition but is not to exceed 9995.

For example,

```
TYPE
  F = FILE OF STRING 20
```

specifies a maximum record length of 20.

When writing to a file of type string, the current lengths of all arguments must be less than or equal to this maximum byte count. The length of the string argument is written, followed by the current string value.

When reading from a file of type string, the string argument is initialized to the contents of the next record in the file. If the string MAXLENGTH is not large enough to accommodate the current record, a run-time error is generated.

This file format is ANSI standard, and can be read, under AOS, using the variable record format.

SP/Pascal Program Structure

SP/Pascal provides for modular design in the program structure itself. While a standard Pascal program must be a self-standing unit that contains all of the information and code necessary for processing and execution, SP/Pascal programs are not limited to a single unit. Separate compilation units (precompiled object modules) can be bound into a final program.

This chapter describes the modular structure of a program written in SP/Pascal. Details of the source statements that make up the various program components are provided in other chapters. Chapter 13 covers program compilation and provides an overview of program implementation from compilation through program testing and execution.

Source Program Components

An SP/Pascal program can include separate program and module compilation units along with various program qualifiers. The modular structure of the language enables an SP/Pascal program to

- accept separate compilation units for modularity of programming;
- share routines and variables among modules;
- interface with assembly language routines or with any language that supports Common Language Run-Time Environment (CLRE);
- send and receive system data through calls to the MP/AOS operating system.

Separate Compilation Units

A compilation unit is an already-compiled object module. There are two kinds of compilation units: *program* and *module.* The units serve as separate components that make up an executable SP/Pascal program. Every executable program developed in SP/Pascal is made up of exactly one program compilation unit; optionally, the program can contain one or more module compilation units. Every program or module that is compiled produces a separate object file.

A *program compilation unit* is an executable unit that contains an executable statement section. A program can also contain a declaration section of program variables and routines along with references to routines. The program could be an independent source program that does not require additional code, or it could be a program that uses routines and declarations external to it (in one or more module compilation units).

A *module compilation unit* is a unit that contains one or more declarations (for constants, variables, procedures and/or functions), but no executable statement section.

Program Components

A source program contains a *program-heading* and a *program-block*. The one-line program-heading gives the whole program its name and lists the program-parameters. The term *program-block* refers to the remainder of the source program. It includes a declaration section, which sets out all of its specific definitions, and an executable section, which specifies the operations to be performed on those definitions.

The syntax for the program heading is

PROGRAM *program_name [program_parameters];*

Program_parameters is a parenthesized comma-list of identifiers representing external data files. The acceptable arguments are INPUT and OUTPUT. (I/O is described in Chapter 7.) Examples of typical program headings follow:

```
PROGRAM MY_PROG(INPUT,OUTPUT);
PROGRAM READFILE;
PROGRAM MY_REPORT(OUTPUT);
```

The program-block of a simple program consists of a *declaration section* and the *statement section*. A large program's program-block can be made up of an extensive declaration section with global declarations, external routine definitions and external variable references to be resolved at bind time, and directives for including other source files in the same compilation. The declaration section can have a number of procedure declarations, each with its local variable declarations and statements. Detailed descriptions of these components appear in Chapter 3.

- **Declaration section**: declarations of all the global variables, constants, and routines used by the program. All program identifiers must be defined before they are used.
- **Statement section**: sequence of executable program statements, delimited by BEGIN and END statements.

For example, a typical source program submitted to the compiler has the following general format:

```
PROGRAM HEADING
DECLARATION SECTION
          CONSTANT DECLARATIONS
          TYPE  DECLARATIONS
          VARIABLE DECLARATIONS
          EXTERNAL DECLARATIONS
          INCLUDE SPECIFICATIONS
          PROCEDURE & FUNCTION DECLARATIONS
   STATEMENT SECTION
          BEGIN
          STATEMENT(S)
          END.
```

Note that the final END statement completing the statement section is followed by a period (.). This syntax indicates the final program statement to the SP/Pascal compiler.

Module Components

Like a program, a module compilation unit contains a *module-heading* and a *module-body*. However, a module cannot execute as an independent unit; it requires binding with a program. The heading names the module, and the block of source text can contain any number of declarations. The text can be one or more complete routines, a number of commonly used procedure or function declaration sections, constant declarations, or variable declarations.

The format of a module heading is

MODULE *module-name*

For example,

```
MODULE MY_IO;
```

identifies the type of compilation unit and names it MY_IO.

For modules with routine declarations, the final END delimiter is followed by a semicolon. For modules without routines, the final declaration is followed by a semicolon.

A module consists of a module header followed by a declaration section. A typical source module submitted to the compiler has the following general format:

```
MODULE HEADING

DECLARATION SECTION
            CONSTANT DECLARATIONS
            TYPE  DECLARATIONS
            VARIABLE DECLARATIONS
            EXTERNAL DECLARATIONS
            INCLUDE SPECIFICATIONS
            PROCEDURE & FUNCTION DECLARATIONS
```

Program Qualifiers

Normally, all routines and variables are defined in a single program unit, and a routine or variable must be declared before it is used. SP/Pascal allows forward reference to routine declarations within the program or module (the FORWARD qualifier); SP/Pascal also provides for separate compilation (or assembly) with the ENTRY and EXTERNAL qualifiers (which can apply to both routine and variable declarations). The INCLUDE and OVERLAY facilities also can be used to expand the modularity of programs. The following sections describe each of these program qualifiers and facilities.

Variable Qualifiers

The syntax of variable declarations is extended to allow storage qualifiers as follows:

[*qualifier*] VAR *var-declaration1 [...;var-declaration2]*

Where *qualifier* for a variable declaration can be ENTRY, EXTERNAL, ZREL, ENTRY ZREL, or EXTERNAL ZREL. Each qualifier is detailed in the following paragraphs. The two external qualifiers (EXTERNAL and EXTERNAL ZREL) can be used in variable declarations that occur at any routine level. The three remaining qualifiers (ENTRY, ZREL, and ENTRY ZREL) can be specified only for variables that are declared at the global level. This restriction is necessary because these variables are allocated static (fixed) program storage locations.

ZREL Variable Designator

A new storage class qualifier for variables is provided in SP/Pascal. The reserved word ZREL can be used to direct the compiler to allocate storage in the zero-relative or ZREL (page 0) partition. The qualifier must precede the keyword 'VAR' in the variable declaration. Only variables declared at the global level can have this qualifier present. For example:

```
ZREL VAR i:integer
```

Variables that reside in the ZREL partition can be accessed more efficiently than other global variables. For this reason, frequently used variables should be placed in this partition. However, the total size of ZREL cannot exceed 216 words.

NOTE: *Not all ZREL storage space is available for program variables; a small amount of ZREL storage is used by some of the run-time routines.*

ENTRY Variable Designator

The ENTRY designator makes a variable visible ("global") for reference by other modules. Once a variable is declared as an ENTRY variable it can be shared among any number of modules. A shared variable is directly accessible to the routines in the modules that share it. Using shared variables provides an alternative to passing variable values as routine parameters.

ENTRY variables are allocated in the ZREL area (when the ZREL qualifier is specified), or in the program impure area. This convention guarantees that the storage area for variables is available for the duration of the program. For example:

```
ENTRY VAR f:text;
ENTRY ZREL VAR t:boolean;
  i,j:integer;
```

To share a variable, one module declares it with an ENTRY qualifier, and all other modules declare it with an EXTERNAL qualifier.

EXTERNAL Variable Designator

The EXTERNAL designator specifies a variable as having its storage allocated by another compilation unit. That is, the variable's value is available for use throughout the entire program. The EXTERNAL designation makes the variable names globally visible from the module (or routine) in which it appears. (Variables declared as EXTERNAL reserve no space in the compilation unit.) These variables then can be referenced to like any other global variables. For example:

```
MODULE utilities;

CONST STACK_MAX = 20;

EXTERNAL VAR stack:array[1..stack_max] of 0..maxint;
EXTERNAL ZREL VAR stack_top:0..stack_max;
                  stack_overflow : boolean; {odd}
ENTRY FUNCTION previous:integer;
begin
   if stack_top > 1 then previous:= stack[pred(stack_top)]
   else previous:= -1;
end;
```

Routine Qualifiers

The syntax of routine declarations is extended to allow program qualifiers as follows:

[storage-qualifier [call-qualifier]] [FORWARD]*routine-declaration*

The storage qualifiers for routines are ENTRY and EXTERNAL. The allowable call qualifiers are CLRE and ASSEMBLY. The permissible combinations of storage and call qualifiers are ENTRY CLRE, EXTERNAL CLRE, and EXTERNAL ASSEMBLY. The FORWARD qualifier also can be used in combination with the ENTRY qualifier. Any of these qualifiers can be omitted from a routine declaration. When the storage qualifier is not specified, the routine name is visible (global) only to the current module. When the call qualifier is omitted, the normal SP/Pascal calling convention is used by default. (SP/Pascal routine calling conventions are detailed in Appendix C.)

The SP/Pascal CLRE qualifier allows for interface with other Common Language Run-time Environment (CLRE) languages. When you declare ENTRY CLRE, SP/Pascal expects all required parameters to be passed in the standard CLRE stack order, rather than by the default accumulator method. The EXTERNAL CLRE declaration causes parameters to be passed in the standard CLRE stack format. Details of CLRE processing are provided in Appendix C. A sample EXTERNAL CLRE declaration is:

```
EXTERNAL CLRE PROCEDURE MOLE (VAR MW:REAL; G:REAL; MOLAR:REAL);
```

External Assembly Routine Designator

SP/Pascal provides several mechanisms for interfacing with routines written in assembly language. The EXTERNAL ASSEMBLY designator directs the compiler to generate the same calling sequence and parameter passing conventions that are generated by the MP/Pascal compiler. This designator is used to permit a common interface to assembly routines for both SP/Pascal and MP/Pascal programs. SP/Pascal assembly language routines also can use the default calling

convention (used for other SP/Pascal routines) or the CLRE calling convention described in Appendix C. (Like SP/Pascal routines, the assembly language routine must have a unique name within the program.)

External assembly procedures are declared in the same way as external procedures. See Appendix C for details about communicating with assembly language.

FORWARD Routine Reference

The FORWARD qualifier allows you to refer to a routine that is not yet declared but that does appear further on in the same source. (The compiler must be able to resolve the routine's name as a defined symbol, or the declaration is considered an error.) The language allows you to identify just the routine name and any parameters it takes without declaring the full procedure or function until further on in the source; the forward reference is declared by specifying the routine identifier, routine name, and parameters, along with the reserved word FORWARD.

A forward reference can be useful with two routines, each of which calls the other. Additionally, forward references provide for several routines that call each other. The keyword FORWARD is used in a routine identifier and can appear either before or after the routine name as follows:

[ENTRY] [FORWARD] *PROCEDURE | FUNCTION routine [(parameter-list)]*

[ENTRY] *PROCEDURE | FUNCTION routine [(parameter-list)]*; FORWARD;

If the first form is used, there is no procedure body. If the second is utilized, then FORWARD replaces the procedure body.

SP/Pascal allows you to repeat a parameter list in a procedure or function declaration that is referred to by a forward declaration. (This is an extension to standard Pascal.) The repeated parameter list is optional. If it is present, it is checked for identity with the one given with the forward definition. For the repeated list to be identical, all the parameter names and modes (VAR or value) and the parameter types must match. This extension makes the source program easier to read, since it is no longer necessary to separate the arguments textually from the routine body that refers to them. For example, the following declarations are accepted:

```
PROCEDURE P(VAR CH:CHAR; LINE:STRING 10); FORWARD;
        .
        .
        .
PROCEDURE P(VAR CH:CHAR; LINE:STRING 10);
```

However, the declaration

```
PROCEDURE P(VAR X:CHAR; LINE:STRING 10);
```

would be diagnosed as an error by SP/Pascal.

ENTRY Routine Designator

The ENTRY designator makes a routine visible to calls from other modules. (The calling module designates the routine as outside by a corresponding EXTERNAL declaration.)

EXTERNAL Routine Designator

The EXTERNAL designator specifies a routine as being declared outside of the program or module compilation unit. A routine designated as EXTERNAL to a compilation unit must be declared as an ENTRY in another unit.

Referring to External Routines or Variables

The routine or variable name must be unique; it cannot be the same as any other routine or variable name in any other module of the program. If there are two global routines or variables with the same name but in different modules, an error is detected at bind time.

NOTE: *SP/Pascal creates external references only for those routines and variables that actually are accessed in the module being compiled. This convention is another aid for program development and the construction of* include *files.*

In SP/Pascal, benign redefinition is permitted only between two EXTERNAL definitions or an EXTERNAL and an ENTRY definition. Parameter lists must use identical names for the parameters and their types. (This is the same rule used for repeated parameter lists in FORWARD referenced routines.)

Example of Separate Routines

In the following example, the module named TWO illustrates the declaration of external variables and routines to match the ENTRY declarations in module ONE. EXTERNAL routine designators, like those for FORWARD and ENTRY routines, consist only of the designator keyword, the routine name, and any parameters.

```
MODULE ONE;
 ENTRY VAR        COUNT: INTEGER;
 ENTRY PROCEDURE PRINT (MESSAGE: STRING 80);
    BEGIN
        COUNT:=COUNT+1;
        {Number of times called}
        .
        .
    END;
```

```
        {End of module ONE}

MODULE TWO;
      .
      .
  EXTERNAL VAR COUNT: INTEGER;
       {Allows Count to be referenced}
   EXTERNAL PROCEDURE PRINT (MESSAGE: STRING 80);
       {Allows Print to be called}
.
.
     {End of module TWO}
```

Include Facility

The INCLUDE facility permits the compiler to take the source text from different files. The format is:

INCLUDE *pathname;*

Pathname is limited to 127 characters; it is not enclosed in quotes, and the syntax is operating-system dependent. INCLUDE nesting is limited to eight levels, including the original program file. Each *pathname* requires its own INCLUDE statement. For example,

```
INCLUDE DOUBLE.PAS;
```

Chapter 10 describes some useful include files supplied by Data General Corporation.

Overlay Facility

The OVERLAY qualifier specifies a separate compilation module as an overlay. The format is:

OVERLAY MODULE *module-name;*

Module-name is initialized to the .ENTO value of the overlay. This information is needed to manage overlays in a program. (Refer to *MP/AOS System Programmer's Reference* (DGC No. 093-400051.)

For example,

```
OVERLAY MODULE INITIALIZE;
    .
    .     {module code}
    .
          {end of module Initialize}

PROGRAM P;

EXTERNAL VAR INITIALIZE: INTEGER;
    .
    .     {Initialize contains the .ENTO for the module Initialize}
    .
```

At this point, the DG-supplied routines in OVLY.PAS can be used to load the overlay module. Refer to Chapter 10 for details about OVLY.PAS.

Overlay module global variables that are not entry points are allocated storage in unlabelled common. Therefore, their values are valid only when the overlay is resident. They do not persist from one use of the overlay area to another.

Overlaying SP/Pascal Programs

Overlaying is a technique that reduces the memory requirements of a program. In a typical program much of the code is used infrequently and does not need to be in memory while it is not being used. With overlays, you can divide a program so that routines reside on disk until they are required for execution.

You can divide an SP/Pascal program with the following steps:

1. Determine areas of code that are called only in rare instances (routines for handling unusual conditions, routines for reducing data, any non-critical processing instruction, etc.)
2. Modularize the selected code in one or more routine declarations.
3. Place OV?LD and OV?RL calls in the program to load the overlay code into memory from disk and to release the code for overwriting by another code segment.
4. Direct the MP/AOS binder to allocate space in the executing program for overlays (one entry for each overlay node).

To utilize overlaying, design your program with one or more overlay nodes. An overlay node is a block of memory allocated during binding. The node receives each segment of overlay code for execution as it is needed. Each node has its own set of overlays, and can hold one overlay at a time.

Managing Program Overlays - ?OVLOD and ?OVREL

The system keeps track of currently loaded overlays. If your program is multitasked, then several tasks can share a node, if all the tasks request the same overlay. If a task requests a new overlay in a node that is already in use by another task, then the requesting task is blocked from execution until the node becomes available.

This handling of tasks under multitasking ensures that the nodes are correctly managed in a way that is transparent to all tasks, provided that your program observes the following conventions:

- All tasks explicitly call and release overlays using the OV?LD and OV?RL routines.
- Tasks that use overlays can tolerate some delays when calling them. (You can minimize these delays by optimizing your overlay structure at bind time.)

The overlay facility is designed to be flexible. The exact distribution of nodes and overlays is not specified until bind time, so no program modification is needed to try out several different strategies. This makes for ease in reorganizing overlays for greatest efficiency.

The binder links together the various inter-module references and allocates space for the overlay nodes. It also builds an overlay file containing all the overlays and places overlay control information in the program file. The overlay file has the same filename as the program, but with a suffix of .OL instead of .PR. For further details on binding, refer to Chapter 13 of this manual, and *MP/AOS Macroassembler, Binder, and Library Utilities* (DGC No. 069-400210).

Source Program Example

The following program example is provided to illustrate program structure, its components, the use of external and entry routines and variables, include files, overlay modules and I/O statements.

The sample program is designed to analyze text files and produce a report showing the relative frequencies of alphabetic characters in the file. The user is prompted for the names of the text files to be analyzed. The program produces statistics for each text file and a summary of all the files analyzed.

The program has three compilation units. The program or main module names text files and calls procedures in the other two modules; the modules are compiled separately for later binding with the program. One of these modules contains procedures for counting characters; the other contains procedures for generating statistics and printing a report. (Certain routines in the program are incomplete; comments are included in these routines to indicate the actions of the omitted code.)

```
PROGRAM example(input,output);

CONST pathlength = 127;
      null_char = '<0>';

TYPE letters = 'A'..'Z';
       stat_rec = RECORD
```

```
                lines:integer;
                                freq:ARRAY[ letters ] OF integer;
      END;
     filename = STRING pathlength;

VAR  source,listfile:filename;
           infile,outfile:text;
           current_stats:stat_rec;
           file_count:integer;
           done:Boolean;

ENTRY VAR totals:stat_rec;

EXTERNAL VAR ovy_one,ovy_two:integer;

EXTERNAL PROCEDURE count_chars(VAR f:text; VAR stats:stat_rec);

EXTERNAL PROCEDURE print_report(VAR f:text; name:filename;
                                        VAR stats:stat_rec);

INCLUDE ovly.pas;  {Overlay management routines}

BEGIN
         {Include code to initialize totals & get listfile name}
         rewrite(outfile,listfile);
         done:=false;
         file_count:=0;
         REPEAT
            write(output,'Type name of file to be analyzed: ');
            IF NOT eof(input) THEN BEGIN
                      read(input,source);
                      IF length(source) < pathlength THEN
                                append(source,null_char);
                      reset(infile,source);
                      file_count:=succ(file_count);
                      ov?ld(ovy_one);
                      count_chars(infile,current_stats);
                      ov?rl(ovy_one);
                     {include code to add current_stats to totals}
                      ov?ld(ovy_two);
                      print_report(outfile,source,current_stats);
                      ov?rl(ovy_two);
                      END
            ELSE done:=true;
         UNTIL done;
```

```
          {Generate summary report if more than one file analyzed}
          IF file_count > 1 THEN BEGIN
             ov?ld(ovy_two);
             print_report(outfile,'Totals',totals);
             ov?rl(ovy_two);
             END;
END. {Example}




OVERLAY MODULE ovy_one;
TYPE letters = 'A'..'Z';
       stat_rec = RECORD
                lines:integer;
                freq:ARRAY[ letters ] OF integer;
       END;

VAR  ok:boolean;
       ch:char;
       upper_case_alphas: SET OF letters;
       lower_case_alphas: SET OF 'a'..'z';

PROCEDURE convert_to_upper(VAR arg:char);
BEGIN
       {Include code to convert arg from lower case to upper case}
END;

FUNCTION get_char(var f:text):char;
VAR result:char;
BEGIN
          result:='<0>'; {Null char}
          IF NOT eof(f) THEN read(f,result)
          ELSE ok:=false;
          get_char:=result;
END;

ENTRY PROCEDURE count_chars(VAR f:text; VAR stats:stat_rec);
BEGIN
       {Include code to initialize stats argument}
       ok:=true;
       REPEAT
              WITH stats DO BEGIN
                     WHILE ok AND eoln(f) DO BEGIN
```

```
                    lines:=lines+1; {Count number of lines in file}

                    ch:=get_char(f);
                  END;
                  IF ok THEN BEGIN
                                  ch:=get_char(f);

  IF ch IN lower_case_alphas THEN convert_to_upper(ch);
                                  IF ch IN upper_case_alphas THEN

  freq[ch]:=freq[ch]+1; {Count occurrences}
                  END;
            END;
      UNTIL NOT ok;
END; {ovy_one}




OVERLAY MODULE ovy_two;

CONST pathlength = 127;
TYPE letters = 'A'..'Z';
       stat_rec = RECORD
                  lines:integer;
                  freq:ARRAY[ letters ] OF integer;
       END;
       filename = STRING pathlength;

VAR  percentages:ARRAY[ letters ] OF real;
       total_chars:real;

PROCEDURE print_header(VAR f:text; name:filename);
BEGIN
      {include code to print the name of the file just analyzed}
END;

PROCEDURE print_distribution(VAR f:text; stats:stat_rec);
VAR index:char;
BEGIN
      {Calculate the total number of alphabetic chars in the file}
      total_chars:=0;
      FOR index:='A' TO 'Z' DO
```

```
            total_chars:=total_chars + stats.freq[index];
      {include code to print total_chars and }
      {the distribution for each alphabetic char}
END;

PROCEDURE analyze(VAR stats:stat_rec);
EXTERNAL VAR totals:stat_rec;
BEGIN
       {Include code to fill in the percentages array and to}
       {calculate any other useful statistics}
END;

PROCEDURE print_statistics(VAR f:text; VAR stats:stat_rec);
BEGIN
       analyze(stats);
       {Include code to print the percentages array, }
       {the average chars per line, and }
       {any other statistics that have been generated}
END;

ENTRY PROCEDURE print_report(VAR f:text; name:filename;
                             VAR stats:stat_rec);
BEGIN
          print_header(f,name);
          print_distribution(f,stats);
          print_statistics(f,stats);
END; {Ovy_two}
```

Predefined Routines

SP/Pascal is equipped with a set of predefined, standard routines that perform commonly-used operations: mathematical calculations, type-handling operations, character and string manipulations, address calculations, and storage space control. This chapter describes these routines. The compiler automatically handles all predefined routines in the source program.

Introduction

The following sections detail several types of predefined SP/Pascal routines:

- mathematical functions
- type handling functions
- string functions and procedures
- dynamic variable handling functions
- functions that return bit fields and addresses
- miscellaneous routines

Tables 9.1 through 9.7 summarize the predefined routines available with SP/Pascal.

The SP/Pascal input and output routines are predefined routines, as well. However, input and output operations involve file management, peripheral device, and media considerations not required by the predefined routines in this chapter. Therefore, the I/O routines are detailed separately in Chapter 7.

The predefined routines described in this chapter adhere to the language's type rules; some operations require an argument of a specific type, while others accept more than one argument type. For example, the predefined function for calculating the square root of an expression is incorporated into the program when you use the function name in a statement, as with:

```
SEMICALC:= SQRT(X + 1.67)
```

Some routines accept one type of argument while returning another as a result and can be used specifically for type-breaking. Some predefined routines return a fatal error at execution time if the argument is unsatisfactory. For example, TRUNC (*R)* causes a run-time error whenever *R* is a real number greater than 32,767.

SP/Pascal offers compilation-time resolution with some of the standard functions. These optimized functions are: ABS, BITSIZE, BYTESIZE, CHR, LENGTH, MAXLENGTH, ODD, ORD, PRED, SQR, and SUCC. When these compile-time functions are used in declarations, the argument must be a constant expression resolvable by the compiler; undefined constants generate a compiler error. (Operands in compile-time expressions cannot be components of structured constants.) The function argument need not be a constant expression if the function appears in executable statements, rather than declarations.

Routine Name	Operation	Argument Type	Result Type
ABS*	Absolute positive value	Integer, whole, real, double_real	Same as argument
ARCTAN	Angle in radians of tangent	Integer, whole, real, double_real	Real, double_real
COS	Cosine of expression	Integer, whole, real, double_real	Real, double_real
EXP	Exponential of expression	Integer, whole, real, double_real	Real, double_real
FLOAT	Converts integer to real	Integer, whole	Real
LN	Natural logarithm of expression	Integer, whole, real, double_real	Real, double_real
ODD*	Boolean test for odd	Integer, whole, real, double_real	Boolean
ROUND	Rounds expression	Integer, whole real, double_real	Integer, whole
SIN	Sine of expression	Integer, whole, real, double_real	Real, double_real
SQR*	Squares expression	Integer, whole, real, double_real	Same as argument
SQRT	Square root of expression	Integer, whole, real, double_real	Real, double_real
TRUNC	Truncates expression	Real, double_real	Integer, whole

Table 9.1 Predefined routines, mathematical functions (means compile-time evaluable)*

Routine Name	Operation	Argument Type	Result Type
CHR*	Returns character for value	Integer, whole	Character
ORD*	Returns value of expression	Integer, whole, CHAR, Boolean, enumeration, or subrange	Integer
WHOLE*	Returns value of expression	Integer, whole, CHAR,Boolean enumeration, or subrange	Whole
REAL	Returns value of expression	Integer, whole, real, double_real	Real
DOUBLE_REAL	Returns value of expression	Integer, whole, real, double_real	Double_real
PRED*	Returns predecessor of expression	Integer, whole, CHAR, Boolean, enumeration, or subrange	Same as argument
SUCC*	Returns successor to expression	Integer, whole, CHAR, Boolean, enumeration, or subrange	Same as argument

*Table 9.2 Predefined routines, type-handling functions (*means compile-type evaluable)*

Routine Name	Operation	Argument Type	Result Type
APPEND	Appends string(s)	Character or String	n/a
LENGTH*	Current length of string	String	Integer
MAXLENGTH*	Maximum declared length of string	String	Integer
SETCURRENT	Set new current string length	String or integer	n/a
STR	Converts character expression	Character	String

Table 9.3 Predefined ASCII string routines (means compile-time evaluable)*

Routine Name	Operation	Argument Type	Result Type
FREESPACE	Gives size of largest free block (space between top-of-stack and bottom-of-heap)	None	Integer
NEW	Creates and allocates space for a dynamic variable pointer	Pointer	n/a
DISPOSE	Deallocates space and destroys a dynamic variable pointer	Pointer	n/a
MARK	Saves free space address for RELEASE to delete NEW variables	Integer	n/a
RELEASE	Deallocates all NEW variables and frees space to MARK address	Integer	n/a

Table 9.4 Predefined routines, pointers for dynamic variables (means compile-time evaluable)*

Routine Name	Operation	Argument Type	Result Type
BYTEADDR	Returns byte address	Any variable	Integer or whole
WORDADDR	Returns word address	Any variable	Integer or whole

Table 9.5 Predefined routines for returning addresses

Routine Name	Operation	Argument Type	Result Type
BITSIZE*	Minimum bits in type	Type name	Whole
BYTESIZE*	Minimum bytes in type	Type name	Whole

Table 9.6 Predefined routines for returning field size (means compile-time evaluable)*

Routine Name	Operation	Argument Type	Result Type
MIN	Returns smaller of two arguments	Integer, whole, CHAR, Boolean, enumeration, or subrange	Same as arguments
MAX	Returns larger of two arguments	Integer, whole, CHAR, Boolean, enumeration, or subrange	Same as arguments
SYSTEM	Executes system call	Whole/integer	n/a

Table 9.7 Predefined miscellaneous routines

Mathematical Functions

SP/Pascal provides the following predefined mathematical functions: ABS, ARCTAN, COS, EXP, FLOAT, LN, ODD, ROUND, SIN, SQR, SQRT and TRUNC.

ABS

The syntax for ABS is

ABS *(expression)*

ABS returns the absolute (positive) value of the *expression*. The *expression* type can be integer, whole, real, or double_real. When resolving an integer or whole expression, ABS returns an integer value; when *expression* is real, ABS returns a real value; when *expression* is double_real, ABS returns a double_real result. SP/Pascal resolves all calls to ABS with constant arguments at compile-time so that ABS can be used in declarations.

ARCTAN

The syntax for ARCTAN is

ARCTAN *(expression)*

ARCTAN returns the value (in radians) of the angle whose tangent is *expression*. *Expression* must be a real, double_real, whole, or integer type. The result returned from ARCTAN is a double_real number if the argument is double_real; otherwise the result is real.

COS

The syntax for COS is

COS *(expression)*

COS returns the cosine of *expression*. The *expression,* which represents the angle in radians, must be a real, double_real, whole, or integer type. The function returns a double_real number if the argument is double_real; otherwise the result is real.

EXP

The syntax for EXP is

EXP *(expression)*

EXP, the exponential function, returns the value of E (approximately 2.71828) raised to the power *expression*. The *expression* must be of real, double_real, whole, or integer type and can have any value. An expression outside a range of plus-or-minus 75_{10} causes a run-time overflow error. The result returned by EXP is a double_real number if *expression* is double_real; otherwise the result is real.

FLOAT

The syntax for FLOAT is

FLOAT *(integer-expression)*

FLOAT converts the *integer-expression* (either integer or whole) to a single precision real number.

LN

The syntax for LN is

LN *(expression)*

LN returns the natural logarithm of *expression*. The *expression* must be of real, double_real, whole or integer type and must have a value greater than zero. The result is a double_real number if *expression* is double_real; otherwise the result is real.

ODD

The syntax for ODD is

ODD *(integer-expression)*

ODD returns the Boolean value true if *integer-expression* resolves to an odd value. When *integer-expression* is even, the function returns the value false. SP/Pascal resolves all calls to ODD with constant arguments on compilation so that ODD can be used in declarations.

ROUND

The syntax for ROUND is

ROUND *(real-expression)*

ROUND returns the rounded value of *real-expression* (real or double_real) as an integer. This function rounds as follows:

IF X > 0.0, ROUND:=TRUNC (X + 0.5), ELSE ROUND:= TRUNC (X-0.5)

SIN

The syntax for SIN is

SIN *(expression)*

SIN returns the circular sine of *expression*. The *expression*, which represents the angle in radians, must be of real, double_real, whole or integer type. The result is a double_real number if *expression* is double_real; otherwise the result is real.

SQR

The syntax for SQR is

SQR *(expression)*

SQR returns the square of *expression*. *Expression* must evaluate to a real, double_real, whole, or integer number. The result type corresponds to *expression's* type. SP/Pascal resolves all calls to SQR with whole or integer constant arguments at compile-time so that SQR can be used in declarations.

SQRT

The syntax for SQRT is

SQRT *(expression)*

SQRT returns the square root of *expression*. *Expression* must evaluate to a non-negative real, double_real, whole, or integer number. The result returned is a double_real number if *expression* is double_real; otherwise the result is real.

TRUNC

The syntax for TRUNC is

TRUNC *(real-expression)*

TRUNC truncates the value of *real-expression* (real or double_real) and returns a value of type integer. A value of *real-expression* outside the limit for type integer or whole (that is, a range of 0..65,535) generates an error. The context of TRUNC determines whether an integer or whole result is returned.

Type-Handling Routines

SP/Pascal provides the following type-handling functions: CHR, ORD, WHOLE, REAL, DOUBLE_REAL, PRED, and SUCC. These routines are grouped as type coercion functions (standard and SP/Pascal-specific) and as standard sequence functions.

Standard Type Coercion

CHR

The syntax for this function is

CHR *(integer-expression)*

CHR returns the value of type character for the ordinal number specified by *integer-expression*. CHR, not a truly executable function, changes the type of the argument value from integer to character. *Integer-expression* must be in the range 0 to 255; out-of-range expressions generate a compilation error when compile-time evaluable; otherwise they cause a run-time error. SP/Pascal resolves all calls to CHR with constant arguments at compile-time so that CHR can be used in declarations.

For instance,

```
'A'=CHR(101B8)
```

and for any character variable,

```
CH=CHR(ORD(CH))
```

ORD

The syntax for this function is

ORD *(expression)*

ORD returns the expression's integer ordinal value. Not a truly executable function, ORD changes the argument value to an integer type. For ASCII character manipulation, the function takes an expression of type character. Additionally ORD takes a Boolean or scalar argument as an expression. ORD of a CHAR expression yields the integer value of the character in ASCII character code. A Boolean expression yields an integer 0 for false and 1 for true. When the expression is integer, ORD yields the integer value. SP/Pascal resolves all calls to ORD with constant arguments at compile-time so that ORD can be used in declarations.

An ORD function with a character type expression:

```
ORD('A')=65
```

For an enumeration type expression, the first (least) constant has the ordinal value zero, the second constant has the ordinal value 1, etc. For instance, with an enumeration type declared as follows:

```
TYPE
    COLOR=(RED, GREEN, BLUE)
```

The following relations are true:

```
ORD (RED)=0
ORD (GREEN)=1
ORD (BLUE)=2
```

NOTE: *The ORD function may not be applied to real or double_real arguments.*

SP/Pascal Type Coercion

In addition to the standard functions CHR and ORD, SP/Pascal permits any scalar type name to be used as a coercion function. WHOLE, REAL and DOUBLE_REAL are provided for these type-handling functions. Each of these functions takes a scalar expression as an argument and changes the type of the argument value to the type of the function. For example, the standard function ORD is equivalent to the type coercion function INTEGER.

WHOLE

The syntax for this function is

WHOLE *(expression)*

WHOLE returns the expression's unsigned numeric value. This function is used to generate unsigned comparison operations or unsigned multiply and divide operations. For example, given the following declarations:

```
VAR W:whole;
    I:integer;
```

The relational expression I < W would generate a signed operation because the value of W would be implicitly treated as an integer value. To force the comparison to be unsigned, the expression should be written as WHOLE(I) < W.

REAL

The syntax for this function is

REAL *(expression)*

REAL returns the *expression's* single precision real value. The *expression* must be a value of integer, whole, real or double_real type. The function is used to control the precision of the arithmetic operations in a real expression. For example, given the following declarations,

```
VAR X,Y:real;
    Z:double_real;
```

the expression Z + X would generate a double precision addition operation. To force the evaluation to be done in single-precision, the *expression* should be written as REAL(Z) + X. The argument *expression* to the REAL function is evaluated via single-precision operations, regardless of the precision of the individual operands. Thus, the preceding expression could also be written as REAL(Z + X).

DOUBLE_REAL

The syntax for this function is

DOUBLE_REAL *(expression)*

DOUBLE_REAL returns the *expression's* double-precision real value. The *expression* must be a value of integer, whole, real or double_real type. The function is used in conjunction with the REAL function to control the precision of the arithmetic operations in a real expression. The argument *expression* to the DOUBLE_REAL function is evaluated via double precision operations, regardless of the precision of the individual operands. The REAL and DOUBLE_REAL functions allow you to control explicitly the precision that is used in the evaluation of any real expression. For example, consider the following expressions:

```
X + Y > 2.0                    % single prec add, single prec comparison
DOUBLE_REAL(X + Y) > 2.0       % double prec add, double prec comparison
REAL(DOUBLE_REAL(X + Y)) > 2.0 % double prec add, single prec comparison
DOUBLE_REAL(REAL(X + Y)) > 2.0 % single prec add, double prec comparison
```

Standard Sequence Functions

The functions PRED and SUCC are the standard sequence functions.

PRED

The syntax for this function is

PRED *(discrete-expression)*

PRED returns the predecessor of *expression*. The function resolves the *expression* and returns the item that precedes it. PRED also can take a scalar expression (except real or double_real) as an argument. When the expression item does not have a predecessor, an error results. SP/Pascal resolves all calls to PRED with constant arguments at compile-time so that PRED can be used in declarations.

Using the variable STOP of the enumeration type, COLOR, for example, the following assignment can be made:

```
STOP := PRED(GREEN);
```

After the assignment, STOP:= RED is true.

SUCC

The syntax for this function is

SUCC *(discrete-expression)*

SUCC returns the successor of *discrete-expression*. The function resolves the expression and returns the item that follows *discrete-expression*. SUCC also can take a scalar expression (except real or double_real) as an argument. When the expression item does not have a successor, an error results. SP/Pascal resolves all calls to SUCC with constant arguments at compile-time so that SUCC can be used in declarations.

Using the enumeration type COLOR, for example:

```
GO := SUCC(RED)
```

After the assignment, the value of GO is GREEN.

String Manipulation

The following string routines are available: APPEND, LENGTH, MAXLENGTH, SETCURRENT, and STR. Note that in SP/Pascal values of type CHAR are converted implicitly to values of type string 1 in all expression contexts. This means that CHAR values can be assigned to string variables, passed as string value parameters, and used as operands in string expressions.

APPEND

This routine appends one or more strings, substrings or characters to the end of an existing string. The syntax is as follows:

APPEND *(string-identifier, append-list)*

where *append-list* is a list of string variables, literals, substrings, or elements of type CHAR. For example:

```
VAR
        S,T: STRING 20;
        .
        .
BEGIN
                APPEND (S, T<<3:5>>, '<12>')
                .
                .
END.
```

LENGTH

The syntax for LENGTH is

LENGTH *(string-expression)*

LENGTH returns the current length of *string-expression* in the form of an integer character-count value. SP/Pascal resolves all calls to LENGTH with constant arguments at compile-time so that LENGTH can be used in declarations.

MAXLENGTH

The syntax for this routine is

MAXLENGTH *(string-identifier)*

MAXLENGTH returns the maximum declared length of *string-identifier* in the form of an integer character-count value. SP/Pascal resolves all calls to MAXLENGTH with constant arguments at compile-time so that MAXLENGTH can be used in declarations.

SETCURRENT

This routine sets the current length of a string. The syntax is

SETCURRENT *(string-identifier, n)*

String-identifier is the string whose length you are setting and *n* is the new current length (number of characters) of that string. For example:

```
VAR
  NAME : STRING 20;
  .
BEGIN
  SETCURRENT(NAME,10);
  .
END.
```

NOTE: *The length* n *cannot exceed MAXLENGTH and must be greater than or equal to zero.*

STR

The syntax for this routine is

STR *(character-expression)*

STR changes a value of type CHAR to a type string value that contains the character specified by *character-expression*. STR can be used to assign a single character to a string, or to pass a single character to a routine that requires a string argument. (The STR function is supplied primarily for MP/Pascal compatibility since SP/Pascal automatically coerces elements of the type CHAR to strings in all contexts requiring it.)

Dynamic Variable Pointers

The routines that control pointers to dynamic variables allow you to create new dynamic variables when they are needed. In order to conserve available program space, the routines also permit you to get rid of dynamic variables when they are no longer useful. Careful management of available space adds to program efficiency.

The available space you are controlling is called the *heap,* a special area of a program's memory space. The space is reserved just for storing data structures that are bound to pointer variables. The heap is dynamic; it acquires available free space in the direction of low memory for storing the instances as you allocate new dynamic variables. Figure 9.1 shows a general layout for memory. For further details on dynamic memory management, refer to Chapter 12.

The heap manipulation routines are: FREESPACE, NEW, DISPOSE, MARK, and RELEASE. The FREESPACE function returns the amount of free memory that can be acquired by the heap for storage of dynamic variables. When FREESPACE reports sufficient area, you can create instances of variables with the NEW procedure. Then, when a variable is obsolete, you can eliminate it and free storage space with the DISPOSE procedure. As an alternative to DISPOSE, MARK and RELEASE work together to free physical heap storage space without freeing space on a variable-by-variable basis. Because the program stack and the heap share the same memory area, you must manage the heap carefully to avoid running out of stack space.

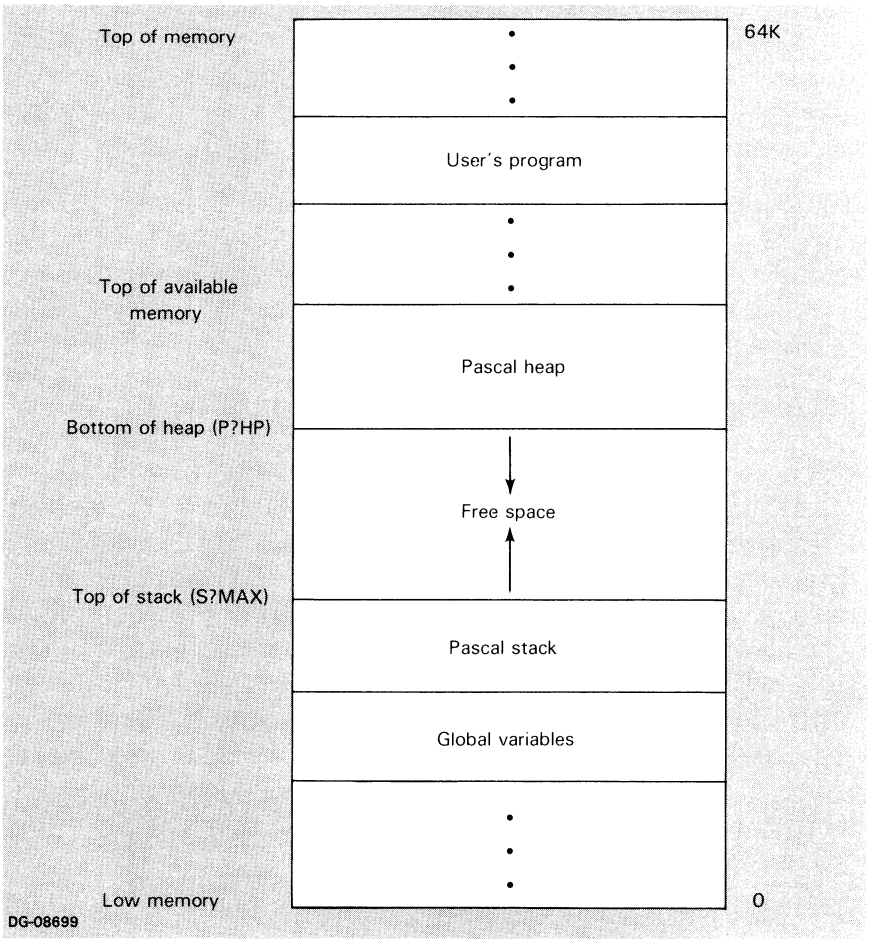


Figure 9.1 Memory organization

FREESPACE

This function is used to determine what the largest amount of available space is before allocating a dynamic variable. FREESPACE returns an integer representing the available space in memory words. It is the largest contiguous area available for allocation.

NEW

The syntax for NEW is

NEW*(pointer-var)*

NEW creates a dynamic variable and allocates memory space for it. The argument to the NEW procedure must be a variable of pointer type. NEW places the address of the newly allocated variable in *pointer-var*. Insufficient free space for the variable causes a run-time error. Once you use NEW to create a variable, the variable continues to occupy storage space in the heap until you remove the variable with DISPOSE or deallocate the area of the heap containing that variable with RELEASE.

DISPOSE

The syntax for DISPOSE is

DISPOSE*(pointer-var)*

DISPOSE deallocates the memory space occupied by the variable referred to by the pointer reference. Once a pointer is removed with DISPOSE, other pointer values that refer to this variable must not be used.

MARK

The syntax for MARK is

MARK *(integer-variable)*

MARK saves the current bottom-of-heap (the low physical address of the heap at the time of marking) in *integer-variable.* The global variable P?HP marks the low bound of the heap, and S?MAX marks the high bound of the stack.

Generally, MARK is used with RELEASE. By marking the original bottom-of-heap before creating dynamic variables, you can deallocate all the variables at one time by using RELEASE to free all space by restoring the bottom-of-heap to the marked address. The alternative to this is to use DISPOSE and deallocate the heap space variable by variable.

RELEASE

The syntax for RELEASE is

RELEASE *(integer-expression)*

RELEASE disposes of all variables with heap addresses less than *integer-expression.* This allows you to recover heap space when you have finished using the associated dynamic variables. Usually, you first determine the current heap bound with MARK, allocate and use several dynamic variables using NEW, and then make that heap area available again by using RELEASE to free storage back to the previous MARK.

NOTE: *Caution must be used with MARK and RELEASE. There is no protection against setting the bottom-of-heap to an invalid address, nor against releasing memory with its variables still in use. Also, files and tasks use the heap for storage. You must not RELEASE areas of the heap used by them.*

Address-Returning Functions

SP/Pascal offers two built-in functions that return variable addresses: BYTEADDR and WORDADDR. These functions are particularly useful in conjunction with the system interface routines (described in Chapter 10).

The following paragraphs detail these functions.

BYTEADDR

The syntax for BYTEADDR is

BYTEADDR *(variable-reference)*

BYTEADDR returns the unsigned logical byte address of its variable argument, *variable-reference.*

NOTE: Variable-reference *can be a qualified (subscripted) variable. If, for example,*

```
VAR X: ARRAY [1..10] OF CHAR;
```

then `BYTEADDR (X[3])` *is valid.*

When BYTEADDR is used with a *(string-identifier)* as *variable-reference,* the function returns the byte address of the first character of the string.

WORDADDR

The syntax of WORDADDR is

WORDADDR *(variable-reference)*

WORDADDR returns the unsigned logical word address of its variable argument, *variable-reference.*

NOTE: *You cannot obtain the word address of a character in a string or in a variable of type ARRAY of CHAR.*

When WORDADDR has *string-identifier* as *variable-reference,* the function returns the word address of the string descriptor.

Returning Field Size

SP/Pascal provides two predefined functions, BITSIZE and BYTESIZE, that accept a type name as an argument and return the number of bits (or bytes) that the type is defined to occupy. The BITSIZE function is intended primarily for setting parameters for definitions of record structure. For example,

```
CONST
  N = 10;
TYPE
  A = 0..N BIT SUCC(N DIV 2);
  B = RECORD
        F: A;
        G: 0..1 BIT 16 - BITSIZE (A);
      END;
```

The BYTESIZE function can be used to set parameters for recast types and to determine the size of a variable. The BYTESIZE of a type is the number of bytes that a variable of that type occupies. Therefore, the minimum bytesize for any type is 2, even for bit-qualified types. For example,

```
TYPE
  SS = SET OF 0..N;
  AR = ARRAY[1..BYTESIZE(SS) DIV 2] OF WHOLE;
```

BITSIZE and BYTESIZE

The syntaxes for these functions are

BITSIZE *(type)*

BYTESIZE *(type)*

BITSIZE returns the number of bits, and BYTESIZE returns the number of bytes, in *type*. The number of bits or bytes is reported as a decimal integer. Both routines allow parameters for the size of records and arrays to be set for I/O and ease of maintenance. SP/Pascal resolves all calls to either routine at compile-time so that BITSIZE (or BYTESIZE) can be used in declarations.

Miscellaneous Routines

SP/Pascal provides three additional predefined routines for use in programs. These are the functions MIN and MAX, and the procedure SYSTEM. MIN and MAX are used to compare two values and return the lesser (MIN) or greater (MAX) of the values. The SYSTEM procedure is used to execute an MP/AOS system call.

MIN

The syntax for MIN is

MIN *(expression1, expression2)*

MIN returns the smaller value of the function's two input argument expressions. The expressions must be compatible and of a discrete type (integer, whole, Boolean, character, enumeration, or subrange). If the type of the input arguments is integer, or a subrange of integer, the comparison is signed; otherwise, the comparison is unsigned. For example:

```
MIN( -1, 1 ) = -1;
MIN( 'A', 'C' ) = 'A';
MIN( WHOLE(-1), 1 ) = 1;
MIN( true, false ) = false;
```

MAX

The syntax for MAX is

MAX *(expression1, expression2)*

MAX returns the larger value of the function's two input argument expressions. The expressions must be compatible and of a discrete type (integer, whole, Boolean, character, enumeration, or subrange). When the type of the input arguments is integer, or a subrange of integer, the comparison is signed; otherwise, the comparison is unsigned. For example:

```
MAX( -1, 1 ) = 1;
MAX( 'A', 'C' ) = 'C'
MAX( WHOLE(-1), 1 ) = -1;
MAX( true, false ) = true;
```

SYSTEM

The syntax for SYSTEM is

SYSTEM *(call number, call options, AC0, AC1, AC2)*

The SYSTEM procedure is used to execute any MP/AOS system call. The DG-supplied include file SYSCALL.PAS (described in Chapter 10) contains the definitions for the call numbers and call options. AC0, AC1, and AC2 are the integer accumulator values required as input and returned as output by the various calls. When the system call executed by this procedure succeeds with a normal return, program execution continues with the next sequential statement. If the system call is unsuccessful (takes an error return), then an exception condition is returned to the routine that executed the SYSTEM procedure. Exception handlers are described in Chapter 11.

To invoke a system call, use the call numbers and options that are parameterized and contained in the include file in SYSCALL.PAS. When invoking a library routine, the call number must be specified as an externally-defined integer. Since library routines cannot take options, the option word must always be zero.

NOTE: *The SYSTEM procedure is very similar to the ?SYS function described in Chapter 10. The only difference between these two routines is the manner in which errors are reported back to the calling routine.*

External Routines Supplied by DGC

The SP/Pascal package provides an assortment of external routines that handle system calls and input-output operations, as well as internal conversion routines for string manipulation and arithmetic operations. Data General Corporation provides these routines (and routine calls) in a number of source files. The files supplied by DGC also contain parameters and constant definitions for useful routines. You can insert the source files for any needed routines or declarations in your program with the INCLUDE facility (described in Chapter 8).

NOTE: *Most of the routines described in this chapter have the same names and functions as corresponding routines for MP/Pascal. However, the MP/Pascal routines must be declared EXTERNAL ASSEMBLY, whereas the SP/Pascal routines must be declared EXTERNAL. If you have declared any of these routines in your source files, rather than by using the DGC-supplied INCLUDE files, you will need to change the declarations.*

Routine Categories

The routines supplied by DGC can be divided into general categories:

- input-output routines,
- routines defining system parameters and system interface parameters,
- numeric string conversion routines,
- multitasking support routines,
- double-precision arithmetic routines.

The following sections of this manual provide details about all the DGC supplied routines that can be incorporated into an SP/Pascal program. Table 10.1 provides an alphabetic list of the routines, along with the name of the supplied source file in which the interface statements are found.

NOTE: *All external routines supplied by DGC use the normal SP/Pascal calling convention. (See Appendix C for a description of this calling convention.)*

Name	Description	Type*	Source File
BYTEREAD	Reads byte pointer location(s)	P	IO_CALLS.PAS
BYTEWRITE	Writes to byte pointer location(s)	P	IO_CALLS.PAS
CHARREAD	Reads specified character count	P	IO_CALLS.PAS
CHARWRITE	Writes specified character count	P	IO_CALLS.PAS
CLDELFILE	Closes and deletes channel's file	P	IO_CALLS.PAS
CLOSEFILE	Closes named channel	P	IO_CALLS.PAS
DATE	Provides current date	F	HEADER.PAS
DDADD	Performs double-precision addition	F	DOUBLE.PAS
DDCOM	Performs double-precision comparison	F	DOUBLE.PAS
DDDIV**	Performs double-precision division	F	DDMATH.PAS
DDMUL**	Performs double-precision multiplication	F	DDMATH.PAS
DDNEG	Performs double-precision sign inversion	F	DOUBLE.PAS
DDSUB	Performs double-precision subtraction	F	DOUBLE.PAS
DELETE	Deletes specified file	P	IO_CALLS.PAS
DI2ST**	Converts double-precision integer to ASCII	P	DINT2ST.PAS
DRSCH	Disables task scheduler	P	TASKING.PAS
DSADD	Performs mixed-type addition	F	DOUBLE.PAS
DSDIV	Performs mixed-type division	F	DOUBLE.PAS
DSMUL	Performs mixed-type multiplication	F	DOUBLE.PAS
DSSUB	Performs mixed-type subtraction	F	DOUBLE.PAS
ERSCH	Reenables task scheduler	P	TASKING.PAS
?EXIT	Terminates program and returns to parent	P	SYSLIB.PAS
FORK	Creates a task at run time	P	TASKING.PAS
GETARG	Extracts one message argument	P	GET_MESSAGE.PAS
GETMESSAGE	Gets CLI format system message	P	GET_MESSAGE.PAS
GETSWITCH	Extracts one message argument switch	F	GET_MESSAGE.PAS
GPOSFILE	Gives file position for random access	P	IO_CALLS.PAS
INDEX	Locates string position within string	F	INDEX.PAS
KILL	Deletes a task and releases locks	P	TASKING.PAS
LINEREAD	Reads one string buffer	P	IO_CALLS.PAS

Table 10.1 External routines supplied by DGC

**P = Procedure and F = Function*

***Requires inclusion of DOUBLE.PAS first for double-precision arithmetic*

Name	Description	Type*	Source File
LINEWRITE	Writes one string buffer	P	IO_CALLS.PAS
LOCK	Locks out other tasks	P	TASKING.PAS
?MESSAGE	Returns error text for an error code	F	MESSAGE.PAS
NEWSTR	Establishes dynamic string variable	P	NEWSTR.PAS
OPENFILE	Opens named file and returns channel	P	IO_CALLS.PAS
OV?LD	Loads specified overlay	P	OVLY.PAS
OV?RL	Releases specified overlay	P	OVLY.PAS
PEND	Blocks task to wait for event	P	TASKING.PAS
RANDOM	Returns random positive integer	F	RANDOM.PAS
RE2ST**	Converts single-precision real to decimal	P	REAL2STR.PAS
RENAME	Renames specified file	P	IO_CALLS.PAS
REVISION	Provides program revision number	F	HEADER.PAS
SETPRIORITY	Changes priority of a task	P	TASKING.PAS
SI2ST**	Converts single-precision integer to ASCII	P	SINT2ST.PAS DINT2ST.PAS
SPOSFILE	Sets new file position offset	P	IO_CALLS.PAS
ST2DI**	Converts ASCII to double-precision integer	P	STR2DINT.PAS
ST2RE**	Converts string to single-precision real	P	STR2REAL.PAS
ST2SI**	Converts ASCII to single-precision integer	P	STR2SINT.PAS STR2DINT.PAS
?SYS	Invokes any system call	F	SYSCALL.PAS
TIME	Provides current time	F	HEADER.PAS
UNLOCK	Unlocks locked tasks	P	TASKING.PAS
UNPEND	Unblocks task on event	P	TASKING.PAS
XAND	Performs AND	F	BOOLEAN.PAS
XEXTRACT	Returns bit field	F	BOOLEAN.PAS
XIOR	Performs inclusive OR	F	BOOLEAN.PAS
XNOT	Performs logical complement	F	BOOLEAN.PAS
XSHFT	Shifts a value	F	BOOLEAN.PAS
XXOR	Performs exclusive OR	F	BOOLEAN.PAS

Table 10.1 External routines supplied by DGC (continued)

**P = Procedure and F = Function*

***Requires inclusion of DOUBLE.PAS first for double-precision arithmetic*

I/O Routines

The I/O package supplied by Data General Corporation is a set of 14 procedures. IO_CALLS.PAS defines these procedures and the data types they use.

These routines give you more control, and a more direct interface to the MP/AOS system, than the routines described in Chapter 7. They are similar to the I/O routines in most other Data General languages, including assembler language. They are distinct from the regular Pascal I/O routines in a number of ways. For instance, regular

Pascal I/O routines use file variables, while these routines use *channel* numbers, which are integers. You should not use Pascal-type and DGC-type I/O simultaneously on one file.

To check routine success or failure, each I/O procedure contains an integer variable parameter, *status.* If the routine encounters an error, the routine returns the system call error code in this argument. With successful completion, the routine returns zero in *status.* It is good practice to test *status* for a nonzero value after calling an I/O procedure.

The data declarations used by the I/O routines are:

```
CONST
   max_path_lth = 128; { maximum size of pathname }
   max_path_lth = 135; { maximum buffer size for data-sens. I/O }
TYPE
   channel = 0 .. maxint;
   pathname = STRING max_path_ith;
   io_buffer = STRING 32767; { see NOTE below }
   RECORD
      high: INTEGER;
      low: INTEGER;
END;
```

NOTE: *The type* io_buffer, *defined above, is intended for use in routine headers only; the size is specified as 32767 to ensure that you may use any size buffer, up to the maximum legal string size. This declaration appears in many of the routine headers in the rest of this chapter. When declaring your buffer variables, you should make their sizes as small as is convenient for your application.*

In addition, INCH and OUCH (which correspond to ?INCH and ?OUCH, standard input and output channels) are defined as constants in IO_CALLS.PAS. Refer to *MP/AOS System Programmer's Reference* (DGC No. 093-400051), for details.

Channel Open Procedure

To open a channel, use the OPENFILE procedure, which follows:

```
EXTERNAL PROCEDURE OPENFILE
     (VAR CHAN: CHANNEL;
           FNAM: PATHNAME;
           OPTIONS: INTEGER;
           FILE_TYPE: INTEGER;
           ELEM_SIZE: INTEGER;
         VAR STATUS: INTEGER);
```

The OPENFILE procedure opens the file specified by *fnam,* a filename that requires a null terminator. OPENFILE returns the opened file's integer channel number in *chan. Options, file_type* and *elem_size* are as described for the ?OPEN system call. Constant definitions for

the options are included in IO_CALLS.PAS. Definitions for *file_type* are contained in FILE_PARS.PAS. For example,

```
OPENFILE (CHN, "TEST", UC + EX ,?DUDF,1,ST);
```

FILE_PARS.PAS defines the MP/AOS file types and attributes for the OPENFILE procedure.

Channel Close

There are two channel closing procedures:

- CLOSEFILE, which closes an open file on the channel specified;
- CLDELFILE, which deletes that channel's file, in addition to closing it.

The declarations for these procedures are:

```
EXTERNAL PROCEDURE CLOSEFILE
  (CHAN:CHANNEL;
   VAR STATUS:INTEGER)

EXTERNAL PROCEDURE CLDELFILE
  (CHAN:CHANNEL;
   VAR STATUS:INTEGER)
```

Data-Sensitive I/O

For data-sensitive I/O, use LINEREAD and LINEWRITE. The declarations for these procedures are

```
EXTERNAL PROCEDURE LINEREAD
  (CHAN:CHANNEL;
   VAR BUFFER: LINE_BUFFER;
   VAR STATUS: INTEGER);

EXTERNAL PROCEDURE LINEWRITE
  (CHAN:CHANNEL;
  BUFFER: LINE_BUFFER;
  VAR STATUS: INTEGER);
```

LINEREAD and LINEWRITE perform data-sensitive I/O to the string specified in *buffer*. The maximum length of *buffer* is used as the maximum byte value to read or write.

Read operations with LINEREAD read data up to the maximum length of *buffer*. If no data-sensitive delimiters (Null, Carriage Return, New Line, or Form Feed) are encountered before maximum length is reached, an error is returned.

Write operations with LINEWRITE write data up to the first data-sensitive delimiter, without regard to current string length. If no delimiter is reached within the maximum length of *buffer*, an error is returned.

String Dynamic I/O

For string dynamic I/O, use CHARREAD and CHARWRITE. The declarations for these procedures are

```
EXTERNAL PROCEDURE CHARREAD
  (CHAN: CHANNEL;
  LTH: INTEGER;
  VAR BUFFER: IO_BUFFER;
  VAR STATUS: INTEGER);

EXTERNAL PROCEDURE CHARWRITE
  (CHAN: CHANNEL;
  BUFFER: IO_BUFFER;
  VAR STATUS: INTEGER);
```

CHARREAD and CHARWRITE perform dynamic I/O to the string specified in *buffer*. On input, *lth* specifies the number of bytes to read. After the read is done, the current length of *buffer* is set to the number of bytes actually read. On output, the current length of *buffer* dictates the number of bytes written.

Buffer Dynamic I/O

For buffer dynamic I/O, use BYTEREAD and BYTEWRITE which follow:

```
EXTERNAL PROCEDURE BYTEREAD
  (CHAN:CHANNEL;
  BUF_BYTE_ADDRESS: INTEGER;
  VAR LTH: INTEGER;
  VAR STATUS: INTEGER);

EXTERNAL PROCEDURE BYTEWRITE
  (CHAN: CHANNEL;
  BUF_BYTE_ADDRESS: INTEGER;
  LTH: INTEGER;
  VAR STATUS: INTEGER);
```

The procedures BYTEREAD and BYTEWRITE perform dynamic I/O to the physical area in memory specified by the byte pointer in *buf_byte_address*. You obtain this address using BYTEADDR, a predefined function explained in Chapter 9.

There is no run-time check that ensures the validity of the specified address. It is the responsibility of the calling program to safeguard memory overwrites.

File-Position

For file positioning, use GPOSFILE and SPOSFILE which follow:

```
EXTERNAL PROCEDURE GPOSFILE
  (CHAN: CHANNEL;
  VAR POSITION: RECAST FILE_POSITION;
  VAR STATUS: INTEGER);
```

```
EXTERNAL PROCEDURE SPOSFILE
   (CHAN: CHANNEL;
   POSITION: RECAST FILE_POSITION;
   VAR STATUS: INTEGER);
```

The file position procedures manipulate the current file position, giving you full random-access capability as an extension to the sequential nature of Pascal's normal file I/O. GPOSFILE returns the file position, and SPOSFILE resets it.

A *file_position* is actually a 32-bit integer, which represents the number of bytes from the start of the file to a given point. If you plan to do any advanced manipulation on these numbers, you will probably want to use double-precision integer arithmetic on them. To do this, you must include the double-precision package DOUBLE.PAS. When the package is included, the variable passed in *position* must be declared as being type DOUBLE. (DOUBLE.PAS is described later in this chapter.)

File Management

Two DG-supplied procedures, DELETE and RENAME, allow you to delete and rename specified files. The declarations for these procedures are:

```
EXTERNAL PROCEDURE DELETE
   (V : PATHNAME;
   VAR STATUS : INTEGER);
```

This procedure deletes the file specified by the pathname in V. No error is returned when the specified file does not exist.

```
EXTERNAL PROCEDURE RENAME
   (F_NOW, F_THEN : PATHNAME;
   VAR STATUS : INTEGER);
```

This procedure changes the name of the file specified in *f_now* to the name specified in *f_then*.

Data Channel Printer Control

DCLP.PINC defines three procedures to open, write to, and close the data channel printer.

```
EXTERNAL PROCEDURE POPEN
   (DEVICE_CODE,PRINTER_TYPE:INTEGER;
   VAR STATUS:INTEGER);

EXTERNAL PROCEDURE PCLOS
   (VAR STATUS:INTEGER);

EXTERNAL PROCEDURE PWRIT
   (BUFF_BYTEADDR,BYTE_COUNT:INTEGER;
   VAR STATUS:INTEGER);
```

In which the following important variables are declared:

device_code	specifies the line printer device code
printer_type	is an integer whose value is either 0 for LP2 or 1 for LPB (non-LP2)
buff_byteaddr	is a byte pointer to an area in memory containing data to be printed
byte_count	specifies the number of bytes to be printed

NOTE: *The routines defined in this include file are available only under MP/AOS.*

Dynamic String Variables

The source file NEWSTR.PAS defines NEWSTR, the procedure for dynamic allocation of a variable-length string.

```
EXTERNAL PROCEDURE NEWSTR
      (VAR PT : RECAST MAXSTRP; SIZE : INTEGER);
```

MAXSTRP is declared in NEWSTR.PAS as follows:

```
TYPE
      MAXSTRP = @STRING 32767;
```

NEWSTR allocates a string of the length specified in *size* and returns a pointer to the string in *pt*. For example:

```
NEWSTR (P,23);
```

This call allocates a 23-character string on the heap and returns a pointer to it in *p*.

System Interfacing

A number of routines define constructs or interact with the MP/AOS operating system and the CLI: GET_MESSAGE.PAS, OVLY.PAS, SYSCALL.PAS, SYSLIB.PAS, and TASKING.PAS. (The multitasking routines provided by TASKING.PAS are described in Chapter 12.)

GET_MESSAGE.PAS

When the CLI begins executing a user program, an inter-program command *message* is translated into a special CLI format. Your program can read and interpret this message by using the three routines (GETMESSAGE, GETARG and GETSWITCH). The routines are defined in the source file GET_MESSAGE.PAS. For details on the message, refer to the *MP/AOS System Programmer's Reference* (DGC. No. 093-400051).

GETMESSAGE

GETMESSAGE places the entire command line in the *MESSAGE* string variable:

```
EXTERNAL PROCEDURE GETMESSAGE
      (VAR MESSAGE: GET_MSG_TYPE);
```

GETARG

The GETARG function allows your program to examine one *argument* (filename, function name, etc.) in the *message;* you must use GETMESSAGE to get the entire message before using GETARG.

```
EXTERNAL FUNCTION GETARG
   (MESSAGE: GET_MSG_TYPE;
   ARG_NUM: INTEGER;
   VAR ARG_VALUE: GET_SW_TYPE): BOOLEAN;
```

If the function finds the argument specified by *arg_num,* it returns the value true. Otherwise, it returns the value false. *Arg_value* is set to the null string if the *arg_num'*th argument does not exist. When the argument exists, *arg_value* contains the text of that argument with all switches removed.

NOTE: *If you call this routine with zero in* arg_num, *it will return the name of the program.*

GETSWITCH

You call the GETSWITCH function to examine a switch to a particular command line (/U, /L=@LPT, etc.) (Before you can call GETSWITCH you must use GETMESSAGE to get the whole message in *message.*)

```
EXTERNAL FUNCTION GETSWITCH
   (MESSAGE: GET_MSG_TYPE;
   ARG_NUM: INTEGER;
   SW_NAME: GET_SW_TYPE;
   VAR KEYWORD_VALUE: GET_SW_TYPE): BOOLEAN
```

Arg_num specifies the integer number of the message argument you would like to examine (0 specifies the program name). *Sw_name* specifies the switch you are looking for. *Sw_name* must terminate with a null byte.

If GETSWITCH cannot find the switch, the function returns false. Otherwise, it returns true. If the switch is not a keyword switch, *keyword_value* is set to the null string. Otherwise, *keyword_value* contains the string that follows the equal sign in the switch.

MESSAGE.PAS

This function returns the text associated with a specific error code. The text is taken from :ERMES, the system error message file.

```
EXTERNAL FUNCTION ?MESSAGE
      (ERROR_NO:INTEGER): STRING 80
```

OVLY.PAS

OVLY.PAS defines the two procedures used for overlay management, OV?LD and OV?RL.

```
EXTERNAL PROCEDURE OV?LD
   (OVLY_NAME:INTEGER);

EXTERNAL PROCEDURE OV?RL
   (OVLY_NAME:INTEGER);
```

The OV?LD and OV?RL routines load or release an overlay specified by the parameter *ovly_name*. Normally, *ovly_name* is the overlay module name. An error during either the OV?LD or the OV?RL procedure terminates the current program with a system error. The SP/Pascal OVERLAY facility is described in Chapter 8, "Program Structure".

SYSCALL.PAS

SYSCALL.PAS defines the function ?SYS, which can be used to invoke any MP/AOS system call. The constants defined in this file also are used with the predefined routine, SYSTEM, described in Chapter 9 of this manual.

```
EXTERNAL FUNCTION
      ?SYS(CALL_NO,OPTIONS:INTEGER;
       VAR AC0, AC1, AC2:INTEGER):BOOLEAN;
```

AC0, AC1 and AC2 are the accumulator values required as input and returned as output by the various calls. The function returns false if the system call succeeds with a normal return or returns true if the call is unsuccessful.

For system calls, the call numbers specified in *call_no* and *options* are parameterized and included in SYSCALL.PAS.

When invoking a library routine, the call number must be specified as an externally-defined integer. Since library routines cannot take options, the option word must always be zero.

A number of DGC-supplied files define parameter packets for system calls (see Table 10.2). These parameter files contain the equivalent of the assembly language files MPARU.SR and SYSID.SR; their primary use is to access system calls. Details of system calls are provided in the *MP/AOS System Programmer's Reference,* (DGC No. 093-400051).

MP/AOS System Call	SP/Pascal Include File Name
?CTASK *	CTASK_PAK.PAS
?SCHAR *	DEV_CHAR.PAS
?DHIS	DHIS_PAK.PAS
?DSTAT *	DSTAT_PAK.PAS
?EHIS	EHIS_PAK.PAS
?EINFO	EINFO_PAK.PAS
?EXEC *	EXEC_ERCLASS.PAS
?FSTAT *	FSTAT_PAK.PAS
?SCHAR *	HW_CHAR.PAS
?IDEF	IDEF_PAK.PAS
?INFO *	INFO_PAK.PAS
?SEND, ?RCV, ?RCVA, ?REPLY	IPC_PAK.PAS
?LDEF	LDEF_PAK.PAS
?MSEG	MSEG_PAK.PAS
?PROC	PROC_PAK.PAS
?RDST, ?WRST	RD_WR_STPAK.PAS
?SD.R	SD.R_PAK.PAS
?READ, ?WRITE (?PKT OPTION)	SEG_IO_PAK.PAS
?SINFO	SINFO_PAK.PAS
?TMSG *	TMSG_PAK.PAS

Table 10.2 MP/AOS system calls and corresponding parameter files

**Supported for cross-development under AOS (see Appendix D).*

SYSLIB.PAS

SYSLIB.PAS contains the ?EXIT routine, which terminates the program normally and returns control to the terminated program's parent program.

```
EXTERNAL PROCEDURE ?EXIT
      (ERROR:INTEGER;
      RET:STRING 2048);
```

The *error* parameter contains the error code to return (zero signifies no error), and *ret* is a string that is returned to the parent. When the parent program is the CLI, it prints the string and the message associated with the error code to @TTO. You can use *ret* for any message that will help users to understand why the program terminated.)

HEADER.PAS

HEADER.PAS defines three functions: DATE, TIME, and REVISION, that are useful for printing the current time and date, or the revision number of the program. The returned information is always a string defined as follows:

```
TYPE
  DATESTR = STRING 29;
  REVSTRING = STRING 7;
  TIMESTR = STRING 11;

EXTERNAL FUNCTION DATE : DATESTR;
```

The returned date format looks like this:

```
  FRIDAY FEBRUARY 30, 1979
```

```
EXTERNAL FUNCTION TIME : TIMESTR;
```

The returned time format looks like this:

```
  12:09:00 AM
```

```
EXTERNAL FUNCTION REVISION : REVSTRING;
```

The returned revision format looks like this:

```
  1.01
```

Numeric String Conversion

Six source files define procedures for converting numbers to ASCII strings or *vice versa,* and handling exceptions. The file names are:

```
DINT2ST.PAS,
REAL2STR.PAS,
SINT2ST.PAS,
STR2DINT.PAS,
STR2REAL.PAS,
STR2SINT.PAS.
```

DINT2ST.PAS

DINT2ST.PAS defines the procedure DI2ST, for converting a double-precision integer to an ASCII string. You must include the files DOUBLE.PAS and DINT2ST.PAS (in that order) before using this procedure in your program. DINT2ST.PAS also defines the procedure SI2ST for converting single-precision integer to an ASCII string for instances when both procedures are needed. (Refer to the explanation of the SINT2ST.PAS file.)

```
EXTERNAL PROCEDURE DI2ST
  (VAL: DOUBLE;
  RADIX_FLAGS: INTEGER;
  VAR OUTST: STRING 32767;
  VAR ERROR: INTEGER);
```

DI2ST converts a double-precision integer value, *val,* to an ASCII output string, *outst,* in the indicated radix. The radix word must be in the range 2 to 35. By default, the output is unsigned and the length of the string will be set to the length of the result. If the output string is a substring, the result is padded with blanks to the end of the substring.

To modify the conversion, add options to the radix word. For signed conversion, use the option *i2st_signd;* then the output string is prefixed by a + or − sign, as appropriate. To right-justify the string, specify *i2st_rjust;* this extends the output string to its maximum length (or the string piece to its current length) by filling it with blanks (the default), or zeroes, if the *i2st_zextn* option is specified. The output number (with sign, if specified) is right-justified in the string. Also, you can specify the *i2st_astfl* option to fill the string with asterisks in the case of an overflow.

When the result of the conversion is too long, *error* returns *i2st_overflow*. In this case, if you specified the *i2st_astfl* conversion option, the entire result string fills with asterisks.

A successful conversion returns a zero in *error*. Test *error* for the status code after every call to DI2ST. Use the error mnemonics documented in DINT2ST.PAS for this purpose.

REAL2STR.PAS

REAL2STR.PAS defines the RE2ST procedure, which converts a single-precision real number to an ASCII string.

```
EXTERNAL PROCEDURE RE2ST
   (INPUT : REAL;
   WIDTH : INTEGER;
   FSIZE : INTEGER;
   VAR OUTST : STRING 32767 );
```

Two parameters control the formatting of the output: *width* and *fsize*. The *width* parameter specifies the maximum number of characters to output. Leading blanks are inserted when the width exceeds the number of characters required for the numeric representation.

The *fsize* parameter determines the type of numeric representation. If *fsize* is greater than 0, a fixed-point notation is generated with *fsize* digits after the decimal point. Otherwise, scientific (E) notation is used. At most, seven non-zero digits are printed. When less than seven digits are requested, the remainder is used for rounding. When both *width* and *fsize* are 0, the default is scientific notation with a *width* of 13 characters.

NOTE: *No error is indicated if the requested numeric representation does not fit into the designated output string. In this case, the output string contains the first MAXLENGTH characters.*

SINT2ST.PAS

SINT2ST.PAS defines the procedure SI2ST for converting a single-precision integer to an ASCII string. (The SI2ST procedure definition also appears in the source file DINT2ST.PAS, explained earlier, an alternative include file when a double-precision integer to ASCII is needed.)

```
EXTERNAL PROCEDURE SI2ST
   (VAL,RADIX_FLAGS:INTEGER;
   VAR OUTST: STRING 32767
   VAR ERROR: INTEGER);
```

SI2ST converts the single-precision integer value, *val,* to an ASCII output string, *outst,* in the indicated radix. The radix word must be in the range 2 to 35. By default, the output is unsigned and the length of the string will be set to the length of the result. If the output string is a substring, the result is padded with trailing blanks to the end of the substring.

You can modify the conversion by adding options to the radix word. For signed conversion, use the option *i2st_signd;* then the output string is prefixed by a + or − sign, as appropriate. To right-justify the string, specify *i2st_rjust.* This extends the output string to its maximum length (or the string piece to its current length) by filling it with blanks (the default), or zeroes when the *i2st_zextn* option is specified. The output number (with sign if specified) is right-justified in the string. You can also specify an asterisk-fill on overflow with *i2st_astfl.*

If the result of the conversion is too long, *error* returns *i2st_overflow.* In this case, if you specified the *i2st_astfl* conversion option, the entire result string fills with asterisks.

When the conversion is successful, *error* returns a zero. After every call to SI2ST, test *error* for the status code. Use the error mnemonics for this purpose.

STR2DINT.PAS

STR2DINT.PAS defines the ST2DI procedure, which converts an ASCII string to a double-precision integer. You must include the files DOUBLE.PAS and STR2DINT.PAS, in that order, before using this procedure in your program. STR2DINT.PAS also defines the procedure ST2SI for converting ASCII to single-precision integer for instances when both procedures are needed. (Refer to STR2SINT.PAS, below.)

```
TYPE S2IN_RADIX = 2..35;

EXTERNAL PROCEDURE ST2DI
   (INSTR: STRING 32767;
   RADIX: S2IN_RADIX;
   VAR RESLT: DOUBLE;
   VAR NXTCH: INTEGER;
   VAR ERCODE: INTEGER);
```

ST2DI converts an ASCII input string *instr* to a double-precision integer value, *reslt,* in the indicated radix. *Radix* must be in the range 2 to 35. Leading blanks and tabs, as well as an optional + or − sign, can begin the input string. *Alphabetic* digits (for radices greater than 10) can be entered as either upper-case or lower-case characters.

The run-time conversion process stops if an invalid digit (not a legal alphanumeric) is found in the user-supplied radix. After the conversion is performed, the *nxtch* actual parameter contains the integer index (character position) of the next character in the string. This value is relative to the beginning of the string parameter passed. (Note that this number would be greater than the length of the string if the whole string were scanned.)

When the conversion succeeds, the returned status word *ercode* contains a zero. If the conversion fails, *ercode* returns either of the following values:

```
CONST
   S2IN_OVERFLOW = 1;
   S2IN_CONVRT_ERR = 2;
```

If *instr* is null, or its sign is not followed by a valid digit, or the input radix is not in the allowable range, *ercode* returns the conversion error (2). If the result is too large, *ercode* returns the overflow error (1).

After every call to ST2DI, test *ercode* for the status code. Use the error mnemonics for this purpose.

STR2REAL.PAS

STR2REAL.PAS defines the ST2RE procedure, which converts an ASCII input string to a single-precision real number.

```
EXTERNAL PROCEDURE ST2RE
   (INSTR: STRING 32767;
   VAR NXTCH: INTEGER;
   VAR RESLT: REAL;
   VAR ERROR: BOOLEAN);
```

Error returns true if an error occurs during conversion. An error occurs if a null string is passed to the routine, if the passed string does not contain a valid digit, or if an underflow or overflow occurs.

The *nxtch* actual parameter contains the integer index of the next character in the string after conversion is performed. This value is relative to the beginning of the string parameter passed. Note that this number can be greater than the length of the string. The range

of absolute values that can be represented is zero to $7.2 \times 10^{+75}$. The smallest non-zero absolute value that can be represented is 5.4×10^{-79}.

STR2SINT.PAS

STR2SINT.PAS defines the ST2SI procedure, which converts an ASCII string to an integer value. (The ST2SI procedure definition also appears in the source file STR2DINT.PAS, explained earlier, an alternative include file when ASCII to double-precision integer is needed.)

```
TYPE S2IN_RADIX = 2..35;

EXTERNAL PROCEDURE ST2SI
  (INSTR: STRING 32767;
  RADIX: S2IN_RADIX;
  VAR RESLT: INTEGER;
  VAR NXTCH: INTEGER;
  VAR ERCODE: INTEGER);
```

The routine converts an ASCII input string *instr* to an integer value *reslt* in the indicated radix. *Radix* must be in the range 2 to 35. Leading blanks and tabs, as well as an optional + or − sign, can begin the input string. *Alphabetic* digits for radices greater than 10 can be entered as either upper-case or lower-case characters.

Conversion stops as soon as an invalid digit is found in the user-supplied radix. After the conversion is performed, the *nxtch* actual parameter contains the integer index (character position) of the next character in the string. This value is relative to the beginning of the string parameter passed.

Consider the following example:

```
S := '000012EF';
ST2SI (S <<5 : 4>>, 10, R, N, E);
```

These statements would cause R, the result, to be set to 12. N would be set to 3, not 7, because ST2SI is operating on a substring, not on all of S. On the other hand:

```
S := '000012EF';
ST2SI (S, 10, R, N, E);
```

would result in N being set to 7, since all of S is being examined. (Note that if the radix were 16 rather than 10, N would be set to 9, since '000012EF' is a valid hexadecimal number.)

When the conversion succeeds, the returned status word (*ercode*) contains a zero. If the procedure fails, *ercode* has either of the two following values:

```
CONST
  S2IN_OVERFLOW = 1;
  S2IN_CONVRT_ERR = 2;
```

When *instr* is null, or if its sign is not followed by a valid digit, or the input radix is not in the allowable range, *ercode* returns the conversion error (2). When the result is too large, *ercode* returns the overflow error (1). After every call to ST2SI, test *ercode* for the status code. Use the error mnemonics for this purpose.

Integers and Bit Manipulation

Table 10.3 lists the functions described in this section:

Routine Name	Operation
INDEX	Returns character position of substring
RANDOM	Returns random positive integer
XAND	Performs AND
XIOR	Performs inclusive OR
XXOR	Performs exclusive OR
XSHFT	Shifts a value
XEXTRACT	Returns bit field
XNOT	Perfoms logical complement

Table 10.3 Miscellaneous integer and bit manipulation routines

INDEX.PAS

INDEX.PAS defines the function INDEX, returning the index of one string located within another string. The routine searches the specified string (first argument) for the first occurrence of the second argument string. When an instance of the second string is found, INDEX returns the numerical character position at which the match begins. For instance, the following example returns 2.

```
INDEX ('abcde', 'bc')
```

When INDEX does not find the second string within the first one, the function returns a value of 0.

RANDOM.PAS

The include file RANDOM.PAS defines the RANDOM function, which generates a positive random integer value each time you call the function, as follows:

```
EXTERNAL FUNCTION RANDOM : INTEGER;
```

RANDOM returns a positive random integer in the range of 0 to 32767. The function seeds itself from the system clock.

BOOLEAN.PAS

BOOLEAN.PAS contains six functions that perform Boolean operations on integer values. These functions are XAND, XIOR, XXOR, XSHFT, XEXTRACT, and XNOT.

XAND

This function performs a logical AND operation on *x* and *y* and returns its integer bit-by-bit result.

```
EXTERNAL FUNCTION XAND
      (X,Y :RECAST INTEGER):INTEGER;
```

XIOR

This function performs a logical inclusive OR operation on *x* and *y* and returns its bit-by-bit result.

```
EXTERNAL FUNCTION XIOR
      (X,Y:RECAST INTEGER):INTEGER;
```

XXOR

This function performs a logical exclusive OR operation on *x* and *y* and returns its bit-by-bit result.

```
EXTERNAL FUNCTION XXOR
      (X,Y:RECAST INTEGER):INTEGER;
```

XSHFT

XSHFT returns the shifted value of *x*.

```
EXTERNAL FUNCTION XSHFT
      (X,S:RECAST INTEGER):INTEGER;
```

S specifies the direction and number of bit positions to be shifted:

positive values specify left shifts;
negative values specify right shifts.

XEXTRACT

This function returns the bit field in X which begins with bit *s* and is of the length specified by *len*. Bits are numbered from left to right, starting from zero.

```
EXTERNAL FUNCTION XEXTRACT
      (X,S,LEN : RECAST INTEGER) : INTEGER;
```

The bits are right-justified in the result. For example, the following call returns 5 (binary 101).

```
XEXTRACT (1_010_011_100_101_110B2, 10, 3)
```

XNOT

This function returns the logical bit-by-bit complement of *x*.

```
EXTERNAL FUNCTION XNOT
      (X : RECAST INTEGER) : INTEGER;
```

DOUBLE.PAS

DOUBLE.PAS defines double-precision unsigned arithmetic functions. The file provides the routines listed in Table 10.4.

Routine Name	Operation
DDADD	Double-precision addition
DDDIV*	Double-precision division
DDMUL*	Double-precision multiplication
DDSUB	Double-precision subtraction
DDNEG	Double-precision sign inversion
DDCOM	Double-precision comparison
DSADD	Mixed-type addition
DSSUB	Mixed-type subtraction
DSMUL	Mixed-type multiplication
DSDIV	Mixed-type division

Table 10.4 Double-precision arithmetic functions

**Source file DDMATH.PAS required after DOUBLE.PAS is included.*

The double-precision arithmetic functions define the DOUBLE type and the double-precision integer arithmetic package.

The first item in the DOUBLE.PAS file is the following declaration for the double-precision INTEGER type:

```
TYPE
     DOUBLE =
     RECORD
        HI,LO: INTEGER;
     END;
```

Double-Precision Arithmetic

Using the following functions, you can add, subtract and invert the signs of variables that are type-compatible with the double-precision integer type. Double-precision integers are unsigned.

In the following descriptions, *x*, *y* and *z* are of type double, and *i* and *n* are integers.

DDADD

The addition function DDADD returns the sum of its arguments.

```
EXTERNAL FUNCTION DDADD
      (X;Y:DOUBLE):DOUBLE;
Z := DDADD (X,Y);
```

DDSUB

The subtraction function DDSUB returns the difference of its arguments.

```
EXTERNAL FUNCTION DDSUB
      (X;Y:DOUBLE):DOUBLE;

Z := DDSUB (X,Y);
```

DDNEG

The sign inversion function DDNEG returns the negative of its argument.

```
EXTERNAL FUNCTION DDNEG
   (X:DOUBLE):DOUBLE;

Z := DDNEG (X);
```

DDCOM

The DDCOM function compares two double-precision arguments

```
I := DDCOM (X, Y);
```

so that I is the result of the unsigned comparison of x and y. If $x<y$, DDCOM returns -1; if $x=y$, it returns 0; if $x>y$, it returns 1.

```
EXTERNAL FUNCTION DDCOM
      (X,Y;DOUBLE):COMPRESULT;

CONST
   DOUBLE_LT = -1; {x<y, double-precision unsigned}
   DOUBLE_EQ = 0; {x=y, double-precision}
   DOUBLE_GT = 1; {x>y, double-precision unsigned}
TYPE
   COMPRESULT = DOUBLE.LT..DOUBLE_GT;
```

DDMATH.PAS

The DDMATH.PAS source file provides for the full unsigned double-precision multiply and divide functions.

DDDIV

The division function DDDIV returns the quotient of its arguments. (Any remainder is discarded.)

```
EXTERNAL FUNCTION DDDIV
      (X;Y:DOUBLE):DOUBLE;

Z := DDDIV (X,Y);
```

NOTE: *DDDIV requires inclusion of the file DDMATH.PAS with DOUBLE.PAS.*

DDMUL

The multiplication function DDMUL returns the product of its arguments.

```
EXTERNAL FUNCTION DDMUL
      (X;Y:DOUBLE):DOUBLE;

Z := DDMUL (X,Y);
```

NOTE: *DDMUL requires inclusion of the file DDMATH.PAS with DOUBLE.PAS.*

Mixed Arithmetic

You can perform unsigned arithmetic using a mix of both single- and double-precision expressions of type integer with the following functions. Note that single-precision values are not sign-extended.

DSADD

The mixed type addition function DSADD returns the sum of its arguments.

```
EXTERNAL FUNCTION DSADD
      (X:DOUBLE;Y:INTEGER):DOUBLE;

Z := DSADD (Y, I);
```

DSSUB

The mixed type subtraction function DSSUB returns the difference of its arguments.

```
EXTERNAL FUNCTION DSSUB
      (X:DOUBLE;Y:INTEGER):DOUBLE;

Z := DSSUB (Y, I);
```

DSMUL

The mixed type multiplication function DSMUL returns the product of its arguments.

```
EXTERNAL FUNCTION DSMUL
      (X:DOUBLE;Y:INTEGER):DOUBLE;

Z := DSMUL (Y, I);
```

DSDIV

The mixed type division function DSDIV computes the quotient and remainder of its arguments. Then it sets the first argument to the quotient, and returns the remainder as the result of the function.

```
EXTERNAL FUNCTION DSDIV
        (VAR X: DOUBLE;
        Y: INTEGER):INTEGER;

N := DSDIV (Y, I);
```

SP/Pascal Exception Handling

Many simple programs become complex and hard to debug when the programmer must add code to handle run-time errors and other unexpected conditions. SP/Pascal addresses this problem by introducing the concept of *exception handlers:* statements that can be automatically executed when an exceptional condition occurs. SP/Pascal programs can handle all types of errors in a simple, coherent manner.

The term *exception* has come into fashion in the computer industry as a polite word for "error." Actually, the new word has some merit, since an "error" is not really "wrong" if the program is designed to respond to it in a helpful way. At any rate, there are three types of exceptions (or errors) that can occur in an SP/Pascal program:

- System errors. These are errors that are detected by the MP/AOS system, such as *FILE DOES NOT EXIST* or *INSUFFICIENT MEMORY AVAILABLE.* These errors are described in detail in the *MP/AOS System Programmer's Reference,* (DGC No. 093-40051).
- Run-time errors. These are errors that are detected within the compiled code of your program, such as *DIVISION BY ZERO* or *STRING OVERFLOW.* A complete list of these errors may be found in Appendix B.
- User-defined errors. There may be times when one routine in your program detects a condition that should be identified to the rest of the program as an error. You can cause these conditions to be handled in the same manner as system or run-time errors.

SP/Pascal provides a simple facility that handles all three types of exceptions. You can use a reserved word, EXCEPTION, to define a group of statements that will be executed automatically if an exception occurs anywhere within a specific part of your program. There is also an ERETURN statement and an ERROR_CODE function that assist you in handling exceptions.

Defining an Exception Handler

You can use the EXCEPTION keyword to associate an exception handler with any compound statement, by expanding the normal BEGIN ... END syntax in the following manner:

```
BEGIN
  .
  . { normal statements }
  .
EXCEPTION
  .
  . { exception handling statements }
  .
END
```

If an exception occurs anywhere within the normal statements, control is immediately transferred to the exception handling statements. If the normal statements execute successfully, then the exception handler is skipped, and control passes in the usual fashion to the first statement after the END.

An exception handler may contain any valid SP/Pascal statements. It may use the ERROR_CODE function (described shortly) to determine the cause of the exception. After the handler is executed, control will pass to the first statement after the END, unless the handler executes a RETURN, ERETURN, or EXITLOOP statement. One of these statements can transfer control to some other point in the program.

Nesting of Exception Handlers

Exception handlers can be nested in the same manner as BEGIN-END blocks. When an exception occurs, control is passed to the innermost exception handler that is currently active, as shown in the following example:

```
BEGIN { level 1 }
  .
  . { exceptions here will activate the level 1 handler }
  .
  BEGIN { level 2 }
    .
    . { exceptions here will activate }
    . { the level 2 handler }
    .
  EXCEPTION
    .
    . { this is the level 2 exception handler }
    .
  END; { end of level 2 }
  .
  . { exceptions here will activate the level 1 handler }
  .
EXCEPTION
  .
  . { this is the level 1 exception handler }
  .
END; { end of level 1 }
```

If an exception handler itself causes an exception, then control is immediately transferred to the next-innermost handler. In the above example, if the level 2 handler causes an exception, then control will immediately pass to the level 1 handler.

System Default Exception Handler

Whenever an exception occurs for which you have not defined your own handler, control passes to a *default* exception handler that is built into all SP/Pascal programs. This handler will simply print a short message and terminate the program. The message consists of a line of text that identifies the exception, followed by the value of the program counter when the error was detected. For example:

```
DIVISION BY ZERO
AT LOCATION 076147
```

If your program uses multitasking (see Chapter 12), and an error occurs in any task other than the program's initial task, then the error message will also identify the task which caused the error. For example:

```
TASK 0004 AT 076147
ERROR: DIVISION BY ZERO
```

Error Codes

An exception always has an *error code* associated with it. This is an integer value that identifies the cause of the exception. Your exception handler can obtain the code by calling the integer function ERROR_CODE. This function (which has no parameters) returns the code for the currently active exception. (If no exception has occurred, ERROR_CODE returns zero.)

The error codes for system errors have values that start at 40000_8. These values are declared in the include file, ERROR_CODES.PAS. The codes for run-time errors have values that start at 43000_8. These values are declared in the file SPC_ERRORS.PAS. Data General reserves error code values between 0_8 and 77777_8 for system and utility use.

User-Defined Errors

The ERETURN procedure gives you the ability to create an exception condition to serve a specific need for your application. ERETURN causes immediate termination of a routine, in a manner similar to RETURN. The difference is that ERETURN causes an exception to occur in the calling routine, so that an exception handler will be activated. The form of the ERETURN statement is:

ERETURN (*number*)

The *number,* an integer, identifies the type of exception. This number will be returned by any subsequent calls to the ERROR_CODE function.

Remember that *ERETURN terminates the routine that executes it.* Thus the exception will occur, not in the routine that does the ERETURN, but in the routine which is resumed. For instance, if a procedure P calls a procedure Q, and Q does an ERETURN, then an exception handler in P, not Q, will be activated.

Examples

Example 1

Example 1 deletes a file. Note that two errors are given specific handling (ERPRM and ERFDE) while other errors are propagated.

```
ACO:= BYTEADDR(FILENAME)
BEGIN
      SYSTEM (?DELETE,0,ACO,AC1,AC2)
EXCEPTION
      IF ERROR_CODE = ERPRM THEN BEGIN
           WRITELN ('CAN'T DELETE PERMANENT FILES');
           ERETURN(ERROR_CODE);
      END ELSE IF ERROR_CODE = ERFDE THEN
           WRITELN ('WARNING: ', FILENAME, ' NOT FOUND');
      ELSE
           ERETURN(ERROR_CODE);
END;
```

Example 2

This example illustrates the handling of a floating-point trap.

```
BEGIN
      DBL_REAL:= VAL/VALUE1;
EXCEPTION
      IF ERROR_CODE = EPFUF THEN
          DBL_REAL:=0.0
      ELSE
          ERETURN(ERROR_CODE);
END;
```

Example 3

The following example illustrates standard Pascal I/O with exception handling.

```
OK:= FALSE;
REPEAT
    BEGIN
        WRITE('ENTER AN INTEGER:  ');
        READLN (INT);
        OK:= TRUE;
    EXCEPTION
        IF ERROR_CODE = EPCUT THEN
           READLN;      %Cleans up rest of input line;
                        %stops at invalid integer.
```

Example 4

This example is a fragment of one program that copies a file.

```
DONE:= FALSE;
WHILE NOT DONE DO BEGIN
        READ (F,Q);
        WRITE (G,Q);
EXCEPTION
        IF ERROR_CODE = EREOF THEN
        DONE:= TRUE;
   ELSE
        ERETURN(ERROR_CODE);
END;
```

Multitasking

Multitasking is a powerful technique that enables you to divide a program into a number of asynchronous subprograms called *tasks*. The MP/AOS operating system provides a scheduling routine, the *task scheduler,* that switches control among the tasks to create the appearance of parallel processing. Multitasking can simplify the code for any program that must do several things at the same time.

Tasks

Multitasking is similar to multiprogramming in that a task operates independently from any other task(s). Still, all tasks are part of a single program and share the same memory, I/O channels, and other system resources. SP/Pascal programs can create, run, suspend, and delete tasks at run-time.

Multitasking has been added to SP/Pascal as a set of external routines with no structural change to the language. The external procedures for tasking are provided as an include file, TASKING.PAS. Table 12.1 is a summary of the tasking procedures, and each procedure is detailed in the remainder of this chapter. The routines used for managing dynamic memory space within task space are described in Chapter 9.

Routine Name	Operation
DRSCH	Disables task scheduler
ERSCH	Re-enables task scheduler
FORK	Creates a task at runtime
KILL	Deletes a task and releases its locks
LOCK	Locks out other tasks
PEND	Blocks a task to wait for an event
SETPRIORITY	Changes the priority of a task
UNLOCK	Unlocks locked tasks
UNPEND	Unblocks a task when an awaited event occurs

Table 12.1 External procedures for multitasking

Multitasking procedures allow you to create or delete tasks at run time. You can assign different priorities to tasks to control which one runs in response to a given event. You also can turn multitasking off and on with procedure calls to ensure that a critical operation is performed without interruption from other tasks. Tasks can use procedure calls to communicate with each other and to suspend or resume operations. The *MP/AOS System Programmer's Reference* (DGC 093-400051), describes how the MP/AOS system controls multitasking.

Managing Tasks

You specify the maximum number of tasks your program requires with the /TASKS=*number* keyword switch at bind time. The number of tasks a program can use must be less than the process maximum. The maximum number of tasks any program can use is specified at system generation. Under MP/AOS, the maximum number of tasks is limited only by the amount of memory available. Under AOS with the call translator, the maximum number of tasks is 30.

Memory Management

For the SP/Pascal run-time environment, each program's stack and heap are allocated in available unshared memory space during program initialization. (When a number of programs runs simultaneously, each program still maintains its independent logical process memory area, with its own program stack and heap, since each program executes within its own process space.)

A program's stack starts in low memory and grows upward; the heap starts in high memory of the program space and grows downward. When a program has one or more tasks, each task is allocated its own private stack in available heap space when the task is activated. Refer to Figure 12.1 for a conceptual representation of logical memory showing a single program with two active tasks. During execution the stack and heap grow toward each other: the program stack is allocated incrementally into the free space, as long as increasing the size does not cause the stack top to reach the bottom of the heap; the heap grows at the bottom provided it does not run into the top of the program stack.

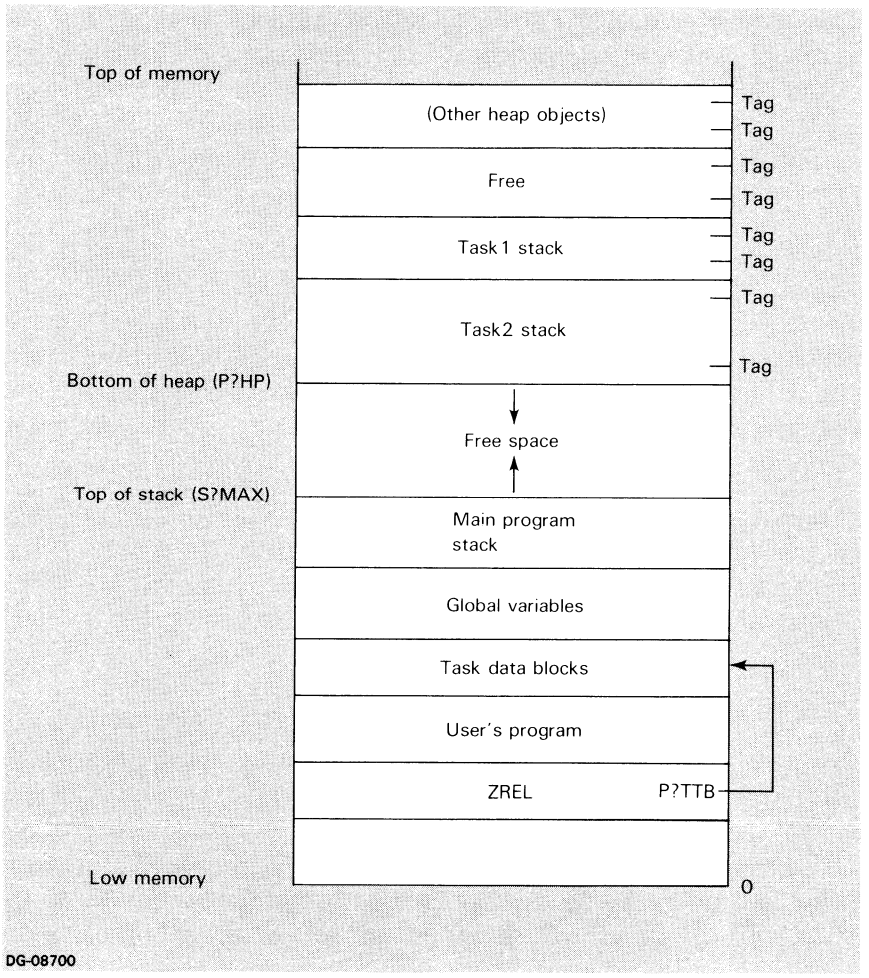


Figure 12.1 Task memory allocation

Two SP/Pascal-defined run-time environment variables are used as heap and stack markers: P?HP marks the low bound of the heap; and S?MAX marks the high bound of the main program's stack. The internal storage management routines guarantee a small buffer area between the stack maximum and the heap minimum. A subsequent task, creating dynamic variables with the predefined NEW routine, must know the main program's stack limit (S?MAX) to ensure an accurate picture of free space.

NOTE: *These details of implementation are provided for informational purposes only. The programmer must not manipulate P?HP or S?MAX.*

Heap Management

Each active task in a multitasking program has a private stack allocated in the main program's heap space. The task stack is used for execution of the task code (for example, calling and returning from procedures, local data storage). Independent task stacks enable each task to execute in the same environment with other asynchronous tasks.

The heap is managed by the boundary-tag method requiring two words of overhead for each heap entry. (Figure 12.1 shows the tags at the boundaries of the active tasks.) This method allows adjacent blocks to be combined quickly. The tag contains a status flag and an absolute block size value. The status flag is a negative value for a block in use, and a positive value for a free block.

When a block is freed at the bottom of the heap, the P?HP is increased; otherwise, the block is added to a list of free blocks. When you perform a NEW operation, the list is scanned first to check for availability of the requested storage; if a block of sufficient size is found on the free list, that block is used. Otherwise, P?HP is reduced, and the new variable instance is allocated at the bottom of the heap.

Generally, the heap is managed for you by the multitasking routines. If you are manipulating the heap with the NEW, DISPOSE, MARK, or RELEASE routines (described in Chapter 9), you must be careful not to de-allocate any part of the heap that is being used by some task.

Task Stack Management

A four-word data block is allocated for each task by the program initialization routine. When the task is activated, the data block is loaded with task-specific information. These words identify the task ID, the task's stack base and the stack limit within the program heap, and the task's *lock chain* (a list of resources acquired by the task). This information is used to release any owned resources when a task is killed. The task data block is pointed to by the ZREL location P?TTB. (Refer to Figure 12.1.) Zero entries in the task ID field indicate unused tasks.

The following sections describe task initialization, scheduling, and task interaction.

> **WARNING:** *SP/Pascal does not check to be sure that only unnested (global) routines with no parameters can be called as tasks. If parameters are present, they are not passed. If the routine is not global, the task within which the routine nests can terminate independently and can have undesirable effects on program execution.*

A task stack overflow is fatal. Tasks use a separate stack overflow handler and error reporter. The SP/Pascal-defined variable P?TSK is set at zero during program initialization, if multi-tasking is not required by the program. (Program initialization determines the multi-tasking status flag from the binder variable ?NTAS. The SP/Pascal binder macro switch /TASKS= allows you to specify the number of tasks when you bind the program.) Testing P?TSK during execution allows routines that manipulate global databases to avoid executing task scheduling calls (?DRSCH and ?ERSCH). All SP/Pascal run-time routines preserve the task scheduling state.

Creating Tasks: FORK Procedure

At run time you create tasks by invoking the following function:

```
EXTERNAL FUNCTION FORK
  (VAR ID: ?TASK_ID;
  PSTK : INTEGER;
  PPRI : ?TASK_PRI): BOOLEAN
```

The procedure arguments are described further in this section.

The recommended method for using a FORK call is illustrated in this example:

```
PROCEDURE TASK 1;
 BEGIN
       IF FORK(TASK1_ID,4000,127) THEN RETURN;
       {start of code for TASK 1}
       .
       .
       .
       RETURN; {TASK 1 is killed on encountering}
            {RETURN}
END; {end of TASK 1}
```

When you use this format, FORK works like a subroutine executed in parallel to the main routine. FORK returns true to the forking task and false to the newly created task. Any local variables of the calling task are copied into the new task's stack space.

The variables ID, PSTK and PPRI, serving as arguments to FORK, are associated with the values for *task identifier*, *task stack size* and *task priority*, respectively, for the ?CTASK system call.

Task Identifier - *id*

When you create a task, the system assigns it a *task identifier*. This is a 16-bit number you use with procedure calls for referring to the task. The task identifier, which should be set up as a global variable, is then available to both the forking and forked tasks.

Task Stack Size - *pstk*

You supply an integer stack size when calling FORK, because each task requires its own stack space. To determine the stack size required for a given task, use the /STACK switch at compile time. The /STACK switch displays the minimum stack size needed for each procedure in the program (see Chapter 13).

The size of the task stack is the largest sum of stack space required by each active routine in any calling sequence path from the procedure calling FORK. (A routine is active from the time it is called until it returns to the calling routine or until it is killed.) Refer to the program listing, (Example 1) to reconstruct the calling sequence. Using the stack size figures provided by the program listing, you can determine the task stack size (see Table 12.2).

Example 1. Task stack size calculation

```
SP/Pascal    Rev. 1.00      Thursday  April 18, 1982  1:51:04  PM
Compiling STK.PAS


 1.  PROGRAM STK;
 2.  %I-
 3.  INCLUDE TASKING.PAS;
50.
51.  VAR
52.      ID: ?TASK_ID;
53.      STKSIZ: INTEGER;
54.
55.  PROCEDURE ONE(P11:INTEGER; P12: REAL);
56.  VAR
57.      V11,V12: ARRAY[1..20] OF INTEGER;
58.  BEGIN
59.      %code...
60.  END;   %procedure one
61.
62.  PROCEDURE TWO(P21: INTEGER; P22: REAL);
63.  VAR
64.      V21,V22,V23: ARRAY[1..5] OF REAL;
65.  BEGIN
```

```
66.      %code...
67.  END;   %procedure two
68.
69.  PROCEDURE THREE;
70.  VAR
71.      V31,V32: REAL;
72.
73.  BEGIN
74.      IF FORK(ID,STKSIZ,127) THEN RETURN;
75.      ONE(10,100.001);
76.      TWO(10,100.001);
77.  END;   %procedure three
78.
79.  BEGIN  %main program
80.      THREE;
81.  END.


Stack allocation summary

Procedure ONE beginning on line 58

        Total save size = 45

        Local vars      Stack offset    Type kind

        V12                  26          array
        V11                   6          array

Procedure TWO beginning on line 65

        Total save size = 35

        Local vars      Stack offset    Type kind

        V23                  26          array
        V22                  16          array
        V21                   6          array

Procedure THREE beginning on line 73

        Total save size = 10

        Local vars      Stack offset    Type kind
```

```
        V32                     8          real
        V31                     6          real

Program STK beginning on line 79

        Total save size = 5



OB generation summary
Total program code size = 36
Total program literal size = 2
Dispatch tables size = 0
Zrel size = 0
Unshared code size = 2
Unlabeled common size = 0

81 source lines were compiled in 18 seconds

No Compilation Errors
```

Procedure(s)	Stack Size
Calling FORK: Procedure THREE	10
Called by Procedure THREE:	
Procedure ONE	45
Procedure TWO	(35)
Called by Procedure ONE	—
Called by Procedure TWO	—
	— —
Stack size needed for task	55

Table 12.2 Determination of task stack size

Note that the stack size required for procedure TWO is not included in the calculation of stack size needed for the task, because procedure TWO is not active at the same time as procedure ONE; since ONE is larger than TWO, you choose the largest possible path. Although the determination of task stack size often is more complex than the example (for instance, when recursive calls are involved), the approach remains essentially the same.

After determining stack size and making the assignment statement to the appropriate variable (here, STKSIZ), recompile the program.

Task Priority - *ppri*

A task priority is a number between 0 and 255; lower numbers represent higher priority. The system always runs higher-priority tasks first. Lower-priority tasks are run only when all higher-priority tasks are blocked from running.

NOTE: *When you start a program, the system assigns a priority of 127 to its initial task.*

You specify a task's priority when you create the task by supplying the priority number as an argument to the FORK call.

SETPRIORITY Procedure

A task can change its priority by invoking the SETPRIORITY procedure. The SETPRIORITY procedure is supplied by DGC in TASKING.PAS. The declaration for this procedure is as follows:

```
EXTERNAL PROCEDURE
SETPRIORITY(pri : ?TASK_PRI);
```

Deleting Tasks: RETURN and KILL

The tasks created by FORK are deleted (*killed)* when any of the following occurs:

- A RETURN procedure call from the forking procedure.
- The end of the forking procedure.
- An invocation of the KILL procedure specifies the task ID returned by the FORK.

The following program sample illustrates the first two conditions.

```
MODULE TASK;
  {$I-}
  INCLUDE TASKING.PAS;

  VAR
     ID, STKSIZ: INTEGER;
     CONDITION: BOOLEAN;

PROCEDURE TASK;

BEGIN
   IF FORK(ID,STKSIZ, 127) THEN RETURN;
      {This return is executed only by the forking}
      {procedure, since FORK returns FALSE in the}
      {task; therefore, the task is not killed.}
```

```
    {Task code}

  IF CONDITION THEN RETURN;
    {This return kills the}
    {task when the condition is TRUE, since}
    {the task has no ''return destination.''}


    {Task code}

END;     {End of procedure calling the task; task is}
     {killed, since END statement forces a return.}
```

The KILL procedure is declared as follows:

EXTERNAL PROCEDURE KILL(*id*:?TASK_ID);

This procedure ignores errors (since the only possible error is an invalid identifier signifying the task is already deleted). It also releases all *locks* owned by the task. Locks are described later in this chapter.

Task Scheduling

At times you need to suspend multitasking activity; for instance, you might need to read and modify a critical memory location without having some other task modify the same location at the same time. You might have a routine, such as a procedure that performs I/O to a global file. The routine could be shared by several tasks, but can be executed only by one task at a time. LOCK and UNLOCK, described further on in this section, are special-purpose alternatives to suspending multitasking.

The external procedures DRSCH and ERSCH support these activities. DRSCH disables the task scheduler, ensuring that no task can run except the one that executed the DRSCH. The task uses the ERSCH procedure to re-enable the scheduler when it completes the critical activity. These procedures correspond to the MP/AOS ?DRSCH and ?ERSCH system calls.

Intertask Communication

Two sets of procedures make it possible to control task operation and task synchronization. They are the procedures LOCK/UNLOCK and PEND/UNPEND.

Procedures LOCK/UNLOCK

When tasks share a common set of global variables, you must be capable of ensuring that only one task at a time updates them. Similar control of task operation is needed in connection with any SP/Pascal I/O if more than one task is writing to a given file.

The procedures LOCK and UNLOCK provide the security of mutual exclusion without having to disabling the task scheduler. These procedures are declared as follows:

EXTERNAL PROCEDURE LOCK(*g* :?TASK_LOCK);

EXTERNAL PROCEDURE UNLOCK(*g* :?TASK_LOCK);

A ?TASK_LOCK is automatically initialized to the unlocked state. When LOCK is called, unlocked ?TASK_LOCKs are placed in the locked state, and the calling task continues. If ?TASK_LOCK already is locked when LOCK is called, the calling task is suspended.

Only the task calling the LOCK can UNLOCK it. Any attempt at unlocking by another task results in a run-time error.

Once procedure UNLOCK is called, the task longest suspended by the lock on the ?TASK_LOCK resumes.

NOTE: *Any of the methods described earlier for terminating a task unlock all of a given task's locks before deleting that task.*

The use of locks to control access to shared variables is illustrated in the following example:

```
VAR  GLOBAL : RECORD
                LOCK: ?TASK_LOCK;
                 .
                 .
              END;
BEGIN
    {code ...}
    LOCK(GLOBAL.LOCK);       {Global is a structure
        shared among tasks.}
  .
  .
  .
  {modify GLOBAL at will}
  .
  .
  .
  UNLOCK(GLOBAL.LOCK); {allow other tasks access
        to GLOBAL}
  .
  .
  .
END.
```

Procedures PEND/UNPEND

Tasks are able to control each other's actions. The system enables tasks to synchronize their activities with the procedures PEND and UNPEND. These procedures correspond to the MP/AOS ?PEND and ?UNPEND system calls and are declared as follows:

```
EXTERNAL PROCEDURE PEND(e :?TASK_EVENT);

EXTERNAL PROCEDURE UNPEND(e :?TASK_EVENT);
```

When a task executes a PEND, that task is blocked from running until a particular event occurs. The event is specified by a 16-bit number (event code). If PEND is issued while the scheduler is disabled, scheduling is re-enabled after the PEND call returns.

The event specified in the 16-bit number must be used by some other task in an UNPEND call. This call signals the occurrence of the specified event and allows resumption of the task waiting for it.

Event number values must be between zero and ?EVMAX. The values between zero and ?EVMIN are reserved for system-defined events. You can specify these reserved values on a PEND, but not on an UNPEND. Values between ?EVMIN and ?EVMAX can be specified on either PEND or UNPEND. ?EVMIN and ?EVMAX are defined in the *MP/AOS System Programmer's Reference,* (DGC No. 093-400051). They are declared in the include file TASKING.PAS.

NOTE: *If locks are used, then events between 40000_8 and EVMAX are reserved.*

Console Interrupt Tasks

It is possible to interrupt an executing SP/Pascal program by typing the CTRL-C CTRL-A sequence on your keyboard. To receive this interrupt, the program must create a task that pends on an event number equal to ?EVCH plus the channel number of the console keyboard (usually ?INCH). The task is unpended when you type the CTRL-C CTRL-A sequence. Then the program can accept a command from you or shut itself down.

Program Examples

Example 2, below, illustrates the use of multitasking, task management, scheduling and intertask communication. The program's function is to copy buffers of the number and length the user specified.

```
SP/Pascal    Rev. 1.00      Thursday  April 8, 1982  4:48:32  PM
Compiling FCOPY.PAS


  1.  MODULE FCOPY;
  2.  {$I-}
  3.  INCLUDE TASKING.PAS;
 50.  INCLUDE IO_CALLS.PAS;
122.  INCLUDE SYSLIB.PAS;
124.  INCLUDE FILE_PARS.PAS;
171.  INCLUDE ERROR_CODES.PAS;
270.
271.  CONST
272.      NO_BUFFER = 0;
273.      BUF_COUNT = 3;
274.      BUF_LENGTH = 1024;
275.      EMPTY_PEND_KEY = 1000;
276.      FULL_PEND_KEY = 1001;
277.
278.  TYPE
279.      BUF_INDEX = NO_BUFFER..BUF_COUNT;
280.
281.      BUF_TYPE = RECORD
282.                 NEXT: BUF_INDEX;
283.                 ENDF: BOOLEAN;
284.                 BUF: STRING BUF_LENGTH;
285.                 END;
286.
287.      HEADER = RECORD
288.                 HDR: BUF_INDEX;
289.                 LOCKS: ?TASK_LOCK;
290.                 KEY: ?TASK_EVENT;
291.                 END;
292.
293.  VAR
294.      INCHAN,OUTCHAN: CHANNEL;
295.      EMPTY,FULL: HEADER;
296.      BUFFER: ARRAY [1..BUF_COUNT] OF BUF_TYPE;
297.
298.  FUNCTION NEXT_BUFFER(VAR Q:HEADER): BUF_INDEX;
299.  %get next buffer from linked list
300.
301.  VAR
302.      ACTIVE: BUF_INDEX;
303.
304.  BEGIN
305.      ACTIVE := NO_BUFFER;
306.      WITH Q DO
```

```
307.      WHILE ACTIVE = NO_BUFFER DO
308.      BEGIN
309.          LOCK(LOCKS);    %only one task can modify the header at a time
310.          ACTIVE := HDR;
311.          IF ACTIVE = NO_BUFFER THEN
312.          BEGIN
313.              DRSCH;  %make sure the other task doesn't run until we're pended
314.              UNLOCK(LOCKS);
315.              PEND(KEY);  %pend renables scheduling
316.              END
317.          ELSE
318.          BEGIN
319.              HDR := BUFFER[ACTIVE].NEXT;
320.              UNLOCK(LOCKS);
321.              END;
322.          END;    % while active = no_buffer
323.      NEXT_BUFFER := ACTIVE;
324.      END;    % next_buffer
325.
326.  PROCEDURE ADD_BUFFER(VAR Q: HEADER; BP: BUF_INDEX);
327.  % add buffer to linked list
328.
329.  VAR
330.      ACTIVE: BUF_INDEX;
331.
332.  BEGIN
333.      WITH Q DO
334.      BEGIN
335.          LOCK(LOCKS);   % only one task can modify the header at a time
336.          ACTIVE := HDR;
337.          IF ACTIVE = NO_BUFFER THEN HDR := BP
338.          ELSE
339.          BEGIN
340.              WHILE BUFFER[ACTIVE].NEXT < > NO_BUFFER DO
341.              ACTIVE := BUFFER[ACTIVE].NEXT;
342.              BUFFER[ACTIVE].NEXT := BP;
343.              END;
344.          BUFFER[BP].NEXT := NO_BUFFER;
345.          UNPEND(KEY);    % signal there is a buffer in the linked list
346.          UNLOCK(LOCKS);
347.          END;    % with q
348.      END;    % add_buffer
349.
350.  PROCEDURE INPUTF;
351.  % input task
```

```
352.
353.  VAR
354.      TASKID: ?TASK_ID;
355.      BP: BUF_INDEX;
356.      STATUS: INTEGER;
357.
358.  BEGIN
359.      IF FORK(TASKID,100,127) THEN RETURN;
360.      WHILE TRUE DO   % loop forever
361.      BEGIN
362.          BP := NEXT_BUFFER(EMPTY);
363.          WITH BUFFER[BP] DO
364.          BEGIN
365.              ENDF := FALSE;
366.              CHARREAD(INCHAN,BUF_LENGTH,BUF,STATUS);
367.              IF STATUS < > 0 THEN
368.              IF STATUS = EREOF THEN ENDF := TRUE
369.              ELSE ?EXIT(STATUS,'INPUT FILE READ');
370.              END;
371.          ADD_BUFFER(FULL,BP);
372.          IF BUFFER[BP].ENDF THEN RETURN;
373.          END;
374.      END;
375.
376.  ENTRY PROCEDURE COPY(INP,OUT: PATHNAME);
377.
378.  VAR
379.      BP: BUF_INDEX;
380.      STATUS: INTEGER;
381.
382.  BEGIN
383.      OPENFILE(INCHAN,INP,0,0,0,STATUS);
384.      IF STATUS < > 0 THEN ?EXIT(STATUS,'INPUT FILE OPEN');
385.      OPENFILE(OUTCHAN,OUT,DE+EX,?DUDF,4,STATUS);
386.      IF STATUS < > 0 THEN ?EXIT(STATUS,'OUTPUT FILE OPEN');
387.      EMPTY.HDR := 1;
388.      EMPTY.KEY := EMPTY_PEND_KEY;
389.      FULL.HDR := NO_BUFFER;
390.      FULL.KEY := FULL_PEND_KEY;
391.      FOR BP := 1 TO BUF_COUNT - 1 DO
392.      BUFFER[BP].NEXT := BP+1;
393.      BUFFER[BUF_COUNT].NEXT := NO_BUFFER;
394.      INPUTF;
395.      WHILE TRUE DO   % buffer output loop
396.      BEGIN
```

```
397.          BP := NEXT_BUFFER(FULL);
398.          WITH BUFFER[BP] DO
399.          BEGIN
400.              CHARWRITE(OUTCHAN,BUF,STATUS);
401.              IF STATUS < > 0 THEN ?EXIT(STATUS,'OUTPUT FILE READ');
402.              IF ENDF THEN EXITLOOP;
403.              END;    % with buffer[bp]
404.          ADD_BUFFER(EMPTY,BP);
405.          END;
406.      CLOSEFILE(INCHAN,STATUS);
407.      IF STATUS < > 0 THEN ?EXIT(STATUS,'INPUT FILE CLOSE');
408.      CLOSEFILE(OUTCHAN,STATUS);
409.      IF STATUS < > 0 THEN ?EXIT(STATUS,'OUTPUT FILE CLOSE');
410.      END;    % copy


Stack allocation summary

Function NEXT_BUFFE beginning on line 304

        Total save size = 7

        Local vars      Stack offset    Type kind

        ACTIVE                6         subrange

Procedure ADD_BUFFER beginning on line 332

        Total save size = 7

        Local vars      Stack offset    Type kind

        ACTIVE                6         subrange

Procedure INPUTF beginning on line 358

        Total save size = 10

        Local vars      Stack offset    Type kind

        STATUS                8         integer
        BP                    7         subrange
        TASKID                6         subrange

Procedure COPY beginning on line 382
```

```
        Total save size = 10

        Local vars      Stack offset    Type kind

        STATUS                  7         integer
        BP                      6         subrange
OB generation summary
Total program code size = 417
Total program literal size = 64
Dispatch tables size = 0
Zrel size = 0
Unshared code size = 1560
Unlabeled common_size = 0

410 source lines were compiled in 58 seconds

No Compilation Errors
```

Operating Procedures

This chapter describes the development process for SP/Pascal programs. It provides instructions for operating the SP/Pascal compiler and for binding, debugging, and executing SP/Pascal programs under the MP/AOS and AOS operating systems. The program development phases covered are:

compiling an SP/Pascal source unit
binding an SP/Pascal program
debugging an SP/Pascal program
executing an SP/Pascal program

Where appropriate, references to other manuals in the MP/AOS documentation set indicate additional program development information.

Compiling

The compiler, a program called SPC, can be invoked at the CLI level with the XEQ SPC command. The command line syntax requires that no space or comma delimiters appear in the designation of the compiler name and switches.

The command line for compiling an SP/Pascal source file is

XEQ SPC[/L[=*listfile*]] [/E=*errorfile*] [/O=*objectfile*] [/STACK] [/OPT=*opt-char*[+|−][...*opt-char*[+|−]][/A[=*asmfile*]] [/NOLEF] [/N] *sourcefile*

Following are three examples of compilation command lines:

```
XEQ SPC/L=MY_LIST/A=ASMLST MY_MOD
SPC MY_PROG
XEQ SPC/L/O=ROUTINE233/STACK/NOLEF/N MY_PROG
```

Compiler Switches

Conditional Code Generation

The SP/Pascal compiler does not generate code for unreachable statements in the program. This feature allows you to insert debugging or diagnostic statements into the program text and to direct the compiler to generate code for these statements only in certain circumstances. For example,

```
IF DEBUG THEN BEGIN
    .
    .   {special statements for debugging purposes}
    .
    END;
```

When the identifier DEBUG is a Boolean constant, code for the special statements is generated only when DEBUG has the value true. A typical use would define DEBUG in an INCLUDE file, and then, using either the searchlist or appropriate link, you can pick up the appropriate definition.

Compiler Program Listing

When you include the /L switch, a list file is generated to the line printer. If you use the keyword switch (/L=*listfile),* the listing is generated to the specified file. If the file doesn't already exist, it is created; if the file already exists, the listing is appended to it. If you omit the optional /L switch, the compiler does not produce a program listing.

Compiler Error File

When you use the keyword switch /E=*errorfile*, the list is generated to the specified file, as well as appearing at the end of the program listing file when that file is specified. If the error file doesn't already exist, it is created; if the file does exist the listing is appended to it.

If you omit the optional /E switch, no separate error file is created.

Compiler Object File

If no errors are found in the source program during compilation, the compiler produces an object file that can be used as input to the binder. No object file is created when compilation errors exist.

When you include the /O=*objectfile* switch, the name of the object file is *objectfile.* If the file doesn't already exist, it is created; if the file does exist the new object file overwrites the old.

When you omit the optional /O switch, the default for the object file is to create the file in the originating directory. The file has the same name as the source program with the .OB extension, rather than .PAS. When the default object file already exists, the new file overwrites the old.

Compiler Stack Calculation

When you use the /STACK switch, the compiler generates a table at the end of the listing file. The table contains one line for each routine that is defined in the compilation unit. The line provides the routine name, the line number of the start of the statement section for that routine, and the stack size to be allocated when the routine is executed. For each routine, the compiler also displays the names stack offset, and type class of every local variable. If you do not specify the /STACK switch, the compiler omits the table.

Compiler Options Override

When you include the /OPT=*opt-char* keyword switch, you can override the compiler's default options. If you omit this keyword switch and its option characters, the existing defaults are in effect.

The *opt-char* specified can be any of the single-character mnemonics representing the options described under "Compiler Options," further on in this chapter. Each option character must be followed by either a plus (+) or a minus (−) sign. The plus sign forces the option on throughout the program; the minus sign turns the option off throughout the program.

You can specify multiple options. For example:

```
SPC/OPT=S-V-P- sourcefile
```

This command directs the compiler to omit generating run-time checks for array subscripts, record variants, and pointer references.

Program Debugging

For SP/Pascal, a separate process debugger can be used during test execution of the program. The compiler switch /A=*asmfile* produces a disassembler listing, a symbolic assembly language listing of the generated code for each SP/Pascal source line. Use the disassembler listing when operating with the process debugger. The process debugger is described in *MP/AOS Debugger and Performance Monitoring Utilities* (DGC No. 069-400205).

When you use the /A switch alone, the symbolic assembly listing is produced with the listfile (or to the console). When you use the keyword switch /A=*asmfile*, the listing is generated to the specified file. If the file doesn't already exist, it is created; if the file does exist the listing is appended to it.

Compiler LEF Instructions

Use the /NOLEF switch to indicate to the compiler that short (single-word) LEF instructions cannot be used in the object code. (The two-word extended LEF instruction ELEF always is permitted.) This switch is necessary for system programs that must issue I/O instructions; short LEF instructions generated by the SP/Pascal compiler would be misinterpreted as I/O instructions. When you do not specify the /NOLEF switch, short LEF instructions are used where appropriate. The SP/Pascal runtimes do not use short LEF instructions.

Compiler Syntactical Check Only

The /N switch indicates that no object file is to be produced by the compiler. Instead, the source file is analyzed for syntactic and semantic errors only.

Program or Module Source File

The source file name is a required argument in the compiler command line. If you provide a .PAS extension to your the compiler searches for a file called *sourcefile*. If you don't provide an extension, the compiler tries to read text from *sourcefile*.PAS, and, if this fails, tries to read source text from *sourcefile*. If both attempts are unsuccessful, the compiler issues an error message and returns you to the CLI. We suggest you always use the .PAS extension.

Compiler Options

You can instruct the compiler to generate code according to certain options. For example, you can request insertion or omission of code for run-time error detection, or you can suppress a listing of include files. The following compiler options are available:

I	List all INCLUDE files. (Default=OFF)
L=5 | 8 | 10	Specify truncation of the identifier for object file external or entry names to a length of either five, eight, or ten characters (default =10).
N	Generate line numbers (not supported) (MP/Pascal compatibility only).
O	Generate code to check for overflow on all integer arithmetic operations (default=OFF).
P	Generate code to initialize to zero all pointers and structures containing pointers. Check all pointer references for a value of nil (default=ON).
R	Generate code to check all subrange assignments (default=OFF).
S	Generate code to check all array subscripts (default=ON).
T	Generate code to check range on current set members (default=OFF).
V	Generate code to check all variant references (default=ON).
W	Generate code to check whole and integer ranges in assignment and value parameter contexts (checking that whole-to-integer assignments do not exceed MAXINT and that negative values are not assigned to whole variables) (default=OFF).
Z	Set divide-by-zero checking (default=ON).

To summarize, the options P, S, V, and Z are on (+) by default, and all other options are off (−).

Specifying Options Within the Source Code

When a compilation unit requires special compiler options, you can set up a directive in the source file to establish specific options for that unit, and then you can clear those options with another directive. (The options that can be specified are those described in the preceding list and specified initially to the compiler with the /OPT= switch.)

The directives are specified in the source program as a comment with a dollar sign ($) as the first character of the comment. Comma delimiters are required between individual option mnemonics. The options directive format is

$[<stack directive>]<option list>
<delimiter><any comment>]

NOTE: *No spaces are allowed in the options string.*

For example,

```
$I-,S+,V+
```

updates the current option list so that INCLUDE files are not listed, but subscripts and variants are checked.

To change the compiler options again or to restore them to their former state you can insert another comment in the source code. The following directive is used when the checking no longer is required but when the include files listing still is suppressed:

```
$S-,V-
```

Alternatively, you could use the option stack directive as explained in the following paragraphs.

Saving Options

When an include file or a portion of a compilation unit requires special compiler options, the existing set of options can be saved in a stack-like fashion and later reinstated by using the push and pop stack directives to save and restore the option set. The beginning directive starts with a push character (a greater-than sign (>)) followed by the option mnemonics and on-off signs. (After a push, all options are off.)

To clear one or more of the options specified with the push directive, you can pop the options with an ending directive comment with a dollar sign followed by the pop character (a less-than sign (<)) and any new options.

If you then wanted to list portions of INCLUDE files, you would specify:

```
$>I+
```

Later in the program, you can pop the original option list, to suppress include file listings as follows:

```
$<
```

The listing control option allows for insertion of the ↑ or ^ character as the first character of a comment string to force a page eject in the listing file.

Compiler Error Messages

The compiler generates a list of error messages with total error count and appends the list to the end of the program listing. Each message lists the number of the source line that triggered the message. The total error count is displayed at the terminal when compilation finishes. Appendix B lists the compiler error messages.

Storing Compiler Output

The object module produced by the compiler can stay in the specified directory or can be made part of a library of compilation units in object form. Use the library editor to establish libraries of object modules. For more information on the MP/AOS library editor, refer to *MP/AOS Macroassembler, Binder, and Library Utilities* (DGC No. 069-400210).

Binding

The object module produced by the compiler must be bound before it can become an executable program. The binder resolves external calls within the program, locates required routines in the library, and binds in any specified separate compilation units stored in object form.

Under the MP/AOS operating system, you bind a program using the following Command Line Interpreter (CLI) bind macro, (SPCBIND.CLI), which is supplied with the SP/Pascal compiler.

SPCBIND*[switches] objectfile [file-1 [file-2[...file-n]]]*

The program name assigned to the binder output is the same as that of the first object module (*objectfile)* in the macro command line. The binder provides the .PR extension to the name.

Objectfile must be an object module file. *File-1* through *file-n* can be either object file names or the names of the user libraries containing the necessary object files. All user libraries and compilation units that are required by the program must be specified in the binder macro.

For example,

```
SPCBIND MYPROG
SPCBIND MYPROG MOD1 MOD2
SPCBIND MYPROG MYMODLIB
SPCBIND PROG2 !* OVY_ONE ! OVY_TWO *!
SPCBIND ROOT !* MOD-A MOD-B ! MOD-G *!
```

The SPCBIND macro produces a binder listing containing the program load map and any binder error messages. The listing file is placed in the working directory and given the same name as *objectfile* along with a .BLS extension.

To use multitasking with an SP/Pascal program, use the /TASKS= switch when binding the program. Task control blocks (TCB's) are not allocated unless you specify the /TASKS=*number-of-tasks* switch. For example,

```
SPCBIND/TASKS=3 PROG1
```

The /STD binder macro switch, specific to SP/Pascal, allows you to request run-time detection of certain non-standard SP/Pascal extended features, such as reading and writing to the same file, or extended syntax for real numbers. (The /STD switch changes the setting of P?STD from the default (0) to a 1. P?STD is described further in Chapter 7.) When you omit the /STD switch, the SP/Pascal extensions are permitted, and the SP/Pascal run-time routines do not flag instances of these extended features. For example,

```
SPCBIND/STD TESTPROG
```

would detect and flag uses of these non-standard extensions.

The following binder switches can also be used with the SPCBIND macro:

/LIBLIST	for an individual listing of all modules loaded from a library,
/ALPHA	to include a list of global symbols sorted in alphabetical order,
/NUMERIC	for a list of global symbols sorted by numeric value.

For details on these and additional binder switches, refer to *MP/AOS Macroassembler, Binder, and Library Utilities* (DGC No. 069-400210).

For binding AOS cross-development programs, refer to Appendix D.

Executing Programs

SP/Pascal programs can be executed with the CLI command XEQ followed by the program name. For example,

```
XEQ MYPROG
```

If a run-time error occurs during program execution, the program aborts, and the associated error message is printed. (The SP/Pascal exception-handling facility provides execution error support. Refer to Chapter 11.)

With system run-time error handling, the message text is preceded by the memory address (program counter value) of the instruction executing when the error occurred (the location of the call to the error routine). For example,

```
AT LOCATION 20114
ERROR FROM PROGRAM
ERROR: HEAPLIMIT EXCEEDED.
```

The following list shows the run-time error messages that can be generated by a program. Some of these errors are generated only when certain checking options have been turned on. (Appendix B lists both the compiler error messages and the run-time error messages.)

```
Implementation error
Subscript out of range
Invalid pointer reference
Invalid variant tag
Range error
Heaplimit exceeded
Integer overflow/underflow
Invalid set element
Invalid case selector
Undefined routine
Stacklimit exceeded
Invalid string length
Uninitialized string
String parameter too long
Invalid string index
String overflow
Real overflow/underflow
Integer division by zero
File not opened with reset/rewrite
File not opened for reading
File not opened for writing
Input conversion error
Record size does not match file type
Bad stack (display links increasing)
Illegal to supply substring here
Invalid width specification
Lock/unlock error
Hardware stacklimit exceeded
Failure in fork
LN of a non-positive number
EXP argument out of range
Negative SQRT argument
Floating-point overflow trap
Floating-point underflow trap
Floating-point zero-divide trap
Floating-point mantissa overflow trap
Fork VAR parameter not declared locally
Invalid format for File of String
```

In addition to these specific errors, other system error messages also can be produced. (See *MP/AOS System Programmer's Reference,* DGC No. 093-400051, for a list of these messages.)

ASCII Character Set

A

DECIMAL	OCTAL	HEX	KEY SYMBOL	MNEMONIC
0	000	00	↑@	NUL
1	001	01	↑A	SOH
2	002	02	↑B	STX
3	003	03	↑C	ETX
4	004	04	↑D	EOT
5	005	05	↑E	ENQ
6	006	06	↑F	ACK
7	007	07	↑G	BEL
8	010	08	↑H	BS (BACKSPACE)
9	011	09	↑I	TAB
10	012	0A	↑J	NEW LINE
11	013	0B	↑K	VT (VERT.TAB)
12	014	0C	↑L	FORM FEED
13	015	0D	↑M	CARRIAGE RETURN
14	016	0E	↑N	SO
15	017	0F	↑O	SI
16	020	10	↑P	DLE
17	021	11	↑Q	DC1
18	022	12	↑R	DC2
19	023	13	↑S	DC3
20	024	14	↑T	DC4
21	025	15	↑U	NAK
22	026	16	↑V	SYN
23	027	17	↑W	ETB
24	030	18	↑X	CAN
25	031	19	↑Y	EM
26	032	1A	↑Z	SUB
27	033	1B	ESC	ESCAPE
28	034	1C	↑\	FS
29	035	1D	↑]	GS
30	036	1E	↑↑	RS
31	037	1F	↑—	US

DECIMAL	OCTAL	HEX	KEY SYMBOL
32	040	20	SPACE
33	041	21	!
34	042	22	" (QUOTE)
35	043	23	#
36	044	24	$
37	045	25	%
38	046	26	&
39	047	27	' (APOS)
40	050	28	(
41	051	29	)
42	052	2A	*
43	053	2B	+
44	054	2C	, (COMMA)
45	055	2D	-
46	056	2E	. (PERIOD)
47	057	2F	/
48	060	30	0
49	061	31	1
50	062	32	2
51	063	33	3
52	064	34	4
53	065	35	5
54	066	36	6
55	067	37	7
56	070	38	8
57	071	39	9
58	072	3A	:
59	073	3B	;
60	074	3C	<
61	075	3D	=
62	076	3E	>
63	077	3F	?
64	100	40	@

DECIMAL	OCTAL	HEX	KEY SYMBOL
65	101	41	A
66	102	42	B
67	103	43	C
68	104	44	D
69	105	45	E
70	106	46	F
71	107	47	G
72	110	48	H
73	111	49	I
74	112	4A	J
75	113	4B	K
76	114	4C	L
77	115	4D	M
78	116	4E	N
79	117	4F	O
80	120	50	P
81	121	51	Q
82	122	52	R
83	123	53	S
84	124	54	T
85	125	55	U
86	126	56	V
87	127	57	W
88	130	58	X
89	131	59	Y
90	132	5A	Z
91	133	5B	[
92	134	5C	\
93	135	5D	]
94	136	5E	↑ OR ^
95	137	5F	← OR —
96	140	60	` (GRAVE)

DECIMAL	OCTAL	HEX	KEY SYMBOL
97	141	61	a
98	142	62	b
99	143	63	c
100	144	64	d
101	145	65	e
102	146	66	f
103	147	67	g
104	150	68	h
105	151	69	i
106	152	6A	j
107	153	6B	k
108	154	6C	l
109	155	6D	m
110	156	6E	n
111	157	6F	o
112	160	70	p
113	161	71	q
114	162	72	r
115	163	73	s
116	164	74	t
117	165	75	u
118	166	76	v
119	167	77	w
120	170	78	x
121	171	79	y
122	172	7A	z
123	173	7B	{
124	174	7C	|
125	175	7D	}
126	176	7E	~ (TILDE)
127	177	7F	DEL (RUBOUT)

DG-05495

Compiler Error Diagnostics

B

This appendix describes four categories of compiler error detection, and a series of tables show the error messages the compiler can issue in each phase of its operation. These error messages are listed alphabetically within each category, along with a brief explanation of each message.

Run-time error messages appear at the end of Chapter 13, "Operating Procedures."

The SP/Pascal compiler error detection and reporting capabilities are divided among three phases of the compilation process: syntax analysis, semantic analysis, and code generation. The compiler identifies errors in the earliest possible phase. When an error is detected, a description is written to the listfile and to the file given with /E=*filename,* if any. These errors are separated into four categories, according to the possible actions the compiler may take at the point the error is detected. The four possible actions are:

1. Issue a warning message and continue.
2. Issue an error message, attempt to correct the error and continue.
3. Issue an error message, but do not generate an object file.
4. Issue an error message and abort execution of the compiler.

When the SP/Pascal compiler completes without detecting any errors, it returns a message to the CLI consisting of the characters "OK"; otherwise it returns a null termination message.

Error Message Occurrence

Category One warnings are issued when the compiler detects a potential inconsistency in an implementation-defined aspect of the language, or when it cannot guarantee that an assertion it has used for the compilation of the program will be inviolate. These situations do not impair the generation of code, but may affect the execution of the program. The user is informed that this condition exists and the compiler ignores the error.

Category Two errors are detected only during syntax analysis. The compiler attempts to perform certain recovery actions when a syntax error is detected. If this action resolves the error condition, then the compilation continues with semantic analysis, but no code is generated. This action is intended to provide a more thorough error analysis of the source text.

Category Three errors are violations of the language requirements serious enough to prevent the compiler from proceeding to the next compilation phase. Both syntactic and semantic errors fall into this category. When a Category Three error is detected, the compiler finishes the compilation phase that is currently executing and then stops the compilation process. Thus, if non-recoverable syntactic errors are present, the source program will not be analyzed for semantic errors. When a Category Three error is detected, no object file is produced.

Category Four errors are issued for internal compiler errors or overflow of compiler data bases required to proceed further. The compiler performs run-time checks to guarantee the integrity of its internal state and to check against limits of its data structures. When it detects a Category Four error, the compiler halts execution.

The texts of the error messages are listed alphabetically in Tables B.1 through B.7, in order by compilation phases.

Syntax Phase Messages

Error Message	Explanation
Illegal octal number in angle brackets	An octal value enclosed in angle brackets in a string constant is greater than 377_8 or contains a non-octal digit.
Illegal push/pop option command	A pop operation on an empty option stack has been detected.
Unknown option	An option character in an option comment is not one of the defined characters. (P,R,S,T,V,O,W,Z,N,L)

Table B.1 Syntax phase messages, Category 1 warnings

Messages
Syntax error before *xxx* Recovery inserted *yyy* before *zzz*
Syntax error before *xxx* Recovery replaces *yyy* by *zzz*
Syntax error before *xxx* Recovery deletes *yyy* and replaces *www* with *zzz*
Syntax error before *xxx* Parse restarted on line *yyy* with *zzz*

Table B.2 Syntax phase messages, Categories 2 and 3

In all cases in the error messages shown in Table B.2, the location of the error is correct. The recovery actions given are included only as hints as to what construct might be correct in this context. Use the "Syntactic error before *xxx*" phrase to localize the error on the line in question. Use the recovery action given to help in deciding how to repair the error.

Error Message	Explanation
Exceeded maximum include depth	The compiler limit on nesting of include files has been exceeded.
Feature not yet implemented	An unimplemented feature of standard Pascal has been used. Examples: a GOTO statement or a routine definition with parametric procedures.
Illegal character in source	An undefined character has been used in the source text.
Internal compiler error	An internal error in the parser has occurred.
Radix out of range	The radix for a non-decimal constant is outside the permissible range.
Real exponent out of range	The exponent part of a real constant is outside the representable range.
Real number out of range	The value of a real constant is outside the representable range.
Source text beyond logical end of module	The source file contains characters beyond the point where the compiler has determined it should end.
Statement nesting is too deep	The compiler limit on nesting of control and compound statements has been exceeded.
Too many names in comma list	The compiler limit of expressions in a list has been exceeded.
Unable to open include file	An include file name cannot be opened at compile time.
Unterminated comment	A comment has not been properly terminated before the end of the source input. Comments that begin with '{' must be terminated with '}', and comments that begin in '(*' must be terminated with '*)'.
Unterminated string	A string constant has not been properly terminated before the end of the current line.

Table B.3 Syntax phase messages, Categories 3 and 4

Semantic Phase Messages

Error Message	Explanation
Bit fields(s) do not end on word boundary	The compiler generates this message when it must leave space at the end of a component in a structure to satisfy the alignment requirements of the next component or to insure that the structure ends on a word boundary.
Control var not declared locally	This message warns that a variable used as the control variable in a FOR loop is not local to the routine containing the FOR statement. The compiler asserts that the control variable is not modified within the FOR statement.
Files in the heap have global persistence	This message is generated for pointer type definitions with object types containing a file. If the file is opened, it will not be closed implicitly when the pointer variable is deallocated.
Incomplete variant definition	This message is generated in a variant record definition when the set of variant labels is not equal to the set of variant values in the tag type.
Only "input", "output" defined as external files	This message appears when arguments other than 'input' or 'output' were supplied in the program heading.
Whole <=> integer operands in expression	This message warns you when a binary operation between an integer type value and a whole type value will be performed. The language rules state that the operation will be signed.

Table B.4 Semantic phase messages, Category 1 warnings

Error Message	Explanation
^p is not allowed	A self-referencing pointer definition is prohibited.
2nd operand must be set for IN operator	The second operand to an 'IN' operator is not a set expression.
Arg not compatible in predefined routine with id *xxx*	An argument to the named predefined routine is not compatible with its definition.
Argument may not be passed as a var parm	A constant, expression or value parameter may not be passed as a VAR parameter.
Argument may not be recasted	A string piece may not be passed as a recasted argument.
Argument not compatible with parameter with id *xxx*	The identifier is the name of the parameter in the routine declaration to which the corresponding argument is not compatible.
Argument not compatible with text file	An argument in a READ, READLN, WRITE, or WRITELN procedure must be scalar, real, or string for text files.
Bit qualifier not in 1 .. 16	The bit qualifier may not exceed the number of bits in a word.
Call name is not a procedure with id *xxx*	An identifier used in a procedure call statement has not been declared as a procedure.
Call name must be a function with id *xxx*	The identifier in a function reference was not defined as a function.
CASE label not compatible with selector	Each label constant must be compatible with the selector type.
CASE selector type must be scalar	The selector expression in a CASE statement must be scalar type.
Constant DIV/MOD by zero	A constant divisor of 0 was detected in an expression.
Constant exp outside set bounds	A member designator in a named set constant is outside the bounds of the set base type.
Constant subscript outside array bounds	Constant array subscripts are checked at compile-time.
Constant value outside subrange bounds	An ordinal constant in a structured-constant definition is outside the bounds of the associated ordinal type.
Control var cannot be a parameter	The control variable in a FOR loop must be a simple variable that is not a parameter.
Control var must be ordinal type	Only ordinal type control variables are permitted in FOR loops.
Control var used in nested FOR	Nested FOR loops must employ unique control variables.
Data representation exceeds bit qualifier	The number of bits necessary to store the values in the bit qualified type exceed the declared size. *E.g.,* 0..255 bit 5
Entry declarations must be at global level	Only global variables may be shared between modules.
Enumeration list exceeds 255 names	The compiler limit on enumeration size has been exceeded.
ERROR_CODE reference not in exception block	The predefined function ERROR_CODE may be invoked only in an exception block.
Error in redefining enumeration type	The number or name of each enumeration constant does not agree with a previous declaration.
[exp .. exp] not allowed as array subscript	This form is valid only for named set constants.
Expression must be integer or whole type	The expression in an ERETURN statement must be integer or whole.
Expression must be ordinal constant	A constant expression must have an ordinal result type.

Table B.5 Semantic phase messages, Category 3 errors

Error Message	Explanation
Expression not compatible with control var	The initial value expression in a FOR statement is not compatible with the type of the control variable.
Expression not compile-time constant	A non-constant expression is used in a declaration.
File buffers are not supported	A dereferenced file variable, *e.g.*, F^, has been used. File buffers are not implemented in SP/Pascal.
File element type must not be file	A component of a file may not include another file.
File of pointers not allowed	A file of pointers is an unsafe construct and is prohibited.
File parameters must be defined VAR	File variables may not be passed as value parameters.
Files may not be compared	The operands for a relational operator may not be file types, or contain components of file type.
File types not allowed in variant part	An implementation restriction on file types in record structures.
FORWARD definitions may not be repeated	Only one FORWARD declaration per routine is permitted.
Fraction width only valid for reals	The form *x:width:fraction* is only permitted when *x* is of type real or double_real.
Global file vars not permitted in overlays	Non-static global file variables are not permitted in overlay modules.
Illegal assignment to function result	An assignment to a function identifier is defined outside the scope of the function body.
Illegal width specifier in arg list	A constant width specifier is negative in the argument list.
Illegal width specifier in argument list	A width specifier on an argument to a procedure that is not the predefined procedure WRITE or WRITELN has been detected.
Incorrect arg list in predefined routine with id *xxx*	The number of arguments to the named predefined routine does not agree with its definition.
Integer constant not in −32768..32767	A signed constant is outside the range for integer types.
Invalid constant structure definition	A record or array constant is incomplete or contains values that do not match the associated type in the structure definition.
Invalid context	A type name or expression appearing in an executable statement is used incorrectly. For example, WORDADDR of a type name or expression evokes this error response.
Invalid pointer object type	The identifier in a pointer definition is not a type identifier.
Invalid variable access	The left side of an assignment statement is not a variable or function identifier.
Left side not assignable	The left side variable is a value parameter or other constant that may not be modified.
Min exceeds max in CASE label range	A CASE element contains a label range with illegal values, *e.g. x..y*, where $x > y$. Empty CASE elements are not permitted.
Min exceeds max in subrange declaration	Subrange bounds are checked with a signed comparison.
Missing array subscript	An array reference of the form *id[ ]* has been used. Empty brackets are not permitted in array qualification.
Missing function result type	A function declaration is missing the result type specifier.
Multi-declared field name with id *xxx*	A field in a record definition has already been defined.
Multi-defined name with id *xxx*	A name has been defined more than once in the current scope.
Multi-defined parameter with id *xxx*	A parameter in a routine definition is defined more than once.
Multi-defined variant label	A variant label has already been defined in the record definition.
Multiply defined CASE label on line *xxx* with ordinal value *yyy*	A duplicate CASE label value has been defined in a CASE statement.

Table B.5 Semantic phase messages, Category 3 errors (continued)

Error Message	Explanation
Name already used in current scope with id *xxx*	A name that is being redefined in the current scope has also been used in another declaration in the same scope. A name must have only one defining point in each scope.
Named WITH type size < > variable size	The recasted type is not the same size as the type of the variable in a type-recasted WITH statement. For example, in the statement WITH x:t = y DO ..., the size(t) < > size(typeof(y)).
Name is referenced while being defined with id *xxx*	A name may not be used in its own definition, except as part of a pointer definition.
Operand is not byte addressable	The argument to BYTEADDR is a bit field or bit array address.
Operand not compatible with set base type	The first operand to an IN operator is not compatible with the set base type of the second operand.
Operands are not compatible in arith exp	The operands for an arithmetic operator must be arithmetic.
Operand(s) must be Boolean	The operands for a Boolean operator must be Boolean.
Operand(s) not compatible in constant exp	A compatibility error was encountered while evaluating a constant expression.
Operands not compatible for assignment	The left side variable is not assignment compatible with the right side expression.
Operands not compatible in relational exp	The operands for a relational operator are not compatible.
Result type may not be file	The result type of a function may not be a file type, or a type containing a file component.
Routine does not match FORWARD def	A repeated argument list of a FORWARD declared routine does not match the initial FORWARD declaration.
Routine redefinitions do not match	A repeated EXTERNAL or ENTRY routine definition is not equivalent to the existing definition.
Set base type must be ordinal	Only ordinal types may be used in set declarations.
Set bounds not in 0 .. 255	The range of the set base type is outside the compiler limits.
Set constants must be typed	Anonymous set constants may not be used in constant definitions.
Set operands are not compatible	The operands for a set operator must have compatible base types.
Standard file has not been defined with id *xxx*	A default reference to INPUT or OUTPUT is specified, but the referenced file has not been declared in the program heading.
Strict inequality not allowed on sets	The relational operators '<' and '>' are not defined for set operands.
String file buffer size exceeds 9995	A file of string has been declared with a string type that has a maximum length greater than 9995.
String length not in 1 .. 32767	String lengths may not be 0 or negative.
Structured constant must be array or set	The identifier in a structured constant definition of the form *'id[...]'* must be an array or set type.
Structured constant must be record	The identifier in a structured constant definition of the form *'id(...)'* must be a record type.
Subrange bounds not compatible	The expression types of the *min* and *max* bounds in a subrange definition are not compatible.
Too many or too few arguments	The number of arguments in a routine reference does not agree with the number of parameters in the routine declaration.
Type must be ordinal	Ordinal types are integer, whole, CHAR, Boolean, subrange, and enumeration. Certain type definitions require ordinal types, for example, the index type of an array declaration.

Table B.5 Semantic phase messages, Category 3 errors (continued)

Error Message	Explanation
Type name expected	An identifier is required to be a type name.
Type of bit qualifier not ordinal	Only ordinal types may have a bit qualifier.
Type of file must be text	The file argument to a READLN, WRITELN, or EOLN routine must be a text file.
Type of expression is not Boolean	The conditional expression in an IF, WHILE, or REPEAT statement must have a Boolean value.
Type of operands must be integer or whole	For example, in certain arithmetic expressions.
Type of variable is not array	A non-array variable is qualified with an array subscript.
Type of variable is not pointer	A non-pointer variable is qualified with a dereference operation.
Type of variable is not record	A non-record variable is qualified with a field selector.
Type of variable is not string	A non-string variable is qualified with a string character or string piece expression.
Undefined field name with id *xxx*	The field name in a record qualification is not defined.
Undefined name reference with id *xxx*	Reference to a name that has not been defined. This error is suppressed on subsequent references.
Unresolved FORWARD declaration	A FORWARD declared routine has not been defined in the scope of the FORWARD declaration.
Unresolved pointer type	An object type specified in a pointer type definition has not been defined in the current scope. The object type must be defined in a type declaration section that precedes any variable or routine declarations.
Variant label not compatible with tag type	The type of a variant label constant is not compatible with the tag type.
Variant label outside tag type bounds	A variant label value is outside the range of tag type values.
Variant tag range must be in 0 ..127	The range of variant tag values is outside the compiler limits.
While evaluating predefined function	A domain error in a constant argument to a predefined function has been detected.
Width specifier must be integer or whole	The type of a width specifier must be integer or whole.
WITH variable must be record type	A WITH variable must specify a record except in a type-recasted WITH.
ZREL declarations must be at global level	Only global variables may be allocated in the ZREL partition.

Table B.5 Semantic phase messages, Category 3 errors (continued)

Error Message	Explanation
FOR/WITH statement nesting exceeds 15	Compiler limit on nesting of FOR and WITH statements has been exceeded.
Pseudo temp overflow in exp	The maximum number of compiler-generated temporaries has been exceeded. This error is generated in very complex expressions and may be prevented by splitting a long expression into two or more simpler expressions.
Record declaration nesting > 15 levels	Compiler limit on nesting of record definitions has been exceeded.
Record variant nesting exceeds 5 levels	Compiler limit on nesting of record variants has been exceeded.
Routine declaration nesting > 8 levels	Compiler limit on nesting of routine definitions has been exceeded.

Table B.6 Semantic phase messages, Category 4 errors

Code Generation Errors

Error Message	Explanation
NREL code size exceeds 32K	The number of words for the module code exceeds 32767.
Unshared data size exceeds 32K	The number of words for the module data exceeds 32767.
ZREL data size exceeds 206 words	The number of words for the module ZREL data exceeds 206. ZREL data only occupy one page, and the first 50 words are reserved for the system.

Table B.7 Code generation errors

NOTE: *Some runtime routines use a small amount of ZREL, so that it is possible to compile programs which will give Binder warnings due to ZREL overflow.*

Calling and Interface Conventions

C

This appendix describes how SP/Pascal uses the accumulators and stack in passing parameters to and from routines. You will need this information if you are writing assembly language routines that will call or be called by SP/Pascal routines. This appendix will also be useful if you are debugging a program with the System Debugger.

Activation Record Format

When an SP/Pascal routine is invoked, an area on the top of the hardware stack (called the activation record) is allocated for the routine. The activation record contains storage for local variables, temporaries needed for expression evaluation, and five special purpose locations. These locations are defined as follows:

Mnemonic	Frame Offset	Usage
—	1	Reserved for future use
—	2	Reserved for future use
S?VCT	3	Exception handler address
S?ERR	4	Returned error code
S?LVL	5	Static nesting level

The S?VCT location contains the address of the currently active, local exception block. If no exception block is active in the routine, then S?VCT is initialized to −1. The S?ERR location is used to store the value of the error condition that caused the last exception to occur. This value may be examined with the predefined function ERROR_CODE. The S?LVL location is needed for accessing non-local variables from nested routines. External routines written in other languages need not conform to this structure unless you wish to use the SP/Pascal exception-handling mechanism (described in Chapter 11).

NOTE: *If this is not a nested routine, S?LVL is not allocated and the activation record is only four words long.*

Calling Sequences

SP/Pascal supports three calling conventions:

- The primary or default convention, which is normally used for EXTERNAL routines, as well as those that are internal to a program or module.
- The ASSEMBLY calling convention, which is used for routines that are declared EXTERNAL ASSEMBLY.
- The Common Language Runtime Environment (CLRE) convention, which is used for routines that are declared EXTERNAL CLRE.

Parameters are passed in such a way that only one word is required, even if the parameter is an array or other multi-word item. In most cases, this is done by passing an address, rather than actual data. The only exception is for value parameters (those not declared as VAR). Routines will pass the actual value of such a parameter, provided it is an integer or other item that will fit into a 16-bit word. Table C.1 details the handling of various types of value parameters.

Argument Type	Passing Convention
Integer, whole, Boolean, CHAR, pointer, sets (single-word)	Actual single-word value
Real and double_real	Address of real-valued frame temporary
Sets (multiple-word)	Address of variable or temporary
Strings, arrays, records	Address of variable

Table C.1 Passing conventions for argument types

The result from a FUNCTION routine is treated as a "zero'th parameter." It is placed where the first parameter would have been, and all other parameters are moved forward.

On return from an external routine, AC3 must contain the frame pointer of the caller (saved frame pointer). All other accumulators are undefined. The state of the floating-point unit also is undefined on return from an external routine.

Default Convention

The default SP/Pascal calling convention passes arguments in registers if at all possible (filling AC2 through AC0 in that order). If more than three arguments are present, then the last three arguments are passed in the accumulators and the first $n-3$ arguments are used on the stack in order (first argument pushed first). The calling routine is responsible for adjusting the stack depth after a call if any arguments are pushed.

SP/Pascal stack layout and accumulators at point of routine entry:

Argument 0 (functions only) ← Old stack top

Argument 1

Argument 2 Program stack

.

.

.

Argument *n*-3 ← New stack top

AC0 = argument $n-2$

AC1 = argument $n-1$

AC2 = argument n

AC3 = return address

ASSEMBLY Convention

The ASSEMBLY calling sequence (detailed in *MP/Pascal Programmer's Reference,* DGC No. 069-400031) is a convention in which AC2 points to the last argument in an argument list. AC0 and AC1 are not used. This sequence is preserved to expedite the conversion of MP/Pascal programs to SP/Pascal.

NOTE: *Because the implementation of sets in MP/Pascal is not identical with SP/Pascal, you should not use sets as parameters to routines that expect the ASSEMBLY calling convention.*

CLRE Convention

The CLRE calling sequence is a set of specifications for inter-language routine interfacing. The CLRE sequence is included in SP/Pascal for future interfacing with those languages that use this convention. It is also for use with DG-supplied run-time library packages that require this convention. Additionally, the CLRE qualifier can be used with SP/Pascal entry routines to permit these routines to be called from the other CLRE languages.

Under the CLRE convention, all parameters are pushed on the stack in reverse order; *i.e.,* the first parameter is at the top of the stack. AC2 contains an address that is one less than the location of the last parameter on the stack; in other words, the contents of AC2 are equal to the value of the stack pointer before the parameters were pushed. AC0 and AC1 are unused.

Examples

Tables C.2, C.3, and C.4 compare the three calling sequences (default, assembly, and CLRE) for the following routine declarations.

```
PROCEDURE P(VAR ARG1:INTEGER; ARG2:INTEGER);

PROCEDURE Q(ARG1:SOME_RECORD; ARG2:STRING 10; ARG3,ARG4:WHOLE);

FUNCTION F(ARG:REAL):BOOLEAN;
```

Sample Call	Accumulators	Hardware Stack
P(I,J+1)	AC2 = value of J+1 AC1 = address of I AC0 = unused	Not modified
Q(R,S,N,M)	AC2 = value of M AC1 = value of N AC0 = address of S	@SP = address of R
F(X/Y)	AC2 = address of frame temporary for X/Y AC1 = address for function result AC0 = unused	Not modified

Table C.2 Default calling sequence

Sample Call	Accumulators	Hardware Stack
P(I,J+1)	AC2 = SP - 1 AC1 = unused AC0 = unused	@SP = address of I @SP-1 = value of J-1
Q(R,S,N,M)	AC2 = SP - 3 AC1 = unused AC0 = unused	@SP = address of R @SP-1 = address of S @SP-2 = value of N @SP-3 = value of M
F(X/Y)	AC2 = SP - 2 (temporary for X/Y) AC1 = unused AC0 = unused	@SP = address for function result @SP-2,SP-1 = single-precision value of X/Y

Table C.3 Assembly calling sequence

NOTE: *The ASSEMBLY calling sequence pushes arguments on the stack in reverse order. All arguments are accessed by positive offsets from AC2. In SP/Pascal, real values are passed as two-word (single-precision) arguments.*

Sample Call	Accumulators	Hardware Stack
P(I,J+1)	AC2 = SP - 2 AC1 = unused AC0 = unused	@SP = address of I @SP-1 = address of frame temporary for J+1
Q(R,S,N,M)	AC2 = SP - 4 AC1 = unused AC0 = unused	@SP = address of R @SP-1 = address of S @SP-2 = address of temporary for value of N @SP-3 = address of temporary for value of M
F(X/Y)	AC2 = SP - 2 AC1 = unused AC0 = unused	@SP = address for function result @SP-1 = address of temporary for X/Y

Table C.4 CLRE calling sequence

Cross Development under AOS

D

SP/Pascal programs can be compiled and executed under the AOS system. The operating procedures for the compiler are identical to those procedures given for MP/AOS in Chapter 13. To execute the compiler, an AOS compilation macro SPC is supplied with the SP/Pascal release.

Binding under AOS

To prepare an SP/Pascal program that can be executed under the AOS system, bind the program using the CLI bind macro, SPCLINK.CLI, which is supplied with the SP/Pascal compiler. The macro format is as follows:

SPCLINK[/*switches] objectfile [file-1 [file-2 [...file-n]]]*

The program name assigned to the binder output is the same as that of the first object module *objectfile* in the macro command line. The binder provides the .PR extension to the name.

Objectfile must be an object module file. *File-1* through *file-n* can be either object file names or can be the names of the user libraries containing the necessary object files. All user libraries and compilation units required by the program must be specified in the binder macro. For example:

```
SPCLINK MYPROG
SPCLINK MYPROG MOD1 MOD2
SPCLINK MYPROG MYMODLIB
SPCLINK PROG2 !* OVY_ONE ! OVY_TWO *!
SPCLINK ROOT !* MOD-A MOD-B ! MOD-C *!
```

The SPCLINK macro produces a binder listing containing the program load map and any binder error messages. The listing file is placed in the working directory and given the same name as *objectfile* along with a .BLS extension.

To use multitasking with an SP/Pascal program, use the /TASKS= switch when binding the program. Task control blocks (TCB's) are not allocated unless you specify the /TASKS=*number-of-tasks* switch. For example:

```
SPCLINK/TASKS=5 PROG1 TASK_MOD
```

The SP/Pascal-specific /STD binder macro switch allows you to request run-time detection of certain non-standard SP/Pascal extended features, such as reading and writing to the same file, or extended syntax for real numbers. (The /STD switch changes the setting of P?STD from the default (0) to 1. P?STD is described further in Chapter 7.) When you omit the /STD switch, the SP/Pascal extensions are permitted, and the SP/Pascal run-time routines do not flag instances of these extended features. For example:

```
SPCLINK/STD TESTPROG
```

would detect and flag uses of non-standard extensions.

The following switches also can be used with the SPCLINK macro:

- /MODSYM for a module by module symbol list,
- /ALPHA to include a list of global symbols sorted in alphabetical order,
- /NUMERIC for a list of global symbols sorted by numeric value.

For details on these and additional switches, refer to *AOS Link User's Manual,* (DGC No. 093-000254).

AOS-MP/AOS Differences

The AOS environment also affects certain program-related actions. Specifically, the precise timing of task scheduling and the requirements for initialization of the hardware floating-point unit are different under AOS. In a multitasked program, the user is responsible for initializing the floating-point unit in each task that uses it. To perform the initialization, the user should call the DG-supplied routine IFPU defined in IFPU.PINC. The handling of default files also is different under AOS. The predefined SP/Pascal file variable INPUT is opened on the AOS generic file '@INPUT', and the predefined SP/Pascal file variable OUTPUT is opened on the AOS generic file '@OUTPUT'.

MP/AOS System Call Translator

The SP/Pascal program built by the SPCLINK macro contains the MP/AOS System Call Translator. This translator emulates both MP/AOS and MP/OS system calls under AOS. Since the AOS environment is somewhat different than the MP/AOS or MP/OS environments, there are some minor differences in the actions of some of the system calls. Therefore, only a subset of MP/AOS and MP/OS system calls can be executed. The remainder of this appendix describes the differences in system calls.

Translated System Calls

Under the AOS environment, the system call translator converts AOS error codes into their MP/AOS counterparts. If the error code has no MP/AOS counterpart, your program receives the AOS code. You should be aware that a different error could be returned by the call translator than by MP/AOS.

Program Management Calls

The ?RETURN call with the BK option uses the AOS convention for the break file name:

?pid.time.BRK

where *pid* is your process I.D. and *time* is the current time of day. Also, a program terminated with a ?RETURN BK cannot pass a message to another program.

The ?EXEC call, where the complete pathname given to ?EXEC is CLI.PR, invokes AOS CLI with the appropriate message format. The user's message must be in the MP/AOS CLI format with the CLI as the zero'th argument.

NOTE: *Attempting to use ?EXEC on any program named CLI.PR other than AOS CLI.PR causes the program to fail on start-up.*

Due to AOS restrictions, some or all the messages passed by ?EXEC are capitalized.

A program activated with ?EXEC fails if it attempts to use a channel exclusively opened by another program.

The ?BOOT call does not perform a bootstrap; it attempts to return to the user's CLI. The message gives the reason for returning and the name of the specified bootstrap device or file in one of two forms:

Emulator shutdown

Emulator booting: <file name>

Under the call translator, ?ERMSG reads from MERMES. Therefore, MERMES must be locatable through the searchlist.

The system call translator's handling of overlays does not use a channel internally. You should also be aware that overlay node sizing is larger under the call translator. Therefore, programs with many nodes that fit under MP/AOS might not fit under the call translator.

Multitasking

The call translator limits a program to 30 tasks.

The allocation scheme for task identifiers used under the call translator is unrelated to that used under MP/AOS.

Avoid running tasks at priority zero (0), since such tasks compete with call translator tasks running at priority zero and could cause them to function incorrectly.

Task scheduling under the system call translator is different than under MP/AOS. Therefore, care should be taken to force scheduling of tasks through ?PEND and ?DRSCH calls, and through setting different priorities.

The ?PEND call for a CTRL-C CTRL-A works only for @TTIO.

File Management Calls

AOS supports file type numbers which are somewhat different from MP/AOS file types. The system call translator converts MP/AOS file types to their AOS counterparts when you create files It also converts AOS file types to their MP/AOS counterparts when you open files created by AOS programs. The correspondences between file types are summarized in Tables D.1 and D.2.

MP/AOS	AOS	Meaning
?DDIR	?FDIR	Directory
?DPRG	?FPRG	Program file
?DUDF	?FUDF	User data file
?DTXT	?FTXT	Text file

Table D.1 Conversion of MP/AOS file types when creating files under AOS

NOTE: *All file types not mentioned in Table D.1 are converted to ?FUDF.*

AOS	MP/AOS	Meaning
?FLPU	?DLPT	Line printer
?FGFN	?DCHR	AOS generic file
?FPRG	?DPRG	Program file
?FDIR	?DDIR	Directory
?FCPD	?DDIR	AOS control point directory
?FTXT	?DTXT	Text file
?FUDF	?DUDF	User data file
?FDKU	?DDVC	Disk unit

Table D.2 Conversion of AOS file types when opening files with MP/AOS programs

NOTE: *All file types not mentioned in Table D.2 are converted to ?DUDF.*

Under AOS the system call ?RENAME is not supported across directories.

When you use the ?OPEN call with a CR (create) option, the system call translator does not use the element size supplied with the call. Instead, the default element size one is used.

No more than three non-pended calls can run concurrently. You must specify additional TCB's (task control blocks).

Under the call translator, the searchlist has a maximum length of 511 characters, and no error is produced if the searchlist contains more than five pathnames.

Due to AOS restrictions, the call translator allows no more than eight directory tree levels.

The call ?FSTAT CH on a disk unit where the channel is exclusively open returns an incorrect file length.

File attributes also are handled differently on the two systems. The MP/AOS system call translator intercepts the references in your program to all file attributes except permanence and translates them into elements on the *access control list* (ACL) of the file. The ACL is

a file protection feature provided by AOS, which is described fully in the *AOS Programmer's Manual* (DGC No. 093-000120).

The correspondences between attributes and access types are summarized in Table D.3.

NOTE: *There is a reversal in polarity between the two systems: setting the MP/AOS read-protect attribute for a file means that it cannot be read; i.e, setting this attribute has the same effect as removing the AOS read-access privilege (R). Conversely, setting the AOS (read access privilege) for a file means that it can be read. (This conversion is handled by the translator.)*

MP/AOS Attributes	AOS Access Privileges
Read protection: may not be read	R : read access
Write protection: may not be written	W : write access
Attribute protection: may not change attributes	O : owner access

Table D.3 Reversal in polarity between MP/AOS attributes and AOS access privileges

The permanence attribute is handled identically under the AOS and MP/AOS systems.

I/O Device Management Calls

The AOS and MP/AOS systems have different formats for device characteristics. The ?GCHAR and ?SCHAR calls perform the conversion between characteristics, so that the difference is transparent to your program. Note, however, that if you use the HC option with ?GCHAR or ?SCHAR, the following occurs: the ?GCHAR call returns a zero, and the ?SCHAR call executes successfully but ignores the HC option.

Special caution also is in order when you use the ?GCHAR and ?SCHAR calls with @TTI and @TTO. (Refer to description following Table D.5.) ?SCHAR with the LL option and a line length set to −1 allows only 256 characters per line. Under AOS, the maximum line length is 256 characters.

Table D.4 summarizes the correspondences between device characteristics for AOS and MP/AOS.

MP/AOS Name	AOS Name
?CBIN	Supported for @TTI, @TTO, @TTI1, @TTO1, @LPT
?CECH	?CEOC
?CEMM	?CEOS
?CESC	?CESC
?CICC	Not supported
?CLST	Not supported
?CNAS	?CNAS
?CNED	Supported for @TTI, @TTO, @TTI1, @TTO1, @LPT
?CST	?CST
?CUCO	?CUCO
?C605	?C605
?C8BT	Supported for @TTI, @TTO, @TTI1, @TTO1, @LPT

Table D.4 Correspondences between device characteristics

?DSTAT does not store information in the packet. It simply validates its input parameters and then exits.

The following calls are unimplemented and produce an error return with code ERISC (illegal system call) when attempted:

?ALMP	?GTPID	?PROC
?ASEG	?IDEF	?PURGE
?BLOCK	?IFPU	?RCV
?CLEAR	?IRMV	?RCVA
?CSEG	?IUNPEND	?RDMEM
?DCLR	?IPEND	?RDST
?DEMP	?IXIT	?REPLY
?DSEG	?KILL	?SD.R
?DHIS	?LDEF	?SEND
?DISMOUNT	?LKUP	?SINFO
?DSBL	?LRMV	?STMP
?EHIS	?LXIT	?TPORT
?EINFO	?MOUNT	?UNBLOCK
?ENBL	?MSEG	?WRMEM
?EQT	?OBITS	?WRST
?GIDS		?WSIG
?GMRP		

There is mapping between the MP/AOS system and AOS for the various devices, shown in Table D.5. The system call translator recognizes the MP/AOS device name and converts it to its AOS counterpart.

MP/AOS Device Name	AOS Device Name
@TTI	@Input (or @null if the program is batched)
@TTO	@Output
@TTI1	@Data
@TTO1	@List

Table D.5 Device name mapping

Because the characteristics for @TTI and @TTO both map into the generic AOS device @CONSOLE, you should exercise caution when using the calls ?GCHAR and SCHAR.

Setting any of the characteristics usable by both input and output on @TTI also affects @TTO, and vice versa. In particular, the following sequence causes problems.

```
?GCHAR @TTO
?SCHAR @TTO ; new characteristics
?GCHAR @TTI
?SCHAR @TTI ; new characteristics
.
.
.
?SCHAR @TTO ; restore characteristics
?SCHAR @TTI ; restore characteristics
```

This sequence does not restore characteristics properly, since the SCHAR @TTO call changes some of the characteristics of @TTI before the GCHAR @TTI saves them.

Instead, use this sequence for both @TTI and @TTO:

```
?GCHAR @TTO
?GCHAR @TTI
?SCHAR @TTO
?SCHAR @TTI
.
.
.
?SCHAR @TTO ; restore characteristics
?SCHAR @TTI ; restore characteristics
```

Transporting AOS programs to MP/AOS

Programs that have been developed to run under AOS using the system call translator can be moved to the MP/AOS system with no modifications except for a rebind. To rebind a program developed under AOS for execution under MP/AOS, use the SPCBIND macro described in Chapter 13. Because the system call translator is bound in with the AOS version of the program, the total program size is different for the two systems. Some programs can be executed under MP/AOS, but not under AOS, because of size considerations.

Assembly Language Parameters Specific to SP/Pascal

E

This appendix contains a listing of SPASC.SR, the assembly language parameter file for SP/Pascal. This file contains symbol definitions that will be useful to you if you are writing assembly language routines that will interface with SP/Pascal routines. The file defines the formats of internal structures such as string descriptors and task control blocks. Definitions also appear for all error codes produced by SP/Pascal routines.

If you will be using this file frequently, you may wish to add it to your assembler's permanent symbol table file. For information on how to do this, see *MP/AOS Macroassembler, Binder, and Library Utilities* (DGC No. 069-400210).

```
0001 SPASC    MP/OS ASSEMBLER REV 99.99

                        .title  spasc   ; PS file for SP/Pascal
02
03
04
05              ; ---------------- The predefined stack layout ----------------------
06              ; offsets 1,2 are reserved for future use
07
08      000001 .dusr   s?vct   = 1             ; exception vector (-1 => none)
09      000002 .dusr   s?err   = 2             ; err code in exception routine
10      000003 .dusr   s?lvl   = 3     ; procedure level (-1 => end of list)


11              ; --------------- The Pascal file descriptor layout ---------------
12
13
14      000004 .dusr   vfhsz   = 4             ; size of header for File of String
15

16      000000 .dusr   elem=   0               ; element size [bytes]
17      000001 .dusr   chan=   elem+1          ; file channel number
18      000002 .dusr   stus=   chan+1          ; file status word
19      000003 .dusr   link=   stus+1          ; ptr linking open files
20      000004 .dusr   strt=   link+1          ; byte addr of start of buffer
21      000005 .dusr   curr=   strt+1          ; current buffer byte address
22      000006 .dusr   endp=   curr+1          ; ending buffer byte address
23      000007 .dusr   ecnt=   endp+1          ; elements per buffer
24      000010 .dusr   fpshi=  ecnt+1          ; hi file position
25      000011 .dusr   fpslo=  fpshi+1         ; low file position
26
27      000012 .dusr   frcsz   = fpslo-elem+1  ; size of file record
28
29              ; Definitions of the flag bit values from the start
30              ; of the file record (in offset STUS) for BTO/BTZ use
31
32      000040 .dusr   stbtb   = stus*16.      ; base
33
34      000040 .dusr   opnbt   = stbtb         ; true if file open
35      000041 .dusr   rwfbt   = stbtb+1       ; true if write
36      000042 .dusr   tmpbt   = stbtb+2       ; true if temp file
37      000043 .dusr   bufbt   = stbtb+3       ; true if multi-buffered
38      000044 .dusr   clobt   = stbtb+4       ; true if doing close
39      000045 .dusr   ntxbt   = stbtb+5       ; non-text I/O bit,
                                                 forces dynamic read
40      000046 .dusr   modbt   = stbtb+6       ; true if buffer modified
```

```
41      000057 .dusr   eofbt   = stbtb+15.      ; true if eof
42
43
44            ; ---------------------------------------
45            ; A SP/Pascal string descriptor looks like this
46
47      000000 .dusr   stlen   = 0      ; length
48      000001 .dusr   stmax   = 1      ; maximum length
49      000002 .dusr   stdat   = 2      ; where the data begins
50
51              ; Note:  if the contents of offset stmax are < 0, then the
52              ;        following template is used (a substring descriptor)
53
54      000000 .dusr   stind   = 0      ; index
55      000001 .dusr   stpln   = 1      ; length of string piece
56      000002 .dusr   stadr   = 2      ; location of string descriptor
57
58                     .eject
59
```

```
0002 SPASC
01
02
03              ; ----------------------------------
04              ;  SP/Pascal task block offsets.
05              ;  The task block is pointed to by the
06              ;  ZREL location P?TTB, allocated at
07              ;  initialization, and contains ?NTASK entries
08              ;  Each entry has the following layout:
09
10       000000 .dusr   pt?id   = 0     ; Task ID, zero => vacant slot
11       000001 .dusr   pt?lc   = 1     ; Lock chain, used to release locks
12       000002 .dusr   pt?sb   = 2     ; Base of stack for this task
13       000003 .dusr   pt?sl   = 3     ; Limit of stack (used to get ID
                                                    w/o system call)
14
15
16       000004 .dusr   pt?ln   = 4     ; Length of per-task block
17
18
19
20              ;  --  the structure of a SP/Pascal task lock
21
22       000000 .dusr   p?lln   = 0             ; link
23       000001 .dusr   p?lpc   = p?lln+1       ; pend code
24       000002 .dusr   p?lfl   = p?lpc+1       ; flag (0 - unlocked,
                                                  locker_id - locked)
25
26
27                      .eject
```

```
0003 SPASC
01
02              ;
03              ; ----------------------------------
04              ;  SP/Pascal error codes ....
05              ;  Note:  Group is same as MP group as are
06              ;         the first 'n' error codes
07
08
09       043000 .dusr   ernum = 43000   ; pascal group = 43
10
11       043001 .dusr   epimp=  ernum+1 ; implementation error
12       043002 .dusr   epsrg=  epimp+1 ; subscript range error
13       043003 .dusr   epptr=  epsrg+1 ; invalid pointer reference
14       043004 .dusr   epvnt=  epptr+1 ; invalid variant tag
15       043005 .dusr   eprge=  epvnt+1 ; range error
16       043006 .dusr   ephep=  eprge+1 ; heaplimit exceeded
17       043007 .dusr   epiov=  ephep+1 ; integer overflow/underflow
18       043010 .dusr   epset=  epiov+1 ; invalid set element
19       043011 .dusr   epcas=  epset+1 ; invalid case selector
20       043012 .dusr   epudr=  epcas+1 ; undefined routine
21       043013 .dusr   epstk=  epudr+1 ; stacklimit exceeded
22       043014 .dusr   epsgl=  epstk+1 ; invalid string length
23       043015 .dusr   epusg=  epsgl+1 ; uninitialized string
24       043016 .dusr   epspl=  epusg+1 ; string parameter too long
25       043017 .dusr   epsgi=  epspl+1 ; invalid string index
26       043020 .dusr   epsov=  epsgi+1 ; string overflow
27       043021 .dusr   epfpt=  epsov+1 ; real overflow/underflow
28       043022 .dusr   epdiv=  epfpt+1 ; division by zero
29       043023 .dusr   epopn=  epdiv+1 ; file not opened with reset/rewrite
30       043024 .dusr   epord=  epopn+1 ; file not opened for reading
31       043025 .dusr   epowt=  epord+1 ; file not opened for writing
32       043026 .dusr   epcvt=  epowt+1 ; input conversion error
33       043027 .dusr   eprsz=  epcvt+1 ; record size does not match file type
34       043030 .dusr   epbsk=  eprsz+1 ; bad stack (display links increasing)
35       043031 .dusr   episs=  epbsk+1 ; illegal to supply substring here
36       043032 .dusr   epwid=  episs+1 ; invalid width specification
37       043033 .dusr   eplok=  epwid+1 ; lock/unlock error
38       043034 .dusr   ephso=  eplok+1 ; hardware stacklimit exceeded
39       043035 .dusr   epfrk=  ephso+1 ; failure in fork
40       043036 .dusr   epfln=  epfrk+1 ; LN of a non-positive number
41       043037 .dusr   epexp=  epfln+1 ; EXP argument out of range
```

```
42      043040 .dusr   epsqr=  epexp+1 ; negative SQRT argument
43      043041 .dusr   epfov=  epsqr+1 ; Floating point overflow trap
44      043042 .dusr   epfuf=  epfov+1 ; Floating point underflow trap
45      043043 .dusr   epfdz=  epfuf+1 ; Floating point zero divide trap
46      043044 .dusr   epmov=  epfdz+1 ; Floating point mantissa overflow trap
47      043045 .dusr   eppfr=  epmov+1 ; Parameter to FORK not local variable
48      043046 .dusr   epvfm=  eppfr+1 ; Invalid format for File of String
49

**00000 TOTAL ERRORS, 00000 PASS 1 ERRORS
```

SP/Pascal Formal Syntax

F

The formal syntax for SP/Pascal is presented in a slightly modified BNF (Backus-Naur form), shown in Table F.1.

Characters	Meaning
" "	Encloses a terminal (non-terminals are not enclosed in " ")
→	Is replaced by
{ }	0 or >0 repetitions of the enclosed constructs
[ ]	0 or 1 repetitions of the enclosed constructs allowed
|	Alternation
( )	Grouping
. (period)	End of a production

Table F.1 Modified BNF

For ease of reference, the productions for identifier, unsigned number, etc., are given; these are intended for reference only since the rules for white space and comments are not presented here, (see Chapter 2).

identifier →	letter-char { letter-char | digit }
letter-char →	'A' | 'B' |... | 'Z' | 'a' | 'b' | ... | z' | '_' | '$' | '?' .
octal-digit →	'0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' .
based_int →	hex_digit { hex_digit }'r' unsigned_int.
hex_letter →	'A' | 'B' |... | 'F' | 'a' | 'b' |... | 'f'.
hex_digit →	hex_letter | digit.
digit →	octal-digit | '8' | '9' | '_' .
unsigned-int →	digit { digit }.
unsigned-real →	unsigned-int '.' unsigned-int | unsigned-int ['.' unsigned-int] scale-factor .
unsigned-num →	unsigned-int | unsigned-real | based_int.
scale-factor →	('E' | 'e') sign unsigned-int .
sign →	'+' | '−' .
character-lit →	" ' " character " ' ".
string-lit →	" ' " { character } " ' " .
character →	"<" $digit_8$ {$digit_8$} > | " ' ' " | any printing character except'.

Table F.2 BNF syntax of SP/Pascal building blocks

The rules above describe the general building blocks of the language. Note that white space is not allowed in the definitions in Table F.2.

In the following rules, case indifference is assumed, and the spacing and comment rules given in Chapter 2 apply. At times descriptive prefixes are attached to previously declared non-terminals. For example, constant_identifiers, type_identifiers, and function identifiers are all produced by the same rules as given for identifiers.

**For the string_type* production, remember that *string* is not a reserved word and *constant* must be either an unsigned integer or an identifier which denotes an unsigned integer.

compilation_unit →	proghead block '.' | modhead decl_list.
modhead →	['OVERLAY'] 'MODULE' identifier ';'.
proghead →	'PROGRAM' identifier ['(' id_list ')'] ';'.
block →	[decl_list] compound_stmt.
id_list →	identifier { ',' identifier }.
constant →	exp.
type →	simple_type ['BIT' constant] | pointer_type | string_type | ['PACKED'] structured_type.
pointer_type →	('@' | '^') type_identifier.
string_type* →	'STRING' constant.
simple_type →	type_identifier | enumeration_type | subrange_type.
enumeration_type →	identifier ',' identifier { ',' identifier }.
subrange_type →	constant '..' constant.
structured_type →	'ARRAY' '[' simple_type_list ']' 'OF' type | 'RECORD' [field_list] 'END' | 'FILE' 'OF' type | 'SET' 'OF' simple_type.

simple_type_list → simple_type { ',' simple_type }.

field_list → (fixed_part [';' variant_part] | variant_part) [';'].

fixed_part → record_section { ';' record_section }.

record_section → id_list ':' type.

variant_part → 'CASE' tag_part 'OF' variant { ';' variant }.

tag_part → [tag_identifier ':'] type_identifier.

variant → constant_list ':' '(' [field_list] ')'.

constant_list → constant { ',' constant }.

variable → identifier | variable var_qual | variable '<<' exp [':' exp] '>>'.

var_qual → ('@' | '^') | '.' field_identifier | '[' index_member_exp { ',' index_member_exp } ']'.

index_member_exp → exp | exp '..' exp.

exp → simple_exp [relation_op simple_exp].

simple_exp → [sign] term { add_op term }.

term → factor { mul_op factor }.

factor → unsigned_num | set_constructor | vfactor | 'NOT' factor | '(' exp ')' | 'NIL'.

set_constructor → '[' [index_member_exp { ',' index_member_exp }]") ']'.

vfactor → vfact | vfactor var_qual | vfactor '<<' exp [':' exp] '>>'.

vfact → identifier | string_lit | function_invocation.

function_invocation → function_identifier '(' exp { ',' exp } ')'.

relation_op → '<' | '>' | '=' | '<>' | '<=' | '=<' | '>=' | '=>' | 'IN'.

add_op → '+' | '-' | 'OR'.

mul_op → '*' | '/' | 'DIV' | 'MOD' | 'AND'.

stmt → [variable ':=' exp | procedure_call | compound_stmt | 'IF' exp 'THEN']_stmt ['ELSE' l_stmt] | 'CASE' exp 'OF' case_list 'END' |
'WHILE' exp 'DO']_stmt | 'REPEAT' stmt_list 'UNTIL' exp | 'FOR' identifier ':=' exp ('TO' | 'DOWNTO') exp 'DO']_stmt |
'WITH' with_list 'DO']_stmt | 'EXITLOOP' | 'RETURN' | 'ERETURN''(' exp ')' | 'GOTO' label].

l_stmt → [label ':'] stmt.

stmt_list →]_stmt { ';']_stmt }.

compound_stmt → 'BEGIN' stmt_list ['EXCEPTION' stmt_list] 'END'.

procedure_call → procedure_identifier ['(' parg { ',' parg } ')'].

parg → exp [[':' exp] ':' exp].

case_list → case_elements [';'] | case_elements ';' 'OTHERWISE' stmt_list.

case_elements → case_element { ';' case_element }.

case_element → constant_list ':' l_stmt.

with_list → with_ref { ',' with_ref }.

```
with_ref ⟶        [ identifier [ ':' type_identifier ] '=' ] variable.
decl_list ⟶       declaration { declaration }.
declaration ⟶     'LABEL' label_list ';' | 'CONST' const_decl_list
                  ';' | 'TYPE' type_decl_list ';' |
                  zqual var_decl_list ';' | rstg_qual pfhead ';'
                  fblock ';' | rstg_qual function_identifier ';' block
                  ';' | frstg_qual pfhead ';'.
label_list ⟶      label { ',' label }.
label ⟶           integer_const.
const_decl_list ⟶ identifier '=' constant { ';' identifier '='
                  constant}
type_decl_list ⟶  identifier '=' type { ';' identifier '=' type }.
zqual ⟶           [ ('EXTERNAL' | 'ENTRY') ] ['ZREL'] 'VAR'.
var_decl_list ⟶   id_list ':' type { ';' id_list ':' type}.
rstg_qual ⟶       ['ENTRY' ['CLRE']].
frstg_qual ⟶      'EXTERNAL' [ ('CLRE' | 'ASSEMBLY') ] |
                   ['ENTRY' ['CLRE']] 'FORWARD'.
pfhead ⟶          'PROCEDURE' identifier [ '(' parm_list ')'] |
                  'FUNCTION' identifier [ '(' parm_list ')' ] ':'
                  ptype.
ptype ⟶           type_identifier | string_type.
parm_list ⟶       formal_parm { ';' formal_parm }.
formal_parm       ['VAR'] id_list ':' recast_type_id { pfhead.
recast_type_id ⟶  ptype | 'RECAST' type_identifier.
fblock ⟶          [ 'FORWARD' ].
```

Differences Among Data General Pascal Compilers

G

Data General provides three Pascal compilers. MP/family computer systems use MP/Pascal or SP/Pascal; these two compilers can also run under AOS. MV/family computer systems use AOS/VS Pascal. This appendix summarizes the differences among the three compilers.

In general, SP/Pascal and AOS/VS Pascal are both supersets of MP/Pascal. Therefore, in the text we take the viewpoint that MP/Pascal is the "basic" version, and we describe those features of the larger systems that are extensions to MP/Pascal.

Reserved Words

SP/Pascal has the reserved words of MP/Pascal except STRING, as well as BIT, CLRE, ERETURN, EXCEPTION, LABEL, GOTO, and ZREL.

AOS/VS Pascal has all the reserved words of MP/Pascal. However, AOS/VS Pascal supports the ENTRY, EXITLOOP, EXTERNAL, MODULE, and RETURN keywords, although they are *not* reserved.

Data Types & Declarations

SP/Pascal requires that enumeration types have more than one identifier. For example,

```
TYPE
  E1 = (ONE, TWO, THREE); { legal }
  E2 = (ONE); { illegal }
```

Memory for set type data is allocated as words: for example; a set with 20 elements will occupy two full words (32 bits).

SP/Pascal supports constants in any radix from 2 to 16.

AOS/VS Pascal handles the PACKED attribute as described in the Pascal standard. (Under MP/Pascal and SP/Pascal, the PACKED word has no effect on data storage, although it is checked for compatibility between items.)

AOS/VS Pascal uses a different internal form for sets, integers, characters, pointers, and packed structures.

SP/Pascal supports both single- and double-precision real numbers.

Characters & Strings

SP/Pascal automatically converts items of type CHAR to type STRING 1, and vice versa.

SP/Pascal does not require that the length of a string argument be less than the maximum length of the formal parameter.

SP/Pascal considers strings to be compatible with character arrays only if the array

is declared PACKED,
has a subscript range starting with 1, and
is an array of CHAR, not a subrange such as 'A' .. 'Z'.

Both SP/Pascal and AOS/VS Pascal support the CHAR type as a full 8-bit range of values (0 to 255).

AOS/VS Pascal does not have a string type.

Routines & Parameters

SP/Pascal permits files to be RECAST parameters.

Under **both AOS/VS and SP/Pascal,** all DG-supplied external routines use the normal calling convention, *i.e.,* they should be declared EXTERNAL, not EXTERNAL ASSEMBLY.

AOS/VS Pascal does not support the RECAST and OVERLAY features.

AOS/VS Pascal passes value parameters by copying, and permits new values to be assigned by the called routine.

Files & Input/Output Operations

PS/Pascal permits the RESET, REWRITE, and FILEAPPEND procedures to have an optional third parameter for specifiying buffering.

SP/Pascal file variables have a fixed size, and buffer space is allocated on the heap when the file is opened.

Optionally, SP/Pascal programs can read and write to the same file.

SP/Pascal string files have a different record format and maximum record size (for AOS compatibility).

An SP/Pascal program does not cause an error if, in reading from a text file into a string, it reads a line that is too long to fit into the string.

Under **AOS/VS Pascal**, a REWRITE to an open file causes repositioning to the beginning of the file, and deletion of its previous contents.

Miscellaneous Features

SP/Pascal does not generate code for unreachable statements. Also, it does not generate references to external routines or variables if they are declared but not used.

SP/Pascal permits the use of NEW, DISPOSE, MARK, and RELEASE in the same program.

Some implementation limits are higher under SP/Pascal (see Appendix H).

SP/Pascal has four additional compiler switches (L, T, W, and Z).

AOS/VS Pascal does not support the following features:

- Nested comments
- Multitasking
- The memory management routines FREESPACE, MARK, and RELEASE
- The N and O compiler switches

SP/Pascal Implementation Limits

H

The maximum length of identifiers is 135 characters.

The maximum length of a string constant is 132 characters.

Enumeration types can contain up to 256 elements.

The range of numeric values for integer constants is $-32768..32767$.

The range of numeric values for whole constants is 0..65535.

The range of absolute values for real constants is approximately $0..7.2 \times 10^{+75}$.

The smallest positive real value is approximately 5.4×10^{-79}.

The range of variant tag fields is 0 .. 127.

The maximum set cardinality is 256.

The range of ordinal values for set elements is 0..255.

The maximum possible length of a string in a type declaration is 32,767 characters.

The maximum depth for nested routines is eight levels.

The maximum depth for nested record definitions is 15 levels.

The maximum depth for nested record variants is five levels.

The maximum depth for nested FOR and WITH statements is 15.

The maximum depth of nesting for include files is eight levels (seven levels plus the original include file).

Index

DG OFFICES

NORTH AMERICAN OFFICES

Alabama: Birmingham
Arizona: Phoenix, Tucson
Arkansas: Little Rock
California: Anaheim, El Segundo, Fresno, Los Angeles, Oakland, Palo Alto, Riverside, Sacramento, San Diego, San Francisco, Santa Barbara, Sunnyvale, Van Nuys
Colorado: Colorado Springs, Denver
Connecticut: North Branford, Norwalk
Florida: Ft. Lauderdale, Orlando, Tampa
Georgia: Norcross
Idaho: Boise
Iowa: Bettendorf, Des Moines
Illinois: Arlington Heights, Champaign, Chicago, Peoria, Rockford
Indiana: Indianapolis
Kentucky: Louisville
Louisiana: Baton Rouge, Metairie
Maine: Portland, Westbrook
Maryland: Baltimore
Massachusetts: Cambridge, Framingham, Southboro, Waltham, Wellesley, Westboro, West Springfield, Worcester
Michigan: Grand Rapids, Southfield
Minnesota: Richfield
Missouri: Creve Coeur, Kansas City
Mississippi: Jackson
Montana: Billings
Nebraska: Omaha
Nevada: Reno
New Hampshire: Bedford, Portsmouth
New Jersey: Cherry Hill, Somerset, Wayne
New Mexico: Albuquerque
New York: Buffalo, Lake Success, Latham, Liverpool, Melville, New York City, Rochester, White Plains
North Carolina: Charlotte, Greensboro, Greenville, Raleigh, Research Triangle Park
Ohio: Brooklyn Heights, Cincinnati, Columbus, Dayton
Oklahoma: Oklahoma City, Tulsa
Oregon: Lake Oswego
Pennsylvania: Blue Bell, Lancaster, Philadelphia, Pittsburgh
Rhode Island: Providence
South Carolina: Columbia
Tennessee: Knoxville, Memphis, Nashville
Texas: Austin, Dallas, El Paso, Ft. Worth, Houston, San Antonio
Utah: Salt Lake City
Virginia: McLean, Norfolk, Richmond, Salem
Washington: Bellevue, Richland, Spokane
West Virginia: Charleston
Wisconsin: Brookfield, Grand Chute, Madison

INTERNATIONAL OFFICES

Argentina: Buenos Aires
Australia: Adelaide, Brisbane, Hobart, Melbourne, Newcastle, Perth, Sydney
Austria: Vienna
Belgium: Brussels
Bolivia: La Paz
Brazil: Sao Paulo
Canada: Calgary, Edmonton, Montreal, Ottawa, Quebec, Toronto, Vancouver, Winnipeg
Chile: Santiago
Columbia: Bogata
Costa Rica: San Jose
Denmark: Copenhagen
Ecuador: Quito
Egypt: Cairo
Finland: Helsinki
France: Le Plessis-Robinson, Lille, Lyon, Nantes, Paris, Saint Denis, Strasbourg
Guatemala: Guatemala City
Hong Kong
India: Bombay
Indonesia: Jakarta, Pusat
Ireland: Dublin
Israel: Tel Aviv
Italy: Bologna, Florence, Milan, Padua, Rome, Tourin
Japan: Fukuoka, Hiroshima, Nagoya, Osaka, Tokyo, Tsukuba
Jordan: Amman
Korea: Seoul
Kuwait: Kuwait
Lebanon: Beirut
Malaysia: Kuala Lumpur
Mexico: Mexico City, Monterrey
Morocco: Casablanca
The Netherlands: Amsterdam, Rijswijk
New Zealand: Auckland, Wellington
Nicaragua: Managua
Nigeria: Ibadan, Lagos
Norway: Oslo
Paraguay: Asuncion
Peru: Lima
Philippine Islands: Manila
Portugal: Lisbon
Puerto Rico: Hato Rey
Saudi Arabia: Jeddah, Riyadh
Singapore
South Africa: Cape Town, Durban, Johannesburg, Pretoria
Spain: Barcelona, Bibao, Madrid
Sweden: Gothenburg, Malmo, Stockholm
Switzerland: Lausanne, Zurich
Taiwan: Taipei
Thailand: Bangkok
Turkey: Ankara
United Kingdom: Birmingham, Bristol, Glasgow, Hounslow, London, Manchester
Uruguay: Montevideo
USSR: Espoo
Venezuela: Maracaibo
West Germany: Dusseldorf, Frankfurt, Hamburg, Hannover, Munich, Nuremburg, Stuttgart

DataGeneral

Ordering Technical Publications

TIPS is the Technical Information and Publications Service—a new support system for DGC customers that makes ordering technical manuals simple and fast. Simple, because TIPS is a central supplier of literature about DGC products. And fast, because TIPS specializes in handling publications.

TIPS was designed by DG's Educational Services people to follow through on your order as soon as it's received. To offer discounts on bulk orders. To let you choose the method of shipment you prefer. And to deliver within a schedule you can live with.

How to Get in Touch with TIPS

Contact your local DGC education center for brochures, prices, and order forms. Or get in touch with a TIPS administrator directly by calling (617) 366-8911, extension 4086, or writing to

Data General Corporation
Attn: Educational Services, TIPS Administrator
MS F019
4400 Computer Drive
Westborough, MA 01580

TIPS. For the technical manuals you need, when you need them.

DGC Education Centers

Boston Education Center
Route 9
Southboro, Massachusetts 01772
(617) 485-7270

Washington, D.C. Education Center
7927 Jones Branch Drive, Suite 200
McLean, Virginia 22102
(703) 827-9666

Atlanta Education Center
6855 Jimmy Carter Boulevard, Suite 1790
Norcross, Georgia 30071
(404) 448-9224

Los Angeles Education Center
5250 West Century Boulevard
Los Angeles, California 90045
(213) 670-4011

Chicago Education Center
703 West Algonquin Road
Arlington Heights, Illinois 60005
(312) 364-3045

Technical Products Publications

Comment Form

Please help us improve our future publications by answering the questions below. Use the space provided for your comments.

Title: _____

Document No. 069-400203-00

Yes	No		
☐	☐	Is this manual easy to read?	○ You (can, cannot) find things easily. ○ Language (is, is not) appropriate. ○ Technical terms (are, are not) defined as needed. ○ Other:
		In what ways do you find this manual useful?	○ Learning to use the equipment ○ As a reference ○ As an introduction to the product ○ To instruct a class. ○ Other:
☐	☐	Do the illustrations help you?	○ Visuals (are,are not) well designed. ○ Labels and captions (are,are not) clear. ○ Other:
☐	☐	Does the manual tell you all you need to know? What additional information would you like?	
☐	☐	Is the information accurate? (If not please specify with page number and paragraph.)	

Name: _____ Title: _____

Company: _____ Division: _____

Address: _____ City: _____

State: _____ Zip: _____ Telephone: _____ Date: _____

DG-06895

CUT ALONG DOTTED LINE

DataGeneral
Data General Corporation, Westboro, Massachusetts 01580

FOLD FOLD

TAPE TAPE

FOLD FOLD

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL
FIRST CLASS PERMIT NO. 26 SOUTHBORO, MA. 01772

Postage will be paid by addressee:

Data General

ATTN: Technical Products Publications (C-138)
4400 Computer Drive
Westboro, MA 01581

Data General

users group

Installation Membership Form

Name _____ Position _____ Date _____

Company, Organization or School _____

Address _____ City _____ State _____ Zip _____

Telephone: Area Code _____ No. _____ Ext. _____

CUT ALONG DOTTED LINE

1. Account Category

- ☐ OEM
- ☐ End User
- ☐ System House
- ☐ Government
- ☐ Educational

2. Hardware

	Qty. Installed	Qty. On Order
M/600		
COMMERCIAL ECLIPSE		
SCIENTIFIC ECLIPSE		
AP/130		
CS Series		
Mapped NOVA		
Unmapped NOVA		
microNOVA		
Other (Specify) _____		
_____		
_____		

3. Software

- ☐ AOS
- ☐ DOS
- ☐ MP/OS
- ☐ RDOS
- ☐ Other

Specify _____

4. Languages

- ☐ Algol
- ☐ DG/L
- ☐ Cobol
- ☐ PASCAL
- ☐ Business BASIC
- ☐ BASIC
- ☐ Assembler
- ☐ Fortran
- ☐ RPG II
- ☐ PL/1
- ☐ Other

Specify _____

5. Mode of Operation

- ☐ Batch (Central)
- ☐ Batch (Via RJE)
- ☐ On-Line Interactive

6. Communications

- ☐ HASP
- ☐ RJE80
- ☐ RCX 70
- ☐ CAM
- ☐ XODIAC
- ☐ Other

Specify _____

7. Application Description

○ _____

8. Purchase

From whom was your machine(s) purchased?

- ☐ **Data General Corp.**
- ☐ Other

Specify _____

9. Users Group

Are you interested in joining a special interest or regional Data General Users Group?

○ _____

Data General

Data General Corporation, Westboro, Massachusetts 01580, (617) 366-8911

FOLD FOLD

TAPE TAPE

FOLD FOLD

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL
FIRST CLASS PERMIT NO. 26 SOUTHBORO, MA. 01772

Postage will be paid by addressee

Data General

ATTN: Users Group Coordinator (C-228)
4400 Computer Drive
Westboro, MA 01581

Data General

Data General Corporation, Westboro, Massachusetts 01580

069-400