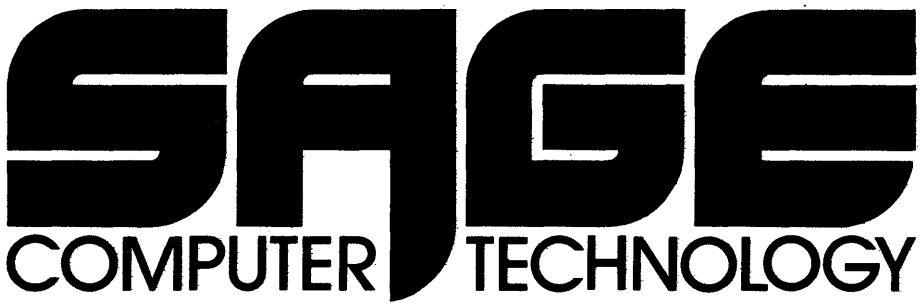


FORTRAN 77

BASIC

Internal Architecture

35 NORTH EDISON WAY, SUITE 4 • RENO, NEVADA 89502 • (702) 322-6868

SOFTECH MICROSYSTEMS

A SUBSIDIARY OF SOFTECH

UCSD p-SYSTEM

A PRODUCT FOR MINI- AND MICRO-COMPUTERS

Version IV.0

FORTRAN REFERENCE MANUAL

First edition: March 1981

SofTech Microsystems, Inc.
San Diego 1981

9494 Black Mountain Rd., San Diego, CA 92126 (714) 578-6105 TWX: 910-335-1594

DISCLAIMER:

This document and the software it describes are subject to change without notice. No warranty expressed or implied covers their use. Neither the manufacturer nor the seller is responsible or liable for any consequences of their use.

ACKNOWLEDGEMENTS:

This manual was written by Jeffrey Barth and R. Steven Glanville of Silicon Valley Software, and edited by Randy Clark and Stan Stringfellow of SofTech Microsystems. Special thanks to Bill Franks, Al Irvine and Mark Overgaard of SofTech Microsystems for contributing useful comments.

TABLE OF CONTENTS

INTRODUCTION - Overview of this Manual and Notational Conventions

0.1. Manual Overview.

This manual is intended as a user reference manual for the SofTech Microsystems FORTRAN 77 language system. SofTech Microsystems FORTRAN 77 is a dialect of FORTRAN which is closely related to the ANSI Standard FORTRAN 77 Subset language defined in ANSI X3.9-1978. Readers familiar with the ANSI standard will find a concise description of the differences between SofTech Microsystems FORTRAN 77 and the standard in Appendix A; in general, this manual does not presume that the reader is familiar with the standard.

SofTech Microsystems FORTRAN 77 runs on the UCSD P-machine architecture, which is available on a variety of host machines as a language system integrated into the UCSD Operating System. The reader is assumed to be somewhat familiar with the use of the UCSD Operating System and Text Editor, although the specifics of how to compile, link, and execute a FORTRAN program in the UCSD environment are covered in this manual. Refer to the UCSD Pascal Users' Manual for more details.

This manual is intended primarily as a reference manual for the FORTRAN system, and contains all of the information necessary to fully utilize it. The reader is assumed to have some prior knowledge of some dialect of FORTRAN, although someone familiar with another high level language should be able to learn FORTRAN from this manual. The manual is not a tutorial in the sense that it does not teach the reader, step by step, the concepts necessary to write successively more complex programs in FORTRAN; rather, each section of the manual fully explains one part of the FORTRAN language system.

The manual is organized as follows: Chapters 0, 1, and 2 are general, and describe the manual and basics necessary in order to successfully use FORTRAN in even a trivial way. Chapters 3, 4, and 5 describe the data types available in the language and how a program assigns a particular data type to an identifier or constant. Chapter 6 deals with the DATA statement, which is used for initialization of memory. Chapters 7, 8, 9, and 10 define the executable parts of programs and the meanings associated with the various executable constructs. I/O statements are presented in chapter 10, and the associated FORMAT statement and formatted I/O are described in Chapter 11. The subroutine structure of a FORTRAN compilation, including parameter passing and intrinsic (system provided) functions, is the topic of Chapter 12. Finally, Chapter 13 discusses the rather sophisticated means which exist for compiling FORTRAN subroutines separately, overlaying, and linking in subroutines which are written in other languages.

0.2. Notational Conventions.

These are the notational conventions used throughout this manual:

Upper Case and Special Characters - are written as they would be in a program.

Lower Case Letters and Words - indicate generalizations which must be replaced by actual FORTRAN syntax in a program, as described in the text. The reader may assume that once a lowercase entity is defined, it retains its meaning for the entire context of discussion.

Example of Upper and Lower Case: The format which describes editing of integers is denoted 'Iw', where w is a nonzero, unsigned integer constant. Thus, in an actual statement, a program might contain I3 or I44. The format which describes editing of reals is 'Fw.d', where d is an unsigned integer constant. In an actual statement, F7.4 or F22.0 are valid. Notice that the period, as a special character, is taken literally.

Brackets - indicate optional items.

Example of Brackets: 'A[w]' indicates that either A or A12 are valid (as a means of specifying a character format).

... - is used to indicate ellipsis. That is, the optional item preceding the three dots may appear one or more times.

Example of ...: The computed GOTO statement is described by 'GOTO (s [, s] ...) [,] i' indicating that the syntactic item denoted by s may be repeated any number of times with commas separating them.

Blanks normally have no significance in the description of FORTRAN statements. The general rules for blanks, covered in chapter 2, govern the interpretation of blanks in all contexts.

INTRODUCTION - Overview of this Manual and Notational Conventions

0.1. Manual Overview.

This manual is intended as a user reference manual for the SofTech Microsystems FORTRAN 77 language system. SofTech Microsystems FORTRAN 77 is a dialect of FORTRAN which is closely related to the ANSI Standard FORTRAN 77 Subset language defined in ANSI X3.9-1978. Readers familiar with the ANSI standard will find a concise description of the differences between SofTech Microsystems FORTRAN 77 and the standard in Appendix A; in general, this manual does not presume that the reader is familiar with the standard.

SofTech Microsystems FORTRAN 77 runs on the UCSD P-machine architecture, which is available on a variety of host machines as a language system integrated into the UCSD Operating System. The reader is assumed to be somewhat familiar with the use of the UCSD Operating System and Text Editor, although the specifics of how to compile, link, and execute a FORTRAN program in the UCSD environment are covered in this manual. Refer to the UCSD Pascal Users' Manual for more details.

This manual is intended primarily as a reference manual for the FORTRAN system, and contains all of the information necessary to fully utilize it. The reader is assumed to have some prior knowledge of some dialect of FORTRAN, although someone familiar with another high level language should be able to learn FORTRAN from this manual. The manual is not a tutorial in the sense that it does not teach the reader, step by step, the concepts necessary to write successively more complex programs in FORTRAN; rather, each section of the manual fully explains one part of the FORTRAN language system.

The manual is organized as follows: Chapters 0, 1, and 2 are general, and describe the manual and basics necessary in order to successfully use FORTRAN in even a trivial way. Chapters 3, 4, and 5 describe the data types available in the language and how a program assigns a particular data type to an identifier or constant. Chapter 6 deals with the DATA statement, which is used for initialization of memory. Chapters 7, 8, 9, and 10 define the executable parts of programs and the meanings associated with the various executable constructs. I/O statements are presented in chapter 10, and the associated FORMAT statement and formatted I/O are described in Chapter 11. The subroutine structure of a FORTRAN compilation, including parameter passing and intrinsic (system provided) functions, is the topic of Chapter 12. Finally, Chapter 13 discusses the rather sophisticated means which exist for compiling FORTRAN subroutines separately, overlaying, and linking in subroutines which are written in other languages.

0.2. Notational Conventions.

These are the notational conventions used throughout this manual:

Upper Case and Special Characters - are written as they would be in a program.

Lower Case Letters and Words - indicate generalizations which must be replaced by actual FORTRAN syntax in a program, as described in the text. The reader may assume that once a lowercase entity is defined, it retains its meaning for the entire context of discussion.

Example of Upper and Lower Case: The format which describes editing of integers is denoted 'Iw', where w is a nonzero, unsigned integer constant. Thus, in an actual statement, a program might contain I3 or I44. The format which describes editing of reals is 'Fw.d', where d is an unsigned integer constant. In an actual statement, F7.4 or F22.0 are valid. Notice that the period, as a special character, is taken literally.

Brackets - indicate optional items.

Example of Brackets: 'A[w]' indicates that either A or A12 are valid (as a means of specifying a character format).

... - is used to indicate ellipsis. That is, the optional item preceding the three dots may appear one or more times.

Example of ...: The computed GOTO statement is described by 'GOTO (s [, s] ...) [,] i' indicating that the syntactic item denoted by s may be repeated any number of times with commas separating them.

Blanks normally have no significance in the description of FORTRAN statements. The general rules for blanks, covered in chapter 2, govern the interpretation of blanks in all contexts.

CHAPTER 1

How to use SofTech Microsystems FORTRAN 77

This chapter describes how to use SofTech Microsystems FORTRAN 77. It assumes that the reader is familiar with the basic operation of the UCSD Pascal Operating System. The mechanics of preparing, compiling, linking, and executing a FORTRAN program are outlined, and an explanation of the Compiler listing file is given.

1.1. How to Compile, Link, and Execute a FORTRAN Program.

1.1.1. Compiling a FORTRAN program.

The SofTech Microsystems FORTRAN 77 Compiler is invoked as the Pascal Compiler would be invoked: by typing 'C' at the command level. The R(un command, which will compile and execute a program, may also be used. If the file has already been compiled, the R(un command will simply execute the code file. For these commands to call FORTRAN, the FORTRAN Compiler must be named SYSTEM.COMPILER. When your disk is shipped, the FORTRAN Compiler is named FORTRAN.CODE. To make it SYSTEM.COMPILER, type 'F' to enter the Filer, C(hange SYSTEM.COMPILER to PASCAL.CODE, and C(hange FORTRAN.CODE to SYSTEM.COMPILER. To start using Pascal again, reverse the renaming process.

Typing 'C' or 'R' at the command level causes the compiler to use the workfiles SYSTEM.WRK.TEXT and SYSTEM.WRK.CODE. If no workfile is present, the Operating System will prompt for the name of a .TEXT file to use.

The FORTRAN Compiler will prompt for a listing file. If a <return> is typed, no listing will be generated.

Once the prompts are all answered, the actual compilation begins. The progress of the compilation will be shown on the console by a successive display of dots. Each dot represents one line of source code.

Remember that anything which applies to the Pascal SYSTEM.COMPILER will now apply to FORTRAN. The UCSD p-System Users' Manual should be referred to for more information.

1.1.2. Providing Runtime Support.

To run any program on the UCSD p-System, some runtime support is needed. The package of routines which do this for FORTRAN is distinct from the package which does this for Pascal, and is originally shipped in the file RTUNIT.CODE. When you change FORTRAN.CODE to SYSTEM.COMPILER, you must also change

SYSTEM.LIBRARY to PASCAL.LIBRARY (or some other name you will remember), and RTUNIT.CODE to SYSTEM.LIBRARY. After this is done, you may run your FORTRAN programs.

It may be that you have placed programs of your own in SYSTEM.LIBRARY. In this case, you will be familiar with the use of the Librarian. RTUNIT.CODE should be added to the SYSTEM.LIBRARY file. The library text file facility described in Section 11.3.1 of the Users' Manual is also available to FORTRAN programmers.

1.1.3. Executing a FORTRAN program.

A compiled, linked FORTRAN program is executed in the same manner as any other user progam, i.e. by typing an 'X' at the command level, followed by the name of the file containing the linked program.

1.2. Form of Input Programs.

All input source files read by FORTRAN must be UCSD .TEXT files. This allows the Compiler to read large blocks of text from a disk file in a single operation, increasing the compile speed significantly. The simplest way to prepare .TEXT files is to use the Screen Oriented Editor. For a more precise description of the fields in a FORTRAN 77 source statement, see Chapter 2 which explains the basic structure of a FORTRAN program.

1.2.1. $INCLUDE Statement.

To facilitate the manipulation of large programs, the SofTech Microsystems Compiler has extended the FORTRAN 77 standard with an $INCLUDE Compiler directive. The format of the directive is:

```
$INCLUDE file.name
```

with the $ appearing in column 1 (see Section 2.5 for an explanation of Compiler directives in general). The meaning is to compile the contents of the file 'file.name', and insert the code into the current code file, before continuing with compilation of the current file. The included file may contain additional $INCLUDE directives, up to a maximum of five levels of files (four levels of $INCLUDE directives). It is often useful to have the description of a COMMON block kept in a single file and to include it in each subroutine that references that COMMON area, rather than making and maintaining many copies of the same source, one in each subroutine. There is no limit to the number of $INCLUDE directives that can appear in a source file.

1.3. Compiler Listing.

The Compiler listing, if requested, contains various information that may be useful to the FORTRAN programmer. The listing consists of the user's source code as read, along with line numbers, symbol tables, error messages, and optional cross-reference information.

The following is a sample listing:

```
FORTRAN Compiler IV.0  [0.0]
     0.      0 C
     1.      0 C -- Example Program #1234
     2.      0 C
     3.      0
     4.      0 $XREF
     5.      0
     6.      0         PROGRAM EX1234
     7.      0
     8.      0         INTEGER A(10,10)
     9.      0         CHARACTER*4 C
    10.      0
    11.      0         CALL INIT(A,C)
    12.      6         I = 1
    13.      9    200  A(I) = 1
***** Error number: 57 in line: 13
    14.     20         I = I + 1
    15.     26         IF (IABS(10-I) .NE. 0) GOTO 200
    16.     37
    17.     37         END

A        INTEGER        3                          8     11     13
C        CHAR*  4      103                         9     11
EX1234 PROGRAM                                     6
I        INTEGER       105                         12     13     13     14

                                                   14     15
IABS     INTRINSIC                                 15
INIT     SUBROUTINE                 2,FWD          11

    18.      0         SUBROUTINE INIT(B,D)
    19.      0         INTEGER B(10,10)
    20.      0         CHARACTER*4 D
    21.      0
    22.      0         RETURN
    23.      2         END

B        INTEGER        2*                         18     19
D        CHAR*  4       1*                         18     20
INIT     SUBROUTINE               2                18

EX1234 PROGRAM
INIT     SUBROUTINE               2,7

24 lines.    1 errors.
```

The first line indicates which version of the Compiler was used for this compilation. In the example it is version 0.0 for the UCSD Operating System version IV.0. The leftmost column of numbers is the source-line number. The next column indicates the procedure-relative instruction counter that the corresponding line of source code occupies as object code. It is only meaningful for executable statements and data statements. To the right of the instruction counter is the source statement.

Errors are indicated by a row of asterisks followed by the error number and line number, as appears in the example between lines 13 and 14. In this case it is error number 57, "Too few subscripts", indicating that there are not enough subscripts in the array reference A(1).

At the end of each routine (function, subroutine, or main program), a local symbol table is printed. This table lists all identifiers that were referenced in that program unit, along with their definition. If the $XREF Compiler directive has been given, a table of all lines containing an instance of that identifier in the current program unit is also printed. If the identifier is a variable, it is accompanied by its type and location. If the variable is a parameter, its location is followed by an asterisk, such as the variables B and D in the SUBROUTINE INIT. If the variable is in a common block, then the name of the block follows enclosed by slashes. If the identifier is not a variable, it is described appropriately. For subroutines and functions, the unit-relative procedure number is given. If it resides in a different segment, then the segment number follows. If the Compiler assumes that it will reside in the same segment, but has not appeared yet, it is listed as a forward program unit by the notation 'FWD'.

At the end of the compilation, the global symbol table is printed. It contains all global FORTRAN symbols referenced in the compilation. No cross-reference is given. The number of source lines compiled and the number of errors encountered follows. If there were any errors, then no object file is produced.

1.4. The Codefile.

The object codefile generated by the FORTRAN Compiler is compatible with the UCSD Linker and Librarian. Indeed, it is hard to tell by examining a codefile whether it was created by the FORTRAN Compiler or the Pascal Compiler. For a description of the binary format of a codefile, see the UCSD p-System Users' Manual.

CHAPTER 2

Basic Structure of a FORTRAN Program

In the most fundamental sense, a FORTRAN program is a sequence of characters which, when fed to the Compiler, are understood in various contexts as characters, identifiers, labels, constants, lines, statements, or other (possibly overlapping) syntactic substructure groupings of characters. The rules which the Compiler uses to group the character stream into certain substructures, as well as various constraints on how these substructures may be related to each other in the source program character stream will be the topic of this chapter.

2.1. Character Set.

A FORTRAN source program consists of a stream of characters, originating in a .TEXT file, consisting of:

> Letters - The 52 upper and lower case letters A through Z and a through z.
>
> Digits - 0, 1, 2, 3, 4, 5, 6, 7, 8, and 9.
>
> Special Characters - The remaining printable characters of the ASCII character set.

The letters and digits, treated as a single group, are called the alphanumeric characters. FORTRAN interprets lower case letters as upper case letters in all contexts except in character constants and Hollerith fields. Thus, the following user-defined names are all indistinguishable to the FORTRAN Compiler:

> ABCDE abcde AbCdE aBcDe

In addition to the above, actual source programs given to the FORTRAN Compiler contain certain hidden (nonprintable) control characters inserted by the Text Editor which are invisible to the user. FORTRAN uses these control characters in exactly the same way as the Text Editor, and transforms them, using the rules of UCSD .TEXT files, into the FORTRAN character set.

The collating sequence for the FORTRAN character set is the ASCII sequence.

2.2. Lines.

A FORTRAN source program may also be considered a sequence of lines, corresponding to the normal notion of line in the Text Editor. Only the first 72 characters in a line are treated as significant by the Compiler, with any trailing characters in a line ignored. Note that lines with fewer than 72 characters are possible and, if shorter than 72 columns, the Compiler does treat as significant the

length of a line (see Section 3.4, which describes character constants, for an illustration of this).

2.3. Columns.

The characters in a given line fall into columns, with the first character being in column 1, the second in column 2, etc. The column in which a character resides is significant in FORTRAN, with columns 1 through 5 being reserved for statement labels, column 6 for continuation indicators and other column conventions, columns 7 through 72 for actual statements.

2.4. Blanks.

The blank character, with the exceptions noted below, has no significance in a FORTRAN source program and may be used for the purpose of improving the readability of FORTRAN programs. The exceptions are:

Banks within string constants are significant

Blanks within Hollerith fields are significant

Blanks on Compiler directive lines are significant

A blank in column 6 is used in distinguishing initial lines from continuation lines

Blanks count in the total number of characters the Compiler processes per line and per statement

2.5. Compiler Directive Lines.

A line is treated as a Compiler directive if the $ character appears in column 1 of an input line. Compiler directives are used to transmit various kinds of information to the Compiler. A Compiler directive line may appear any place that a comment line can appear, although certain directives are restricted to appear in certain places. Blanks are significant on Compiler directive lines, and are used to delimit keywords and filenames. The set of directives is described below:

2.5.1 $INCLUDE

$INCLUDE filename

Include textually the file 'filename' at this point in the source. Nested includes are implemented to a depth of nesting of five files. Thus, for example, a program may include various files with subprograms, each of which includes various files which describe COMMON areas (which would be a depth of nesting of three files).

2.5.2 $USES

$USES ident
    [IN filename]
    [OVERLAY]

Similar to the USES command in the UCSD Pascal Compiler. The already compiled FORTRAN subroutines or Pascal procedures contained in the .CODE file 'filename', (or in the file '*SYSTEM.LIBRARY' if no file name is present), become callable from the currently compiling code. This directive must appear before the initial non-comment input line. For more details, see Chapter 13.

2.5.3 $XREF

$XREF

Produce a cross-reference listing at the end of each procedure compiled.

2.5.4 $EXT

$EXT SUBROUTINE name #params
or
$EXT [type] FUNCTION name #params

The subroutine or function called 'name' is an Assembly Language routine. The routine has exactly '#params' reference parameters.

2.6. Comment Lines.

A line is treated as a comment if any one of the following conditions are met:

A 'C' (or 'c') in column 1.

A '*' in column 1.

Line contains all blanks.

Comment lines do not effect the execution of the FORTRAN program in any way. Comment lines must be followed immediately by an initial line or another comment line. They must not be followed by a continuation line. Note that extra blank lines at the end of a FORTRAN program result in a compile time error since the system interprets them as comment lines but they are not followed by an initial line.

2.7. Statements, Initial Lines, Continuation Lines, and Labels.

Sections 2.7.1 through 2.7.4 define a FORTRAN statement in terms of the input character stream. The Compiler recognizes certain groups of input characters as complete statements according to the rules specified here. The remainder of this manual will further define the specific statements and their properties. When it is necessary to refer to specific kinds of statements here, they are simply referred to by name.

2.7.1. Labels.

A statement label is a sequence of from one to five digits. At least one digit must be nonzero. A label may be placed anywhere in columns 1 through 5 of an initial line. Blanks and leading zeros are not significant.

2.7.2. Initial Lines.

An initial line is any line which is not a comment line or a Compiler directive line and contains a blank or a 0 in column 6. The first five columns of the line must either be all blank or contain a label. With the exception of the statement following a logical IF, FORTRAN statements all begin with an initial line.

2.7.3. Continuation Lines.

A continuation line is any line which is not a comment line or a Compiler directive line and contains any character in column 6 other than a blank or a 0. The first five columns of a continuation line must be blanks. A continuation line is used to increase the amount of room to write a given statement. If it will not fit on a single initial line, it may be extended to include up to 9 continuation lines.

2.7.4. Statements.

A FORTRAN statement consists of an initial line, followed by up to 9 continuation lines. The characters of the statement are the up to 660 characters found in columns 7 through 72 of these lines. The END statement must be wholly written on an initial line, and no other statement may have an initial line which appears to be an END statement.

2.8. Main Program and Subprogram Units and Ordering of Statements within Program Units.

The FORTRAN language enforces a certain ordering among statements and lines which make make up a FORTRAN compilation. In general, a compilation consists of some number of subprograms (possibly zero), and at most one main program (see Sections on compilation units and subroutines). The various rules for ordering statements appear below.

2.8.1. Program Units - Main Program and Subprogram Program Units.

A subprogram begins with either a SUBROUTINE or a FUNCTION statement and ends with an END statement. A main program begins with a PROGRAM statement, or any other than a SUBROUTINE or FUNCTION statement, and ends with an END statement. A subprogram or the main program is referred to as a program unit.

2.8.2. Statement Ordering Within a Program Unit.

Within a program unit, whether a main program or a subprogram, statements must appear in an order consistent with the following rules:

A SUBROUTINE or FUNCTION statement, or PROGRAM statement, if present, must appear as the first statement of the program unit.

FORMAT statements may appear anywhere after the SUBROUTINE or FUNCTION statement, or PROGRAM statement if present.

All specification statements must precede all DATA statements, statement function statements, and executable statements.

All DATA statements must appear after the specification statements and precede all statement function statements and executable statements.

All statement function statements must precede all executable statements.

Within the specification statements, the IMPLICIT statement must precede all other specification statements.

These rules are illustrated in the following chart:

<table>
<tr><td rowspan="6">Comment Lines</td><td colspan="2">PROGRAM, FUNCTION, or SUBROUTINE Statement</td></tr>
<tr><td rowspan="5">FORMAT Statements</td><td>IMPLICIT Statements</td></tr>
<tr><td>Other Specification Statements</td></tr>
<tr><td>DATA Statements</td></tr>
<tr><td>Statement Function Statements</td></tr>
<tr><td>Executable Statements</td></tr>
<tr><td colspan="2">END Statement</td></tr>
</table>

Table 2.1. Order of Statements within Program Units.

The chart is to be interpreted as follows:

Classes of lines or statements above or below other classes must appear in the designated order.

Classes of lines or statements may be interspersed with other classes which appear across from one another.

2.8.3. The Final Statement of a Source Program.

When creating FORTRAN programs with the UCSD Editor, the final END statement must be entered as a complete line. That is, there must be a "return" character following the statement. Otherwise, the Compiler will not find the END statement and will issue an error message. In addition, that "return" character must be the final character in the program source file. Any further characters, even blanks, might be considered part of a subsequent subprogram by the Compiler.

CHAPTER 3

Data Types

There are four basic data types in SofTech Microsystems FORTRAN 77: integer, real, logical, and character. This chapter describes the properties of each type, the range of values for each type, and the form of constants for each type.

3.1. The Integer Type.

The integer data type consists of a subset of the integers. An integer value is an exact representation of the corresponding integer. An integer variable occupies one word (two bytes) of storage and can contain any value in the range -32768 to 32767. Integer constants consist of a sequence of one or more decimal digits preceeded by an optional arithmetic sign, + or -, and must be in range. A decimal point is not allowed in an integer constant. The following are examples of integer constants:

```
123     +123     -123     0     00000123     32767     -32768
```

3.2. The Real Type.

The real data type consists of a subset of the real numbers. A real value is normally an approximation of the real number desired. A real variable occupies two consecutive words (4 bytes) of storage. The range of real values is approximately:

```
-1.7E+38 ... -5.8E-39   0.0   5.8E-39 ... 1.7E+38     (LSI-11)
```

The actual range depends upon which computer is being used. The precision is greater than 6 decimal digits.

A basic real constant consists of an optional sign followed by an integer part, a decimal point, and a fraction part. The integer and fraction parts consist of 1 or more decimal digits, and the decimal point is a period, '.'. Either the integer part or the fraction part may be omitted, but not both. Some sample basic real constants follow:

```
-123.456        +123.456        123.456
-123.           +123.           123.
-.456           +.456           .456
```

An exponent part consists of the letter 'E' followed by an optionally signed integer constant. An exponent indicates that the value preceding it is to be multiplied by 10 to the value of the exponent part's integer. Some sample exponent parts are:

```
E12          E-12          E+12          E0
```

A real constant is either a basic real constant, a basic real constant followed by an exponent part, or an integer constant followed by an exponent part. For example:

+1.000E-2	1.E-2	1E-2
+0.01	100.0E-4	.0001E+2

all represent the same real number, one one-hundredth.

3.3. The Logical Type.

The logical data type consists of the two logical values true and false. A logical variable occupies one word (two bytes) of storage. There are only two logical constants, .TRUE. and .FALSE., representing the two corresponding logical values. The internal representation of .FALSE. is a word of all zeros, and the representation of .TRUE. is a word of all zeros but a one in the least significant bit. If a logical variable contains any other bit values, its logical meaning is undefined.

3.4. The Character Type.

The character data type consists of a sequence of ASCII characters. The length of a character value is equal to the number of characters in the sequence. The length of a particular constant or variable is fixed, and must be between 1 and 127 characters. A character variable occupies one word of storage for each two characters in the sequence, plus one word if the length is odd. Character variables are always aligned on word boundaries. The blank character is allowed in a character value and is significant.

A character constant consists of a sequence of one or more characters enclosed by a pair of apostrophes. Blank characters are allowed in character constants, and count as one character each. An apostrophe within a character constant is represented by two consecutive apostrophes with no blanks inbetween. The length of a character constant is equal to the number of characters between the apostrophes, with doubled apostrophes counting as a single apostrophe character. Some sample character constants are:

'A' ' ' 'Help!' 'A very long CHARACTER constant' ''''

Note the last example, '''', that represents a single apostrophe, '.

FORTRAN allows source lines with up to 72 columns. Shorter lines are not padded out to 72 columns, but left as input. When a character constant extends across a line boundary, its value is as if the portion of the continuation line

beginning with column 7 is juxtapositioned immediately after the last character on the initial line. Thus, the FORTRAN source:

```
200   CH = 'ABC<cr>
     X DEF'
```

(where the '<cr>' indicates a carriage return, or the end of the source line) is equivalent to:

```
200   CH = 'ABC DEF'
```

with the single space between the C and D being the equivalent to the space in column 7 of the continuation line. Very long character constants can be represented in this manner.

CHAPTER 4

FORTRAN Names

A FORTRAN name, or identifier, consists of an initial alphabetic character followed by a sequence of 0 through 5 alphanumeric characters. Blanks may appear within a FORTRAN name, but have no significance. A name is used to denote a user- or system-defined variable, array, function, subroutine, etc. Any valid sequence of characters may be used for any FORTRAN name. There are no reserved names as in other languages. Sequences of alphabetic characters used as keywords are not to be confused with FORTRAN names. The Compiler recognizes keywords by their context and in no way restricts the use of user chosen names. Thus, a program can have, for example, an array named IF, READ, or GOTO, with no error indicated by the Compiler (as long as it conforms to the rules that all arrays must obey). Using such names, however, is not a recommended practice.

4.1. Scope of FORTRAN Names.

The scope of a name is the range of statements in which that name is known, or can be referenced, within a FORTRAN program. In general, the scope of a name is either global or local, although there are several exceptions. A name can only be used in accordance with a single definition within its scope. The same name, however, can have different definitions in distinct scopes.

A name with global scope may be used in more than one program unit (a subroutine, function, or the main program) and still refer to the same entity. In fact, names with global scope can only be used in a single, consistent manner within the same program. All subroutine, function subprogram, and common names, as well as the program name, have global scope. Therefore, there cannot be a function subprogram that has the same name as a subroutine subprogram or as a common data area. Similarly, no two function subprograms in the same program can have the same name.

A name with local scope is only visible (known) within a single program unit. A name with a local scope can be used in another program unit with a different meaning, or with a similar meaning, but is in no way required to have similar meanings in a different scope. The names of variables, arrays, parameters, and statement functions all have local scope. A name with a local scope can be used in the same compilation as another item with the same name but a global scope as long as the global name is not referenced within the program unit containing the local name. Thus, a function can be named FOO, and a local variable in another program unit can be named FOO without error, as long as the program unit containing the variable FOO does not call the function FOO. The Compiler detects all scope errors, and issues an error message when they occur, so the user need not worry about undetected scope errors causing bugs in programs.

One exception to the scoping rules is the name given to common data blocks. It is possible to refer to a globally scoped common name in the same program unit that an identical locally scoped name appears. This is allowed because common

names are always enclosed in slashes, such as /NAME/, and are therefore always distinguishable from ordinary names by the Compiler.

Another exception to the scoping rules is made for parameters to statement functions. The scope of statement function parameters is limited to the single statement forming that statement function. Any other use of those names within that statement function is not allowed, and any other use outside that statement function is allowed.

4.2. Undeclared FORTRAN Names.

When a user name that has not appeared before is encountered in an executable statement, the Compiler infers from the context of its use how to classify that name. If the name is used in the context of a variable, the Compiler creates an entry into the symbol table for a variable of that name. Its type is inferred from the first letter of its name. Normally, variables beginning with the letters I, J, K, L, M, or N are considered integers, while all others are considered reals. These defaults can be overridden by an IMPLICIT statement (see Chapter 5). If an undeclared name is used in the context of a function call, a symbol table entry is created for a function of that name, with its type being inferred in the same manner as that of a variable. Similarly, a subroutine entry is created for a newly encountered name that is used as the target of a CALL statement. If an entry for such a subroutine or function name exists in the global symbol table, its attributes are coordinated with those of the newly created symbol table entry. If any inconsistencies are detected, such as a previously defined subroutine name being used as a function name, an error message is issued.

In general, one is encouraged to declare all names used within a program unit, since it helps to assure that the Compiler associates the proper definition with that name. Allowing the Compiler to use a default meaning can sometimes result in logical errors that are quite difficult to locate. Indeed, most modern programming languages require the programmer to declare all names, to avoid any such potential difficulties.

CHAPTER 5

Specification Statements

This chapter describes the specification statements in SofTech Microsystems FORTRAN 77. Specification statements are non-executable. They are used to define the attributes of user defined variable, array, and function names. There are eight kinds of specification statements:

5.1. IMPLICIT
5.2. DIMENSION
5.3. Type Statements
5.4. COMMON
5.5. EXTERNAL
5.6. INTRINSIC
5.7. SAVE
5.8. EQUIVALENCE

Specification statements must precede all executable statements in a program unit. If present, any IMPLICIT statements must precede all other specification statements in a program unit as well. Otherwise, the specification statements may appear in any order within their own group.

5.1. IMPLICIT Statement.

An IMPLICIT statement is used to define the default type for user-declared names. The form of an IMPLICIT statement is:

IMPLICIT type (a [,a]...) [,type (a [,a]...)]...

The 'type' is one of INTEGER, LOGICAL, REAL, or CHARACTER[*nnn]

The 'a' is either a single letter or a range of letters. A range of letters is indicated by the first and last letters in the range separated by a minus sign. For a range, the letters must be in alphabetical order.

The 'nnn' is the size of the character type that is to be associated with that letter or letters. It must be an unsigned integer in the range 1 to 127. If *nnn is not specified, it is assumed to be *1.

An IMPLICIT statement defines the type and size for all user-defined names that begin with the letter or letters that appear in the specification. An IMPLICIT statement applies only to the program unit in which it appears. IMPLICIT statements do not change the type of any intrinsic functions.

Implicit types can be overridden or confirmed for any specific user-name by the appearance of that name in a subsequent type statement. An explicit type in a FUNCTION statement also takes priority over an IMPLICIT statement. If the type

in question is a character type, the user-name's length is also overridden by a latter type definition.

The program unit can have more than one IMPLICIT statement, but all implicit statements must precede all other specification statements in that program unit. The same letter cannot be defined more than once in an IMPLICIT statement in the same program unit.

5.2. DIMENSION Statement.

A DIMENSION statement is used to specify that a user-name is an array. The form of a DIMENSION statement is:

DIMENSION var(dim) [,var(dim)]...

where each 'var(dim)' is an array declarator. An array declarator is of the form:

name(d [,d]...)

'name' is the user defined name of the array.

'd' is a dimension declarator.

5.2.1. Dimension Declarators.

The number of dimensions in the array is the number of dimension declarators in the array declarator. The maximum number of dimensions is three. A dimension declarator can be one of three forms:

An unsigned integer constant.

A user-name corresponding to a non-array integer formal argument.

An asterisk.

A dimension declarator specifies the upper bound of the dimension. The lower bound is always one. If a dimension declarator is an integer constant, then the array has the corresponding number of elements in that dimension. An array has a constant size if all of its dimensions are specified by integer constants. If a dimension declarator is an integer argument, then that dimension is defined to be of a size equal to the initial value of the integer argument upon entry to the subprogram unit at execution time. In such a case the array is called an adjustable-sized array. If the dimension declarator is an asterisk, the array is an assumed-sized array and the upper bound of that dimension is not specified.

All adjustable- and assumed-sized arrays must also be formal arguments to the program unit in which they appear. Additionally, an assumed-size dimension declarator may only appear as the last dimension in an array declarator.

The order of array elements in memory is column-major order. That is to say, the leftmost subscript changes most rapidly in a memory-sequential reference to all array elements.

5.2.2. Array Element Name.

The form of an array element name is:

arr(sub [,sub]...)

'arr' is the name of an array.

'sub' is a subscript expression.

A subscript expression is an integer expression used in selecting a specific element of an array. The number of subscript expressions must match the number of dimensions in the array declarator. The value of a subscript expression must be between 1 and the upper bound for the dimension it represents.

5.3. Type Statements.

Type statements are used to specify the type of user- defined names. A type statement may confirm or override the implicit type of a name. Type statements may also specify dimension information.

A user-name for a variable, array, external function, or statement function may appear in a type statement. Such an appearance defines the type of that name for the entire program unit. Within a program unit, a name may not have its type explicitly specified by a type statement more than once. A type statement may confirm the type of an intrinsic function, but is not required. The name of a subroutine or main program cannot appear in a type statement.

5.3.1. INTEGER, REAL, and LOGICAL Type Statements.

The form of an INTEGER, REAL, or LOGICAL type statement is:

type var [,var]...

'type' is one of INTEGER, REAL, or LOGICAL.

'var' is a variable name, array name, function name, or an array declarator. For a definition of an array declarator, see Section 5.2, which describes the DIMENSION statement.

5.3.2. CHARACTER Type Statement.

The form of a CHARACTER type statement is:

CHARACTER [*nnn [,]] var [*nnn] [, var [*nnn]]...

'var' is a variable name, array name, or an array declarator. For a definition of an array declarator, see 5.2. DIMENSION Statement.

'nnn' is the length in number of characters of a character variable or character array element. It must be an unsigned integer in the range 1 to 127.

The length nnn following the type name CHARACTER is the default length for any name not having its own length specified. If not present, the default length is assumed to be one. A length immediately following a variable or array overrides the default length for that item only. For an array, the length specifies the length of each element of that array.

5.4. COMMON Statement.

The COMMON statement provides a method of sharing storage between two or more program units. Such program units can share the same data without passing it as arguments. The form of the COMMON statement is:

COMMON [/ [cname] /] nlist [[,] / [cname] / nlist]...

'cname' is a common block name. If a 'cname' is omitted, then the blank common block is specified.

'nlist' is a comma seperated list of variable names, array names, and array declarators. Formal argument names and function names cannot appear in a COMMON statement.

In each COMMON statement, all variables and arrays appearing in each nlist following a common block name cname are declared to be in that common block. If the first cname is omitted, all elements appearing in the first nlist are specified to be in the blank common block.

Any common block name can appear more than once in COMMON statements in

the same program unit. All elements in all nlists for the same common block are allocated storage sequentially in that common storage area in the order that they appear.

All elements in a single common area must be either all of type CHARACTER or none of type character. Furthermore, if two program units reference the same named common containing character data, association of character variables of different length is not allowed. Two variables are said to be associated if they refer to the same actual storage.

The size of a common block is equal to the number of bytes of storage required to hold all elements in that common block. If the same named common block is referenced by several distinct program units, the size must be the same in all program units.

5.5. EXTERNAL Statement.

An EXTERNAL statement is used to identify a user-defined name as an external subroutine or function. The form of an EXTERNAL statement is:

EXTERNAL name [,name]...

'name' is the name of an external subroutine or function.

Appearance of a name in an EXTERNAL statement declares that name to be an external procedure. Statement function names cannot appear in an EXTERNAL statement. If an intrinsic function name appears in an EXTERNAL statement, then that name becomes the name of an external procedure, and the corresponding intrinsic function can no longer be called from that program unit. A user-name can only appear once in an EXTERNAL statement.

5.6. INTRINSIC Statement.

An INTRINSIC statement is used to declare that a user-name is an intrinsic function. The form of an INTRINSIC statement is:

INTRINSIC name [,name]...

'name' is an intrinsic function name.

Each user-name may only appear once in an INTRINSIC statement. If a name appears in an INTRINSIC statement, it cannot appear in an EXTERNAL statement. All names used in an INTRINSIC statement must be system-defined INTRINSIC functions. For a list of these functions, see Chapter 12.

5.7. SAVE Statement.

A SAVE statement is used to retain the definition of a common block after the return from a procedure that defines that common block. Within a subroutine or function, a common block that has been specified in a SAVE statement does not become undefined upon exit from the subroutine or function. The form of a SAVE statement is:

SAVE /name/ [,/name/]...

where: 'name' is the name of a common block.

Note: In SofTech Microsystems FORTRAN 77 all common blocks are statically allocated, so the SAVE statement is not necessary. Common blocks are never disposed on exiting a procedure. The SAVE statement is implemented here for the sake of program portability.

5.8. EQUIVALENCE Statement.

An EQUIVALENCE statement is used to specify that two or more variables or arrays are to share the same storage. If the shared variables are of different types, the EQUIVALENCE does not cause any kind of automatic type conversion. The form of an EQUIVALENCE statement is:

EQUIVALENCE (nlist) [, (nlist)]...

where: 'nlist' is a list of at least two variable names, array names, or array element names. Argument names may not appear in an EQUIVALENCE statement. Subscripts must be integer constants and must be within the bounds of the array they index.

An EQUIVALENCE statement specifies that the storage sequences of the elements that appear in the list nlist have the same first storage location. Two or more variables are said to be associated if they refer to the same actual storage. Thus, an EQUIVALENCE statement causes its list of variables to become associated. An element of type character can only be associated with another element of type character with the same length. If an array name appears in an EQUIVALENCE statement, it refers to the first element of the array.

5.8.1. Restrictions on EQUIVALENCE Statements.

An EQUIVALENCE statement cannot specify that the same storage location is to appear more than once, such as:

```
REAL R,S(10)
EQUIVALENCE (R,S(1)),(R,S(5))
```

which forces the variable R to appear in two distinct memory locations. Furthermore, an EQUIVALENCE statement cannot specify that consecutive array elements are not stored in sequential order. For example:

```
REAL R(10),S(10)
EQUIVALENCE (R(1),S(1)),(R(5),S(6))
```

is not allowed.

When EQUIVALENCE statements and COMMON statements are used together, several further restrictions apply. An EQUIVALENCE statement cannot cause storage in two different common blocks to become equivalenced. An EQUIVALENCE statement can extend a common block by adding storage elements following the common block, but not preceding the common block. For example:

```
COMMON /ABCDE/ R(10)
REAL S(10)
EQUIVALENCE (R(1),S(10))
```

is not allowed because it extends the common block by adding storage preceding the start of the block.

CHAPTER 6

DATA Statement

The DATA statement is used to assign initial values to variables. A DATA statement is a non-executable statement. If present, it must appear after all specification statements and prior to any statement function statements or executable statements. The form of a DATA statement is:

DATA nlist / clist /[[,] nlist / clist /]...

'nlist' is a list of variable, array element, or array names.

'clist' is a list of constants or constants preceded by an integer constant repeat factor and an asterisk, such as:

5*3.14159 3*'Help' 100*0

A repeat factor followed by a constant is the equivalent of a list all of constants of that constant's value repeated a number of times equal to the repeat constant.

There must be the same number of values in each clist as there are variables or array elements in the corresponding nlist. The appearance of an array in an nlist is the equivalent to a list of all elements in that array in storage sequence order. Array elements must be indexed only by constant subscripts.

The type of each non-character element in a clist must be the same as the type of the corresponding variable or array element in the accompanying nlist. Each character element in a clist must correspond to a character variable or array element in the nlist, and must have a length that is less than or equal to the length of that variable or array element. If the length of the constant is shorter, it is extended to the length of the variable by adding blank characters to the right. Note that a single character constant cannot be used to define more than one variable or even more than one array element.

Only local variables and array elements can appear in a DATA statement. Formal arguments, variables in common, and function names cannot be assigned initial values with a DATA statement.

CHAPTER 7

Expressions

This chapter describes the four classes of expressions found in the FORTRAN language. They are:

7.1. Arithmetic Expressions.
7.2. Character Expressions.
7.3. Relational Expressions.
7.4. Logical Expressions.

7.1. Arithmetic Expressions.

An arithmetic expression produces a value which is either of type integer or of type real. The simplest forms of arithmetic expressions are:

Unsigned integer or real constant.
Integer or real variable reference.
Integer or real array element reference.
Integer or real function reference.

The value of a variable reference or array element reference must be defined for it to appear in an arithmetic expression. Moreover, the value of an integer variable must be defined with an arithmetic value, rather than a statement label value previously set in an ASSIGN statement.

Other arithmetic expressions are built up from the above simple forms using parentheses and these arithmetic operators:

Operator	Representing Operation	Precedence
**	Exponentiation	Highest
/	Division	Intermediate
*	Multiplication	
-	Subtraction or Negation	Lowest
+	Addition or Identity	

Table 7.1. Arithmetic Operators.

All of the operators are binary operators, appearing between their arithmetic expression operands. The + and - may also be unary, preceding their operand.

Operations of equal precedence are left-associative except exponentiation which is right- associative. Thus, A / B * C is the same as (A / B) * C and A ** B ** C is the same as A ** (B ** C). Arithmetic expressions may be formed in the usual algebraic sense, as in most programming languages, except that FORTRAN prohibits two operators from appearing consecutively. Thus, A ** -B is prohibited, although A ** (-B) is permissible. Note that unary minus is also of lowest precedence so that - A * B is interpreted as - (A * B). Parentheses may be used in a program to control the associativity and the order of operator evaluation in an expression.

7.1.1. Integer Division.

The division of two integers results in a value which is the quotient of the two values, truncated toward 0. Thus, 7 / 3 evaluates to 2, (-7) / 3 evaluates to -2, 9 / 10 evaluates to 0 and 9 / (-10) evaluates to 0.

7.1.2. Type Conversions and Result Types of Arithmetic Operators.

Arithmetic expressions may involve operations between operands which are of different type. The general rules for determining type conversions and the result type for an arithmetic expression are:

> An operation between two integers results in an expression of type integer.
>
> An operation between two reals results in an expression of type real.
>
> For any operator except **, an operation between a real and an integer converts the integer to type real and performs the operation, resulting in an expression of type real.
>
> For the operator **, a real raised to an integer power is computed without conversion of the integer, and results in an expression of type real. An integer raised to a real power is converted to type real and the operation results in an expression of type real. Note that for integer I and negative integer J, I ** J is the same as 1 / (I ** IABS(J)) which is subject to the rules of integer division so, for example, 2 ** (-4) is 1 / 16 which is 0.
>
> Unary operators result in the same result type as their operand type.

The type which results from the evaluation of an arithmetic operator is not dependent on the context in which the operation is specified. For example,

evaluation of an integer plus a real results in a real even if the value obtained is to be immediately assigned into an integer variable.

7.2. Character Expressions.

A character expression produces a value which is of type character. The forms of character expressions are:

Character constant.
Character variable reference.
Character array element reference.
Any character expression enclosed in parenthesis.

There are no operators which result in character expressions.

7.3. Relational Expressions.

Relational expressions are used to compare the values of two arithmetic expressions or two character expressions. It is not allowed to compare an arithmetic value with a character value. The result of a relational expression is of type logical.

Relational expressions may use any of these operators to compare values:

Operator	Representing Operation
.LT.	Less than
.LE.	Less than or equal to
.EQ.	Equal to
.NE.	Not equal to
.GT.	Greater than
.GE.	Greater than or equal to

Table 7.2. Relational Operators.

All of the operators are binary operators, appearing between their operands. There is no relative precedence or associativity among the relational operands since an

expression of the form A .LT. B .NE. C violates the type rules for operands. Relational expressions may only appear within logical expressions.

Relational expressions with arithmetic operands may have an operand of type integer and one of type real. In this case, the integer operand is converted to type real before the relational expression is evaluated.

Relational expressions with character operands compare the position of their operands in the ASCII collating sequence. An operand is less than another if it appears earlier in the collating sequence, etc. If operands of unequal length are compared, the shorter operand is considered as if it were blank extended to the length of the longer operand.

7.4. Logical Expressions.

A logical expression produces a value which is of type logical. The simplest forms of logical expressions are:

Logical constant.
Logical variable reference.
Logical array element reference.
Logical function reference.
Relational expression.

Other logical expressions are built up from the above simple forms using parentheses and these logical operators:

Operator	Representing Operation	Precedence
.NOT.	Negation	Highest
.AND.	Conjunction	
.OR.	Inclusive disjunction	Lowest

Table 7.3. Logical Operators.

The .AND. and .OR. operators are binary operators, appearing between their logical expression operands. The .NOT. operator is unary, preceding its operand. Operations of equal precedence are left associative so, for example, A .AND. B .AND. C is equivalent to (A .AND. B) .AND. C. As an example of the precedence rules, .NOT. A .OR. B .AND. C is interpreted the same as (.NOT. A) .OR. (B .AND. C). It is not permitted to have two .NOT. operators adjacent to each

other, although A .AND. .NOT. B is an example of an allowable expression with two operators being adjacent.

The meaning of the logical operators is their standard semantics, with .OR. being "nonexclusive"; that is, .TRUE. .OR. .TRUE. evaluates to the value .TRUE..

7.5. Precedence of Operators.

When arithmetic, relational, and logical operators appear in the same expression, their relative precedences are:

Operator	Precedence
Arithmetic	Highest
Relational	
Logical	Lowest

Table 7.4. Relative Precedence of Operator Classes.

7.6. Evaluation Rules and Restrictions for Expressions.

Any variable, array element, or function referenced in an expression must be defined at the time of the reference. Integer variables must be defined with an arithmetic value, rather than a statement label value as set by an ASSIGN statement.

Certain arithmetic operations are prohibited if they cannot be evaluated (e.g., dividing by zero). Other prohibited operations are raising a zero valued operand to a zero or negative power and raising a negative valued operand to a power of type real.

CHAPTER 8

Assignment Statements

An assignment statement is used to assign a value to a variable or an array element. There are two basic kinds of assignment statements: computational assignment statements, and label assignment statements.

8.1. Computational Assignment Statements.

The form of a computational assignment statement is:

var = expr

'var' is a variable or array element name, and

'expr' is an expression.

Execution of a computational assignment statement evaluates the expression and assigns the resulting value to the variable or array element appearing on the left. The type of the variable or array element and the expression must be compatible. They must both be either numeric, logical, or character, in which case the assignment statement is called an arithmetic, logical, or character assignment statement.

If the type of the elements of an arithmetic assignment statement are not identical, automatic conversion of the value of the expression to the type of the variable is done. The following table gives the conversion rules:

Type of variable or array element	Type of expression	
	integer	real
integer	expr	INT(expr)
real	REAL(expr)	expr

Table 8.1. Type conversion for arithmetic assignment statements.

If the length of the expression does not match the size of the variable in a character assignment statement, it is adjusted so that it does. If the expression is shorter, it is padded with enough blanks on the right to make the sizes equal before the assignment takes place. If the expression is longer, characters on the right are truncated to make the sizes the same.

8.2. Label Assignment Statement.

The label assignment statement is used to assign the value of a format or statement label to an integer variable. The form of the statement is:

ASSIGN label TO var

'label' is a format label or statement label, and

'var' is an integer variable.

Execution of an ASSIGN statement sets the integer variable to the value of the label. The label can be either a format label or a statement label, and it must appear in the same program unit as the ASSIGN statement. When used in an assigned GOTO statement, a variable must currently have the value of a statement label. When used as a format specifier in an I/O statement, a variable must have the value of a format statement label. The ASSIGN statement is the only way to assign the value of a label to a variable.

CHAPTER 9

Control Statements

Control statements are used to control the order of execution of statements in the FORTRAN language. This chapter describes the following control statements:

9.1. Unconditional GOTO.
9.2. Computed GOTO.
9.3. Assigned GOTO.
9.4. Arithmetic IF.
9.5. Logical IF.
9.6. Block IF THEN ELSE.
9.6.1. Block IF.
9.6.2. ELSEIF.
9.6.3. ELSE.
9.6.4. ENDIF.
9.7. DO.
9.8. CONTINUE.
9.9. STOP.
9.10. PAUSE.
9.11. END.

The two remaining statements which control the order of execution of statements are the CALL statement and the RETURN statement, both of which are described in Chapter 12.

9.1. Unconditional GOTO.

The syntax for an unconditional GOTO statement is:

GOTO s

where s is a statement label of an executable statement that is found in the same program unit as the GOTO statement. The effect of executing a GOTO statement is that the next statement executed is the statement labeled s. It is not legal to jump into a DO, IF, ELSEIF, or ELSE block from outside the block (see the various sections for an explanation of the kinds of blocks).

9.2. Computed GOTO.

The syntax for a computed GOTO statement is:

GOTO (s [, s] ...)[,] i

where i is an integer expression and each s is a statement label of an executable statement that is found in the same program unit as the computed GOTO

statement. The same statement label may appear repeatedly in the list of labels. The effect of the computed GOTO statement can be explained as follows: Suppose that there are n labels in the list of labels. If i < 1 or i > n then the computed GOTO statement acts as if it were a CONTINUE statement, otherwise, the next statement executed will be the statement labeled by the ith label in the list of labels. It is not allowed to jump into a DO, IF, ELSEIF, or ELSE block from outside the block (see the various sections for an explanation of the kinds of blocks).

NOTE: computed GOTOs are often used to implement a CASE construct.

9.3. Assigned GOTO.

The syntax for an assigned GOTO statement is:

GOTO i [[,] (s [, s] ...)]

where i is an integer variable name and each s is a statement label of an executable statement that is found in the same program unit as the assigned GOTO statement. The same statement label may appear repeatedly in the list of labels. When the assigned GOTO statement is executed, i must have been assigned the label of an executable statement that is found in the same program unit as the assigned GOTO statement. The effect of the statement is that the next statement executed will be the statement labelled by the label last assigned to i. If the optional list of labels is present, a runtime error is generated if the label last assigned to i is not among those listed. It is not legal to jump into a DO, IF, ELSEIF, or ELSE block from outside the block (see the various sections for an explanation of the kinds of blocks).

9.4. Arithmetic IF.

The syntax for an arithmetic IF statement is:

IF (e) s1, s2, s3

where e is an integer or real expression and each of s1, s2, and s3 are statement labels of executable statements found in the same program unit as the arithmetic IF statement. The same statement label may appear more than once among the three labels. The effect of the statement is to evaluate the expression and then select a label based on the value of the expression. Label s1 is selected if the value of e is less than 0, s2 is selected if the value of e equals 0, and s3 is selected if the value of e exceeds 0. The next statement executed will be the statement labeled by the selected label. It is not legal to jump into a DO, IF, ELSEIF, or ELSE block from outside the block (see the various sections for an

explanation of the kinds of blocks).

9.5. Logical IF.

The syntax for a logical IF statement is:

```
IF (e) st
```

where e is a logical expression and st is any executable statement except a DO, block IF, ELSEIF, ELSE, ENDIF, END, or another logical IF statement. The statement causes the logical expression to be evaluated and, if the value of that expression is .TRUE., then the statement, st, is executed. Should the expression evaluate to .FALSE., the statement st is not executed and the execution sequence continues as if a CONTINUE statement had been encountered.

9.6. Block IF THEN ELSE.

Sections 9.6.1 through 9.6.4 describe the block IF statement and the various statements associated with it. These statements are new to FORTRAN 77 and can be used to dramatically improve the readability of FORTRAN programs and to cut down the number of GOTOs of the various forms. As an overview of these sections, the following three code skeletons illustrate the basic concepts:

Skeleton 1 - Simple Block IF which skips a group of statements if the expression is false:

```
IF(I.LT.10)THEN
    .
    .  Some statements executed only if I.LT.10
    .
ENDIF
```

Skeleton 2 - Block IF with a series of ELSEIF statements:

```
IF(J.GT.1000)THEN
    .
    .  Some statements executed only if J.GT.1000
    .
ELSEIF(J.GT.100)THEN
    .
    .  Some statements executed only if J.GT.100 and
    .  J.LE.1000
```

```
ELSEIF(J.GT.10)THEN
     .
     . Some statements executed only if J.GT.10 and
     . J.LE.1000 and J.LE.100
ELSE
     .
     . Some statements executed only if none of above
     . conditions were true
ENDIF
```

Skeleton 3 - Illustrates that the constructs can be nested and that an ELSE statement can follow a block IF without intervening ELSEIF statements (indentation solely to enhance readability):

```
IF(I.LT.100)THEN
     .
     . Some statements executed only if I.LT.100
     .
       IF(J.LT.10)THEN
       .
       . Some statements executed only if I.LT.100
       . and J.LT.10
       ENDIF
     .
     . Some statements executed only if I.LT.100
     .
ELSEIF(I.LT.1000)THEN
     .
     . Some statements executed only if I.GE.100 and
     . I.LT.1000
       IF(J.LT.10)THEN
       .
       . Some statements executed only if I.GE.100
       . and I.LT.1000 and J.LT.10
       ENDIF
     .
     . Some statements executed only if I.GE.100 and
     . I.LT.1000
ENDIF
```

In order to understand, in detail, the block IF and associated statements, the concept of an IF-level is introduced. For any statement, its IF-level is

```
n1 - n2
```

where n1 is the number of block IF statements from the beginning of the program unit that the statement is in, up to and including that statement, and n2 is is the number of ENDIF statements from the beginning of the program unit up to, but not including, that statement. The IF-level of every statement must be greater than or equal to 0 and the IF-level of every block IF, ELSEIF, ELSE, and ENDIF must be greater than 0. Finally, the IF-level of every END statement must be 0. The IF-level will be used to define the nesting rules for the block IF and associated statements and to define the extent of IF blocks, ELSEIF blocks, and ELSE blocks.

9.6.1. Block IF.

The syntax for a block IF statement is:

IF (e) THEN

where e is a logical expression. The IF block associated with this block IF statement consists of all of the executable statements, possibly none, that appear following this statement up to, but not including, the next ELSEIF, ELSE, or ENDIF statement that has the same IF-level as this block IF statement (the IF-level defines the notion of "matching" ELSEIF, ELSE, or ENDIF). Executing the block IF statement first causes the expression to be evaluated. If it evaluates to .TRUE. and there is at least one statement in the IF block, the next statement executed is the first statement of the IF block. Following the execution of the last statement in the IF block, the next statement to be executed will be the next ENDIF statement at the same IF-level as this block IF statement. If the expression in this block IF statement evaluates to .TRUE. and the IF block has no executable statements, the next statement executed is the next ENDIF statement at the same IF level as the block IF statement. If the expression evaluates to .FALSE., the next statement executed is the next ELSEIF, ELSE, or ENDIF statement that has the same IF-level as the block IF statement. Note that transfer of control into an IF block from outside that block is not allowed.

9.6.2. ELSEIF.

The syntax of an ELSEIF statement is:

ELSEIF (e) THEN

where e is a logical expression. The ELSEIF block associated with an ELSEIF statement consists of all of the executable statements, possibly none, that follow the ELSEIF statement up to, but not including, the next ELSEIF, ELSE, or ENDIF statement that has the same IF-level as this ELSEIF statement. The execution of

an ELSEIF statement begins by evaluating the expression. If its value is .TRUE. and there is at least one statement in the ELSEIF block, the next statement executed is the first statement of the ELSEIF block. Following the execution of the last statement in the ELSEIF block, the next statement to be executed will be the next ENDIF statement at the same IF-level as this ELSEIF statement. If the expression in this ELSEIF statement evaluates to .TRUE. and the ELSEIF block has no executable statements, the next statement executed is the next ENDIF statement at the same IF level as the ELSEIF statement. If the expression evaluates to .FALSE., the next statement executed is the next ELSEIF, ELSE, or ENDIF statement that has the same IF-level as the ELSEIF statement. Note that transfer of control into an ELSEIF block from outside that block is not allowed.

9.6.3. ELSE.

The syntax of an ELSE statement is:

```
ELSE
```

The ELSE block associated with an ELSE statement consists of all of the executable statements, (possibly none), that follow the ELSE statement up to, but not including, the next ENDIF statement that has the same IF-level as this ELSE statement. The "matching" ENDIF statement must appear before any intervening ELSE or ELSEIF statements of the same IF-level. Note that transfer of control into an ELSE block from outside that block is not allowed.

9.6.4. ENDIF.

The syntax of an ENDIF statement is:

```
ENDIF
```

There is no effect of executing an ENDIF statement. An ENDIF statement is required to "match" every block IF statement in a program unit in order to specify which statements are in a particular block IF statement.

9.7. DO.

The syntax of an DO statement is:

```
DO s [,] i=e1, e2 [, e3]
```

where s is a statement label of an executable statement. The label must follow this DO statement and be contained in the same program unit. In the DO

statement, i is an integer variable, and e1, e2, and e2 are integer expressions. The statement labeled by s is called the terminal statement of the DO loop. It must not be an unconditional GOTO, assigned GOTO, arithmetic IF, block IF, ELSEIF, ELSE, ENDIF, RETURN, STOP, END, or DO statement. If the terminal statement is a logical IF, it may contain any executable statement except those not permitted inside a logical IF statement.

A DO loop is said to have a "range", beginning with the statement which follows the DO statement and ending with (and including) the terminal statement of the DO loop. If a DO statement appears in the range of another DO loop, its range must be entirely contained within the range of the enclosing DO loop, although the loops may share a terminal statement (not recommended). If a DO statement appears within an IF block, ELSEIF block, or ELSE block, the range of the associated DO loop must be entirely contained in the particular block. If a block IF statement appears within the range of a DO loop, its associated ENDIF statement must also appear within the range of that DO loop. The DO variable, i, may not be set by the program within the range of the DO loop associated with it. It is not allowed to jump into the range of a DO loop from outside its range.

The execution of a DO statement causes the following steps to happen in order:

> The expressions e1, e2, and e3 are evaluated. If e3 is not present, e3 defaults to 1 (e3 must not evaluate to 0).
>
> The DO variable, i, is set to the value of e1.
>
> The iteration count for the loop is computed to be MAX0(((e2 - e1 + e3)/e3),0) which may be zero (Note: unlike FORTRAN 66) if either e1 > e2 and e3 > 0, or e1 < e2 and e3 < 0.
>
> The iteration count is tested, and if it exceeds zero, the statements in the range of the DO loop are executed.

Following the execution of the terminal statement of a DO loop, the following steps occur in order:

> The value of the DO variable, i, is incremented by the value of e3 which was computed when the DO statement was executed.
>
> The iteration count is decremented by one.
>
> The iteration count is tested, and if it exceeds zero, the statements in the range of the DO loop are executed again.

The value of the DO variable is well-defined after execution of the loop, regardless of whether the DO loop exits as a result of the iteration count becoming zero, as the result of a transfer of control out of the DO loop, or as

the result of a RETURN statement.

Example of the final value of a DO variable:

```
C This program fragment prints the number 1 to 11 on
C the CONSOLE:
        DO 200 I=1,10
        200 WRITE(*,'(I5)')I
        WRITE(*,'(I5)')I
```

9.8. CONTINUE.

The syntax of a CONTINUE statement is:

```
CONTINUE
```

There is no effect associated with execution of a CONTINUE statement. The primary use for the CONTINUE statement is a convenient statement to label, particularly as the terminal statement in a DO loop.

9.9. STOP.

The syntax of an STOP statement is:

```
STOP [n]
```

where n is either a character constant or a string of not more than 5 digits. The effect of executing a STOP statement is to cause the program to terminate. The argument, n, if present, is displayed on CONSOLE: upon termination.

9.10. PAUSE.

The syntax of an PAUSE statement is:

```
PAUSE [n]
```

where n is either a character constant or a string of not more than 5 digits. The effect of executing a PAUSE statement is to cause the program to be suspended pending an indication from the CONSOLE: that it is to continue. The argument, n, if present, is displayed on the CONSOLE: as part of the prompt requesting input from the CONSOLE:. If the indication from the CONSOLE: is received to continue execution of the program, execution resumes as if a CONTINUE statement had been executed.

9.11. END.

The syntax of an END statement is:

```
END
```

Unlike other statements, an END statement must wholly appear on an initial line and contain no continuation lines. No other FORTRAN statement, such as the ENDIF statement, may have an initial line which appears to be an END statement. The effect of executing the END statement in a subprogram is the same as execution of a RETURN statement and the effect in the main program is to terminate execution of the program. The END statement must appear as the last statement in every program unit.

CHAPTER 10

I/O System

Chapters 10 and 11 of this manual describe the FORTRAN I/O System. Chapter 10 describes the basic FORTRAN I/O concepts and statements and Chapter 11 describes the FORMAT statement. The four major Sections of these chapters are:

10.1. I/O System Overview - Provides an overview of the FORTRAN file System. Defines the basic concepts of I/O records, I/O units, and the various kinds of file access available under the System.

10.2. General Discussion of I/O System Concepts and Limitations - The definitions made in Section 10.1 are related to how to accomplish various simple, as well as complex, tasks using the I/O System. There is a general discussion of I/O System limitations.

10.3. I/O Statements - The statements of the I/O System are presented with the exception of the FORMAT statement.

11. Formatted I/O and the FORMAT Statement - The FORMAT statement and formats in general are described.

NOTE: the reader is directed to Section 10.2 for a brief discussion of the most commonly used forms of files and I/O statements, and a complete sample program illustrating the most commonly used forms of I/O.

10.1. I/O System Overview.

In order to fully understand the I/O statements, it is necessary to be familiar with a variety of terms and concepts related to the structure of the FORTRAN I/O System. Most I/O tasks can be accomplished without a complete understanding of this material and the reader is encouraged to skip to Section 10.2 on first reading and subsequently use 10.1 primarily for reference.

10.1.1. Records.

The building block of the FORTRAN file system is the Record. A Record is a sequence of characters or a sequence of values. There are three kinds of records:

Formatted.
Unformatted.
Endfile.

A formatted record is a sequence of characters terminated by the character value which corresponds to the "return" key on a terminal (character value 13).

Formatted records are processed on input consistent with the way that the Operating System and Text Editor process characters. Thus, reading characters from formatted records in FORTRAN is identical to reading characters in other System programs and other languages on the System. Formatted files are normally transportable between different UCSD interpreters.

An unformatted record is a sequence of values, with no System alteration or processing; no physical representation for the end of record exists. Unformatted files generated on different processors are not generally interchangable, since the internal representations of integers and reals differ among the various UCSD interpreters.

The System makes it appear as though an endfile record exists after the last record in a file, although no physical endfile mark ever exists.

10.1.2. Files.

A FORTRAN file is a sequence of records. FORTRAN files are one of two kinds:

External.
Internal.

An external FORTRAN file is a file on a device, or the device itself. An internal FORTRAN file is a character variable which serves as the source or destination of some I/O action. From this point on, both FORTRAN files and the notion of a file as known to the Operating System and the Editor will be referred to simply as files, with the context determining which meaning is intended. (The OPEN statement provides the linkage between the two notions of files, and in most cases the ambiguity disappears, since after a file has been opened, the two notions are one and the same.)

10.1.3. File Properties.

A file which is being acted upon by a FORTRAN program has a variety of properties. These properties are described in Sections 10.1.3.1 through 10.1.3.4.

10.1.3.1. File Name.

A file may have a name. If present, a name is a character string identical to the name by which it is known to the UCSD File System. There may be more than one name for the same file, such as SYS:A.TEXT and #4:A.TEXT.

10.1.3.2. File Position.

A file has a position property which is usually set by the previous I/O operation. There is a notion of the initial point in the file, the terminal point in the file, the current record, the preceding record, and the next record of the file. It is possible to be between records in a file, in which case the next record is the successor to the previous record and there is no current record. The file position after sequential writes is at the end of file, but not beyond the endfile record. Execution of the ENDFILE statement positions the file beyond the endfile record, as does a read statement executed at the end of file (but not beyond the endfile record). Reading an endfile record may be trapped by the user using the END= option in a READ statement.

10.1.3.3. Formatted and Unformatted Files.

An external file is opened as either formatted or unformatted. All internal files are formatted. Files which are formatted consist entirely of formatted records and files which are unformatted consist entirely of unformatted records. Files which are formatted obey all the structural rules of .TEXT files, so that they are fully compatible with the System Text Editor.

10.1.3.4. Sequential and Direct Access Properties.

An external file is opened as either sequential or direct. Sequential files contain records with an order property determined by the order in which the records were written (the normal sequential order). These files must not be read or written using the REC= option which specifies a position for direct access I/O. The System will attempt to extend sequential access files if a record is written beyond the old terminating boundary of the file, but the success of this depends on the existence of room on the physical device at the appropriate location.

Direct access files may be read or written in any order (they are random access files). Records in a direct access file are numbered sequentially, with the first record numbered one. All records in a direct access file have the same length, which is specified at the time the file is opened. Each record in the file is uniquely identified by its record number, which was specified when the record was written. It is entirely possible to write the records out of order, including, for example, writing record 9, 5, and 11 in that order without the records in between. It is not possible to delete a record once written, but it is possible to overwrite a record with a new value. It is an error to read a record from a direct access file which has not been written, but the System will not detect this error unless the record which is being read is beyond the last record written in the file (a non-written record before the end-of-file contains garbage). Direct access files must

reside on block-structured peripheral devices such as the diskette, so that it is meaningful to specify a position in the file and reference it. The System will attempt to extend direct access files if an attempt is made to write to a position beyond the previous terminating boundary of the file, but the success of this depends on the existence of room on the physical device at the appropriate location.

10.1.4. Internal Files.

Internal files provide a mechanism for using the formatting capabilities of the I/O System to convert values to and from their external character representations, within the FORTRAN internal storage structures. That is, reading a character variable converts the character values into numeric, logical, or character values and writing into a character variable allows values to be converted into their (external) character representation.

10.1.4.1. Special Properties of Internal Files.

An internal file is a character variable or character array element. The file has exactly one record, which has the same length as the character variable or character array element. Should less than the entire record be written by a WRITE statement, the remaining portion of the record is filled with blanks. The file position is always at the beginning of the file prior to I/O statement execution. Only formatted, sequential I/O is permitted with internal files, and only the I/O statements READ and WRITE may specify an internal file.

10.1.5. Units.

A unit is a means of referring to a file. A unit specified in an I/O statement is one of:

External unit specifier.
Internal file specifier.

External unit specifiers are either integer expressions which evaluate to non-negative values, or the character *, which stands for the CONSOLE: device. In most cases, external unit specifier values are bound to physical devices (or files resident on those devices) by name (using the OPEN statement). Once this binding of value to System file name occurs, FORTRAN I/O statements refer to the unit number as a means of referring to the appropriate external entity. Once opened, the external unit specifier value is uniquely associated with a particular external entity until an explicit CLOSE occurs or until the program terminates. The only exception to the above binding rules is that the unit value 0 is initially associated

with the CONSOLE: device for reading and writing and no explicit OPEN is necessary. The character * is interpreted by the System as specifying unit 0.

An internal file specifier is a character variable or character array element which directly specifies an internal file.

10.2. General Discussion of I/O System Concepts and Limitations.

10.2.1. General Discussion of FORTRAN I/O System.

FORTRAN provides a rich combination of possible file structures. Choosing from among these many structures may at first seem somewhat confusing. However, two kinds of files will suffice for most applications.

* - CONSOLE:, a sequential, formatted file, also known as unit 0 - This particular unit has the special property that an entire line terminated by the return key, must be entered when reading from it, and the various backspace and line delete keys familiar to the System user serve their normal functions. Note that a READ from any other unit will not have these properties, even if that unit is bound to CONSOLE: by an explicit OPEN statement.

Explicitly opened external, sequential, formatted files - These files are bound to a System file by name in an OPEN statement. They can be read and written in the System Text Editor compatible format.

10.2.2. Example Program Illustrating Most Common I/O Operations.

Here is a sample program which uses the kinds of files discussed in Section 10.2.1 for reading and for writing. The various I/O statements are explained in detail in Section 10.3.

```
C Copy a file with three columns of integers, each 7
C columns wide, from a file whose name is input by the
C user to another file named OUT.TEXT, reversing the
C positions of the first and second column.
      PROGRAM COLSWP
      CHARACTER*23 FNAME
C Prompt to the CONSOLE: by writing to *
      WRITE(*,900)
900   FORMAT('Input file name - ')
C Read the file name from the CONSOLE: by reading from *
      READ(*,910) FNAME
910   FORMAT(A)
```

```
C Use unit 3 for input, any unit number except 0 will do
      OPEN(3,FILE=FNAME)
C Use unit 4 for output, any unit number except 0 and 3
C will do
      OPEN(4,FILE='OUT.TEXT',STATUS='NEW')
C Read and write until end of file
100   READ(3,920,END=200)I,J,K
      WRITE(4,920)J,I,K
920   FORMAT(3I7)
      GOTO 100
200   WRITE(*,910)'Done'
      END
```

10.2.3. Use of Less Common File Operations.

The less commonly used file structures are appropriate for certain classes of applications. A very general indication of the intended usages for them are as follows: if the I/O is to be random access, such as in maintaining a database, direct access files are probably necessary. If the data is to be written by FORTRAN and reread by FORTRAN (on the same brand of processor), unformatted files are more efficient both in file space and in I/O overhead. The combination of direct and unformatted is ideal for a database to be created, maintained, and accessed exclusively by FORTRAN. If the data must be transferred without any System interference, especially if all 256 possible bytes will be transferred, unformatted I/O will be necessary, since .TEXT files are constrained to contain only the printable character set as data. An example of a usage of unformatted I/O would be in the control of a device which has a single byte, binary interface. Formatted I/O would, in this example, interpret certain characters, such as the ASCII representation for carriage return, and fail to pass them through to the program unaltered. Internal files are not I/O in the conventional sense but rather provide certain character string operations and conversions within a standard mechanism.

Use of formatted direct files requires special caution. FORTRAN formatted files attempt to comply with the Operating System rules for .TEXT files (for a discussion of .TEXT files, see the Users' Manual). FORTRAN I/O is able to enforce these rules for sequential files, but it cannot always enforce them for direct files. Direct files are not necessarily legal .TEXT files, since any unwritten records contain undefined values which do not follow .TEXT file constraints. Direct files do, of course, obey FORTRAN I/O rules.

A file opened in FORTRAN is either "old" or "new", but there is no concept of "opened for reading" as distinguished from "opened for writing". Therefore, you may open "old" (existing) files and write to them, with the effect of modifying existing files. Similarly, you may alternately write and read to the same file (providing

that one avoids reading beyond end of file, or reading unwritten records in a direct files). A write to a sequential file effectively deletes any records which had existed beyond the freshly written record. Normally, when a device is opened as a file (such as CONSOLE: or PRINTER:) it makes no difference whether the file is opened as "old" or "new". With diskette files, opening "new" creates a new temporary file. If that file is closed using the "keep" option, or if the program is terminated without doing a CLOSE on that file, a permanent file is created with the name given when the file was opened. If a previous file existed with the same name, it is deleted. If closed using the "delete" option, the newly created temporary file is deleted, and any previous file of the same name is left intact. Opening a diskette file as "old" will generate a run time error if the file does not exist and alter the existing file if written.

10.2.4. Limitations of the FORTRAN I/O System.

10.2.4.1. Direct Files must be Associated with Blocked Devices.

The Operating System uses two kinds of devices: block- structured and sequential. Sequential files may be thought of as streams of characters, with no explicit motion allowed except reading and/or writing. CONSOLE: and PRINTER: are examples of sequential devices. Block-structured devices, such as diskette files, allow the additional operation of seeking a specific location. They can be accessed either sequentially or randomly and thus can support direct files. Since there is no notion of seeking a position on a file which is not block- structured, FORTRAN I/O does not allow direct file access to sequential devices.

10.2.4.2. BACKSPACE only Applies to Files Associated with Blocked Devices.

Sequential devices can not be backspaced meaningfully under the UCSD Operating System, so FORTRAN I/O disallows backspacing a file on a sequential device (see 10.2.2.1).

10.2.4.3. BACKSPACE may not be Used on Unformatted Sequential Files.

It is not possible to implement BACKSPACE on unformatted sequential files since there is no indication in the file itself of the record boundaries. It would be possible to append end of record marks to unformatted sequential files, but this would interfere with the notion of an unformatted file being a "pure" sequence of values, and would interfere with the most common usage for such files, such as the direct control of an external device. Direct files contain records of fixed and

specified length, so it is possible to backspace direct unformatted files.

10.2.4.4. Side Effects of Functions Called in I/O Statements.

During the course of executing any I/O statement, the evaluation of an expression may cause a function to be called. That function call must not cause any I/O statement to be executed.

10.3. I/O Statements.

This Section describes these I/O statements which are available from FORTRAN:

10.3.2.1. OPEN
10.3.2.2. CLOSE
10.3.2.3. READ
10.3.2.4. WRITE
10.3.2.5. BACKSPACE
10.3.2.6. ENDFILE
10.3.2.7. REWIND

In addition, there is an I/O intrinsic function EOF, presented in Chapter 12, which returns a logical value indicating whether the file associated with the unit specifier passed to it is at end-of-file. A familiarity with the FORTRAN file system, units, records, and access methods as described in the previous Sections is assumed.

10.3.1. Elements of I/O Statements.

The various I/O statements take certain parameters and arguments which specify sources and destinations of data transfer, as well as other facets of the I/O operation. The abbreviations are used throughout Section 10.3 are defined in Sections 10.3.1.1 through 10.3.1.3.

10.3.1.1. The Unit Specifier ('u').

The unit specifier, 'u', can take one of these forms in an I/O statement:

* - refers to the CONSOLE:.

integer expression - refers to external file with unit number equal to the value of the expression (* is unit number 0).

name of a character variable or character array element - refers to

the internal file which is the character datum.

10.3.1.2. The Format Specifier ('f').

The format specifier, 'f', can take one of these forms in an I/O statement:

statement label - refers to the FORMAT statement labeled by that label.

integer variable name - refers to the FORMAT label which that integer variable has been assigned to using the ASSIGN statement.

character expression - the format which is specified is the current value of the character expression provided as the format specifier.

10.3.1.3. The Input-Output List ('iolist').

The input-output list, 'iolist', specifies the entities whose values are transferred by READ and WRITE statements. An iolist is a possibly empty list, separated by commas, of items which consist of:

Input or Output entities - see 10.3.1.3.1 and 10.3.1.3.2.

Implied DO lists - see 10.3.1.3.3.

10.3.1.3.1. Input Entities.

An input entity may be specified in the iolist of a READ statement and is of one of these forms:

Variable name.

Array element name.

Array name - this is a means of specifying all of the elements of the array in storage sequence order.

10.3.1.3.2. Output Entities.

An output entity may be specified in the iolist of a WRITE statement, and is of one of these forms:

Variable name;

Array element name;

Array name - this is a means of specifying all of the elements of the array in storage sequence order;

Any other expression not beginning with the character '(' - to distinguish implied DO lists from expressions.

10.3.1.3.3. Implied DO lists.

Implied DO lists may be specified as items in the I/O list of READ and WRITE statements, and are of the form:

(iolist, i = e1, e2 [, e3])

where the iolist is as above (including nested implied DO lists) and i, e1, e2 and the optional e3 are as defined for the DO statement. That is, i is an integer variable and e1, e2, and e3 are integer expressions. In a READ statement, the DO variable i (or an associated entity) must not appear as an input list item in the embedded iolist, but may have been read in the same READ statement outside of the implied DO list. The embedded iolist is effectively repeated for each iteration of i with appropriate substitution of values for the DO variable i.

10.3.2. I/O Statements.

The following I/O statements are supported by FORTRAN. The possible form for each statement is specified first, with an explanation of the meanings for the forms following. Certain items are specified as required if they must appear in the statement, and are specified as optional if they need not appear. Typically, optional items have defaults. Examples are provided.

10.3.2.1. OPEN Statement.

OPEN(

u,

Required, must appear as the first argument. Must not be internal unit specifier.

FILE=fname,

The file name, 'fname', is a character expression. This argument to OPEN is required and must appear as the second argument.

The following arguments are all optional, and may appear in any order. The options are character constants with optional trailing blanks (except RECL=). Defaults are indicated.

STATUS='OLD'

Default, for reading or writing existing files.

STATUS='NEW'

For writing new files.

ACCESS='SEQUENTIAL' (Default)

ACCESS='DIRECT'

FORM='FORMATTED' (Default)

FORM='UNFORMATTED'

RECL=rl)

The record length, 'rl' is an integer expression. This argument to OPEN is for DIRECT access files only, for which it is required.

The OPEN statement binds a unit number with an external device or file on an external device by specifying its file name. If the file is to be direct, the RECL=rl option specifies the length of the records in that file.

Example program fragment 1:

```
C Prompt user for a file name
      WRITE(*,'(A\)') 'Specify output file name -
C Presume that FNAME is specified to be CHARACTER*23
C Read the file name from the CONSOLE:
```

```
      READ(*,'(A)') FNAME
C Open the file as formatted sequential as unit 7, note
C that the ACCESS specified need not have appeared since
C it is the default.
      OPEN(7,FILE=FNAME,ACCESS='SEQUENTIAL',STATUS='NEW');
```

Example program fragment 2:

```
C Open an existing file created by the editor called
C DATA3.TEXT as unit 3
      OPEN(3,FILE='DATA3.TEXT')
```

10.3.2.2. CLOSE Statement.

CLOSE(

u,

Required, must appear as the first argument. Must not be internal unit specifier.

STATUS='KEEP'
STATUS='DELETE'

Optional argument which applies only to files opened NEW, default is KEEP. The option is character constant.

)

CLOSE disconnects the unit specified and prevents subsequent I/O from being directed to that unit (unless the same unit number is reopened, possibly bound to a different file or device). Files opened NEW are temporaries and discarded if STATUS='DELETE' is specified. Normal termination of a FORTRAN program automatically closes all open files as if CLOSE with STATUS='KEEP' had been specified.

Example program fragment:

```
C Close the file opened in OPEN example, discarding the file
      CLOSE(7,STATUS='DELETE')
```

10.3.2.3. READ Statement.

READ(

u,

Required, must be first argument.

f,

Required for formatted read as second argument, must not appear for unformatted read.

REC=rn

For direct access only, otherwise error. Positions to record number rn, where rn is a positive integer expression. If omitted for direct access file, reading continues from the current position in the file.

END=s)

Optional, statement label. If not present, reading end of file results in a run time error. If present, encountering an end of file condition results in the transfer to the executable statement labeled s which must be in the same program unit as the READ statement.

iolist

The READ statement sets the items in iolist (assuming that no end of file or error condition occurs). If the read is internal, the character variable or character array element specified is the source of the input, otherwise the external unit is the source.

Example program fragment:

```
C Need a two dimensional array for the example
      DIMENSION IA(10,20)
C Read in bounds for array off first line, hopefully less
C than 10 and 20.   Then read in the array in nested
```

```
C implied DO lists with input format of 8 columns of width
C 5 each.
      READ(3,990)I,J,((IA(I,J),J=1,J),I=1,I,1)
990   FORMAT(215/,(815))
```

10.3.2.4. WRITE Statement.

WRITE(

u,

Required, must be first argument.

f,

Required for formatted write as second argument, must not appear for unformatted write.

REC=rn)

For direct access only, otherwise error. Positions to record number rn, where rn is a positive integer expression. If omitted for direct access file, writing continues at the current position in the file.

iolist

The WRITE statement transfers the iolist items to the unit specified. If the write is internal, the character variable or character array element specified is the destination of the output, otherwise the external unit is the destination.

Example program fragment:

```
C Place message: "One = 1, Two = 2, Three = 3" on the
C CONSOLE:, not doing things in the simplest way!
      WRITE(*,980)'One =',1,1+1,'ee = ',+(1+1+1)
980   FORMAT(A,I2,', Two =',1X,I1,', Thr',A,I1)
```

10.3.2.5. BACKSPACE Statement.

BACKSPACE u

> Unit is not internal unit specifier. Can only be issued on units which are bound to blocked devices. Can only be issued on units which are direct or sequential formatted (i.e., not on sequential unformatted).

BACKSPACE causes the file connected to the specified unit to be positioned before the preceding record. If there is no preceding record, the file position is not changed. Note that if the preceding record is the endfile record, the file becomes positioned before the endfile record.

10.3.2.6. ENDFILE Statement.

ENDFILE u

> Unit is not an internal unit specifier.

ENDFILE "writes" an end of file record as the next record of the file connected to the specified unit. The file is then positioned after the end of file record, so further sequential data transfer is prohibited until either a BACKSPACE or REWIND is executed. An ENDFILE on a direct access file makes all records written beyond the position of the new end of file disappear.

10.3.2.7. REWIND Statement.

REWIND u

> Unit is not an internal unit specifier.

Execution of a REWIND statement causes the file associated with the specified unit to be positioned at its initial point.

10.3.3. Restriction on I/O Side Effects of Functions.

Any function referenced in an expression within any I/O statement must not cause any I/O statement to be executed.

CHAPTER 11

Formatted I/O and the FORMAT Statement

This chapter describes formatted I/O and the FORMAT statement. A familiarity with the FORTRAN file system, units, records, access methods, and I/O statements as described in the previous chapters is assumed.

11.1. Format Specifications and the FORMAT Statement.

If a READ or WRITE statement specifies a format, it is considered a formatted, rather than an unformatted I/O statement. Such a format may be specified in one of three ways, as explained in the previous chapter. Two ways refer to FORMAT statements and one is an immediate format in the form of a character expression containing the format itself. The following are all valid and equivalent means of specifying a format:

```
      WRITE(*,990)I,J,K
990   FORMAT(2I5,I3)

      ASSIGN 990 TO IFMT
990   FORMAT(2I5,I3)
      WRITE(*,IFMT)I,J,K

      WRITE(*,'(2I5,I3)')I,J,K

      CHARACTER*8 FMTCH
      FMTCH = '(2I5,I3)'
      WRITE(*,FMTCH)I,J,K
```

The format specification itself must begin with "(",
possibly following initial blank characters, and end with a matching ")". Characters beyond the matching ")" are ignored.

FORMAT statements must be labelled, and like all nonexecutable statements, may not be the target of a branching operation.

Between the initial "(" and terminating ")" is a list of items, separated by commas, each of which is one of:

[r] ed - repeatable edit descriptors

ned - nonrepeatable edit descripors

[r] fs - a nested format specification. At most 3 levels of nested parenthesis are permitted within the outermost level.

where r is an optionally present, nonzero, unsigned, integer constant called a

repeat specification. The comma separating two list items may be omitted if the resulting format specification is still unambiguous, such as after a P edit descriptor or before or after the / edit descriptor.

The repeatable edit descriptors, explained in detail below, are:

Iw
Fw.d
Ew.d
Ew.dEe
Lw
A
Aw

where I, F, E, L, and A indicate the manner of editing and, w and e are nonzero, unsigned, integer constants, and d is an unsigned integer constant.

The nonrepeatable edit descriptors, which are also explained in detail below, are:

'xxxx' - character constants of any length, see special rules below

nHxxxx - another means of specifying character constants, see rules below

nX
/
\
kP
BN
BZ

where apostrophe, H, X, slash, backslash, P, BN, and BZ indicate the manner of editing and, x is any ascii character, n is a nonzero, unsigned, integer constant, and k is an optionally signed integer constant.

11.2. Interaction between Format Specification and I/O List.

Before describing in greater detail the manner of editing specified by each of the above edit descriptors, it must be explained how the format specification interacts with the input/output list (iolist) in a given READ or WRITE statement.

If an iolist contains at least one item, at least one repeatable edit descriptor

must exist in the format specification. In particular, the empty edit specification, (), may be used only if no items are specified in the iolist (in which case the only action caused by the I/O statement is the implicit record skipping action associated with formats). Each item in the iolist will become associated with a repeatable edit descriptor during the I/O statement execution in turn. In contrast to this, the other format control items interact directly with the record and do not become associated with items in the iolist.

The items in a format specification are interpreted from left to right. Repeatable edit descriptors act as if they were present r times (omitted r is treated as a repeat factor of 1). Similarly, a nested format specification is treated as if its items appeared r times.

The formatted I/O process proceeds as follows: The "format controller" scans the format items in the order indicated above. When a repeatable edit descriptor is encountered, either

> a corresponding item appears in the iolist in which case the item and the edit descriptor become associated and I/O of that item proceeds under format control of the edit descriptor, or
>
> the "format controller" terminates I/O.

If the format controller encounters the matching final) of the format specification and there are no further items in the iolist, the "format controller" terminates I/O. If, however, there are further items in the iolist, the file is positioned at the beginning of the next record and the "format controller" continues by rescanning the format starting at the beginning of the format specification terminated by the last preceding right parenthesis. If there is no such preceding right parenthesis, the "format controller" will rescan the format from the beginning. Within the portion of the format rescanned, there must be at least one repeatable edit descriptor. Should the rescan of the format specification begin with a repeated nested format specification, the repeat factor is used to indicate the number of times to repeat that nested format specification. The rescan does not change the previously set scale factor or BN or BZ blank control in effect. When the "format controller" terminates, the remaining characters or an input record are skipped or an end of record is written on output, except as noted under the \ edit descriptor.

11.3. Edit Descriptors.

Here are the detailed explanations of the various format specification descriptors, beginning with the nonrepeatable edit descriptors:

11.3.1. Nonrepeatable Edit Descriptors.

11.3.1.1. 'xxxx' (Apostrophe Editing).

The apostrophe edit descriptor has the form of a character constant. Embedded blanks are significant and double '' are interpreted as a single '. Apostrophe editing may not be used in a READ statement. It causes the character constant to be transmitted to the output unit.

11.3.1.2. H (Hollerith Editing).

The nH edit descriptor cause the following n characters, with blanks counted as significant, to be transmitted to the output. Hollerith editing may not be used in a READ.

Examples of Apostrophe and Hollerith editing:

```
C Each write outputs characters between the
C slashes: /ABC'DEF/
      WRITE(*,970)
970   FORMAT('ABC''DEF')
      WRITE(*,'(''ABC''''DEF'')')
      WRITE(*,'(7HABC''DEF)')
      WRITE(*,960)
960   FORMAT(7HABC'DEF)
```

11.3.1.3. X (Positional Editing).

On input (a READ), the nX edit descriptor causes the
file position to advance over n characters, thus the next n characters are skipped. On output (a WRITE), the nX edit descriptor causes n blanks to be written, providing that further writing to the record occurs, otherwise, the nX descriptor results in no operation.

11.3.1.4. / (Slash Editing).

The slash indicates the end of data transfer on the current record. On input, the file is positioned to the beginning of the next record. On output, an end of record is written and the file is positioned to write on the beginning of the next record.

11.3.1.5. \ (Backslash Editing).

Normally when the "format controller" terminates, the end of data transmission on the current record occurs. If the last edit descriptor encountered by the "format controller" is the backslash, this automatic end of record is inhibited. This allows subsequent I/O statements to continue reading (or writing) out of (or into) the same record. The most common use for this mechanism is to prompt to the CONSOLE: and read a response off the same line as in:

```
WRITE(*,'(A\)') 'Input an integer ->
READ(*,'(BN,I6)') I
```

The backslash edit descriptor does not inhibit the automatic end of record generated when reading from the * unit. Input from the CONSOLE: must always be terminated by the return key. This permits the backspace character and the line delete key to function properly.

11.3.1.6. P (Scale Factor Editing).

The kP edit descriptor is used to set the scale factor for subsequent F and E edit descriptors until another kP edit descriptor is encountered. At the start of each I/O statement, the scale factor equals 0. The scale factor affects format editing in the following ways:

On input, with F and E editing, providing that no explicit exponent exists in the field, and F output editing, the externally represented number equals the internally represented number multiplied by 10**k.

On input, with F and E editing, the scale factor has no effect if there is an explicit exponent in the input field.

On output, with E editing, the real part of the quantity is output multiplied by 10**k and the exponent is reduced by k (effectively altering the column position of the decimal point, but not the value that is output).

11.3.1.7. BN and BZ (Blank Interpretation).

These edit descriptors specify the interpretation of blanks in numeric input fields. The default, BZ, is set at the start of each I/O statement. This makes blanks, other than leading blanks, identical to zeros. If a BN edit descriptor is processed by the "format controller", blanks in subsequent input fields will be ignored unless, and until, a BZ edit descriptor is processed. The effect of ignoring blanks is to take all the nonblank characters in the input field, and treat them as if they were

right justified in the field with the number of leading blanks equal to the number of ignored blanks. For instance, the following READ statement shown accepts the characters shown between the slashes as the value 123 (where <cr> indicates hitting the return key):

```
      READ(*,100) I
100   FORMAT(BN,I6)

      /123      <cr>/,
      /123   456<cr>/,
      /123<cr>/, or
      /   123<cr>/.
```

The BN edit descriptor, in conjunction with the infinite blank padding at the end of formatted records, makes interactive input very convenient.

11.3.2. Repeatable Edit Descriptors.

11.3.2.1. I, F, and E (Numeric Editing, General Description).

The I, F, and E edit descriptors are used for I/O of integer and real data. The following general rules apply to all three of them:

> On input, leading blanks are not significant. Other blanks are interpreted differently depending on the BN or BZ flag in effect, but all blank fields always become the value 0. Plus signs are optional.
>
> On input, with F and E editing, an explicit decimal point appearing in the input field overrides the edit descriptor specification of the decimal point position.
>
> On output, the characters generated are right justified in the field with padding leading blanks if necessary.
>
> On output, if the number of characters produced exceeds the field width or the exponent exceeds is specified width, the entire field is filled with asterisks.

11.3.2.2. I (Integer Editing).

The edit descriptor Iw must be associated with an iolist item which is of type integer. The field width is w characters in length. On input, an optional sign may appear in the field. The general rules of Section 11.3.2.1 apply to the I edit descriptor.

11.3.2.3. F (Real Editing).

The edit descriptor Fw.d must be associated with an iolist item which is of type real. The width of the field is w positions, the fractional part of which consists of d digits. The input field begins with an optional sign followed by a string of digits optionally containing a decimal point. If the decimal point is present, it overrides the d specified in the edit descriptor, otherwise the rightmost d digits of the string are interpreted as following the decimal point (with leading blanks converted to zeros if necessary). Following this is an optional exponent which is one of

plus or minus followed by an integer, or

E or D followed by zero or more blanks followed by an optional sign followed by an integer (E and D are treated identically).

The output field occupies w digits, d of which fall beyond the decimal point, and the value output is controlled both by the iolist item and the current scale factor. The output value is rounded rather than truncated.

The general rules of Section 11.3.2.1 apply to the F edit descriptor.

11.3.2.4. E (Real Editing).

An E edit descriptor either takes the form Ew.d or Ew.dEe. In either case the field width is w characters. The e has no effect on input. The input field for an E edit descriptor is identical to that described by an F edit descriptor with the same w and d. The form of the output field depends on the scale factor (set by the P edit descriptor) which is in effect. For a scale factor of 0, the output field is a minus sign (if necessary), followed by a decimal point, followed by a string of digits, followed by an exponent field for exponent, exp, of one of the following forms:

Ew.d -99 <= exp <= 99
E followed by plus or minus followed by the two digit exponent.

Ew.d -999 <= exp <= 999
Plus or minus followed by three digit exponent.

Ew.dEe -((10**e) - 1) <= exp <= (10**e) -1
E followed by plus or minus followed by e digits which are the exponent with possible leading zeros.

The form Ew.d must not be used if the absolute value of the exponent to be printed exceeds 999.

The scale factor controls the decimal normalization of the printed E field. If the scale factor, k, is in the range -d < k <= 0 then the output field contains exactly -k leading zeros after the decimal point and d + k significant digits after this. If 0 < k < d+2 then the output field contains exactly k significant digits to the left of the decimal point and d - k - 1 places after the decimal point. Other values of k are errors.

The general rules of Section 11.3.2.1 apply to the E edit descriptor.

11.3.2.5. L (Logical Editing).

The edit descriptor is Lw, indicating that the field width is w characters. The iolist element which becomes associated with an L edit descriptor must be of type logical. On input, the field consists of optional blanks, followed by an optional decimal point, followed by a T (for .TRUE.) or and F (for .FALSE.). Any further characters in the field are ignored, but accepted on input, so that .TRUE. and .FALSE. are valid inputs. On output, w - 1 blanks are followed by either T or F as appropriate.

11.3.2.6. A (Character Editing).

The forms of the edit descriptor are A or Aw. If w is not present, the number of characters in the iolist item with which it becomes associated determines the length (an implicit w). The iolist item must be of character type if it is to be associated with an A or Aw edit descriptor. On input, if w exceeds or equals the number of characters in the iolist element, the rightmost characters of the input field are used as the input characters, otherwise the input characters are left justified in the input iolist item and trailing blanks are provided. On output, if w should exceed the characters produced by the iolist item, leading blanks are provided, otherwise, the leftmost w characters of the iolist item are output.

CHAPTER 12

Programs, Subroutines and Functions

This chapter describes the format of program units. A program unit is either a main program, a subroutine, or a function program unit. The term procedure is used to refer to either a function or a subroutine. This chapter also describes the CALL and RETURN statements as well as function calls.

12.1. Main Program.

A main program is any program unit that does not have a FUNCTION or SUBROUTINE statement as its first statement. It may have a PROGRAM statement as its first statement. The execution of a program always begins with the first executable statement in the main program. Consequently, there must be precisely one main program in every executable program. The form of a PROGRAM statement is:

PROGRAM pname

where: 'pname' is a user defined name that is the name of the main program.

The name 'pname' is a global name. Therefore, it cannot be the same as another external procedure's name or a common block's name. It is also a local name to the main program, and must not conflict with any local name in the main program. The PROGRAM statement may only appear as the first statement of a main program.

12.2. Subroutines.

A subroutine is a program unit that can be called from other program units by a CALL statement. When envoked, it performs the set of actions defined by its executable statements, and then returns control to the statement immediately following the statement that called it. A subroutine does not directly return a value, although values can be passed back to the calling program unit via parameters or common variables.

12.2.1. SUBROUTINE Statement.

A subroutine begins with a SUBROUTINE statement, and ends with the first following END statement. It may contain any kind of statement other than a PROGRAM statement or a FUNCTION statement. The form of a SUBROUTINE statement is:

SUBROUTINE sname [([farg [, farg]...])]

'sname' is the user defined name of the subroutine.

'farg' is a user defined name of a formal argument.

The name 'sname' is a global name, and it is also local to the subroutine it names. The list of argument names defines the number and, with any subsequent IMPLICIT, type, or DIMENSION statements, the type of arguments to that subroutine. Argument names cannot appear in COMMON, DATA, EQUIVALENCE, or INTRINSIC statements.

12.2.2. CALL Statement.

A subroutine is executed as a consequence of executing a CALL statement in another program unit that references that subroutine. The form of a CALL statement is:

CALL sname [([arg [,arg]...])]

'sname' is the name of a subroutine.

'arg' is an actual argument.

An actual argument may be either an expression or the name of an array. The actual arguments in the CALL statement must agree in type and number with the corresponding formal arguments specified in the SUBROUTINE statement of the referenced subroutine. If there are no arguments in the SUBROUTINE statement, then a CALL statement referencing that subroutine must not have any actual arguments, but may optionally have a matched pair of parentheses following the name of the subroutine. Note that a formal argument may be used as an actual argument in another subprogram call.

Execution of a CALL statement proceeds as follows: All arguments that are expressions are evaluated. All actual arguments are associated with their corresponding formal arguments, and the body of the specified subroutine is executed. Control is returned to the statement following the CALL statement upon exiting the subroutine, by executing either a RETURN statement or an END statement in that subroutine.

A subroutine specified in any program unit may be called from any other program unit within the same executable program. Recursive subroutine calls, however, are not allowed in FORTRAN. That is, a subroutine cannot call itself directly, nor can it call another subroutine that will result in the first subroutine being called

again before it returns control to its caller.

12.3. Functions.

A function is referenced in an expression and returns a value that is used in the computation of that expression. There are three kinds of functions: external functions, intrinsic functions, and statement functions. This section describes the three kinds of functions.

A function reference may appear in an arithmetic expression. Execution of a function reference causes the function to be evaluated, and the resulting value is used as an operand in the containing expression. The form of a function reference is:

fname ([arg [,arg]...])

'fname' is the name of an external, intrinsic, or statement function.

'arg' is an actual argument.

An actual argument may be an arithmetic expression or an array. The number of actual arguments must be the same as in the definition of the function, and the corresponding types must agree.

12.3.1. External Functions.

An external function is specified by a function program unit. It begins with a FUNCTION statement and ends with an END statement. It may contain any kind of statement other than a PROGRAM statement or a SUBROUTINE statement. The form of a FUNCTION statement is:

[type] FUNCTION fname ([farg [, farg]...])

'type' is one of INTEGER, REAL, or LOGICAL.

'fname' is the user defined name of the function.

'farg' is a formal argument name.

The name 'fname' is a global name, and it is also local to the function it names. If no type is present in the FUNCTION statement, the function's type is determined by default and any subsequent IMPLICIT or type statements that would determine the type of an ordinary variable. If a type is present, then the function name cannot appear in any additional type statements. In any case, an external

function cannot be of type character. The list of argument names defines the number and, with any subsequent IMPLICIT, type, or DIMENSION statements, the type of arguments to that subroutine. Neither argument names nor 'fname' can appear in COMMON, DATA, EQUIVALENCE, or INTRINSIC statements.

The function name must appear as a variable in the program unit defining the function. Every execution of that function must assign a value to that variable. The final value of this variable, upon execution of a RETURN or of an END statement, defines the value of the function. After being defined, the value of this variable can be referenced in an expression, exactly like any other variable. An external function may return values in addition to the value of the function by assignment to one or more of its formal arguments.

12.3.2. Intrinsic Functions.

Intrinsic functions are functions that are predefined by the FORTRAN compiler and are available for use in a FORTRAN program. Table 12.1 gives the name, definition, number of parameters, and type of the intrinsic functions available in UCSD System FORTRAN 77. An IMPLICIT statement does not alter the type of an intrinsic function. For those intrinsic functions that allow several types of arguments, all arguments in a single reference must be of the same type.

All intrinsic functions used in a program unit must appear in an INTRINSIC statement.

An intrinsic function name may appear in an INTRINSIC statement, but only those intrinsic functions listed in table 12.1 may do so. An intrinsic function name also may appear in a type statement, but only if the type is the same as the standard type of that intrinsic function.

Arguments to certain intrinsic functions are limited by the definition of the function begin computed. For example, the logarithm of a negative number is undefined, and therefore not allowed.

12.3.3. Statement Functions.

A statement function is a function that is defined by a single statement. It is similar in form to an assignment statement. A statement function statement can only appear after the specification statements and before any executable statements in the program unit in which it appears. A statement function is not an executable statement; since it is not executed in order as the first statement in its particular program unit. Rather, the body of a statement function serves to define the meaning of the statement function. It is executed, as any other function, by the execution of a function reference. The form of a statement

function is:

fname ([arg [, arg]...]) = expr

'fname' is the name of the statement function.

'arg' is a formal argument name.

'expr' is an expression.

The type of the 'expr' must be assignment compatible with the type of the statement function name. The list of formal argument names serves to define the number and type of arguments to the statement function. The scope of formal argument names is the statement function. Therefore, formal argument names may be used as other user defined names in the rest of the program unit enclosing the statement function definition. The name of the statement function, however, is local to the enclosing program unit, and must not be used otherwise (except as the name of a common block, or as the name of a formal argument to another statement function). The type of all such uses, however, must be the same. If a formal argument name is the same as another local name, then a reference to that name within the statement function defining it always refers to the formal argument, never to the other usage.

Within the expression 'expr', references to variables, formal arguments, other functions, array elements, and constants are allowed. Statement function references, however, must refer to statement functions that have been defined prior to the statement function in which they appear. Statement functions cannot be recursively called, either directly or indirectly.

A statement function can only be referenced in the program unit in which it is defined. The name of a statement function cannot appear in any specification statement, except in a type statement which may not define that name as an array, and in a COMMON statement as the name of a common block. A statement function cannot be of type character.

12.4. RETURN Statement.

A RETURN statement causes return of control to the calling program unit. It may only appear in a function or subroutine. The form of a RETURN statement is:

RETURN

Execution of a RETURN statement terminates the execution of the enclosing subroutine or function. If the RETURN statement is in a function, then the value

of that function is equal to the current value of the variable with the same name as the function. Execution of an END statement in a function or subroutine is equivalent to execution of a RETURN statement.

12.5. Parameters.

This section discusses the relationship between formal and actual arguments in a function or subroutine call. A formal argument refers to the name by which the argument is known within the function or subroutine, and an actual argument is the specific variable, expression, array, etc., passed to the procedure in question at any specific calling location.

Arguments are used to pass values into and out of procedures. Variables in common can be used to perform this task as well. The number of actual arguments must be the same as formal arguments, and the corresponding types must agree.

On entry to a subroutine or function, the actual arguments become associated with the formal arguments, much as an EQUIVALENCE statement associates two or more arrays or variables, and COMMON statements in two or more program units associate lists of variables. This association remains in effect until execution of the subroutine or function is terminated. Thus, assigning a value to a formal argument during execution of a subroutine or function may alter the value of the corresponding actual argument. If an actual argument is a constant, function reference, or an expression other than a simple variable, assigning a value to the corresponding formal argument is not allowed, and may have some strange side effects. In particular, assigning a value to a formal argument of type character, when the actual argument is a literal, can be disastrous.

If an actual argument is an expression, it is evaluated immediately prior to the association of formal and actual arguments. If an actual argument is an array element, its subscript expression is evaluated just prior to the association, and remains constant throughout the execution of the procedure, even if it contains variables that are redefined during the execution of the procedure.

A formal argument that is a variable may be associated with an actual argument that is a variable, an array element, or an expression.

A formal argument that is an array may be associated with an actual argument that is an array or an array element. The number and size of dimensions in a formal argument may be different than those of the actual argument, but any reference to the formal array must be within the limits of the storage sequence in the actual array. While a reference to an element outside these bounds is not detected as an error in a running FORTRAN program, the results are unpredictable.

Table 12.1 Intrinsic Functions

Intrinsic Function	Definition	No. Args	Name	Type of Argument	Type of Function
Type Conversion	Conversion to Integer int(a) See Note 1	1	INT IFIX	Real Real	Integer Integer
	Conversion to Real See Note 2	1	REAL FLOAT	Integer Integer	Real Real
	Conversion to Integer See Note 3	1	ICHAR	Character	Integer
	Conversion to Character	1	CHAR	Integer	Character
Truncation	int(a) See Note 1	1	AINT	Real	Real
Nearest Whole Number	int(a.5) a>=0 int(a.5) a<0	1	ANINT	Real	Real
Nearest Integer	int(a.5) a>=0 int(a.5) a<0	1	NINT	Real	Integer
Absolute Value	a	1 1	IABS ABS	Integer Real	Integer Real
Remaindering	alint(a1/a2)*a2 See Note 1	2	MOD AMOD	Integer Real	Integer Real
Transfer of Sign	a1 if a2>=0 a1 if a2<0	2	ISIGN SIGN	Integer Real	Integer Real
Positive Difference	a1a2 if a1>a2 0if a1<=a2	2	IDIM DIM	Integer Real	Integer Real

Intrinsic Function	Definition	No. Args	Name	Type of Argument	Function
Choosing Largest Value	max(a1,a2,...)	>=2	MAX0 AMAX1 AMAX0 MAX1	Integer Real Integer Real	Integer Real Real Integer
Choosing Small est Value	min(a1,a2,...)	>=2	MIN0 AMIN1 AMIN0 MIN1	Integer Real Integer Real	Integer Real Real Integer
Square Root	a**0.5	1	SQRT	Real	Real
Exponential	e**a	1	EXP	Real	Real
Natural Logarithm	log(a)	1	ALOG	Real	Real
Common Logarithm	log10(a)	1	ALOG10	Real	Real
Sine	sin(a)	1	SIN	Real	Real
Cosine	cos(a)	1	COS	Real	Real
Tangent	tan(a)	1	TAN	Real	Real
Arcsine	arcsin(a)	1	ASIN	Real	Real
Arccosine	arccos(a)	1	ACOS	Real	Real
Arctangent	arctan(a) arctan(a1/a2)	1 2	ATAN ATAN2	Real Real	Real Real
Hyperbolic Sine	sinh(a)	1	SINH	Real	Real
Hyperbolic Cosine	cosh(a)	1	COSH	Real	Real
Hyperbolic Tangent	tanh(a)	1	TANH	Real	Real
Lexically Greater Than or Equal	a1 >= a2 See Note 4	2	LGE	Character	Logical

Intrinsic Function	Definition	No. Args	Name	Type of Argument	Type of Function
Lexically Greater Than	a1 > a2 See Note 4	2	LGT	Character	Logical
Lexically Less Than or Equal	a1 <= a2 See Note 4	2	LLE	Character	Logical
Lexically Less Than	a1 < a2 See Note 4	2	LLT	Character	Logical
End of File	End_Of_File(a) See Note 5	1	EOF	Integer	Logical

Table 12.1 Notes

(1) For a of type real, if a >= 0 then int(a) is the largest integer not greater than a, if a < 0 then int(a) is the most negative integer not less than a. IFIX(a) is the same as INT(a).

(2) For a of type integer, REAL(a) is to the greatest possible precision. This varies from processof to processor. FLOAT(a) is the same as REAL(a).

(3) ICHAR converts a character value into an integer value. The integer value of a character is the ASCII internal representation of that character, and is in the range 0 to 127. For any two characters, c1 and c2, (c1 .LE. c1) is .TRUE. if and only if (ICHAR(c1) .LE. ICHAR(c2)) is .TRUE..

(4) LGE(a1,a2) returns the value .TRUE.. if a1 = a2 or if a1 follows a2 in the ASCII collating sequence. Otherwise it returns .FALSE..

LGT(a1,a2) returns .TRUE. if a1 follows a2 in the ASCII collating sequence, otherwise it returns .FALSE..

LLE(a1,a2) returns .TRUE. if a1 = a2 or if a1 precedes a2 in the ASCII collating sequence, otherwise it returns .FALSE..

LLT(a1,a2) returns .TRUE. if a1 precedes a2 in the ASCII collating sequence, otherwise it returns .FALSE..

The operands or LGE, LGT, LLE, and LLT must be of the same length.

(5) EOF(a) returns the value .TRUE. if the unit specified by its argument is at or

past the end of file record, otherwise it returns .FALSE.. The value of a must correspond to an open file, or to zero (which indicates CONSOLE:).

(6) All angles are expressed in radians.

(7) All arguments in an intrinsic function reference must be of the same type.

CHAPTER 13

Compilation Units

This chapter describes the relationship between FORTRAN and the UCSD Pascal segment mechanism. In normal use, the user need not be aware of such intricacies. However, if the user desires to interface FORTRAN with Pascal, to create overlays, or to take advantage of separate compilation or libraries, the details contained here are helpful. This chapter consists of the following sections:

13.1. Units, Segments, Partial Compilation, and FORTRAN.
13.2. The $USES Compiler Directive.
13.3. Linking Pascal and FORTRAN.
13.4. The $EXT Compiler Directive.

The first section discusses the general form of a FORTRAN program in terms of the UCSD operating system object code structure. The next section describes the $USES compiler directive. This directive provides access libraries or already compiled procedures, and provides overlays in FORTRAN. The next section describes how one links FORTRAN with Pascal. The final section explains the $EXT compiler directive.

13.1. Units, Segments, Partial Compilation, and FORTRAN.

If a FORTRAN compilation contains no main procedure, then it is output as if it were a Pascal unit compilation. The unit is given the name 'U' followed by the name of its first procedure. For example:

```
C -- No PROGRAM statement present
      SUBROUTINE X
      ...
      END
      SUBROUTINE Y
      ...
      END
      ...
      SUBROUTINE Z
      ...
      END
```

would be compiled into a single unit named 'UX'. (Assume for later examples that the object code is output to file 'X.CODE'.) All procedures called from within unit UX must be defined within unit UX, unless a $USES or a $EXT statement has shown them to reside in another unit. Similarly, procedures in unit UX cannot be called from other units unless the other units contain a $USES UX statement. Thus, a typical main program that would call X might be:

```
C
C -- This is the main program BIGGIE
C

$USES UX IN X.CODE

      PROGRAM BIGGIE
      ...
      CALL X
      ...
      END
      SUBROUTINE W
      ...
      CALL Y
      ...
      END
```

If the $USES statement were not present, the FORTRAN compiler would expect subroutines X and Y to appear in the same compilation, somewhere after subroutine W. Assume that the object code for this compilation is output to the file 'BIGGIE.CODE'.

Thus, the user can create libraries of functions, partial compilations, etc., and save compilation time and disk space, by a simple use of the $USES statement. For more inforation on the $USES statement, see the section on the $USES statement.

13.2. The $USES Compiler Directive.

The $USES compiler directive provides several distinct functions to the user. It allows procedures and functions in separately compiled units, such as the system library, to be called from FORTRAN. It provides the user a relatively secure form of separate compilation for FORTRAN compilations. It allows the user to call Pascal routines that have been compiled into Pascal units.

The format of the $USES control statement is:

$USES unitname [IN filename] [OVERLAY]

where: 'unitname' is the name of a unit.

'filename' is a valid UCSD file name.

As with all $ control statements, the $ must appear in column one. This compiler directive directs the compiler to open the .CODE file 'filename', locate the unit

'unitname', and process the INTERFACE information associated with that unit, generating a reasonable FORTRAN equivalent declaration for the FORTRAN compilation in progress. All $USES commands must appear before any FORTRAN statements, specification or executable, but they are allowed to follow comment lines and other $ control lines. If the optional 'IN filename' is present, the name 'filename' is used as the file to process. If it is not, the file '*SYSTEM.LIBRARY' is used as a default. The optional field OVERLAY has no effect on program execution, and is included in version IV.0 only for compatibility with version II.0.

Warning: If a FORTRAN main program $USES a Pascal unit, any global variables in the INTERFACE part of that unit will not be accessible from FORTRAN. See section 13.3 for further information.

13.2.1. Separate Compilation.

Separate compilation is accomplished by compiling a set of subroutines and functions without any main program. Each such compilation creates a code file containing a single UCSD unit. Then, when the main program is compiled, possibly along with many subroutines or functions, it $USES the separately compiled units. The routines compiled with the main program obtain the correct definition of each externally compiled procedure through the $USES directive.

In the simplest form, when no $USES statements appear in any of the separate compilations, the user simply $USES all separately compiled FORTRAN units in the main program. However, this limits the procedure calls in each of the separately compiled units to procedures defined in the same unit. If there are calls to procedures in unit A from unit B, then unit B must contain a $USES A statement. The main program must then contain a $USES A statement as its first $USES statement, followed by a $USES B statement. This is necessary for the compiler to get the unit numbers allocated consistently.

In more complicated cases, the user must insure that all references to procedures in outside units are preceded by the proper $USES statement in the same order, and not missing any units. If unit B $USES unit A, and unit C $USES unit B, then unit C must first $USES unit A. Likewise, if units D and E both $USES unit F, they both must contain exactly the same $USES statements prior to the $USES F statement.

13.3. Linking Pascal and FORTRAN.

In order to call Pascal routines form FORTRAN, the Pascal routines must first be compiled into a Pascal unit. The FORTRAN program can then $USES that unit. Unfortunately, the exceedingly rich type structure present in Pascal is not present in FORTRAN. Also, the I/O systems of FORTRAN and Pascal are not compatible.

Therefore it is not possible to do everything one might desire. This section does, however, help the user do what is possible in interfacing the two languages.

It is not generally possible to do I/O from Pascal routines called from a main program that is written in FORTRAN. Normal Pascal I/O to and from the console, however, can always be done from Pascal routines providing that there is no file name in the I/O statement. The Pascal routines RESET, REWRITE, CLOSE, etc., should not be called from Pascal routines running under a FORTRAN main program.

It is possible to do I/O from a FORTRAN procedure that is called from a Pascal main program. In general, however, this practice should be avoided. This section is provided to allow the user who absolutely must mix I/O operations from both languages to do what is possible. While the following information is believed to be correct, it is neither warranted to work nor guaranteed to remain valid in future releases. Again, mixed I/O is not supported. It is done at the user's risk.

There are several precautions that the user must take for FORTRAN I/O to work from Pascal programs. The FORTRAN I/O procedures use the heap for the allocation of file related storage, so the user should not force the deallocation of heap memory via calls to MARK and/or RELEASE. Other restrictions may apply in special cases. As stated above, one should avoid doing I/O from both FORTRAN and Pascal in the same program as the two systems are not totaly compatible.

Since there are Pascal types that have no FORTRAN equivalent, the way FORTRAN looks at Pascal parameters is somewhat limited. FORTRAN does recognize both reference and value parameters when calling Pascal subroutines. The following table shows how FORTRAN views Pascal declarations:

Pascal Declaration:	FORTRAN's View:
CONST anything ... ;	Ignored.
TYPE anything ... ;	Ignored.
VAR anything ... ;	Ignored.
PROCEDURE X(arg-list);	SUBROUTINE X(arg-list)
FUNCTION X(arg-list): type;	type FUNCTION X(arg-list) Note: type must be INTEGER, LOGICAL, or REAL.
type:	
REAL	REAL
BOOLEAN	LOGICAL
CHAR	CHARACTER*1
STRING or PACKED ARRAY of CHAR	CHARACTER*n 1 <= n <= 127
any other identifier	INTEGER
arg-list:	
(VAR I,J: type)	(I,J) type I,J
(I,J: type)	*** There is no proper FORTRAN equivalent to value parameters, but the FORTRAN compiler does generate the correct calling sequence for Pascal routines with value parameters.

Likewise, when the INTERFACE information for a FORTRAN program is output, it must be mapped onto Pascal declarations. The following table gives the corresponding declarations:

FORTRAN Declaration:	Pascal's View:
SUBROUTINE X(arg-list)	PROCEDURE X(arg-list);
type FUNCTION X(arg-list)	FUNCTION X(arg-list): type;
type:	
INTEGER	INTEGER
REAL	REAL
LOGICAL	BOOLEAN
CHARACTER*n	CHAR n = 1
	STRING or PACKED ARRAY of CHAR 2 <= n <= 127

arg-list:	
(1)	(VAR I: type)
type 1	

Note: When a Pascal compilation USES a FORTRAN unit, it is the responsibility of the Pascal program to make sure that any needed type declarations for the ALFAn types are properly defined. This cannot consistently be done by FORTRAN as it would lead to duplicate type definitions should a user use two FORTRAN units in which each declares the same type. There is another point that must be made for Pascal programs that call FORTRAN subroutines. If the subroutine has a REAL parameter that is in actuality an array, the Pascal program must pass a scalar instead of an array. This should not be a problem. Since the Pascal program can pass the first element of the array, and all FORTRAN parameters are reference parameters, the FORTRAN subroutine has access to the whole array. The user is cautioned to remember that Pascal stores its arrays in row- major order, while FORTRAN stores them in column-major order.

When a FORTRAN program $USES a Pascal unit, the interface section variables in that Pascal unit are not accessible from FORTRAN.

13.4. The $EXT Compiler Directive.

The $EXT compiler directive is used when one desires to call assembly language routines, or routines in $SEPARATE FORTRAN or Pascal units, from a FORTRAN 77 routine. The form of the $EXT directive is:

$EXT { SUBROUTINE / [type] FUNCTION } procname #params

where: 'type' is either INTEGER, LOGICAL, or REAL,

'procname' is the name of the subroutine or function, and

'#params' is an integer equal to the number of parameters that this procedure requires.

This directive must appear before any FORTRAN statements, either specification or executable, but may follow comment lines or other $ compiler directives. All parameters are passed by reference (called VAR parameters if Pascal) to procedures defined by the $EXT directive. It is up to the user to follow this convention, as the linker does not enforce it. The linker does, however, check the number of parameters.

APPENDIX A - Differences Between SofTech Microsystems FORTRAN 77 and ANSI Standard Subset FORTRAN 77

This appendix is directed at the reader who is familiar with the ANSI Standard FORTRAN 77 Subset language as defined in ANSI X3.9-1978. It concisely describes how SofTech Microsystems FORTRAN 77 differs from the standard language. The differences fall into three general categories:

A.1. Unsupported Features.
A.2. Full-Language Features.
A.3. Extensions to Standard.

A.1. Unsupported Features.

There are two significant places where SofTech Microsystems FORTRAN 77 does not comply with the standard. One is that procedures cannot be passed as parameters and the other is that INTEGER and REAL data types do not occupy the same amount of storage. Both differences are due to limitations of the UCSD P-machine architecture.

Parametric procedures are not supported simply because there is no practical way to do so in the USCD P-machine. The instruction set does not allow the loading of a procedure's address onto the stack, and more significantly, does not provide for the calling of a procedure whose address is on the stack.

REAL variables require 4 bytes of storage while INTEGER and LOGICAL variables only require 2 bytes. This is due to the fact that the UCSD P-machine supported operations on those types are implemented in those sizes.

A.2. Full-Language Features.

There are several features from the full language that have been included in this implementation for a variety of reasons. Some were done at either minimal or zero cost, such as allowing arbitrary expressions in subscript calculations. Others were included because it was felt that they would significantly increase the utility of the implementation, especially in an engineering or laboratory application. An example is the generalized I/O that allows easier control of peripherals. In all cases, a program which is written to comply with the subset restrictions will compile and execute properly, since the full language properly includes the subset constructs. A short description of full language features included in the implementation follows.

Subscript Expressions - The subset does not allow function calls or array element references in subscript expressions, but the full language and this implementation do.

Do Variable Expressions - The subset restricts expressions that define the limits of a DO statement, but the full language does not. SofTech Microsystems FORTRAN also allows full integer expressions in DO statement limit computations. Similarly, arbitrary integer integer expressions are allowed in implied DO loops associated with READ and WRITE statements.

Unit I/O Number - SofTech Microsystems FORTRAN allows an I/O unit to be specified by an integer expression, as does the full language.

Expressions in I/O list - The subset does not allow expressions to appear in an I/O list whereas the full language does allow expressions in the I/O list of a WRITE statement. SofTech Microsystems FORTRAN allows expressions in the I/O list of a WRITE statement, providing that they do not begin with an initial left parenthesis.

NOTE: the expression (A+B)*(C+D) can be specified in an output list as +(A+B)*(C+D) which, incidently, does not generate any extra code to evaluate the leading +.

Expression in computed GOTO - SofTech Microsystems FORTRAN allows an expression for the value of a computed GOTO, consistent with the full language rather than the subset language.

Generalized I/O - SofTech Microsystems FORTRAN allows both sequential and direct access files to be either formatted or unformatted. The subset language restricts direct access files to be unformatted, and sequential files to be formatted. SofTech Microsystems FORTRAN also contains an augmented OPEN statement which takes additional parameters that are not included in the subset. There is also a form of the CLOSE statement, which is not included at all in the subset. I/O is described in more detail in Chapters 10 and 11.

A.3. Extensions to Standard.

The language implemented has several minor extensions to the full language standard. These are briefly described below:

Compiler Directives - Compiler directives have been added to allow the programmer to communicate certain information to the Compiler. An additional kind of line, called a Compiler directive line, has been added. It is characterized by a dollar sign '$' appearing in column 1. A Compiler directive line may appear any place that a comment line can appear, although certain directives are restricted to appear in certain places. A Compiler directive line is used to convey certain compile-time information to the System about the nature of the current compilation. The set of directives is briefly listed below:

$INCLUDE filename

Include textually the file 'filename' at this point in the source. Nested includes are implemented to a depth of nesting of five files. Thus, for example, a program may include various files with subprograms, each of which includes various files which describe common areas (which would be a depth of nesting of three files).

$USES ident
    [IN filename]
    [OVERLAY]

Similar to a USES command in the UCSD Pascal Compiler. The already compiled FORTRAN subroutines or Pascal procedures contained in the .CODE file 'filename', or in the file '*SYSTEM.LIBRARY' (if no file name is present), become callable from the currently compiling code. This directive must appear before the initial non- comment input line. For more details, see Chapter 13.

$XREF

Produce a cross-reference listing at the end of each procedure compiled.

$EXT SUBROUTINE name #parms
or
$EXT [type] FUNCTION name #params

The subroutine or function named 'name' is either an assembly language routine or a routine in a $SEPARATE unit (either FORTRAN or Pascal). The routine has exactly '#params' reference parameters.

Backslash Edit Control - The edit control character \ can be used in formats to inhibit the normal advancement to the next record which is associated with the completion of a READ or a WRITE statement. This is particularly useful when prompting to an interactive device, such as CONSOLE:, so that a response can be on the same line as the prompt.

End of File Intrinsic Function - An intrinsic function, EOF, has been provided. The function accepts a unit specifier as an argument and returns a logical value which indicates whether the specified unit is at its end of file.

Lower Case Input - Upper and lowercase source input is allowed. In most

contexts, lowercase characters are treated as indistinguishable from their uppercase counterparts. Lower case is significant in character constants and Hollerith fields.

Appendix B - FORTRAN Error Messages

B.1. Compile-Time Error Messages.

1 Fatal error reading source block
2 Nonnumeric characters in label field
3 Too many continuation lines
4 Fatal end of file encountered
5 Labeled continuation line
6 Missing field on $ compiler directive line
7 Unable to open listing file specified on $ compiler directive line
8 Unrecognizable $ compiler directive
9 Input source file not valid textfile format
10 Maximum depth of include file nesting exceeded
11 Integer constant overflow
12 Error in real constant
13 Too many digits in constant
14 Identifier too long
15 Character constant extends to end of line
16 Character constant zero length
17 Illegal character in input
18 Integer constant expected
19 Label expected
20 Error in label
21 Type name expected (INTEGER, REAL, LOGICAL, or CHARACTER[*n])
22 Integer constant expected
23 Extra characters at end of statement
24 '(' expected
25 Letter IMPLICIT'ed more than once
26 ')' expected
27 Letter expected
28 Identifier expected
29 Dimension(s) required in DIMENSION statement
30 Array dimensioned more than once
31 Maximum of 3 dimensions in an array
32 Incompatible arguments to EQUIVALENCE
33 Variable appears more than once in a type specification statement
34 This identifier has already been declared
35 This intrinsic function cannot be passed as an argument
36 Identifier must be a variable
37 Identifier must be a variable or the current FUNCTION
38 '/' expected
39 Named COMMON block already saved
40 Variable already appears in a COMMON block

41 Variables in two different COMMON blocks cannot be equivalenced
42 Number of subscripts in EQUIVALENCE statement does not agree with variable declaration
43 EQUIVALENCE subscript out of range
44 Two distinct cells EQUIVALENCE'd to the same location in a COMMON block
45 EQUIVALENCE statement extends a COMMON block in the negative direction
46 EQUIVALENCE statement forces a variable to two distinct locations, not in a COMMON block
47 Statement number expected
48 Mixed CHARACTER and numeric items not allowed in same COMMON block
49 CHARACTER items cannot be EQUIVALENCE'd with non-character items
50 Illegal symbol in expression
51 Can't use SUBROUTINE name in an expression
52 Type of argument must be INTEGER or REAL
53 Type of argument must be INTEGER, REAL, or CHARACTER
54 Types of comparisons must be compatible
55 Type of expression must be LOGICAL
56 Too many subscripts
57 Too few subscripts
58 Variable expected
59 '=' expected
60 Size of EQUIVALENCE'd CHARACTER items must be the same
61 Illegal assignment - types do not match
62 Can only call SUBROUTINES
63 Dummy parameters cannot appear in COMMON statements
64 Dummy parameters cannot appear in EQUIVALENCE statements
65 Assumed-size array declarations can only be used for dummy arrays
66 Adjustable-size array declarations can only be used for dummy arrays
67 Assumed-size array dimension specifier must be last dimension
68 Adjustable bound must be either parameter or in COMMON prior to appearance
69 Adjustable bound must be simple integer variable
70 Cannot have more than 1 main program
71 The size of a named COMMON must be the same in all procedures
72 Dummy arguments cannot appear in DATA statements
73 COMMON variables cannot appear in DATA statements
74 SUBROUTINE names, FUNCTION names, INTRINSIC names, etc. cannot appear in DATA statements
75 Subscript out of range in DATA statement

76 Repeat count must be >= 1
77 Constant expected
78 Type conflict in DATA statement
79 Number of variables does not match number of values in DATA statement list
80 Statement cannot have label
81 No such INTRINSIC function
82 Type declaration for INTRINSIC function does not match actual type of INTRINSIC function
83 Letter expected
84 Type of FUNCTION does not agree with a previous call
85 This procedure has already appeared in this compilation
86 This procedure has already been defined to exist in another unit via a $USES command
87 Error in type of argument to an INTRINSIC FUNCTION
88 SUBROUTINE/FUNCTION was previously used as a FUNCTION/SUBROUTINE
89 Unrecognizable statement
90 Functions cannot be of type CHARACTER
91 Missing END statement
92 A program unit cannot appear in a $SEPARATE compilation
93 Fewer actual arguments than formal arguments in FUNCTION/SUBROUTINE call
94 More actual arguments than formal arguments in FUNCTION/SUBROUTINE call
95 Type of actual argument does not agree with type of format argument
96 The following procedures were called but not defined:
97 This procedure was already defined by a $EXT directive
98 Maximum size of type CHARACTER is 255, minimum is 1

100 Statement out of order
101 Unrecognizable statement
102 Illegal jump into block
103 Label already used for FORMAT
104 Label already defined
105 Jump to format label
106 DO statement forbidden in this context
107 DO label must follow DO statement
108 ENDIF forbidden in this context
109 No matching IF for this ENDIF
110 Improperly nested DO block in IF block
111 ELSEIF forbidden in this context
112 No matching IF for ELSEIF
113 Improperly nested DO or ELSE block
114 '(' expected

115 ')' expected
116 THEN expected
117 Logical expression expected
118 ELSE statement forbidden in this context
119 No matching IF for ELSE
120 Unconditional GOTO forbidden in this context
121 Assigned GOTO forbidden in this context
122 Block IF statement forbidden in this context
123 Logical IF statement forbidden in this context
124 Arithmetic IF statement forbidden in this context
125 ',' expected
126 Expression of wrong type
127 RETURN forbidden in this context
128 STOP forbidden in this context
129 END forbidden in this context

131 Label referenced but not defined
132 DO or IF block not terminated
133 FORMAT statement not permitted in this context
134 FORMAT label already referenced
135 FORMAT must be labeled
136 Identifier expected
137 Integer variable expected
138 'TO' expected
139 Integer expression expected
140 Assigned GOTO but no ASSIGN statements
141 Unrecognizable character constant as option
142 Character constant expected as option
143 Integer expression expected for unit designation
144 STATUS option expected after ',' in CLOSE statement
145 Character expression as filename in OPEN
146 FILE= option must be present in OPEN statement
147 RECL= option specified twice in OPEN statement
148 Integer expression expected for RECL= option in OPEN statement
149 Unrecognizable option in OPEN statement
150 Direct access files must specify RECL= in OPEN statement
151 Adjustable arrays not allowed as I/O list elements
152 End of statement encountered in implied DO, expressions beginning with '(' not allowed as I/O list elements
153 Variable required as control for implied DO
154 Expressions not allowed as reading I/O list elements
155 REC= option appears twice in statement
156 REC= expects integer expression
157 END= option only allowed in READ statement
158 END= option appears twice in statement

159 Unrecognizable I/O unit
160 Unrecognizable format in I/O statement
161 Options expected after ',' in I/O statement
162 Unrecognizable I/O list element
163 Label used as format but not defined in format statement
164 Integer variable used as assigned format but no ASSIGN statements
165 Label of an executable statement used as a format
166 Integer variable expected for assigned format
167 Label defined more than once as format

200 Error in reading $USES file
201 Syntax error in $USES file
202 SUBROUTINE/FUNCTION name in $USES file has already been declared
203 FUNCTIONS cannot return values of type CHARACTER
204 Unable to open $USES file
205 Too many $USES statements
206 No .TEXT info for this unit in $USES file
207 Illegal segment kind in $USES file
208 There is no such unit in this $USES file
209 Missing UNIT name in $USES statement
210 Extra characters at end of $USES directive
211 Intrinsic units cannot be overlayed
212 Syntax error in $EXT directive
213 A SUBROUTINE cannot have a type
214 SUBROUTINE/FUNCTION name in $EXT directive has already been defined

400 Code file write error
401 Too many entries in JTAB
402 Too many SUBROUTINES/FUNCTIONS in segment
403 Procedure too large (code buffer too small)
404 Insufficient room for scratch file on system disk
405 Read error on scratch file

B.2. Run-Time Error Messages.

600 Format missing final ')'
601 Sign not expected in input
602 Sign not followed by digit in input
603 Digit expected in input
604 Missing N or Z after B in format
605 Unexpected character in format
606 Zero repetition factor in format not allowed
607 Integer expected for w field in format
608 Positive integer required for w field in format
609 '.' expected in format
610 Integer expected for d field in format
611 Integer expected for e field in format
612 Positive integer required for e field in format
613 Positive integer required for w field in A format
614 Hollerith field in format must not appear for reading
615 Hollerith field in format requires repetition factor
616 X field in format requires repetition factor
617 P field in format requires repetition factor
618 Integer appears before '+' or '-' in format
619 Integer expected after '+' or '-' in format
620 P format expected after signed repetition factor in format
621 Maximum nesting level for formats exceeded
622 ')' has repetition factor in format
623 Integer followed by ',' illegal in format
624 '.' is illegal format control character
625 Character constant must not appear in format for reading
626 Character constant in format must not be repeated
627 '/' in format must not be repeated
628 '\' in format must not be repeated
629 BN or BZ format control must not be repeated
630 Attempt to perform I/O on unknown unit number
631 Formatted I/O attempted on file opened as unformatted
632 Format fails to begin with '('
633 I format expected for integer read
634 F or E format expected for real read
635 Two '.' characters in formatted real read
636 Digit expected in formatted real read
637 L format expected for logical read
639 T or F expected in logical read
640 A format expected for character read
641 I format expected for integer write
642 w field in F format not greater than d field + 1
643 Scale factor out of range of d field in E format

644 E or F format expected for real write
645 L format expected for logical write
646 A format expected for character write
647 Attempt to do unformatted I/O to a unit opened as formatted
648 Unable to write blocked output, possibly no room on device for file
649 Unable to read blocked input
650 Error in formatted textfile, no <cr> in last 512 bytes
651 Integer overflow on input
652 Too many bytes read out of direct access unit record
653 Incorrect number of bytes read from a direct access unit record
654 Attempt to open direct access unit on unblocked device
655 Attempt to do external I/O on a unit beyond end of file record
656 Attempt to position a unit for direct access on a nonpositive record number
657 Attempt to do direct access to a unit opened as sequential
658 Attempt to position direct access unit on unblocked device
659 Attempt to position direct access unit beyond end of file for reading
660 Attempt to backspace unit connected to unblocked device
661 Attempt to backspace sequential, unformatted unit
662 Argument to ASIN or ACOS out of bounds (ABS(X) .GT. 1.0)
663 Argument to SIN or COS too large (ABS(X) .GT. 10E6)
664 Attempt to do unformatted I/O to internal unit
665 Attempt to put more than one record into internal unit
666 Attempt to write more characters to internal unit than its length
667 EOF called on unknown unit

697 Integer variable not currently assigned a format label
698 End of file encountered on read with no END= option
699 Integer variable not ASSIGNed a label used in assigned goto

1000+ Compiler debug error messages - should never appear in correct programs

NOTES

SOFTECH MICROSYSTEMS

A SUBSIDIARY OF SOFTECH

UCSD p-SYSTEM

A PRODUCT FOR MINI- AND MICRO-COMPUTERS

Version IV.0

FORTRAN REFERENCE MANUAL

First edition: March 1981

SofTech Microsystems, Inc.
San Diego 1981

9494 Black Mountain Rd., San Diego, CA 92126 (714) 578-6105 TWX: 910-335-1594

DISCLAIMER:

This document and the software it describes are subject to change without notice. No warranty expressed or implied covers their use. Neither the manufacturer nor the seller is responsible or liable for any consequences of their use.

ACKNOWLEDGEMENTS:

This document was written by Stan Stringfellow at SofTech Microsystems. Special thanks are due to Dan LaDage of LaDage Computer Systems, as well as Gail Anderson, Blake Berry, and Randy Clark of SofTech Microsystems, for providing information and assistance. Also, thanks are due to Texas Instruments Incorporated, for providing useful information.

TABLE OF CONTENTS

CHAPTER 1

THE UCSD p-SYSTEM AND SOFTECH MICROSYSTEM BASIC

1.1 Introduction

SofTech Microsystems' BASIC is a compiled BASIC that runs under the UCSD p-System. Since it was designed to be used with the Screen Oriented Editor, it has an expanded syntax that allows indented and unnumbered statements as well as statements which are not in numeric order by line number. Because it is intended to be one language in a multi-language software development environment, BASIC subroutines can be separately compiled and linked into Pascal, FORTRAN, and BASIC host programs without recompilation. Also, BASIC programs may host separately compiled Pascal procedures and FORTRAN subroutines.

SofTech Microsystem's BASIC allows arrays with unlimited dimensions and subroutines with unlimited numbers of parameters. Virtual arrays which reside on disk, and may be very large, are permitted. Large programs may be split into many disk files, which can include each other using the $1 compile option, and be compiled into a single codefile.

The following sections describe how to use the UCSD p-System to create and compile BASIC programs. The compiler options are described, and the simple BASIC constructs relating to the text of the program itself are explained.

1.2 Editing BASIC Programs

The UCSD p-System Screen Oriented Editor is used to create and modify BASIC programs. This section will give a cursory explanation of how to use this Editor. For a more complete description of the Screen Oriented Editor, see the UCSD p-System Users' Manual.

To enter the Editor from the main system promptline, type "E". The system will respond by asking what file is to be edited. An existing text file may be specified (e.g., #5:PROG<return> will indicate the file PROG.TEXT on the disk in drive #5:) or a new file may be created by simply typing <return>.

Once the Editor has been entered, text may be inserted after typing an "I". A new line of text will automatically be indented to correspond to the line above it. This makes it easy to use the indentation feature of SofTech Microsystem's BASIC to increase readability:

```
      FOR I=1 TO 100
         FOR J=1 TO 100
            A(I,J)=0
            B(I,J)=J
         NEXT J
         IF I >= 50 THEN GOTO 10
         DISPLAY "I < 50"
10    NEXT I
      END
```

Note that line numbers are optional, and are really only necessary if a statement is to be the target of a GOTO or GOSUB statement (described in Chapter 4). When line numbers are used, they do NOT need to appear in increasing order.

Once the program has been typed in, the ETX key is typed to accept the text, and the Editor is exited by typing "Q" for quit. Then the user may U(pdate or W(rite the file. In the first case, the file will be saved as SYSTEM.WRK.TEXT and in the second case a name may be specified (e.g., #5:PROG<return> will save the file as PROG.TEXT on the disk in drive #5:).

1.3 Compiling BASIC Programs

Before the BASIC compiler can be invoked, the following steps must be followed:

1. Enter the Filer (by typing "F") and use the C(hange command to change the names of SYSTEM.COMPILER and SYSTEM.LIBRARY to some other names (such as PASCAL.COMPILER and SAVE.LIBRARY).

2. Change the names of BASIC.CODE and BASIC.LIBRARY to SYSTEM.COMPILER and SYSTEM.LIBRARY. If the new SYSTEM.LIBRARY is not on the * system disk, it must be T(ransfered there.

It may be desirable to create an entirely separate system disk to be used only with the BASIC language.

When the BASIC compiler is properly set up, it may be invoked from the main system promptline by typing "C". If the program text was U(pdated from the Editor, it will be automatically compiled.

If the program was W(ritten to a file (and no SYSTEM.WRK.TEXT already existed), the system will prompt for the name of the file to be compiled. The user should respond, for example #5:PROG<return> if the file is PROG.TEXT on #5. Then the system will prompt for the name of the codefile to be produced. The response

should be #5:PROG<return> if PROG.CODE on #5 is to be created.

At this point, the two pass compiler will execute, printing a dot for each line in the text. After compilation, "R" (for run) should be typed, and the program will execute (usually after linking the object with code from the Library).

If a syntax error is encountered by the compiler, the bell will sound, compilation will temporarily halt, and an error message will be displayed. The editor may be re-entered at this point by typing "E", and the error may be corrected immediately. Compilation may also be either continued or aborted.

Other errors, such as runtime arithmetic overflows, will be caught during program execution.

1.4 Compiler Options

The Compiler Options are used to control various aspects of the BASIC compiler's output. These options are specified in the following manner:

```
(*$<option> <parameters> *)
```

The INCLUDE option indicates to the Compiler that the specified file is to be compiled as though it were placed directly in line within the current file. The following is an example of this directive:

```
(*$I #5:PROG2.TEXT *)
```

The LIST option causes the compiler to emit a compiled listing to the CONSOLE, PRINTER or specified disk file:

```
(*$L CONSOLE: *)
(*$L PRINTER: *)
(*$L #4:LIST.TEXT *)
```

The listing can be optionally turned on or off at any point in the source text (after it has been started using one of the above forms of this directive) by following the directive with a "+" for on, or a "-" for off:

```
(*$L+*)
(*$L-*)
```

The PAGE option causes a form feed in a compiled listing at the point where it occurs in the source text:

```
(*$P*)
```

The FLIP option is used only in the II.0 version of the compiler. This causes the byte sex of the object to be of opposite sex from the host machine:

```
(*$F*)
```

The QUIET option controls the "quiet/noisy" mode of the compiler. In the noisy mode, $Q-, the compiler displays a dot (period) on the system console for each statement compiled. In quiet mode, $Q-, the dots are not displayed. The default is $Q- unless the machine has a "slow terminal" (designated by a data item in SYSTEM.MISCINFO), in which case the default is $Q+.

```
(*$Q+*)
(*$Q-*)
```

The RANGE CHECKING option controls runtime range checking on references to array variables and string variables. When $R+ is in effect, runtime range errors cause the program to abort with an execution error. If $R- is in effect, the compiler does not emit code to do range checking during execution. The default is $R+.

```
(*$R+*)
(*$R-*)
```

The I/O CHECK option directs the compiler to emit code which will cause I/O errors to be handled by the system if the $I+ option is on. The $I- option causes the I/O status to be recorded and made available, through the built in function IORESULT, but no execution error results from I/O errors. The default is $I+.

```
(*$I+*)
(*$I-*)
```

The T option, when $T+ is used, causes code to be emitted which handles the transcendentals in the library in a manner consistent with the TI machines. The default is T+.

```
(*$T+*)
(*$T-*)
```

The Copyright option will place a copyright notice within the codefile:

```
(*$C Copyright (c) 1981, SofTech Microsystems *)
```

1.5 Comment Delimiters

The REM statement and the exclamation point (!) are treated identically by the BASIC compiler. They represent the start of a comment which is terminated by the end of the line:

```
REM  This is a comment
A=1  REM This is a comment
! This is a comment
A=2  ! This is a comment
```

The two delimiters (* and *) can also be used to enclose comments. The comments between these two delimiters may cross line boundaries:

```
A=1  (* This is a comment *)
B=2  (* This is a comment: At this point we have
        decided to set B equal to 2 *)
```

1.6 Multiple Line Statements

Since placing an end of line between the delimiters (* and *) essentially causes that end of line to be invisible to the compiler, statements which are allowed to fill only one line may be expanded to several lines by commenting out the EOL character. For example, the following DIM statement (see Chapter 2) is used to declare two lines of arrays:

```
DIM  A(4), B(5,6,7), C(10,10), (*
     *) D(8,9), E(20)
```

And the following FUNCTION (see Chapter 6) is defined with more parameters than might fit on one line:

```
DEF A_FUNC(A,B,C,D,E,F,G,H,I,J,K, (*
           *) L,M,N,O,P,Q,R,S)
```

1.7 Multiple Statement Lines

Normally, only one statement is allowed on a line. The double colon (::) is used to separate statements so that two or more may appear on a single line as in the following example:

```
A=1 :: B=2 :: C=3
```

CHAPTER 2

DATA TYPES AND EXPRESSIONS

2.1 Data Types

SofTech Microsystems' BASIC handles both numeric and character string data types. Numeric data is expressed as INTEGER, REAL, or DECIMAL numbers. In the current version, the DECIMAL data type is identical to the REAL type. Arithmetic operations can be performed on this type of data. Character string data consists of sequences of printable ASCII characters. String operations may be performed on data of this type.

2.1.1 Numeric Data

Integers have no decimal point and are allowed to have values between 32767 and -32767.

Real or floating point numbers may have a decimal point and/or an 'E' followed by an exponent. If an exponent is specified, the floating point number will be raised to that power of ten. For example 2.0E7 is equivalent to 2.0 times ten to the seventh power. The minimum and maximum values for real numbers are machine dependent.

Decimal numbers are identical to real numbers. There are some syntax differences, but there is no difference at all in the way they are treated in the current version.

2.1.2 String Data

String data is nonnumeric information expressed as words or other character sequences. A string may contain numeric symbols, but arithmetic may not be performed on it. The string operations are described in Sections 2.7, 2.8, and Chapter 5.

2.2 Constants

Data may be in the form of constants. The value of a constant is specified at the time the program is written and does not change during program execution.

2.2.1 Numeric Constants

A numeric constant may be an integer or floating point number, and may be either positive or negative. The following are examples of numeric constants:

```
27
1981
123.4567
0.333
.333
333.0
10.0E3
10E3
-1
-765.4321
-0.001
-001.0
-12.234E5
-1E-15
```

The following are incorrect numeric constants:

25x2	The 'x' is not allowed
7,999.99	Commas are not allowed
10.0E2E3	Only one 'E' is allowed

2.2.2 String Constants

A string constant is a sequence of printable ASCII characters enclosed within double quotes. A quote may be inserted into a string by entering two consecutive quotes (""). The following are examples of string constants:

```
"Now is the time for all good men..."
"765.4321"
"<>,. ?/;: !@#$ %%&* ()'+ =_"
"Quoth the Raven, ""NEVER MORE!"""
```

The following are incorrect string constants:

'Incorrect'	Single quotes are not string delimiters
"WOW	Second quote missing
"She said, "Hi!""	Second quote is taken as end of string

2.3 Variables

Variables are data items which may have their values changed during the execution of a program. Like constants, variables may be numeric data or character strings. Variables may also be grouped into arrays. Within an array they may be accessed individually by specifying the array name and a subscript. For more information about arrays, see Section 2.4.

2.3.1 Numeric Variables

Numeric variables may be INTEGER, REAL, or DECIMAL format. The range of values which are valid for numeric variables is the same as for numeric constants. If, during program execution, an attempt is made to assign a variable to a value outside that range, a runtime error will result. (The exception to this is when a computed integer value overflows. An error will be produced only if the sign bit is changed due to an overflow. Other integer overflow errors are not detected.)

2.3.1.1 Numeric Variable Naming Conventions

Numeric variable names must begin with a letter of the alphabet. This letter may be followed by as many as 254 alpha-numeric characters or any of the special characters: @, [,], \ or _. All the characters in a variable name (up to 255) are used to distinguish it from other variables. The following are valid numeric variable names:

```
ONE
F[2]NUM
P_123
VERY_LONG_IDENTIFIERS_ARE_OK
L@PTR
```

Variable names may not be the same as reserved words used in SofTech Microsystems Basic. For example, a variable can not be named GOTO. A compiler error will result if this is attempted. A list of these reserved words appears in Appendix A.

2.3.1.2 Numeric Variable Declarations

All numeric variables are assumed to be of type REAL unless otherwise specified. The default type can be changed by using the ALL clause as shown below:

```
INTEGER ALL     Changes default type to INTEGER
REAL ALL        Changes default type REAL
DECIMAL ALL     Changes default type to DECIMAL
```

The ALL statement, if used, must precede the first occurrence of any of the following statements:

```
INTEGER
REAL
DECIMAL
DIM
DEF
SUB
```

Numeric variables can be individually declared to be of a particular type, regardless of what the default type is by using the INTEGER, REAL or DECIMAL statements. (Numeric variables can also be declared within the DIM statement, see section 2.4.2.) The type name is followed by a list of variables separated by commas as follows:

```
INTEGER I,J,K
REAL R
DECIMAL A1,A2,A3,A4,A5
DECIMAL (2) B1,B2,B3,B4,B5,B6
DECIMAL (-4) C1,C2,C3,C4,C5,C6,C7
```

The INTEGER declaration above specifies I, J, and K as integers. The REAL statement declares R as a REAL. The rest of the variables are declared to be DECIMAL numbers. The second and third DECIMAL statements contain an optional number in parentheses. In the current version, this number has no meaning, and DECIMAL statements are equivalent to REAL statements.

2.3.2 String Variables

String variables, like numeric variables, may have their values altered during program execution. A string may contain up to 255 printable ASCII characters. Strings are used to hold data for input and output and to express nonnumeric data such as names, descriptions, etc.

2.3.2.1 String Variable Naming Conventions

String variables are named according to the same conventions as numeric variables. The only difference is that string variable names must end with a dollar sign. The following are correct examples of string variables:

```
ASTRING$
ANOTHER_STRING$
S[22]$
A\B\C\$
```

2.3.2.2 String Declarations

Strings do not have to be declared, but declaring them may save memory space. If they are not declared, or if the declaration does not specify a size, they are allocated a default length of 255 bytes at compile time. This allocated space does not change dynamically during program execution. But a maximum size can be specified using the DIM statement:

```
DIM ASTRING$*20
```

This limits ASTRING$ to a maximum length of 20 bytes. For further information concerning the DIM statement in this context, see section 2.4.3.

2.4 Arrays

Variables may be grouped together into an ARRAY. The array is given a name, a number of dimensions, and a size for each dimension. By specifying the array name followed by the one or more index values, a specific variable within the array may be accessed. For example, a two dimensional array, AR1, would be refered to as AR1(4,3) in order to obtain the indicated element. An array may have any number of dimensions, but the total number of elements may not exceed 32767.

Virtual Arrays are arrays which reside on disk. This allows programs to use large arrays which will not fit into memory. This type of array is discussed in Chapter 7.

2.4.1 The DIM Statement

In order to declare an array, the DIM statement is used. This statement defines the number of dimensions and the number of entries within each dimension of the array.

In the declaration, DIM is followed by the array name. Then, in parentheses, one or more integers are specified, separated by commas. The array name should follow the conventions for numeric variable names if it is an array of numeric variables. Likewise, it should follow the conventions for string variable names if it is an array of strings.

The integers in parentheses are zero-based. (This may be changed, however, by the OPTION BASE statement. See Section 2.4.4.) This means that the array can be indexed from zero up to the number specified, and that the number of entries is one greater than that number.

More than one name may be defined in a DIM statement, if each name is separated by a comma.

The following are correct examples of array declarations:

```
DIM AR10 (9)
DIM ONE_ELEMENT_ARRAY(0)
DIM S\ARRAY$ (20,20)
DIM MULTI@ (3,7,15,31)
DIM A7(100,10,10),A8(100,10,10)
DIM LARGE(32000), W$(9,9,9), LIST_NUMS(0,17)
```

It is not always necessary to declare arrays. If the maximum dimension is less than or equal to 10 (e.g. DIM A(10,10,5)) then the array may be implicitly declared when it is first referenced. The default lower boundary for each dimension is 0 and the default upper boundary is 10. However, it may save considerable space to declare small multi-dimensional arrays anyway. For example, an integer array of dimension (2,2,2,2) would only take up 3 to the fourth (81) words of memory. If undeclared, however, this same array would default to a (10,10,10,10) dimensional array and take up 14641 words.

2.4.2 Type Specification of Numeric Arrays

All of the arrays declared in Section 2.4.1 above, except for S\ARRAY$ and W$, are numeric arrays. The variable type of the entries within those arrays will be the numeric variable default type (REAL unless otherwise specified by using the ALL statement). Numeric arrays may be declared to be of a particular type using either a DIM statement or an INTEGER, REAL, or DECIMAL statement as in the following examples:

```
DIM A(7,2), INTEGER B(2,3), C(12,1,0)
DIM REAL D(5,7), DECIMAL E(10,10,10), INTEGER F(0)
INTEGER I,J,K(7,7,7)
REAL    R(12),S
```

In the first line above, B is an array of type integer. A and C are arrays of the default type. In the second line, D is a real array, E is a decimal array, and F is an integer array. In the third line, I and J are integer variables and K is an integer array. The fourth line declares R to be a real array and S to be a real variable.

A DIM statement may also include single variables. These variables may optionally be preceded by a type declaration:

```
DIM A(99,9),B,C(49)
DIM E(4,5), INTEGER F1,F2, G(6,7)
DIM REAL H
```

The first line above declares arrays A and C and variable B. Both arrays and the

variable are of the default type. The second line declares arrays E and G to be of the default type. Variable F1 is declared to be an integer and variable F2 is declared as the default type. In the third line, real variable H is declared.

2.4.3 Size Specification of String Arrays

An array is declared as a string array by naming it according to string variable naming conventions:

```
DIM S_ARRAY$(2,3,4)
```

However each string in S_ARRAY$ above will consume 256 bytes of memory. A maximum size for each string in an array can be indicated by following the declaration with an asterisk (*) and a number between 1 and 255 inclusive. Also, string variables can be declared within DIM statements in the same way. The following are examples:

```
DIM S_ARRAY$(2,3,4)*20
DIM S1$(2)*10, S2$(20)*11
DIM A$(10,10)*1, B$*25, C$(5,6,7), D$*1
```

The first line above declares three-dimensional S_ARRAY$ to consist of strings with a maximum length of 20 bytes. The second line declares S1$ and S2$ as one-dimensional arrays containing strings with a maximum length of 10 and 11 bytes. In the third line, A$ is a 10 by 10 array of one character strings, B$ is a string variable with a maximum length of 25, C$ is a three dimensional string array of 256 byte strings, and D$ is a string variable containing at most one character.

This string length specification may occur anywhere that a string declaration is legal.

2.4.4 The OPTION BASE Statement

Array indices, whether declared in a DIM statement or not, are zero-based by default. This means that the statement, DIM A(10), declares A to be indexed from 0 to 10. By using the OPTION BASE statement, array indices can be based at 1 or 0:

```
OPTION BASE 0
OPTION BASE 1
```

The first statement leaves the indexing base at the default value of 0 and the second makes the base 1. The OPTION BASE statement may be used, at most, once in a program. If it is used, it must come before any statement which declares or references array elements. If 1 is declared to be the base, no statement may declare or reference an array with an index of zero.

2.5 The LET Statement

The LET statement is used to make numeric or string assignments. The word LET, which is optional, is followed by the variable to which a value is to be assigned. This is followed by an equal sign and an expression, the value of which will be assigned to the variable.

LET statement syntax:

```
LET numeric_variable = numeric_expression
LET string_variable   string_expression
numeric_variable = numeric_expression
string_varible = string_expression
```

LET statement examples:

```
LET A=1
LET R1=2.0 * (R1+1)
LET S1$="STRING"
LET S2$="ANOTHER "&S1$
A=2
R1=R2+1.0
S2$=S1$
```

The variables on the left of the equal sign may be subscripted variables (indexed arrays). Likewise, the expressions on the right of the equal sign may contain subscripted variables. Both sides of the LET statement must be of the same type.

2.6 Arithmetic Operators

The arithmetic operators are used to combine numeric constants and variables into expressions. The following table illustrates these operators:

SYMBOL	OPERATION	EXAMPLE
-	Negation	-A
+	Addition	A+B
-	Subtraction	A-B
*	Multiplication	A*B
/	Division	A/B
^	Exponentiation	A^2

FIGURE 2.6.1 Arithmetic Operators

The unary minus negates the value following it. The four arithmetic operators perform the standard arithmetic functions. And the exponentiation symbol raises the first value to the power of the second.

2.7 String Operator

String variables, constants and expressions can be concatenated (joined together into a single string) by using the the '&' operation. When this symbol is placed between two strings, they are concatenated. In the following example, S1$ is set equal to "ABCDEFGHIJ":

```
S2$ = "EF" & "G"
S1$ ="ABCD" & S2$ & "HIJ"
```

2.8 Relational Operators

Relational operators are used to compare two expressions of the same type (numeric or string). Relational expressions can be employed within control flow statements. They may also be used to evaluate to the numeric values of -1 for true and 0 for false. These values (-1 and 0) may then be used within arithmetic expressions or they may be printed. The relational operators all have the same precedence. The following table lists them:

SYMBOL	OPERATION	EXAMPLE
=	equal to	A = B
<	less than	A < B
>	greater than	A > B
<= or =<	less or equal	A <= B
>= or =>	greater or equal	A >= B
<> or ><	not equal	A <> B

FIGURE 2.8.1 Relational Operators

The following example shows how relational expressions can be evaluated to numeric quantities:

```
DISPLAY 7+8=15; 2 = 2.0/.1; 100 >= 1

-1 0 -1
```

Comparisons between strings are based on the ASCII value of their characters. For example "A BIRD" < "A bird" because the ASCII value for lower case b is greater then the value for upper case B. Also "$ZEBRA" <= "AARDVARK" because the code for $ is less then (or equal to) the code for A. If strings are identical except

that one string has addtional characters, then the longer string is greater: "COW" < "COWBOY".

2.9 Logical Operators

Logical operators are used within expressions to create results which have the values of TRUE or FALSE. The three logical operators are NOT, AND and OR. The following truth table illustrates the actions of these operators:

X	Y	NOT X	X AND Y	X OR Y
F	F	T	F	F
F	T	T	F	T
T	F	F	F	T
T	T	F	T	T

FIGURE 2.9.1 Logical Operators

The NOT operator yields the value which is logically opposite the value of the argument.

The AND operator produces a TRUE if and only if both arguments are true.

The OR operator produces a FALSE if and only if both arguments are false.

The precedence of these operators is: NOT, AND, OR. This precedence may be overridden by using parentheses. The following examples illustrate the use of the logical operators:

```
A=1
B=2
C=3
IF NOT A > 0 THEN DISPLAY "TRUE" ELSE DISPLAY "FALSE"
IF A < B AND C < B THEN DISPLAY "TRUE" ELSE DISPLAY "FALSE"
IF A < B OR  C < B THEN DISPLAY "TRUE" ELSE DISPLAY "FALSE"
IF NOT A<B AND C<B THEN DISPLAY "TRUE" ELSE DISPLAY "FALSE"
IF NOT (A<B AND C<B) THEN DISPLAY "TRUE" ELSE DISPLAY "FALSE"

FALSE
FALSE
TRUE
FALSE
TRUE
```

These operators can also be used to manipulate integer values. The meaning of the logical operations on arithmetic bits is given by replacing every F with a 0, and

every T with a 1 in FIGURE 2.9.1. The NOT of an integer is equal to the NOT of each individual bit within it (the one's complement value). Likewise, the AND or OR of two integers is the bitwise AND or OR operation performed on them. For example:

```
DISPLAY NOT 0
DISPLAY NOT -2
DISPLAY 1981 AND 255
DISPLAY 1981 OR  255

-1
1
189
2047
```

Negative one is the bitwise complement of zero. Similarly, one is the complement of negative two. The number 255 is a byte of all ones. The AND of 255 and 1981 represents the lower byte of 1981. The OR of 255 and 1981 is the upper byte of 1981 and a lower byte of 255.

Floating point numbers may be used with the logical operators. However, they are converted to integer form before they are operated upon.

2.10 Precedence of Operators

The arithmetic and logical operations discussed in this chapter are evaluated according to the following priorities:

PRIORITY	OPERATION	SYMBOL
1	Exponentiation	^
2	Unary Minus	-
3	Multiplication and Division	*,/
4	Addition and Subtraction	+,-
5	Relational Operators	=,<,>,<=,>=,<>
6	Logical NOT	NOT
7	Logical AND	AND
8	Logical OR	OR

FIGURE 2.10.1 Precedence of Operators

Using these priorities, expressions are evaluated from left to right. A portion of an expression may be placed inside parentheses. In this case, it will be evaluated separately before being combined with the rest of the expression. Within the parentheses, the same order of precedence is held. Parentheses may be nested, and

2.11.4 Relational Expressions

Constants and variables may be combined with arithmetic and logical operators to form relational expressions. The only requirement is that the last operation performed must be a relational operation. Relational operations are often used within control flow constructs such as IF THEN ELSE statements.

CHAPTER 3

I/O STATEMENTS

3.1 The PRINT and DISPLAY Statements

String or numeric expressions may be output using the DISPLAY or PRINT statements. Usually the expressions to be output are constants or single variables. The formatting of the output can be controlled with these statements.

The PRINT statement directs output to the printer unless another unit is specified. If there is no printer, the PRINT statement directs its output to the console. The DISPLAY statement directs output to the console (video terminal). If a PRINT statement is re-directed to the console, it acts on all of the options described in this chapter. If not, it ignores the following options: ERASE ALL, AT, SIZE and BELL. Except for these differences, the two statements are identical. The description in this chapter of the DISPLAY statement and its options applies equally to the PRINT statement.

The DISPLAY statement may be used with or without an expression following it:

```
DISPLAY
DISPLAY expression
```

If no expression is indicated, then a carriage return is output. Otherwise, the value of the (numeric or string) expression is output, followed by a carriage return. When string values are displayed, no automatic formatting is done. Numeric values, however, are displayed with a leading character and a trailing blank. The leading character is a space if the value is positive, a minus sign is the value is negative. The trailing blank is used to separate numbers which are directly adjacent on the same line. The following are examples of the DISPLAY statement using expressions:

```
DISPLAY "Roses are red, Violets are blue"
DISPLAY 2+2

Roses are red, Violets are blue
 4
```

The DISPLAY statement may be followed by a list of expressions:

```
DISPLAY list
```

A list is several expressions, separated by commas, semicolons, or apostrophes. These expressions may be mixtures of string and numeric types. The commas, semicolons, and apostrophes are known as data separators. The effect of each data separator is different. But ending a list with any of them causes the cursor to stay on the current line after the DISPLAY statement is executed. In this way, it is possible to use more than one DISPLAY statement to print characters on a single line.

An output line is divided into display zones which are 16 characters wide. The comma causes the cursor to advance to the next display zone. If the cursor is currently in the last zone, then it advances to the first zone on the next line. Data is left-justified within each zone. The following is an example of the use of the comma as a data separator:

```
DISPLAY 1,2,3
DISPLAY "DOG","CAT","BIRD"

 1               2               3
DOG             CAT             BIRD
```

This same effect can be achieved by using separate DISPLAY statements separated by commas:

```
DISPLAY 1,
DISPLAY 2,
DISPLAY 3
DISPLAY "DOG",
DISPLAY "CAT",
DISPLAY "BIRD"

 1               2               3
DOG             CAT             BIRD
```

Using a semicolon between expressions causes NO separation between them:

```
DISPLAY 1;2;3
DISPLAY "DOG";"CAT";"BIRD"

 1  2  3
DOGCATBIRD
```

The spaces between the 1, 2, and 3 represent the leading and trailing blanks which always accompany numeric values.

Using an apostrophe between expressions causes a comma to be inserted between them:

```
DISPLAY 1'2'3
DISPLAY "DOG"'"CAT"'"BIRD"

 1 , 2 , 3
DOG,CAT,BIRD
```

3.2 Options available with the PRINT and DISPLAY statements

This section discusses the several options which can be used with the DISPLAY and PRINT statements. These options may be combined to fully format the output as desired. The list of options follow the DISPLAY or PRINT command. A colon is placed after the last option, and the expressions to be output are then specified. If more than one of the options ERASE ALL, AT, SIZE, or BELL are used, they must be in the order shown in this sentence.

3.2.1 The ERASE ALL Option

The ERASE ALL option causes the screen to be cleared before values are displayed. The following example illustrates the syntax for this option:

```
DISPLAY ERASE ALL: "DOGS"," CATS"," and lots of BIRDS"
```

3.2.2 The AT Option

The AT option can be used to indicate a starting line and column number for the display to appear on the screen. Column and line numbers start at 1. The format for this option is:

```
AT (line_number,column_number)
```

The default line number is 24 (the bottom line on the screen). The default column number is 1. If a DISPLAY statement which uses an AT option is followed by a DISPLAY statement which does not, the second statement starts in the default position regardless of where the first was positioned. The following example illustrates the use of the AT option:

```
COL=10
DISPLAY AT (12,COL): "Where am I?"
```

3.2.3 The SIZE Option

The SIZE option can be used to specify the maximum number of characters to be output by a DISPLAY statement. The format for the SIZE option is:

```
SIZE (n)
```

If this option is not used, the default size is large enough to hold all of the characters to be output, plus enough extra blank spaces to fill the end of the last line onto which the DISPLAY statement is writing. When this option is used, the line will be cleared, after the output is performed, only as far as the indicated size. If it is desired to leave portions of a line intact when displaying to the same

line, this can be done with the SIZE option. The semicolon data separator should be placed at the end of the display list so that no clearing will be done beyond the end of the last character being displayed. Strings which have a length greater than the specified size will be truncated on the right. If a negative size is given, its absolute value will be used. The following example illustrates the use of the SIZE option:

```
DISPLAY SIZE(38): "There are 38 characters in this string"
```

3.2.4 The BELL Option

The BELL option causes the terminal bell to ring when the display statement is executed. The following are examples:

```
DISPLAY BELL: "There were bells on the hills"
DISPLAY AT(10,10) BELL: "But I never heard them ringing"
```

3.2.5 The USING Option

The USING option controls the output of a DISPLAY statement. (The USING option may also be employed within the ASSIGN statement in conjunction with Virtual Arrays, see Section 7.1.7, and within the PRINT statement in conjunction with file I/O, see Section 7.2.1.) It has the following format:

```
USING line_number
USING string_expression
```

The line number is the number of a line containing an IMAGE statement (see Section 3.2.6). The string expression contains the elements of an IMAGE statement.

If a DISPLAY statement employs a USING statement, then the list of expressions to be displayed must use commas as data separators. Also, the only data separator that may terminate this list is a semicolon. The following illustrates the use of the USING option:

```
      NUM=999.999
      FORMAT$="###.##"
10    IMAGE ###.##
      DISPLAY USING "###.##":NUM
      DISPLAY USING FORMAT$:NUM
      DISPLAY USING 10:NUM

      999.99
      999.99
      999.99
```

All of the above statements produce the same output. If a DISPLAY statement list contains more expressions than the corresponding IMAGE statement contains formats, a new line is begun. This new line is formatted according to the same IMAGE statement. If there are fewer expressions than IMAGE formats, the DISPLAY terminates after the last expression.

3.2.6 The IMAGE Statement

The IMAGE statement is referred to by line number within the USING clause of a DISPLAY or PRINT statement. It provides a format for the expressions to be output. It has the format:

IMAGE string_constant

The quotes around the string constant are optional. Text may be inserted into the string constant and this text will appear in exactly the same position in the actual output. Text consists of all characters which are not format control characters.

3.2.6.1 Format Control Characters

There are nine format control characters: #, ^, -, +, ., <>, ,, $, and *.

The # (number sign) indicates the place of a data character.

The ^ (exponent sign) indicates how many places an exponent should fill. If there are more places indicated then the actual exponent has, leading zeros are inserted. Either four or five exonent signs should be used. If less than four exponent signs are used, they will be printed as a literal string rather than used to indicate an exponent field.

The - (minus sign) specifies the position of the minus sign if the value is negative. If the value is positive, this position will be left blank. The minus sign may be placed before or after the value.

The + (plus sign) may be placed to the left of a numeric field. It indicates that positive numbers are to be displayed with a plus sign preceding them. Negative numbers are displayed with a minus sign as usual.

The . (decimal point) is used to indicate the position of the decimal place.

The <> (angle brackets) are used to enclose numeric IMAGE fields if it is desired to have negative numbers appear within angle brackets. Positive numbers will appear without the brackets.

The , (comma) will produce a comma at the specified position within a numeric value.

The $ (dollar sign) will cause a dollar sign to appear at the beginning of the indicated field. A $$ (double dollar sign) allows the dollar sign to float (otherwise it is left justified).

Two ** (asterisks) produce asterisk fill wherever the numeric value does not fill the field. This is used in protecting checks.

The following is an example of the use of these format characters:

```
      A=999.999
      B=88.8888
      S$="OCEAN"
10    IMAGE ##### BLUE <###.##>
      DISPLAY USING 10:A,B
      DISPLAY USING 10:S$,-B

        1000 BLUE    88.89
      OCEAN BLUE   <88.89>
```

3.2.6.2 Fields Within IMAGE Definitions

An integer field of an IMAGE definition or string has no decimal point. It may have a sign. If the value overflows the field, asterisks will be produced instead of the value. The integer is right-justified within the field, and is rounded. The following example illustrates integer fields:

```
10    IMAGE "#### ##### ###"
      I=999 :: J=-88 :: K=7777
      DISPLAY USING 10: I,J,K

       999  -88  ***
```

A decimal field consists of a string of number signs and may have a plus/minus sign. A decimal point may appear within it, just before it, or just after it. The value is rounded according to the quantity of number signs which follow the decimal point in the IMAGE format. The number is right-justified within the field. The decimal point is placed in the position indicated in the field definition. If the number overflows, asterisks are displayed instead of the value. The following example illustrates decimal fields:

```
10    IMAGE "#### ##### ###"
      I=111.11 :: J=-88.888 :: K=7777.7777
      DISPLAY USING 10: I,J,K

       111  -89  ***
```

An exponent field is a series of four or five exponent signs (^) which reserve space for the exponent. The number is rounded similarly to decimal fields. A leftmost plus/minus sign reserves space for the appropriate sign. If the minus sign is used, a blank will appear if the value is positive. There must be at least one character (#, + or -) to the left of the decimal point if the number to be displayed is negative. The following example illustrates exponent fields:

```
20      IMAGE "#.#####^^^^ ##.###^^^^ ###.^^^^ #.##^^^^"
        A=111.111 :: B=-66.666 :: C=55.5 :: D=-.077
        DISPLAY USING 20:A,B,C,D

         .11111E+03 -6.667E+01   56.E+00   -.78E-01
```

A string field may be indicated by any sequence of control characters. If the string is shorter than the indicated field, blank spaces will be padded on the right. If the string exceeds the specified length, it will be truncated on the right.

Fields consisting of characters other than the control characters are taken as text to be literally inserted into the displayed output.

3.2.6.3 The PUNCTUATION Statement

The PUNCTUATION statement can be used to alter the monetary symbols for currency ($), digit separators (,) and decimal point (.). This statement takes the following form:

```
PUNCTUATION string_expression
```

The first character in the string expression is used for the currency symbol. The second is used for the decimal point. The third character is used for the digit separator symbol. The default values for these are the same as they would be if the following statement was executed:

```
PUNCTUATION "$.,"
```

The following example demonstrates the use of the PUNCTUATION statement:

```
10      IMAGE $$###,###.##
        AMOUNT=999999.25
        DISPLAY USING 10:AMOUNT
        PUNCTUATION "L,."
        DISPLAY USING 10:AMOUNT

        $999,999.25
        L999.999,25
```

3.3 The INPUT Statement

The INPUT statement accepts values typed in from the keyboard during program execution. The basic form of this statement is:

```
INPUT variable
```

A question mark, followed by a blank space, appears when this statement is executed. When a value is entered, followed by a <return>, the variable is assigned accordingly. If it is a string variable, the input will be interpreted as a string. If it is a numeric variable, the input must represent a correct numeric value. Leading and trailing blanks are removed from string variables.

Several variables may be included within the INPUT statement if they are separated by commas. The keyboard input must be made on a single line and the input values must be separated by commas. All the variables must be within a single INPUT statement for each line input form the keyboard. This last constraint does not apply when input is being done from files.

A quoted string followed by a colon may precede the variable list. The string will be used as a prompt to replace the question mark. If no prompt and no question mark are desired, a null string ("") may be used. The following are examples of the INPUT statement:

```
INPUT RATE
INPUT "":HEIGHT,WIDTH,NAME(1)
INPUT "Type a character string:":STRING1$
```

3.3.1 The AT Option With the INPUT statement

The AT option may be used with the INPUT statement in a manner similar the DISPLAY statement. The cursor is positioned according to the AT clause specifications. The following are examples of the AT clause:

```
INPUT AT(10,10):PAY
INPUT AT(10,10)"Enter Pay":PAY
```

3.3.2 The SIZE Option With the INPUT statement

The SIZE option can be used to specify the maximum number of characters that may be input. If the number specified is positive, the line will be cleared before a prompt for input is made. If that number is negative the line will not be cleared. The bell will sound if more characters are entered than the SIZE clause allows. The default size is the remainder of the line after the input prompt. If a size is

specified, it does not include the length of an input prompt if one is issued. The following are examples of the use of the SIZE option with the INPUT statement:

```
INPUT AT (10,18) SIZE(2),"What year?": YEAR
INPUT SIZE(-3):S$
```

If the input exceeds the specified size, the bell will sound for each extra character typed until a <space> or <return> is input.

3.3.3 The BELL Option With the INPUT statement

When the BELL option is used with the INPUT statement, the bell will ring, prompting the user to input. The following is an example of this:

```
INPUT AT(10,20) BELL,"I hear bells, do you?": S$
```

3.4 The ACCEPT Statement

The ACCEPT statement is used like the INPUT statement, except that it can take only one variable. The ACCEPT statement reads the entire line as input and does not edit out commas or quotes. Since commas are used as data separators for the INPUT statement, the ACCEPT statement is useful because a comma can be a part of an input string. The ACCEPT statement, therefore, is most useful when reading into a string variable. The INPUT statement options described above may also be used with the ACCEPT statement. The following are examples of the ACCEPT statement:

```
ACCEPT S$
ACCEPT "What Company? ": CO$
ACCEPT AT(10,10) SIZE(2): DAY_OF_MONTH
```

3.5 The DATA Statement

The DATA statement defines values that will be used as data within a program. These values may be numeric or string constants. Quotes may optionally be used to enclose string data. Strings must be enclosed in quotes if commas are contained within them. Otherwise, commas are interpreted as data separators. Also, leading and trailing blanks will be removed from strings which are not within quotes. The following is the DATA statement format:

```
DATA list
```

The list is one or more constants separated by commas. These constants may be numeric or string types.

Several DATA statements may appear within a program. They may be placed anywhere in the program source text and need not be grouped together. As one DATA statement is exhausted, the next one in the file will be used. The following are examples of the DATA statement:

```
DATA 20,40,60,80
DATA 100,120,140,160
      .
      .
      .
DATA "CALIFORNIA"
DATA "TEXAS"
```

3.6 The READ Statement

The READ statement uses the values specified in DATA statements. It assigns these values to variables which are listed in the READ statement. The READ statement variables may be numeric or string and they may be subscripted or unsubscripted. The DATA statements will be used in the order that they appear in the source text. The specified variables and the corresponding data values must be of the same type and have the same range. The READ statement has the following form:

```
READ list
```

The list is one or more variables separated by commas.

If a READ statement is encountered and no corresponding DATA statement has been declared, or if all the DATA statements have been exhausted, then an error will occur. The following illustrates the use of the READ statement:

```
READ I,J
DISPLAY I;J;
READ I,J
DISPLAY I;J
DATA 2,4,6,8

 2  4  6  8
```

3.7 The RESTORE Statement

During program execution an internal data pointer is kept. This pointer indicates the next DATA statement value to be read. The RESTORE statement resets this pointer to the first DATA statement in the program. Alternatively, the line number of a particular DATA statement may be specified, and the RESTORE statement will reset the data pointer to that statement. After the RESTORE statement is executed, the next READ statement will take its input from where the reset data pointer indicates. The RESTORE statement takes the following forms:

```
RESTORE
RESTORE line_number
```

If a line number is indicated and that line does not contain a DATA statement, then the next line which does contain a DATA statement is used. If there is no DATA statement on the indicated line or following it, then an error will occur at the next READ statement. The following illustrates the use of the RESTORE statement:

```
      DATA 1,2
20    DATA 3,4
30    DATA 5,6
      READ I,J,K,L
      DISPLAY I;J;K;L
      RESTORE
      READ I,J,K,L
      DISPLAY I;J;K;L
      RESTORE 20
      READ I,J,K,L
      DISPLAY I;J;K;L

 1  2  3  4
 1  2  3  4
 3  4  5  6
```

CHAPTER 4

CONTROL FLOW STATEMENTS

4.1 The GOTO Statement

The GOTO statement unconditionally transfers control to a specified line number. It has the following format:

```
GOTO line_number
GO TO line_numer
```

The following sample program shows the use of the GOTO statement:

```
      1=1
10    DISPLAY 1
      1=1*2
      GOTO 10

       1  2  4  8  16  ...
```

4.2 The ON-GOTO Statement

The ON-GOTO statement allows a multiple switch mechanism for control flow. This statement has the format:

```
ON expression GOTO line_num_1, line_num_2, line_num_3 ...
```

The expression is any valid numeric expression. If necessary it will be rounded to an integer. If the expression evaluates to 1, control is transfered to the first line number. If the expression evaluates to 2, control is transfered to the second, etc. If the expression is less than 1 or greater than the number of listed line numbers, an error will result. The following example illustrates the use of the ON-GOTO statement:

```
        I=0
10      I=I+1
        ON I GOTO 20,30,40
20      DISPLAY "AT LINE 20
        GOTO 10
30      DISPLAY "AT LINE 30
        GOTO 10
40      DISPLAY "AT LINE 40
END

        AT LINE 20
        AT LINE 30
        AT LINE 40
```

4.3 The IF-THEN-ELSE Statement

The IF-THEN-ELSE statement provides conditional transfer of control flow based on the value of a relational expression. It has the following forms:

```
IF condition THEN action
IF condition THEN action_a ELSE action_b
```

The condition is a relational expression. If the expression evaluates to true, the THEN clause is executed. Otherwise, the ELSE clause is executed (if there is no ELSE clause, the next statement in the program is executed). The action is either a single executable statement, or a line number to which control will be transfered. The entire statement must fit on one line (or several lines joined together with comment delimiters, see Chapter 1). The following example demonstrates the use of this statement:

```
        S1$="ABC"
        S2$="123"
        IF S1$=S2$ THEN 10 ELSE 20
10      DISPLAY "WHAT ??"
        STOP
20      DISPLAY "GOOD"
        END

        GOOD
```

The IF clause condition may be a numeric expression. In this case the resulting value is taken to be false if its least significant bit is zero, and true otherwise.

```
A=10
IF A=10 THEN GOOD=1 ELSE GOOD=0
IF GOOD THEN DISPLAY "GOOD" ELSE DISPLAY "BAD"

GOOD
```

4.4 The FOR-TO-STEP and NEXT Statements

These statements are used to create programming loops. They have the following formats:

```
FOR variable = init_val TO final_val
FOR variable = init_val TO final_val STEP increment
NEXT
NEXT variable
```

The variable is any subscripted or unsubscripted numeric variable. If it is a subscripted variable such as A(10,J), its actual location is confirmed the first time the loop is executed and does not change within the loop. Init_val, final_val and increment are any valid numeric expressions. They are also bound at the first execution of the loop, and do not change. When the FOR statement is first executed, the variable is assigned the value of init_val. When the NEXT statement encountered, the value of increment is added to it. If no increment is specified, 1 is added. If the value of the increment is positive and the new value of variable does not exceed final_val, the loop is re-executed. Likewise, if the value of the increment is negative and the new value of variable is not less than final_val, the loop is performed again.

The loop consists of those statements which lie between the FOR statement and the NEXT statement. It is possible that the loop will never be executed if the increment is positive and final_val is less than init_val, or if the increment is negative and final_val is greater than init_val.

The NEXT statement may be followed by the loop variable. If this is the case, that variable must match the variable specified in the preceding FOR statement. If the loop variable is an array element, the NEXT statement should only specify the array name.

The following demonstrates the use of these statements:

```
FOR J=0 TO 10 STEP 2
  DISPLAY " J=";
  DISPLAY J;
NEXT

 J= 0   J= 2   J= 4   J= 6   J= 8   J= 10
```

FOR statement loops may be nested as follows:

```
FOR I=1 TO 10
    .
    .
    .
  FOR J(2,3,4)=A TO B STEP C
      .
      .
  NEXT J
    .
    .
    .
NEXT I
```

4.5 The GOSUB and RETURN Statements

Basic programs may have procedure blocks within them. A procedure block is a group of statements which are called by the GOSUB statement. When a RETURN statement is encountered the block is exited and execution is continued at the first statement after the GOSUB call. These statements have the following format:

```
GOSUB line_number
    .
    .
    .
RETURN
```

The line_number indicates the start of the procedure block. When the RETURN is encountered the block is exited. The following example illustrates the use of these statements:

```
        I=10
        J=20
        GOSUB 100
        I=100
        J=200
        GOSUB 100
            .
            .
            .
100     DISPLAY I;J;
        DISPLAY I+J
        RETURN

         10   20   30
         100   200   300
```

Procedure blocks may be nested in the following fashion:

```
        I=10
        GOSUB 100
        I=20
        GOSUB 100
            .
            .
            .
100     IF I=20 THEN GOSUB 200 ELSE DISPLAY "In block 100"
        DISPLAY "This is the second statement in block 100"
        RETURN

200     DISPLAY "In block 200"
        RETURN

        In block 100
        This is the second statement in block 100
        In block 200
        This is the second statement in block 100
```

4.6 The ON-GOSUB Statement

The GOSUB statement has a computed format similar to the computed GOTO statement:

```
ON expression GOSUB line_1, line_2, ...
```

If the expression is equal to 1, control is transfered to the first line indicated. If

the expression is equal to 2, the second line indicated is chosen, etc. Like the computed GOTO statement, an error will result if the expression is less than one or greater than the number of listed line numbers. The following example illustrates the use of the computed GOSUB statement:

```
      I=1
10    ON I GOSUB 100,200,300
      I=I+1
      IF I <= 3 THEN 10
          .
          .
          .
100   DISPLAY "Block 100"
      RETURN
200   DISPLAY "Block 200"
      RETURN
300   DISPLAY "Block 300"
      RETURN

      Block 100
      Block 200
      Block 300
```

4.7 The END and STOP Statements

The END statement is used to indicate that the end of a program has been reached. It must be the last statement in a program. It has the format:

```
      END
```

The STOP statement causes execution to terminate. There may be more than one STOP statement in a program. It has the form:

```
      STOP
```

The following illustrates the use of these statements:

```
      ACCEPT "Enter a number": I
      IF I > 0 THEN 10
      DISPLAY I
      STOP
10    I=-I
      DISPLAY I
      END
```

CHAPTER 5

STANDARD FUNCTIONS

5.1 The Numeric Functions

The numeric functions take as an argument a numeric constant, variable, or expression. These functions may be used within assignment statements, PRINT or DISPLAY statements, ON statements, and function definitions.

5.1.1 The ABS Function

The ABS function returns the absolute value of the argument. A nonnegative argument will be returned unaltered. The following is an example of the ABS function:

```
I=2
J=-3
DISPLAY ABS(I); ABS(J)

 2  3
```

5.1.2 The SIN Function

The SIN function returns the sine of the argument passed in radians. In order to convert an angle from degrees to radians, multiply the number of degrees by PI/180. The following example illustrates the SIN function:

```
I=25.0
DISPLAY SIN(I)

-.13235175
```

5.1.3 The COS Function

The COS function returns the cosine of the argument passed in radians. In order to convert an angle from degrees to radians, multiply the number of degrees by PI/180. The following example illustrates the COS function:

```
I=25.0
DISPLAY COS(I)

.99120292
```

5.1.4 The TAN Function

The TAN function returns the tangent of the argument passed in radians. In order to convert an angle from degrees to radians, multiply the number of degrees by PI/180. The following example illustrates the TAN function:

```
I=25.0
DISPLAY TAN(I)

-.1335264
```

5.1.5 The ATN Function

The ATN (arctangent) function returns the angle in radians, which has the tangent equal to the argument passed. If degrees are desired, multiplying the output of the ATN function by 180/PI performs the conversion. The following example illustrates the ATN function:

```
I=25.0
DISPLAY ATN(I)

1.5308176
```

5.1.6 The EXP Function

The EXP function yields the value of e, the base of natural logarithms, raised to the power of the argument passed. The following example illustrates the EXP function:

```
I=25.0
DISPLAY EXP(I)

72005171000.0
```

5.1.7 The LOG Function

The LOG function yields the natural logarithm (base e) of the argument passed. The following example illustrates this function:

```
I=25.0
DISPLAY LOG(I)

3.2188754
```

5.1.8 The INT Function

The INT function returns the largest integer which is not greater than the argument:

```
DISPLAY INT(25.9); INT(-3.2)

 25 -4
```

5.1.9 The SGN Function

The SGN function returns 0 if the argument is zero, 1 if the argument is positive, and -1 if the argument is negative:

```
I=0
J=749
K=-1024
DISPLAY SGN(I); SGN(J); SGN(K)

 0  1 -1
```

5.1.10 The SQR Function

The SQR function yields the square root of the value passed. If the argument is negative, an error results. The following example illustrates the use of the SQR function:

```
I=25
DISPLAY SQR(I)

5
```

5.1.11 The RND Function and RANDOMIZE Statement

The RND function produces evenly distributed pseudo-random numbers which fall in the range X: 0 <= X < 1. It has the format:

```
RND
```

The RANDOMIZE statement can be used to specify a "seed value" which will generate a new sequence of numbers. It has the following format:

```
RANDOMIZE numeric_expression
RANDOMIZE
```

The same sequence of values will be produced by the RND function in different programs whenever no RANDOMIZE statement is used, or whenever there are two occurrences of the RANDOMIZE statement with the same seed value. If the RANDOMIZE statement is used without specifying a value, the seed value will be taken from the real-time clock. If there is no clock, the user will be prompted to enter a seed value. This will provide for an uncontrolled sequence of numbers.

The following example illustrates the use of these statements:

```
RANDOMIZE (.12345678)
FOR I=1 TO 5
  DISPLAY RND
NEXT I

.6213
.97039985
.41409874
.51999116
.2612381
```

5.2 String Functions

The string functions are used in conjunction with string variables. These functions may be used within assignment statements, PRINT or DISPLAY statements, ON statements, and function definitions.

5.2.1 The ASC Function

The ASC function returns the decimal value of the ASCII code for the first character in the string argument. It has the format:

```
ASC (string)
```

The string may be any valid string expression. The following example illustrates the use of the ASC function:

```
S$="&"
DISPLAY ASC(S$)

 38
```

5.2.2 The BREAK Function

The BREAK function finds the first character in a string that appears in a second string. It has the format:

BREAK (string_1, string_2)

Strings 1 and 2 are any valid string expressions. This function compares the first character in string_1 to all the characters in string_2. If there is no match, it compares the second character in string_1 to all the characters in string_2, etc. It returns the number of characters in string_1 which did not match any character in string_2 before a matching character was found. If no match was found, it returns the total number of characters in string_1. The following example illustrates this function:

```
S1$="ABC"
S2$="COW"
S3$="XXXX1"
DISPLAY BREAK (S1$,S2$)
DISPLAY BREAK (S2$,S1$)
DISPLAY BREAK (S1$,S3$)
DISPLAY BREAK (S3$,S1$)

 2
 0
 3
 5
```

5.2.3 The SPAN Function

The SPAN function compares the characters in one string with the characters in a second string until a character in the first string is not found in the second string. It has the format:

SPAN (string_1, string_2)

String_1 and string_2 are any valid string expressions. Consecutive characters of string_1 are compared to characters in string_2. When a character in string_1 is found which is not in string_2, SPAN returns the number of characters that did match. The following example shows the use of the SPAN function:

```
S1$="$$$ Hi there"
S2$="$"
DISPLAY SPAN(S1$,S2$)

 3
```

5.2.4 The LEN Function

The LEN function yields the number of characters in the string passed. It has the following format:

```
LEN (string)
```

The string is any valid string expression. The following example shows the use of the LEN function:

```
S1$="ABC"
S2$="ABCDEFGHIJKLMNOPQRSTUVWXYZ"
DISPLAY LEN (S1$)
DISPLAY LEN (S2$)

 3
 26
```

5.2.5 The NUMERIC Function

The NUMERIC function will determine whether or not the string passed represents a valid number. A -1 will be returned if it does, and a 0 will be returned if it does not. If a string represents a valid number, it may be passed to the VAL function (see Section 5.2.6). The following example illustrates the use of the NUMERIC function:

```
DISPLAY NUMERIC ("1234"); NUMERIC ("12ABC")

-1  0
```

5.2.6 The VAL Function

The VAL function returns the numeric value of the string argument. Leading and trailing blanks are permitted. Any string expression which is accepted by the NUMERIC function (see Section 5.2.5) may be passed to the VAL function without error. The following example illustrates the use of the VAL function:

```
S$="1.234"
K=VAL(S$)
DISPLAY K+0

 1.234
```

5.2.7 The STR$ Function

The STR$ function provides a counterpart to the VAL function. The STR$ function is passed a numeric value and returns the corresponding string value. It has the format:

```
STR$(numeric_expression)
```

The string returned is identical to the numeric value as it would appear on the console, i.e. it is preceded by a blank space or a minus sign, etc. The following example illustrates the use of the STR$ function:

```
1=1.234
S$=STR$(1)
DISPLAY (S$)

 1.234
```

5.2.8 The POS Function

The POS function is used to determine the position of one string within another. It has the following format:

```
POS (string_1, string_2, start)
```

String_1 and string_2 are any valid strings, and start is a numeric value. This function returns the position of the first occurrence of string_2 within string_1. The search will begin at character position start within string_1. Start is rounded to an integer value if necessary. If string_2 is not found within string_1, a 0 will be returned. The following example illustrates the POS function:

```
DISPLAY POS("ROW ROW ROW YOUR BOAT", "ROW", 1);
DISPLAY POS("ROW ROW ROW YOUR BOAT", "ROW", 2);

 1  5
```

5.2.9 The RPT$ Function

The RPT$ function returns a string which is a specified number of repetitions of the argument string. It has this format:

```
RPT$ (string, numeric_expression)
```

The string is any valid string. The numeric expression may evaluate to any nonnegative number less than 256. The string returned is that number of repetitions

of the string passed. If the resulting string has a length greater than 255 an error will result. The following example shows the use of this function:

```
DISPLAY RPT$("Cats ",3)

Cats Cats Cats
```

5.2.10 The UPRC$ Function

The UPRC$ function changes all lowercase letters in the string passed to upper case letters. Nonalphabetic characters remain the same. The following example illustrates the use of this function:

```
S$="Once upon a time..."
DISPLAY UPRC$(S$)

ONCE UPON A TIME...
```

5.2.11 The CHR$ Function

The CHR$ function takes a numeric argument, the value of which must fall between 0 and 255 inclusive. It returns a single-character string whose ASCII value is equal to that number. This complements the ASC function (see Section 5.2.1). Special control characters within DISPLAY or PRINT statements can be generated with this function. The following example shows the use of this statement:

```
DISPLAY CHR$(65)

A
```

5.2.12 The SEG$ Function

The SEG$ function extracts a segment of a string. It has the following format:

```
SEG$(string, position, length)
```

The string is any valid string expression. Position and length are numeric expressions which will be rounded to integers if necessary. Starting at position characters into the string, length characters will be extracted by the SEG$ function. If position is less than or equal to zero an error will result. If position is greater than the length of the string, a null string will be returned. If length is less than 0, an error will result. If length is equal to 0, a null string will be returned. If length plus position are greater than the remaining portion of the string, all of the string will be extracted starting at position. The following is an example of the use of the SEG$ function:

```
S$="But don't you step on my Blue Suede Shoes..."
DISPLAY SEG$ (S$,26,16)

Blue Suede Shoes
```

5.3 Miscellaneous Functions

The remaining standard functions discussed in this section are used in the same way as numeric and string functions.

5.3.1 The DAT$ Function

The DAT$ function returns the month, day and year in the form:

```
month/day/year
```

Month, day, and year are two-digit numbers. The following example displays the date using this function:

```
DISPLAY DAT$

02/16/81
```

5.3.2 The FREESPACE Function

The FREESPACE function returns the number of bytes available in memory. It has the following format:

```
FREESPACE(0)
```

If there were 5000 bytes available, the following would occur:

```
DISPLAY FREESPACE(0)

 5000
```

5.3.3 The INKEY and INKEY$ Functions

The INKEY function always returns zero. The INKEY$ function reads and removes a character from the keyboard input buffer, and returns a string consisting of that character. These functions have the following formats:

```
INKEY(0)
INKEY$(0)
```

The following example shows the use of the INKEY$ function:

```
S$=INKEY$(0)
```

5.3.4 The EOF Function

The EOF function is used to determine whether or not the end of a file has been read. It has the following format:

```
EOF(X)
```

X is a numeric expression that evaluates to the file number which was assigned when the file was opened. (See Chapter 7 for further information about files.) A zero is returned by the EOF function if the last record of the file has not yet been read. A 1 is returned if the last record has been read. A 2 is returned if an attempt has been made to read beyond the end of the file. A 4 is returned if the specified file number is not in use.

5.3.5 The FTYPE Function

This function always returns 0. The type of a file is determined by the name associated with it. See Chapter 7 for further information on files.

5.3.6 The TAB Function

The TAB function advances the cursor or printer head to a specified position. It has the following form:

```
TAB(position)
```

Position is the column number where the next output will begin. The position may be a numeric constant, variable, or expression. This number will be rounded to an integer if necessary. It must be nonnegative, and the value actually used is this number modulus the output width of the device. If the cursor is already past the specified position, it will be advanced to that position on the next line. The following illustrates the use of this function:

```
DISPLAY TAB(10);CAT;
DISPLAY TAB(12);CAT;

          CAT
            CAT
```

5.3.7 The ERR Function

The ERR function returns an integer error number which indicates the last exception which took place. If there has been no error, a zero is returned. The following illustrates the use of the ERR function:

```
IF ERR > 0 THEN DISPLAY "ERROR ";ERR
```

5.3.8 The TIME$ Function

The TIME$ function returns a string which represents the current time based on the real time clock (if the computer is equipped with one). The following example shows the use of this function:

```
DISPLAY TIME$

11:24:10
```

This call to TIME$ yielded 24 minutes and 10 seconds after 11.

CHAPTER 6

USER-DEFINED FUNCTIONS AND SUBROUTINES

6.1 Functions

Functions are defined using the DEF and FNEND statements. A function consists of one or more statements which are executed each time the function is called. A function may have parameters passed to it, and it may have local variables. Functions may also reference variables which are global to them. In the body of the function, a value is assigned to the function name. This value will be evaluated when the function is called, and then returned as the value of the function. A function may call itself and other functions recursively. Subroutines may also be called by functions and call functions in the same way (see Section 6.2). Functions and subroutines may be nested to any depth.

6.1.1 The DEF Statement

The DEF statement is used to indicate a function definition. It has the following formats:

```
DEF func_name
DEF func_name (param_list)
DEF func_type func_name
DEF func_type func_name (param_list)
```

The func_type and param_list are optional and are described in the following paragraphs.

For single-statement functions, the DEF statement defines the entire function. For multiple-statement functions, the DEF statement defines the function name, type, and parameters. Function definitions are commonly placed at the beginning of a program. The following example shows the use of the DEF statement to define a single statement-function:

```
DEF A_FUNC(I,J,K)=I*J*K+200
```

A_FUNC is the name of this function. I, J, and K are parameters passed to it. The specified number of parameters must be passed each time the function is called. The value returned is the expression on the right side of the equal sign. The following is an example of a multiple statement function-definition:

```
DEF A_FUNC(I,J,K)
A_FUNC=I*J*K+200
FNEND
```

This function is equivalent to the one line function above. If the function name is assigned a value more than once in the body of the function definition, the last assignment before the execution of the FNEND statement determines the runtime function value.

Function parameters are always passed by value. (Subroutine parameters are passed by reference, see Section 6.2.) Any number of parameters may be passed to a function. If necessary, the comment delimiters (* and *) may be used to comment out the end of the line so that more than one line may be used to declare the parameters. The parameters may be subscripted variables. In this case, they are defined exactly as they would be within a DIM statement. The following illustrates the use of comments to extend parameter lists and subscripted variables within parameter lists:

```
DEF A_FUNC(A,B,C,D,E,F,G,H,I, (*
          *) J,K,L,M)
DEF A_FUNC( A(3,4), B$(5)*4 )
```

The function name may be any legal variable name which corresponds to the function type (numeric or string). If the function returns a numeric value, the function type may be optionally specified as INTEGER, REAL or DECIMAL. For example:

```
DEF INTEGER A_FUNC (I,J,K)=I*J*K+200
DEF REAL B_FUNC (I,J,K)=I*J*K+200
```

String functions may be defined to return a string that has a maximum length:

```
DEF MONTH$ (JULIAN)*3
```

6.1.2 The FNEND Statement

The FNEND statement indicates the end of a multiple statement function definition. This statement must be the last statement within the function body. A single statement function does not require an FNEND statement. The following example illustrates the use of this statement:

```
DEF F
   .
   .
   .
FNEND
```

6.1.3 Calling Functions

User-defined functions, like the standard functions, may appear any place an expression of the function type is permitted. The type and number of parameters specified in the function definition must be passed at the call. An array element or an entire array may be passed (or any other type of variable which matches the definition). Only the array name is specified when an entire array is passed. The following are examples of function calls:

```
I=FUNC1(I,J,K)
S$=FUNC2$( S1_ARRAY$(I,J), S2_ARRAY$, INT )
DISPLAY FUNC3
DEF FUNC4(I)=I*FUNC3
```

6.2 Subroutines

Subroutines are similar to functions. There are three differences between them. First, the subroutine name may not be assigned a value as in a function. Rather than being used within assignment statements like functions, subroutines are called using the CALL statement (see Section 6.3). Second, parameters are passed to subroutines by reference (not by value as they are to functions). This means that if an assignment is made to a parameter within a subroutine, the contents of the actual location of that variable within the caller will be altered. And third, a subroutine may not reference variables which are global to it (see Section 6.4). This includes other subroutines or functions.

Subroutines, like functions, may be nested to any depth. Subroutines may call themselves recursively. Subroutines may also have any number of parameters.

6.2.1 The SUB Statement

Subroutines are defined using the SUB statement. The SUB statement has the following formats:

```
SUB subr_name
SUB subr_name (param_list)
```

This statement is similar to the DEF statement for functions (see Section 6.1.1).

Like functions, this heading is followed by one or more statements. The parameter list follows the same rules as for functions.

6.2.2 The SUBEND and SUBEXIT Statements

Subroutines are terminated with the SUBEND statement in the same way that functions are terminated with the FNEND statement. But subroutines must always include the SUBEND statement, unlike one-line functions which do not need the FNEND statement. A subroutine may be specified on a single line by using double colons to separate statements. The SUBEND statement must be the last statement of subroutine.

There may be zero or more SUBEXIT statements within the subroutine body. When a SUBEXIT statement is encountered, the subroutine is exited in the same manner as it would be had the SUBEND statement been executed. The following examples illustrate the use of the SUBEND and SUBEXIT statements:

```
      SUB S1 (A) :: A=1 :: SUBEND

      SUB S2 (I,J)
      IF I=J THEN 100
            .
            .
            .
      SUBEXIT
100         .
            .
            .
      SUBEND
```

6.3 The CALL Statement

The CALL statement is used to invoke a subroutine. It has the following formats:

```
CALL subroutine_name
CALL subroutine_name (param_list)
```

When the subroutine returns, execution resumes at the statement immediately following the CALL statement. The number and type of the parameters within the CALL statement must match the definition of the subroutine. The actual variables passed as parameters may be altered by the subroutine. The following examples demonstrate the CALL statement;

```
CALL SUB1
CALL SUB2 (I,J,K)
```

```
CALL SUB3 ( A(1,J), S_ARRAY$(,) )
```

6.4 Local Variables and Parameters

Parameters and local variables are valid only within the body of the function or subroutine. Numeric parameters may be defined to be INTEGER, REAL, or DECIMAL by using those statements when the parameters are declared. String parameters may be declared using the DIM statement if desired. Local variables are declared using the INTEGER, REAL, DECIMAL, and DIM statements on the lines which follow the function or subroutine heading. The following examples illustrate the declaration of function and subroutine parameters and local variables:

```
DEF FUNC1
  INTEGER A,B
  REAL R1,R2
  DIM S$*10
     .
     .
     .
SUB SUB1(INTEGER J,K, S_ARRAY$(2,3)*10, S$ )
  DIM C$*20
  INTEGER AN_ARRAY(2,3,4)
     .
     .
     .
```

In the above examples, FUNC1 is defined to have local variables A and B as integers, R1 and R2 as reals, and S$ as a ten-byte string. Subroutine SUB1 has parameters integer J, default type K (real unless otherwise specified), two-dimensional string array S_ARRAY$ and string S$. SUB1 also has local variables C$, a twenty-byte string, and AN_ARRAY, a three-dimensional integer array.

All local variables are cleared to zero, and local strings are set to null, each time the function or subroutine is invoked.

All variables within a subroutine are local to it. A subroutine may not access any variables external to it except through its parameter list.

A function may access variables in the standard block-structured manner. Any variables which are global to a function up to the subroutine (or program) definition may be accessed. If variables are not declared as parameters or local variables within functions, they will be assumed to be global.

6.5 Line Numbers and Data Lists Within Subroutines and Functions

Statement line numbers are local to functions and subroutines. This means that any GOTO, GOSUB, ON, RESTORE line_number or PRINT USING line_number statements will refer to line numbers within the function or subroutine only.

DATA lists and READ data statements are local to subroutines and functions. A RESTORE is executed automatically on entry to subroutines and functions.

6.6 The USES and LIBRARY Statements

BASIC programs may use separately compiled BASIC, Pascal, or FORTRAN units. These units contain functions and subroutines (or procedures) which may be called by a BASIC program. Units reside on disk, either in SYSTEM.LIBRARY or in some user-created file.

Units may have interface text which declares variables, functions, and subroutines (or procedures) to be externally recognized by the host program. This text is compiled when, during compilation of the host BASIC program, a USES statement is encountered. The USES statement indicates that one or more specified unit(s) within SYSTEM.LIBRARY are to be used by the BASIC program. It has the following format:

```
USES unit_name1, unit_name2, unit_name3, ...
```

If the unit(s) reside in a file other than SYSTEM.LIBRARY, that file may be specified with the LIBRARY statement. It has the format:

```
LIBRARY "file_name"
```

The default library is SYSTEM.LIBRARY. Once the LIBRARY statement has been executed, the indicated file will serve as the library where all units are to be found by all USES statements, until another LIBRARY statement is executed. The following is an example of these two statements:

```
USES PASCALIO

LIBRARY "MY.LIB.CODE"
USES MY_UNIT1, MY_UNIT2, MY_UNIT3
USES MY_UNIT4

LIBRARY "#5:ANOTHER.CODE"
USES ANOTHER_UNIT
```

In this example, PASCALIO will be found in SYSTEM.LIBRARY, MY_UNIT1 through

MY_UNIT4 will be found in MY.LIB.CODE on the prefixed disk, and ANOTHER_UNIT will be found in ANOTHER.CODE on the disk in drive #5.

In the II.0 version of the UCSD p-System, after compilation the unit must be linked into the codefile. This may be done by invoking the Linker or, if the unit resides within SYSTEM.LIBRARY, by simply R(UNning the program from the main system prompt which will automatically use the Linker. In the IV.0 version of the UCSD p-System, the unit must be connected to the host codefile using the Librarian, or if the unit resides in SYSTEM.LIBRARY, it will be called directly from there during program execution. More information on units, the Linker and the Librarian may be found in the UCSD Pascal Users' Manual.

6.7 Pascal Interface Text Restrictions

The BASIC compiler will parse the interface sections of Pascal units subject to some restrictions. First, only the following simple types (which corespond to BASIC types) are allowed:

```
INTEGER
REAL
STRING or STRING[ <size> ]
ARRAY [ <dimensions> ] OF <one of the preceding simple types>
ARRAY [ <dimensions> ] OF ARRAY[ <dimensions> ]  OF ...
```

No other types, and no user-defined types are permitted. A second restriction is that no constants are allowed within interface sections. For example, the following would not be correct:

```
ARRAY [LOW_INDEX, HIGH_INDEX] OF INTEGER;
```

Arrays such as this may be declared if the indices are ordinary integers.

Procedure and Function declarations are allowed, as long as the type of the parameters and the function type conform to these same restrictions.

The BASIC program should refer to any numeric Pascal variables which exceed eight characters in length by the first eight characters only. Alternatively, the Pascal program may be written so that no externally reconizable numeric variables exceed that length. If a Pascal string variable does not exceed eight characters in length, it should be refered to by BASIC with a dollar sign appended. If it does exceed that length, if should be referred to by BASIC as its first eight characters with a dollar sign appended.

6.8 The UNIT Statement

In order to create a BASIC unit, which is separately compiled and used by a host BASIC, Pascal, or FORTRAN program, the UNIT statement is used. This statement has the following format:

```
UNIT unit_name
```

The UNIT statement should be at the beginning of the text. Following this heading, an interface section may be declared using the INTEGER, REAL, DECIMAL, and DIM statements. Following this, Functions and Subroutines may be declared. These routines will be accessible to the host program. There should be no main program in the unit. The following is an example of a BASIC unit:

```
UNIT MY_UNIT
  INTEGER I,J
  DIM S$*20

DEF A_FUNC (A,B,C)
  IF A > 0 THEN GOTO 10
     .
     .
     .
10   FNEND

SUB A_SUB (PARAM$)
  IF S$=PARAM$ THEN I=2
     .
     .
     .
  SUBEND

END
```

In the above unit, integers I and J, and string S$ are able to be referenced from the host program. They are essentially global variables in the host. A_FUNC and A_SUB are accessible to the host program also. They are global routines within the host as if they had been declared, like any other function or procedure, at the beginning of the host.

All variable, function and subroutine names should be distinguishable by their first eight characters if the unit is to be used by a Pascal host program. This is because Pascal only distinguishes identifiers by their first eight characters. Also, no special characters may be used in the BASIC variable, function and subroutine names because Pascal allows only alpha-numeric characters within identifiers. The

single exception to this is the dollar sign required at the end of a string variable name.

The BASIC compiler will convert the externally recognizable BASIC text into a Pascal interface section so that the Pascal compiler may use the compiled BASIC unit.

CHAPTER 7

FILE I/O AND VIRTUAL ARRAYS

7.1 Opening and Closing Files

SofTech Microsystems BASIC allows the user to access disk files. The file must be created, or opened if it already exists, before it can be accessed. The file should be closed before the program terminates. If an error occurs during program execution, the files left open will remain open.

7.1.1 The OPEN Statement

The OPEN statement will either open an existing file or create a new one. Once the file is open, records within it may be accessed until the file is closed. The OPEN statement has the following format:

```
OPEN #file_num: "file_name", attributes
```

File_num is a numeric expression which has a positive value less than 256. This number will be associated with the file as long as it is open. File_num should not be assigned to any other file until this file is closed. File_name is a valid UCSD p-System file name. This name may include a unit number (such as #4:FILE.TEXT) or a unit name (such as DISK1:FILE.TEXT). The attributes are one or more of the file attributes which determine: File Access Mode, File Organization, File Length, File Format, Record Type, and Record Length. These are discussed in the following sections.

If any of the attributes in the following two lists are used, they must appear in the order shown:

```
SEQUENTIAL - DISPLAY - VARIABLE - File Access Modes
RELATIVE - INTERNAL - FIXED - File Access Modes
```

7.1.2 File Access Modes

The OUTPUT access mode indicates that the file to be opened is to be created as a new file. New records may be written to a file declared with this mode. This mode must be used if a new file is to be created. A device, such as a disk drive, may not be opened with this mode, because a device cannot be created. The following example will create FILE.TEXT on the disk in drive #4 and associate the file with file number 1:

```
OPEN #1: "#4:FILE.TEXT", OUTPUT
```

The INPUT access mode indicates that the file may be read from. If INPUT is the only attribute used, an attempt to write to the file will result in an error. If both the OUTPUT and INPUT attributes are used, a new file will be created which can be written to and read from. The following examples illustrate the use of the INPUT attribute:

```
OPEN #1: "#4:FILE.TEXT", INPUT
OPEN #1: "#4:FILE.TEXT", INPUT, OUTPUT
```

The UPDATE access mode indicates that the file may be read from and written to. This is the default mode. If this mode is used with the OUTPUT mode, a new file will be created which can be read from and written to (this is equivalent to using the combination of INPUT and OUTPUT access modes). The following is an example of the use of the UPDATE mode:

```
OPEN #1: "#4:FILE.TEXT", UPDATE
```

The APPEND access mode is used only with sequential files and indicates that records may be written to the end of the file. No reads may be done, nor may a RESTORE statement be used on the file. All records will be written, sequentially, starting at the end of the file. The following is an example of the use of the APPEND access mode:

```
OPEN #1: "#4:FILE.TEXT", APPEND
```

7.1.3 File Organization

Files may be opened with either of two file organization attributes: SEQUENTIAL or RELATIVE. If no attribute is specified, SEQUENTIAL is assumed. Virtual Arrays, which allow a file to be accessed as though it were an array in memory, are RELATIVE files.

SEQUENTIAL files are identical to Pascal text files. SEQUENTIAL files are written to and read from in sequential order, beginning with the first record in the file. Also, records may be appended to the end of existing SEQUENTIAL files.

Peripheral devices are treated as SEQUENTIAL files. A device may be opened and, if the device allows, written to or read from. The following examples illustrate opening the Printer (LP01 and PRINTER: are treated identically):

```
OPEN #1:"PRINTER:"
OPEN #2:"LP01"
```

An ordinary SEQUENTIAL file may be opened as follows:

```
OPEN #1:"#5:FILE.TEXT", INPUT, SEQUENTIAL
```

RELATIVE files allow sequential access and random access to the records within a file. If a RELATIVE file is to be opened, the keyword RELATIVE must be one of the specified attributes. If a new RELATIVE file is to be created, the attributes

INTERNAL and FIXED must also be specified. The following examples illustrate how relative files are opened:

```
OPEN #1:"#5:FILE1", RELATIVE, INPUT
OPEN #2:"#5:FILE2", RELATIVE, INTERNAL, FIXED, OUTPUT
```

If a record in a RELATIVE file is to be accessed out of sequence, that record is specified by a number which represents its position within the file. The first record within a file is record number zero. If it is desired to access the 12'th record in the file, the number eleven should be specified. This record number is specified within the REC clause of an INPUT, ACCEPT, RESTORE, or PRINT statement (see Section 7.2.2).

7.1.4 File Length

When a RELATIVE file is created, it is assigned a maximum file length. It may not expand beyond that size. This file length may be specified as one of the attributes. If it is not, a default size of 144 logical records will be assumed. After a file reaches its maximum size, it must be copied into a larger file before more records may beadded to it. The following is an example of file length specification:

```
OPEN #1:"#5:FILE2", RELATIVE 700, INTERNAL, FIXED, OUTPUT
```

7.1.5 File Format

The information within files may be stored in either of two formats: DISPLAY or INTERNAL.

The DISPLAY format is used with sequential files and indicates that the data is stored in ASCII format. This type of file typically contains text and string data. DISPLAY is the default file format attribute, and may be used only with SEQUENTIAL files. The following example shows the use of the DISPLAY attribute:

```
OPEN #1:"#5:F1.TEXT", SEQUENTIAL, DISPLAY, UPDATE
```

The INTERNAL format must be specified when opening or creating RELATIVE files. The INTERNAL format indicates that the data within the file is stored in binary format. This type of format is especially useful when dealing with numeric quantities. When a value is written to this type of file, it is stored in the format which corresponds to its declaration within the program, e.g. INTEGER, REAL or DECIMAL. It is important, therefore, that the data be read and written using variables of the same type. The following example illustrates the use of the INTERNAL file format attribute:

```
OPEN #2:"#5:FILE2", RELATIVE 700, INTERNAL, FIXED, OUTPUT
```

7.1.6 Record Length

Records in a SEQUENTIAL file may have a fixed or variable length. The length attribute should be specified in the OPEN statement when a file is created. This "logical record length" should be greater than or equal to 2, and less than 32767 (bytes), and must be an even number.

The VARIABLE attribute indicates that records in the file may be of different lengths. An optional maximum length may be specified by a number following the keyword VARIABLE. If no maximum length is specified, a default of 80 bytes will be assumed. If a record being written to the file exceeds the maximum record size, the current record slot is terminated, and the remaining data is written into the next record. The VARIABLE attribute is assumed if no record length attribute is indicated. The VARIABLE attibute may be used in conjunction with sequential files only. In the following example, records have a variable length with a maximum size of 200 bytes:

```
OPEN #1:"#5:F1.TEXT", SEQUENTIAL 500, VARIABLE 200, OUTPUT
```

The FIXED attribute indicates that the records in the file are all of the same size. The size is specified to the right of the keyword FIXED. Any valid INTEGER expression may be used to specify this length. An attempt to read or write records which are not of the correct size will result in an error. RELATIVE files must be created with the FIXED attribute. Also, when an existing RELATIVE file is opened, the FIXED attribute must be specified and the record size must match the size specified when the file was created. The default length for fixed records is 256 bytes. The following example creates a RELATIVE file with 32-byte records:

```
OPEN #2:"#5:FILE2", RELATIVE 700, INTERNAL, FIXED 32, OUTPUT
```

7.1.7 The ASSIGN Statement and Virtual Arrays

The ASSIGN statement is used to associate a Virtual Array with a disk file. After the ASSIGN statement has been executed, assignments may be made to the Virtual Array, in which case the disk file is written to. Also, variables may be assigned from the Virtual Array, in which case the disk file is read in order to obtain the needed values. In this way Virtual Arrays may be used as ordinary arrays, but instead of taking up space in main memory, they actually exist on disk. Very large arrays may be used in this manner, without the danger of running out of memory space. The total number of elements within an array, however, may not exceed 32767. The following is an example of the use of the ASSIGN statement to create a Virtual Array:

```
ASSIGN "#4:REAL.FILE" USING V_ARRAY(100,100)
```

In this example, REAL.FILE on disk #4: is opened and associated with V_ARRAY. V_ARRAY may now be used as any other array, but whenever it is accessed, #4:REAL.FILE is really used. V_ARRAY is declared as it would be within a DIM statement and may be preceded by INTEGER, REAL or DECIMAL. The following example shows the use of Virtual Arrays:

```
ASSIGN "#4:REALFILE" USING INTEGER V_ARRAY(100,100)
ASSIGN "#5:REALFILE2" USING STRINGS$(10,10,100)*10
V_ARRAY(99,97) = 1234
STRINGS$(2,8,97) = STR$( V_ARRAY(99,97) )
      .
      .
CLOSE V_ARRAY
CLOSE STRINGS$
```

The CLOSE statements are used to close the disk files, see Section 7.1.8.

7.1.8 The CLOSE Statement

The CLOSE statement ends the association between an opened file and its file number. The CLOSE statement is also used to end the association between a disk file and a virtual array name. The file number or virtual array name is then available to be re-used if desired. The closed disk file is inaccessible to the program unless it is re-opened. If the file number is not associated with an open file when the CLOSE statement is executed, an error will result. The EOF function can be used to determine if this association exists. The CLOSE statement has the following formats:

```
CLOSE #file_number
CLOSE virtual_array_name
```

Virtual arrays are also implicitly closed when a STOP, END, or RUN statement is executed.

When a new file is opened, it must be explicitly closed with the CLOSE statement if it is to remain on disk after the program has finished execution. By adding the word DELETE to a CLOSE statement, the closed file will be removed from the disk directory even if it existed before the program opened it. The following shows the use of the DELETE option:

```
OPEN #2:"#4:JUNK.TEXT"
CLOSE #2:DELETE
```

7.2 File I/O Statements

Records within a file may be accessed using the INPUT, ACCEPT, and PRINT statements. After the file has been opened, the execution of one of these statements causes a record to be read from or written to the indicated file. If a data separator (comma, semicolon, or apostrophe) is used to terminate one of these statements, I/O will be defered. The RESTORE statement is used to select, in a random access manner, the next record on which I/O statements will perform their function. The following are the simplest formats of these statements:

```
PRINT #file_number: variable_list
INPUT #file_number: variable_list
ACCEPT #file_number: variable
RESTORE #file_number
```

The PRINT statement outputs the variables listed to the indicated file. The INPUT and ACCEPT statements input from the file indicated, to the variables in the list. After one of these statements is executed, an internal record pointer is advanced to the next record. The RESTORE statement points the internal record pointer to the first record in the file.

If a record contains more variables than can be listed on one line, the I/O statement may be terminated with a data separator and further statements can be used to complete the I/O.

7.2.1 Sequential File I/O

Because sequential files use the DISPLAY format, variables written to them will be in the same format as if they were written to the console. In the following example the variables I, J, and K are written with trailing blanks, and a preceding blank or minus sign (because of the comma data separator, see Section 3.1). The file is then restored, and the variables are read back into a string. The Standard Functions could then be used to convert the string into three separate numeric values again.

```
PRINT #1: I,J,K
RESTORE #1
INPUT #1:THREE_VARS$
```

In the next example I, J, and K are written with commas between them (because of the apostrophe data separator, see Section 3.1). This allows the INPUT statement to read the three separate variables directly as numeric values. Alternatively, the ACCEPT statement (which doesn't treat commas as data separators) can be used to read the variables into a string.

```
PRINT #2: I'J'K
RESTORE #2
INPUT #2: I,J,K
RESTORE #2
ACCEPT #2:THREE_VARS$
```

If there are fewer variables in a record than in the variable list of an INPUT statement, subsequent records will be read until enough variables are obtained. If there are more variables in a record than in the variable list of an INPUT statement, the remaining variables will be discarded unless the variable list is terminated with a data separator. In this case the next input operation will read further data from the same record.

The following example shows how 40 variables may be written to the same record:

```
FOR I=1 TO 40 DO
    PRINT #1:DATA_ARRAY(I)'
NEXT
PRINT #1:
```

In this example, the apostrophe indicates that more data is still to be written to the same record after the current PRINT statement is finished. (The apostrophe also inserts commas between each value written to the file.) The PRINT statement after the loop completes the record. Future PRINT statements to this file will write to the next record.

The USING option (see Section 3.2.5) can be used in conjunction with the PRINT statement to files. This allows formatting of the file contents. The following is an example of the PRINT-USING statement to a file:

```
PRINT #1 USING ###.##: NUM
```

7.2.2 Relative File I/O

Relative files always contain data in the INTERNAL format. This format may be INTEGER, REAL, DECIMAL, or string. Because of this, the apostrophe data separator, which produces commas in the output, may not be used with relative files. The input variables used to read in data should be identical in number and type to the output variables used to write the data originally. Otherwise, incorrect results will occur. If string variables are written to relative files, the runtime length of the string determines the number of characters written.

A particular record in a relative file may be accessed using the REC clause. The keyword REC is followed by the number of the record (zero is the first record in the file).

In the following example two strings are written and then read from the 10'th record in a file:

```
S1$="STRING"
S2$="FELLOW"
PRINT #1, REC 9: S1$,S2$
INPUT #1, REC 9: S3$,S4$
```

Fourteen bytes are written, a length byte and six characters for both S1$ and S2$. The INPUT statement assumes the first character is a length byte and assigns S3$ accordingly. The byte which follows the first string is then assumed to be the length byte for the second string, which is assigned to S4$.

If several statements are needed to write to or read from a single record, a data separator can be placed at the end of each statement. This defers the final I/O to that record. Only the first such statement should contain a REC clause, however, since each occurrence of a REC clause will cause a new record to be accessed. The following example writes 40 values to record N:

```
PRINT #1, REC N: INFO(1);
FOR I=2 TO 40
   PRINT #1: INFO(I);
NEXT
PRINT #1
```

7.2.3 The RESTORE Statement

The RESTORE statement is used to reposition the internal file record pointer to a specific record. SEQUENTIAL files use the RESTORE statement with the following format:

```
RESTORE file_number
```

The indicated SEQUENTIAL file is repositioned to the first record within it.

RELATIVE files use the RESTORE statement with the following formats:

```
RESTORE file_number
RESTORE file_number, REC record_number
```

The indicated file's internal record pointer is set to the record number specified in the REC clause. If no REC clause is specified, the file is repositioned to the beginning. In the following example, the file is reset to the tenth record:

```
RESTORE #1, REC 9
```

The next read operation reads that record.

APPENDIX A

BASIC RESERVED WORDS

ABS
APPEND
AT
BELL
CHR$
DAT$
DEF
ELSE
ERASE
EXP
FOR
GO
IF
INKEY$
INTEGER
LET
NOT
OPTION
PRINT
READ
RELATIVE
RETURN
SEG$
SIN
SQR
STR$
SUBEXIT
THEN
UNIT
VAL

ACCEPT
ASC
ATN
BREAK
CLOSE
DATA
DIM
END
ERR
FIXED
FREESPACE
GOSUB
IMAGE
INPUT
INTERNAL
LIBRARY
ON
OUTPUT
PUNCTUATION
REAL
REM
RND
SEQUENTIAL
SIZE
STEP
SUB
TAB
TIME$
USES
VARIABLE

ALL
ASSIGN
BASE
CALL
COS
DECIMAL
DISPLAY
EOF
ERROR
FNEND
FTYPE
GOTO
INKEY
INT
LEN
LOG
OPEN
POS
RANDOMIZE
REC
RESTORE
RPT$
SGN
SPAN
STOP
SUBEND
TAN
TO
USING

APPENDIX B

BASIC ERROR NUMBERS

Compiler Errors

1. Illegal or missing label
2. Illegal or missing variable name
3. Duplicate ALL statements
4. Doubly-defined variable
5. Right parenthesis or comma is expected here
6. Bad format
7. Integer is expected here
8. No scale in decimal statement
9. Illegal character in text
10. Illegal statement
11. FOR without matching NEXT
12. Array is too large
13. Illegal variable type
14. Illegal operator in statement
15. Wrong number of dimensions
16. Wrong number of arguments to function or subroutine
17. Missing BASE in OPTION statement
18. Bad number in OPTION statement
19. Too many OPTION statements
20. Function reference is not allowed here
21. Expression should start with a constant or variable
22. Doubly-defined label
23. Too much stuff in statement
24. Missing equal-sign in assignment statement
25. Missing THEN in IF statement
26. Missing GOTO in ON statement
27. Missing equal-sign in FOR statement
28. NEXT statement without FOR statement
29. Missing TO in FOR statement
30. Undefined label
31. Colon is expected here
32. Missing ALL after ERASE
33. Left parenthesis expected here
34. REC clause is expected here
35. Too many input variables in the ACCEPT statement
36. Variable is expected here
37. String is expected here
38. Array dimension is too small
39. Pound (#) is expected here
40. Delete is expected here
41. Comma is expected here
42. File types conflict or are inconsistent
43. Modes conflict

44. USING is expected here
45. Missing exponent in number
46. FNEND not expected
47. SUBEND not expected
48. Function name not expected
49. Too many jumps
50. Too much object code
51. Out of memory space
52. Number is too large
53. Number is expected here
54. Missing GOTO or GOSUB
55. Right parenthesis or semicolon is expected here
56. Semicolon is expected here
57. Too many units are included
58. Unit not found in library
59. Error attempting to open include or uses file
60. Variable in NEXT statement does not match FOR statement
61. Too many UNIT statements
62. Too many subroutines and functions
63. Subroutine call is expected here
64. SUBEND or FNEND is expected here
65. END expected

Execution Errors

1. String size error
2. Missing or bad number
3. File is not open
4. Not enough input
5. Bad number (conversion from string)
6. Too much input
7. Too many variables for print image
8. Image field error
9. End of data list
10. Wrong type of data in data list
11. File types do not match
12. You tried to open an open file
13. You cannot restore a sequential file
14. Read record overflow of relative file
15. Write record overflow of relative file
16. Bad arguments to SEG$ function
17. Number too large for exponentiation
18. Negative argument in exponentiation
19. ON statement index is out of bounds
20. You cannot write to a read-only file
21. You cannot read from a write-only file
22. You cannot close file #0.
23. You cannot close a closed file
24. You cannot open-for-output an existing file
25. You cannot open-for-output a device
26. Relative record number is too large or too small
27. You cannot restore an APPEND file
28. The number of records in the OPEN statement is bad
29. The record size in the OPEN statement is bad
30. Too many returns from GOSUB
31. Too many GOSUB statements
32. FREESPACE argument is not zero

NOTES

NOTES

NOTES

SOFTECH MICROSYSTEMS

A SUBSIDIARY OF SOFTECH

UCSD p-SYSTEM and UCSD PASCAL

A PRODUCT FOR MINI- AND MICRO-COMPUTERS

Version IV.0

INTERNAL ARCHITECTURE GUIDE

First edition: March 1981

9494 Black Mountain Rd., San Diego, CA 92126 (714) 578-6105 TWX: 910-335-1594

This guide was written by Gail Anderson, Randy Clark, Chip Chapin, Bill Franks, Mark Overgaard, and Stan Stringfellow, and edited by Randy Clark. Much advice and information was supplied by Rich Gleaves, Steve Koehler, and Mark Overgaard.

The editor feels this is the appropriate place to thank all the people at Arts & Crafts Press, both for the quality of their work, and their truly admirable patience.

TABLE OF CONTENTS

FIGURES:

I. INTRODUCTION

I.1 Purpose of this Guide

This guide describes the internal design of the UCSD p-System: the P-machine, Operating System, basic I/O, and the way in which these elements are organized to support the running of a program written in UCSD Pascal (or BASIC or FORTRAN).

It should serve as a guide and reference for more advanced users of the System, but is not intended to be a standalone definition for the use of implementors. Such a definition does not yet exist; if one is written, it will probably be based on the format of this book.

Perhaps the best way to use this guide is to read it sequentially, skipping those sections (such as the list of P-codes) that go into very specific detail. This should give the reader a fairly complete picture of what goes on within the System. If the user then needs to know specific internal details, the relevant section can be referred to later.

While few users will want or need to implement a p-System from scratch, the internal descriptions provided in this guide should be useful to a number of audiences.

The largest audience is probably those who will make no specific use of the information. To these users, the benefit will be a better understanding of the System's operation and a general improvement in their ability to engineer programs for effective execution in the p-System environment.

Second, there are the implementors of system software facilities that complement existing System capabilities: for instance, new language translators, new System utilities, or Interpreters for additional processors. For this group of programmers, the Architecture Guide presents more information than was available in the past.

Finally, there are the implementors with a compelling need to use facilities such as, for instance, the ability to explicitly generate P-codes in a Pascal program, where an ordinary Pascal construct would not suffice (we take it for granted that only a compelling need would lead a user to take such steps).

All of these audiences (but particularly the last) should understand that the principal commitment of SofTech Microsystems (and its licensees) is to the user facilities described in the Users' Manual, and not to any of the specific implementation strategies that are described in this guide. Programmers who take advantage of "internal tricks" do so at their own risk.

1.2 A Brief History of the System

The software system that is now called the UCSD p-System began when Kenneth Bowles was responsible for teaching the introductory programming course at the University of California, San Diego. In late 1974, under Bowles' direction, a group of undergraduate and graduate students began to implement Pascal for microcomputers.

Before this time, the introductory programming course had been taught using a large time-shared computer (on campus it was popularly called "The Beast"). This presented a bottleneck: many people used the machine, so its turnaround was sometimes quite slow, and a student's productivity was to some extent limited by the availability of the card punches. Furthermore, the machine's time-sharing environment, its accounting system, its complexity, and the amount of sensitive information that it stored prevented the student from any extensive "hands on" use of the machine or its facilities. In brief, the Beast was intimidating.

These were the main reasons for the decision to change the nature of the beginning programming course. It would be self-paced, to accommodate the large number of students, and each individual student's study habits (UC -- Irvine's physics program had been doing this successfully for a couple of years). It would use Pascal, rather than the dialect of Algol that was specific to the University's large time-sharing computer. And it would use microcomputers.

The decision to use small computers was motivated partly by their low cost, and partly by the desire to give students an opportunity to program in an interactive environment. The System was first implemented for a number of PDP-11/10's with floppy disks and VT-50 terminals. Students were expected to buy their own floppy disk, and use it for storing the System and their own programs.

It was the interactive environment that led to some of UCSD Pascal's deviations from the standard language, mostly as regards INTERACTIVE files and the handling of EOF and EOLN. The type STRING came about from the desire to teach basic programming concepts without recourse to numerical problems (which distracted many students from the actual problems of programming).

The user interface of the System, by which we mean the philosophy of displaying a promptline at every level of the System, and organizing these promptlines in a tree structure, was intended to be easy to learn for the complete novice, yet usable (i.e., not cumbersome) for the experienced user. This proved very successful, and has been retained.

The interpretive approach to executing Pascal was present from the beginning. P-code, adapted from the original design by Urs Amman of the Eidgenossische Technische Hochschule in Zurich, was designed to be compact and easily generated

by a Compiler; because of the constraints of the microprocessor environment, the goal was to keep the Compiler and the codefiles as small as possible. The tradeoff in execution time was felt to be an affordable cost (time has borne out this decision).

All of the original implementations were on PDP-11/LSI-11 machines. Because of the interpretive approach, it was a relatively straightforward matter to re- write the Interpreter for the 8080 and Z80, and subsequently for many other processors.

This adaptation of the Interpreter was originally motivated by the search for cheaper hardware, but it was soon recognized that software portability was valuable in itself. The economics of the computer business, especially the microprocessor field, dictated this: it is not a new observation that hardware costs continue to plummet, while software, being "hand-made", continues to be very expensive; it is relatively new to encounter a software system that, through modularity and portability, addresses the problem as thoroughly as does the p-System.

This is a brief view of the System as it was first created at UCSD. It was created to fill a need within the University, and other issues were subordinate to that need. Thus, despite the innovations within the System, it came as quite a surprise to learn that there was considerable commercial interest in the System. This commercial interest ultimately led the University to turn the "Pascal Project" over to a licensee, and proceed with other projects. The firm of SofTech Microsystems was created with the original purpose of supporting, maintaining, licensing, and further developing UCSD Pascal and the System that supports it.

II. THE P-MACHINE

II.1 Overview

The P-machine is an idealized machine. The Operating System itself, System programs such as the Filer, and compiled user programs all run on the P-machine. Code for the P-machine is known as P-code, and all codefiles in the System consist of either P-code or native code (that is, code for a particular physical processor).

P-code is designed to be compact, so that programs in P-code are much shorter than equivalent programs in native code. P-code is also designed to be easily generated by a compiler.

Because P-code is compact and simple, relative to native codes, it is fairly easy to implement the P-machine on a variety of actual processors. It is also easier (and cheaper) to maintain a System that runs on one P-machine, rather than a family of Systems, each dedicated to a particular physical processor. This is the essential key to the portability of the p-System.

II.1.1 Interpretive Execution

The "P" in "P-code" and "P-machine" stands for "pseudo." The P-machine may be implemented as a physical processor, or emulated by an interpreter. The Interpreter is a program written in the native code of some particular processor. It is responsible for executing P-code instructions, and controlling machine-dependent I/O.

At runtime, the user's program (or a portion of it) is in main memory. The Interpreter fetches each P-code instruction, in sequence, and performs the appropriate action. The process of bootstrapping involves loading the Interpreter (if necessary) and starting its execution (the next step is to call the Operating System, which runs on the P-machine).

II.1.2 The Stack and the Heap

The System maintains memory-resident data in two dynamic structures called the Stack and the Heap. The Stack is used for static variables, bookkeeping information about procedure and function calls, and evaluation of expressions. The Heap is used for dynamic variables, including the structures that describe a program's environment.

The Stack can be considered part of the P-machine. Most P-code instructions affect the Stack in one way or another.

The Heap is an integral part of the System, but is primarily supported by the Operating System, rather than the P-machine.

Both the Stack and the Heap reside in main memory, and grow toward each other in a (largely) First-In-First-Out manner. Between them is an area of memory that is partly unused, but also contains the Codepool (see below).

The Heap is more fully described in Chapter IV.

II.1.3 Code Segments

In the p-System, program code is stored in one or more segments. A code segment may contain either P-code or native code (or both). Besides the code itself, each code segment contains bookkeeping information for the System's use, and (usually) a pool of constants.

Every "compilation unit" (a separately compiled Pascal PROGRAM or UNIT) results in a "principal segment" of code. In addition, there may be "subsidiary segments," if the program or unit contained SEGMENT routines or EXTERNAL native code routines. Information embedded in the compilation's codefile contains the references among the (possibly) various compilation units that are part of the full program.

When a program is eX(ecuted, the Operating System reads this reference information and resolves the references by finding the location of all compilation units needed by the program (including subsidiary segments and indirect references, such as a UNIT using another UNIT). Tables are built that may be used at runtime to make references (such as procedure calls) from one segment to another.

The segments of a running program compete for space in main memory with each other and with the Stack and the Heap. The principal constraint (as far as code segments are concerned) is that both the calling and called segment must both be present in main memory for an inter-segment call to succeed.

Segments in main memory are all stored contiguously in an area called the Codepool. The Codepool resides between the Stack and the Heap, and may be moved about to create more room.

Code segments are described in this chapter. Codepool handling is described in Chapter IV.

II.1.4 Device I/O

Device I/O and control is accomplished by calls from the language level to routines within the Interpreter. The device I/O routines then call on the routines of the Interpreter's BIOS (for Basic I/O Subsystem), and the BIOS routines control the peripheral hardware directly. I/O environment dependencies are thus isolated in the BIOS, and it is possible to adapt the p-System to a new hardware environment by changing only the BIOS (not the entire Interpreter).

On Adaptable Systems, the BIOS itself has a standard interface to the SBIOS, or Simplified BIOS. The SBIOS is a set of simple I/O routines, and is intended to allow the user to rapidly adapt the System to a new I/O environment.

The BIOS is dealt with in Chapter III, and the SBIOS is fully described in the Installation Guide.

II.2 Program Code

II.2.1 Code Segments

A code segment is a collection of routines, together with descriptive information. The code and information in a segment is contiguous, since the code segment is the "unit of movement" for code; code is loaded into memory a segment at a time.

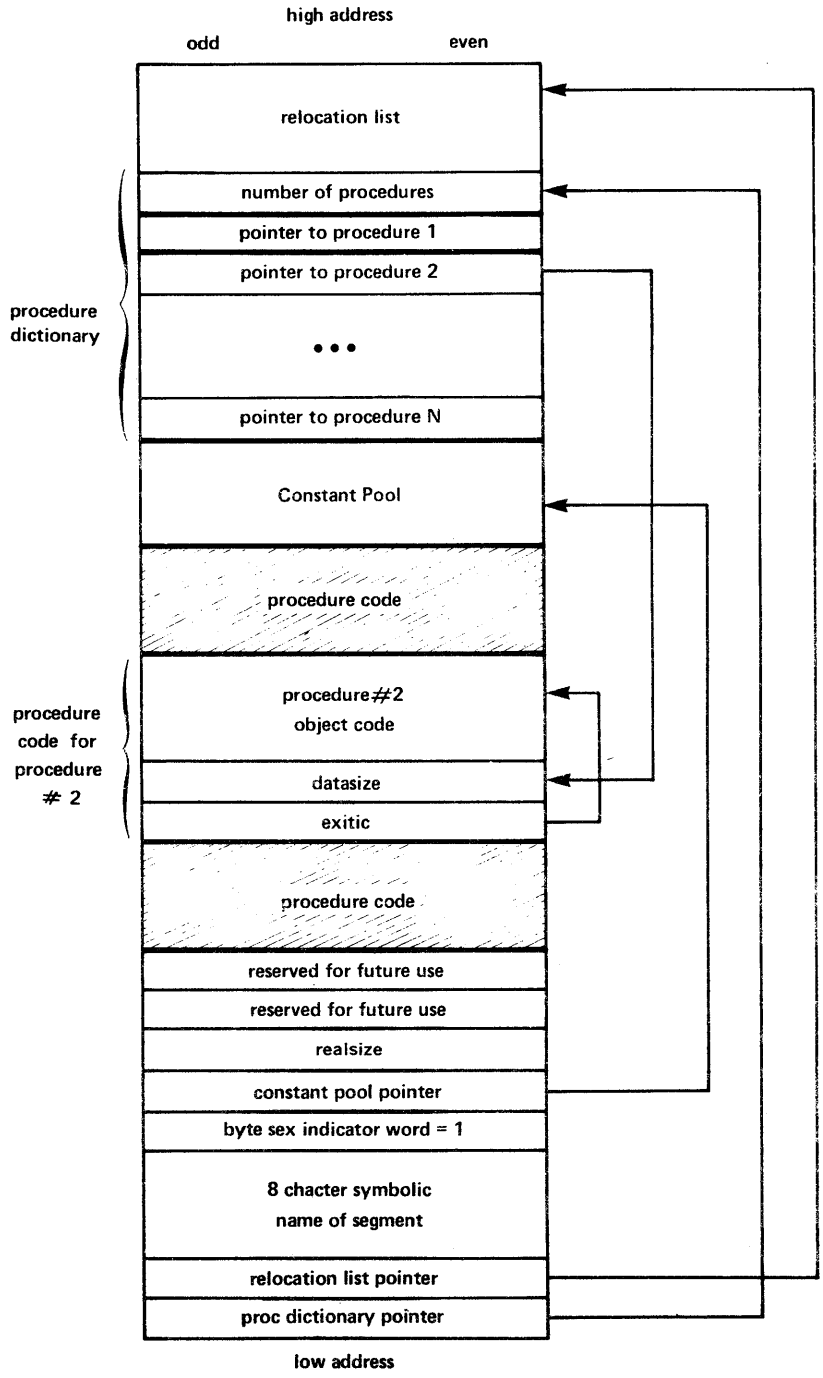
There are up to 255 routines within a segment, numbered 1..255.

At compile time, segments are assigned a name and a number. The name is 8 characters long. It is used by the Operating System to handle inter-segment references at associate time. It is also used when maintaining codefiles with LIBRARY. The number is used to reference the segment at runtime.

The beginning (low address) of a code segment is a record that contains the following information about the segment:

pointer to the routine dictionary
pointer to the relocation list
the 8-character name of the segment (4 words)
byte sex indicator word
pointer to the constant pool
real size word
space reserved for future use (2 words)

Figure 1 illustrates a code segment as it would be loaded into memory. The various substructures of a code segment are described below.



EXECUTABLE CODE SEGMENT FORMAT

FIGURE 1

II.2.1.1 Code Segments and Byte Sex

Code segments are independent of the byte sex of the host processor. A number of System components cooperate to achieve this independence.

There are two groups of word-oriented (byte-sex-dependent) information. The first is superstructure information, such as the routine dictionary. This information is flipped by the Operating System when a segment is loaded. The second is embedded information, such as, for example, constants (accessed by LDC) or XJP tables. This sort of information is flipped by the Interpreter.

The Compiler produces code segments that contain word information in the natural order of the machine on which the Compiler was run. Immediately following the segment's 8-character name is a flag that always contains the constant 1, in the byte sex of the original machine; if read in the opposite byte sex, it appears to be a 256.

When a segment is loaded by the Operating System, and its byte sex flag indicates that the sex of the segment is opposite that of the running machine, routine dictionaries are byte-swapped. Embedded information is then flipped by the Interpreter.

The net result is that segments of either sex can run on any machine.

II.2.1.2 Routine Dictionaries

The first word in a code segment points to word 0 of the segment's routine dictionary (also called the "procedure dictionary"). The routine dictionary is a list of pointers to the code for each routine in the segment. Each routine dictionary pointer is a seg-relative word pointer.

Routines within a segment are numbered 1..255. A routine's number is an index into the routine dictionary: the n'th word in the dictionary contains a pointer to the code for routine n.

The first word (word 0) of the dictionary contains the number of routines in the segment.

In the case of EXTERNAL and FORWARD routines, the source code may contain a routine's declaration but not its code. The corresponding routine dictionary entry is zero (at least, before linking).

11.2.1.3 Routine Code

The code of a routine consists of two words: DATASIZE and EXITIC, followed by the executable object code. The object code may be entirely P-code, entirely native code, or a mixture of the two.

DATASIZE is the number of words of local data space that must be allocated when the procedure is called. DATASIZE does not include parameters: the routine's parameters are assumed to already be on the Stack. The first executable instruction starts at the byte or word immediately following the DATASIZE word. If the first executable instruction is native code, DATASIZE is one's-complemented.

If this first instruction is a P-code instruction, then EXITIC is a seg-relative byte pointer to the code that must be executed when the procedure is exited. If this first instruction is a native code instruction, then EXITIC is undefined at runtime.

If the code of the routine contains both P-code and native code, it is still the first instruction of the routine that determines these conditions.

II.2.1.4 The Constant Pool

In Version IV.0, multi-word constants are stored together in a single constant pool for the entire segment. The constant pool begins immediately after the last body of procedure code in the segment.

The location of the constant pool is contained in the constant pool pointer, a seg-relative word pointer that immediately follows the byte sex indicator word at the beginning of the segment. It points to the low address of the constant pool. If the constant pool pointer is equal to zero, the segment does not contain a constant pool.

Constants are referenced by word offsets relative to the beginning (low address) of the constant pool.

The constant pool is divided into two subpools: the real pool and the main pool.

The first word of the constant pool points to the beginning of the real pool. This is a word pointer relative to the start of the constant pool; if there are no real constants in the code segment, this word must be 0. The first word of the real pool contains the number of real constants in the real pool.

Figure 2 illustrates a constant pool with an embedded real subpool.

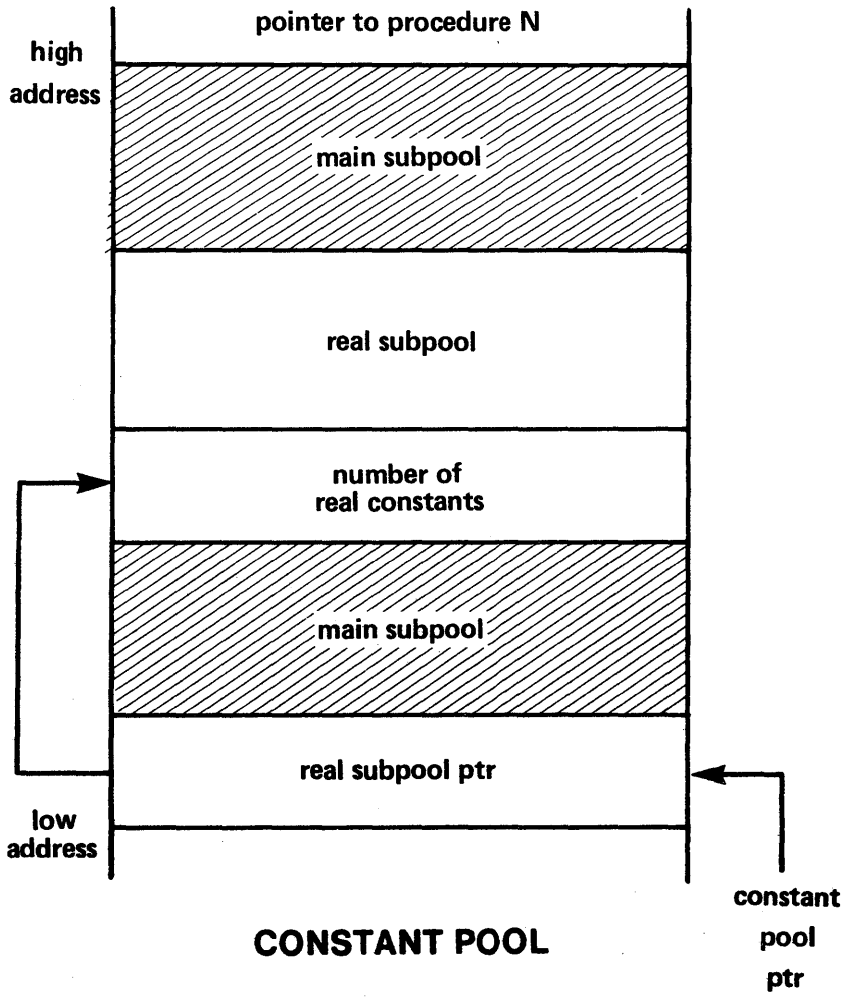

pointer to procedure N
high
address
main subpool
real subpool
number of
real constants
main subpool
real subpool ptr
low
address
constant
pool
ptr
CONSTANT POOL

FIGURE 2

Real constants are generated for either 32- or 64-bit floating point BCD (Binary Coded Decimal) data formats: real values (and operations upon them) can be transported across all processors with the same-sized representation of floating point numbers, but cannot be transported to machines with floating point formats of a different size.

Only one size is likely to be available for a particular processor, since real constant handling is done by machine-dependent software (i.e., within the Interpreter). Within a single program, all compilation units must share the same size for real constants and variables.

The Pascal Compiler is configured (when it is compiled) to default either to 32-bit or 64-bit reals. A directive is available to override the default:

{$R2} - sets realsize to 2 words (32 bits)
{$R4} - sets realsize to 4 words (64 bits)

This directive must occur before the first symbol in a compilation that is not a comment. The active realsize for a particular compilation is displayed after the Compiler's version number at the beginning of the console output during a compilation (and in a compiled listing).

The realsize at compilation time is also embedded in every code segment (even though it may not reference any reals). The word REALSIZE at the base of the segment contains this value.

A 32-bit real constant is represented by a three-word record. The first word contains a signed integer representing the exponent value. The following two words contain the mantissa digits. A mantissa word representing significant mantissa digits contains an integer whose absolute value is between 0 and 9999; its value corresponds to four mantissa digits. The first mantissa word is signed, and thus contains the mantissa sign. The second mantissa word may contain a negative value; in this case, it does not contain any significant digits and is disregarded when constructing the internal representation of the real constant. It serves as a terminator word for the constant conversion routines. The decimal point is defined to lie to the right of the four digits in the last valid (used) mantissa word. The digits in the last mantissa word are left-justified.

For example, if the real value is 1.1, the first mantissa word contains 1100 (BCD).

Example:

1 .. 4 significant mantissa digits:
The first mantissa word contains a signed value between 0

and 9999. The second word contains a negative value. The implied decimal point position is at the end of the first word.

5 .. 8 significant mantissa digits:
The second mantissa word contains a positive value between 1 and 9999, and represents up to 4 low-order digits. The first word contains a signed value between 1 and 9999; it represents the 4 high-order digits. The implied decimal point position is at the end of the second word.

A 64-bit real constant is represented by a record whose length may vary between 4 and 6 words, depending upon the number of significant digits in the constant. The first 2 words of a 64-bit constant are identical in format to those of a 32-bit real constant; thus, the format always contains an exponent word and a first mantissa word. An enumeration of the remaining words for all cases follows:

1 .. 4 significant mantissa digits:
Mantissa word 2 contains a negative terminator.
Mantissa word 3 is zeroed and is present solely to provide sufficient space for the native format.

5 .. 8 significant mantissa digits:
Mantissa word 2 contains 1 to 4 digits (left-justified).
Mantissa word 3 contains a negative terminator.

9 .. 12 significant mantissa digits:
Mantissa word 2 contain 4 digits.
Mantissa word 3 contains 1 to 4 digits (left-justified).
Mantissa word 4 contains a negative terminator.

13 .. 16 significant mantissa digits:
Mantissa words 2 - 3 contain 4 digits.
Mantissa word 4 contains 1 to 4 digits.
Mantissa word 5 contains a negative terminator.

17 .. 20 significant mantissa digits:
Mantissa words 2 - 4 contain 4 digits.
Mantissa word 5 contains 1 to 4 digits.

Real constants are converted to native machine format when a code segment is loaded into memory; this may result in a significant runtime overhead for programs that are memory-bound. Time-critical programs of this nature may sacrifice portability for execution speed by using a native constant generator utility program

(not yet available) to convert their real subpools into native machine format. This is done by replacing the canonical form of each real constant in the codefile with a native real constant. The modified subpool is merged with the main pool by setting the real pool pointer to zero, thus eliminating the usual conversion process during a segment load. Because the constant pool is transformed in place, constant offsets embedded in the codefile do not require updating.

II.2.1.5 The Relocation List

The last (high address) body of information in a (memory-resident) code segment is the relocation list. The second pointer at the beginning of the code segment points to the last (highest address) word in the relocation list. This pointer is a seg-relative word pointer; if there is no relocation list, it is equal to zero.

The relocation list contains all the information necessary to fix any absolute addresses used by code within the segment, whenever the segment is loaded or moved in memory. Such absolute addresses are only needed by native code: Segments containing exclusively P-code are completely position-independent; no relocation list is needed.

A relocation list consists of zero or more relocation sublists. Each sublist contains code offsets for objects that must be relocated, and specifies the type of relocation that must be done. Sublists can occur in any order, and more than one sublist can have the same type of relocation.

The following code fragment shows the format of the heading of a sublist:

```
LocTypes=(RelocEnd, {signals end of entire relocation list}
          SegRel,   {relative to address of base of this segment}
          BaseRel,  {relative to data segment given in DATASEGNUM}
          InterpRel,{relative to Interpreter's interp-relative table}
          ProcRel); {relative to address of 1st instruction in proc}

ListHeader=PACKED RECORD
             ListSize: integer;    {number of pointers in sublist}
             DataSegNum: 0..255;   {local segment number for BaseRel}
             RelocType: LocTypes; {relocation type of sublist entries}
           END;
```

Each sublist contains a ListHeader and zero or more seg-relative byte pointers to the objects which must be relocated. The RelocType field in the ListHeader defines what kind of relocation will be applied to all objects designated by the sublist.

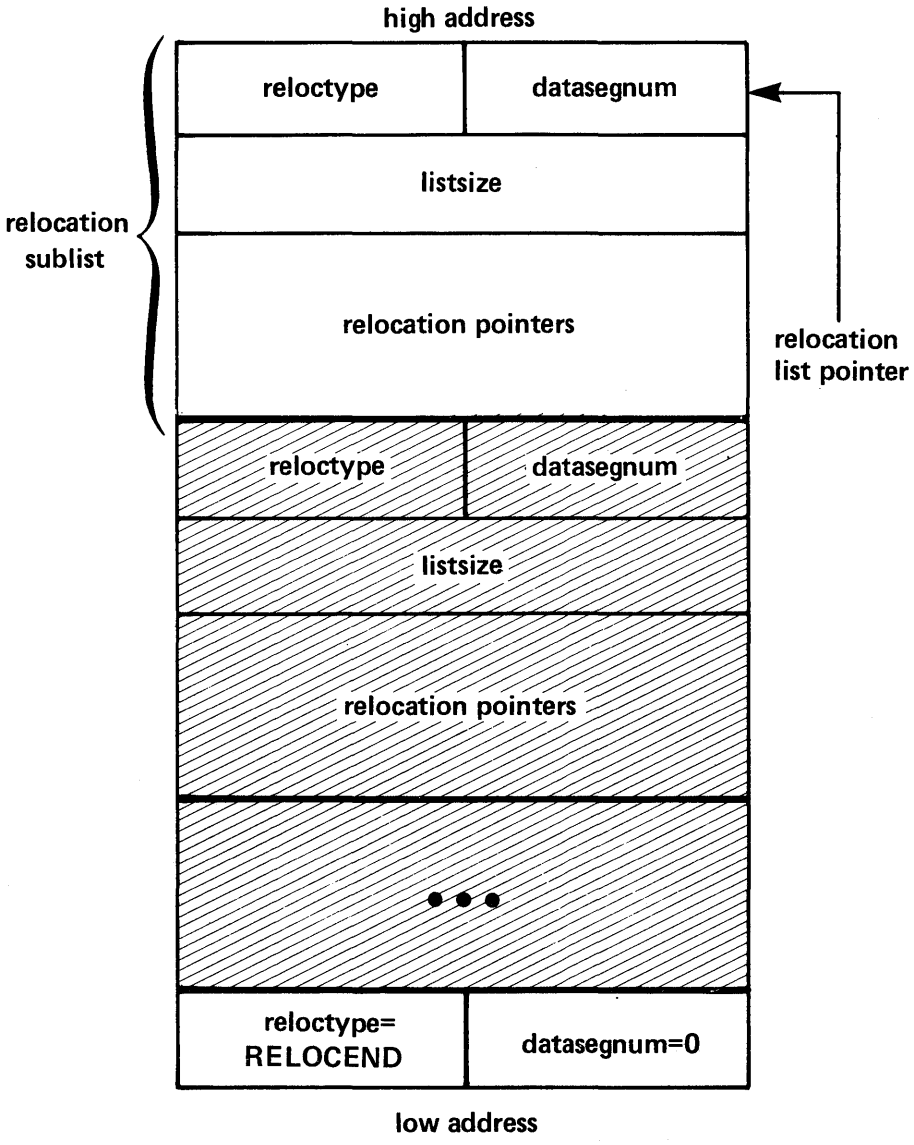
The relocation type ProcRel is generated by the Assembler, but changed by the Linker into SegRel. ProcRel sublists should never be encountered when loading and relocating assembly code.

The DataSegNum field in the ListHeader is only used in sublists with a RelocType of BaseRel, and in all other cases should be zeroed. It specifies the local segment number of the data segment that all of the sublist's pointers are relative to.

Since the Assembler cannot know this segment number in advance, it should zero-fill the field and leave the responsibility for correctly setting this field to the Linker.

The ListSize field in the ListHeader contains the number of pointers in the sublist.

Figure 3 illustrates a relocation list with multiple sublists:



RELOCATION LIST

FIGURE 3

The relocation list is intended to be used from high address down to low address. Each sublist in turn from high to low is processed until a sublist with a relocation type of RelocEnd is encountered. The DataSegNum and ListSize should be 0 for this terminating entry.

The relocation list is located at the end of the code segment, since it is sometimes possible to discard the relocation information after the segment has been loaded into memory.

II.2.1.6 Segment Reference List

In the P-machine, Version IV.0, each code segment is associated at runtime with an "environment vector" that defines the mapping of each segment number to the segment or unit that it designates. Each compilation unit has its own independent (i.e., local) series of segment numbers, and its own environment vector. In this way, a particular unit may be referenced by more than one unit, and each unit that references it may use a different segment number. (More about environment vectors appears in Section II.2.3.)

When a compilation unit references one or more other compilation units, the principal segment of the compilation contains a segment reference list. This list defines the connection between the segment numbers that appear in the object code (they are created by the Compiler), and the names of the units to which they refer. Only principal segments contain segment reference lists.

The segment reference list, when present, is located above the relocation list (it grows toward higher memory addresses). The list is used by the Operating System at associate time. It does not occupy any space in memory during the program's execution.

The segment reference list associates the name of each compilation unit (which does not change) with the number by which that that compilation unit is referenced.

The following fragment of Pascal code describes a record in the segment reference list:

```
SegRec=PACKED RECORD
          SegName: PACKED ARRAY [0..7] OF CHAR; {referenced segment name}
          SegNum: 0..255; {associated segment number}
          Filler: 0..255; {reserved for future use}
        END;
```

The Seg_Refs entry in the segment dictionary (described below) contains the number of words in the segment reference list. The Code_Leng field in the segment dictionary can be used as a seg-relative word pointer to the start of the segment reference list. The segment reference list consists of one or more SegRec's, starting directly above the relocation lists and continuing towards higher memory addresses. A SegRec consists of SegName, which contains the name of the segment, SegNum, which contains the number by which the segment is reference within this current code segment, and some Filler.

The segment reference list is terminated by a SegRec with a blank-filled SegName and SegNum of zero.

SegRec's with a SegName of '*** are generated so the Operating System can execute the initialization and termination code sections of a unit: before executing a host program, the Operating System constructs a list of all used units that contain a reference to '***', and uses this list to execute the initialization/termination sections of all used units before/after the invocation of the host program.

When the initialization/termination section of a unit (which is procedure 1) is compiled, a <CXG <***'s seg num>, 1> instruction is emitted between the initialization and termination parts. A local segment number is reserved for the '***' segment reference, and the Operating System creates a linear list that links together the units of a program that require initialization. At the end of this list is the outer body of the main program. The Operating System invokes the program by calling the first initialization code on this list, which calls the next, and so forth up to the body of the main program itself. When the main program terminates, the calling chain is "popped", and termination sections are executed in the reverse order.

II.2.1.7 Linker Information

Linker information (Linker info) is a portion of a code segment that allows the Linker to resolve references between P-code and native code. Segments output by an assembler always have Linker info. Segments output by a compiler have Linker info only if they contain an EXTERNAL routine. Only principal segments may contain EXTERNAL routines.

Linker info is a sequence of 8-word records, starting on the block boundary following the end (high address) of the segment reference list. The end of the sequence contains the value EOFMark. Linker info records are always 8 words long: unused records and unused fields are zero-filled.

If a code segment has Linker info, the HasLinkerInfo Boolean in Seg_Misc in the segment dictionary is TRUE. The starting block of Linker info, relative to the start of the codefile, can be calculated from the formula:

Code_Addr + ((Code_Leng + Seg_Refs + 255) DIV 256)

... where Code_Addr, Code_Leng, and Seg_Refs are all values in the segment dictionary (see below).

Two fields are common to all Linker info records. The Name field contains an 8-character segment name. The LIType field determines the nature of the Linker information in the remainder of the record.

The following fragment of psuedo-Pascal code describes a Linker info record:

```
PtrRecNum = {an integral number of 8-word pointer records}
            {this is variable from record to record};

LITypes = (EOFMark, GlobRef, PublRef, PrivRef, ConstRef, GlobDef, PublDef,
           ConstDef, ExtProc, ExtFunc, SepProc, SepFunc);

LIEntry = RECORD
            Name: PACKED ARRAY [0..7] OF CHAR;
            CASE LIType: LITypes OF

              GlobRef, PublRef, ConstRef
                       : (Format: (Word, Byte, Big);
                          NRefs:  integer);

              PrivRef: (Format: (Word, Byte, Big);
                          NRefs:  integer;
                          NWords: integer);
```

```
        ExtProc, ExtFunc
                : (SrcProc: integer;
                   NParams: integer);

        SepProc, SepFunc
                : (SrcProc: integer;
                   NParams: integer;
                   KoolBit: Boolean);

        GlobDef:  (HomeProc: integer;
                   ICOffSet: integer);

        PublDef:  (BaseOffset: integer;
                   PubDataSeg: integer);

        ConstDef: (ConstVal: integer);

        EOFMark:
    END {CASE};

    PtrList: ARRAY [0..PtrRecNum] OF
                ARRAY [0..7] OF integer

END {LIEntry};
```

GlobRef, PublRef, ConstRef, and PrivRef are all Linker info types generated by an assembler. They all consist of two fields that precede a list (PtrList) of seg-relative byte pointers into the associated segment. Format contains the size of the fields pointed to by the accompanying list. NRefs contains the number of pointers in the list. PtrList contains multiples of 8 words; all unused words should be zero.

For these types of Linker info records, PtrRecNum = ceiling(NRefs/8), where ceiling(n) is the smallest integer >= n.

GlobRef is used to link identifiers in two or more assembled routines. Name is an identifier that is referenced within the segment, and defined in some other assembled routine. Format should always be Word. The Linker must add the final segment offset of the referenced object to all words pointed at by PtrList. This offset must be in the correct addressing mode: i.e., bytes or words, depending on the processor being used.

PublRef is used to link an identifier in an assembled routine to a global variable in a compilation unit. Name is an identifier that is referenced in the segment, and

defined as a global variable in some other compilation unit. Format should always be Word. The Linker must add the offset of the referenced object to all words pointed at by PtrList.

ConstRef is used to link an identifier in an assembled routine to a global constant in a compilation unit. Name is an identifier that is referenced in the segment, and defined as a global constant in some compilation unit. Format may be either Byte or Word. The Linker must place the constant value into all locations pointed at by PtrList.

PrivRef is used to allocate space in the global data segment. Format should always be Word. NWords specifies the number of words to allocate. The Linker must add the offset of the start of the allocated area within the global data segment to all words pointed at by PtrList.

ExtProc and ExtFunc are generated by a compiler to reference EXTERNAL routines. There is no PtrList. SrcProc is the number assigned to the routine. NParams is the number of words allocated for parameter passing.

SepProc and SepFunc are generated by an assembler for routine declarations. There is no PtrList. SrcProc is the number assigned to the routine. NParams is the number of words allocated for parameter passing. KoolBit is TRUE if the routine is relocatable, FALSE otherwise. Thus, .PROC and .FUNC generate SepProc or SepFunc records with KoolBit = FALSE, and .RELPROC and .RELFUNC generate SepProc or SepFunc records with KoolBit = TRUE.

GlobDef declares a global identifier in an assembled routine. A GlobDef record is generated for each label defined by a .DEF, .PROC, .FUNC, .RELPROC, or .RELFUNC directive. There is no PtrList. Name is an identifier defined within the segment, and may be referenced by any other assembled routines within the same segment. HomeProc contains the number of the routine in which Name is defined. ICOffset is a byte offset to Name, relative to the start of the routine in which Name is defined.

PublDef declares a global variable in a compilation unit. A PublDef record is generated for each global variable in a compilation unit that is visible to any EXTERNAL routines. There is no PtrList. BaseOffset is the word offset of the variable, relative to the start of the data segment that contains it. PubDataSeg is the local number of the data segment that contains the variable.

ConstDef declares a global constant in a compilation unit. A ConstDef record is generated for each global constant in a compilation unit that is visible to any EXTERNAL routines. There is no PtrList. ConstVal contains the value of the constant.

EOFMark indicates the end of used Linker info records. Name should be blank-filled.

The following table shows the types of segments (as defined in the segment dictionary), and the types of segment reference records that can be contained in the associated Linker info. Note that Proc_Seg's cannot have Linker info at all:

	Prog_Seg	Unit_Seg	Seprt_Seg
GlobRef			yes
PublRef			yes
PrivRef			yes
ConstRef			yes
ExtProc	yes	yes	
ExtFunc	yes	yes	
SepProc			yes
SepFunc			yes
GlobDef			yes
PublDef	yes	yes	
ConstDef	yes	yes	
EOFMark	yes	yes	yes

II.2.2 Codefile Organization

II.2.2.1 The Segment Dictionary

The first block of a codefile contains the first record of that file's segment dictionary. In Version IV.0, a segment dictionary consists of a linked list of dictionary records; if the dictionary is longer than one record, subsequent records are embedded in the codefile. These are each one block long, and are located between code segments.

A single dictionary record can describe up to 16 distinct segments. The information describing a segment is contained in 6 different arrays: the information describing a segment is found by using a single index value to select a component from each of these arrays. Entries in the segment dictionary describe only segments whose code bodies are included in the codefile.

The following fragment of Pascal code describes a segment dictionary record:

```
CONST Max_Dic_Seg = 15; {maximum segment dictionary record entry}

TYPE  Seg_Dic_Range = 0..Max_Dic_Seg; {range for segment dictionary entries}

      Segment_Name = PACKED ARRAY [0..7] OF CHAR; {segment name}

      {segment types}
           Seg_Types = (No_Seg,     {empty dictionary entry}
                        Prog_Seg,   {program outer segment}
                        Unit_Seg,   {unit outer segment}
                        Proc_Seg,   {segment procedure inside program or unit}
                        Seprt_Seg); {native code segment}

      {machine types}
           M_Types = (M_Psuedo, M_6809, M_PDP_11, M_8080, M_Z_80,
                                M_GA_440, M_6502, M_6800, M_9900,
                                M_8086, M_Z8000, M_68000);

      {p-machine versions}
           Versions = (Unknown, II, II_1, III, IV, V, VI, VII);

      {segment dictionary record}
           Seg_Dict = RECORD
             Disk_Info:
               ARRAY [Seg_Dic_Range] OF {disk info entries}
                 RECORD
                   Code_Addr: integer;  {segment starting block}
                   Code_Leng: integer;  {number of words in segment}
                 END {of RECORD};
             Seg_Name:
               ARRAY [Seg_Dic_Range] OF Segment_Name; {segment name entries}
             Seg_Misc:
               ARRAY [Seg_Dic_Range] OF {misc entries}
                 PACKED RECORD
                   Seg_Type: Seg_Types;      {segment type}
                   Filler: 0..31;            {reserved for future use}
                   Has_Link_Info: Boolean; {need to be linked?}
                   Relocatable: Boolean;     {segment relocatable?}
                 END {of PACKED RECORD};
             Seg_Text:
               ARRAY [Seg_Dic_Range] OF integer; {start blk of interface text
             Seg_Info:
               ARRAY [Seg_Dic_Range] OF {segment information entries}
                 PACKED RECORD
                   Seg_Num: 0..255;          {local segment number}
```

```
            M_Type: M_Types;              {machine type}
            Filler: 0..1;                 {reserved for future use}
            Major_Version: Versions; {P-machine version}
         END {of PACKED RECORD};
   Seg_Famly:
      ARRAY [Seg_Dic_Range] OF {segment family entries}
         RECORD
            CASE Seg_Types OF
               Unit_Seg, Prog_Seg:
                  (Data_Size: integer;    {data size}
                   Seg_Refs: integer;     {segments in compilation unit}
                   Max_Seg_Num: integer; {number of segments in file}
                   Text_Size: integer);  {# of blks interface text}
               Seprt_Seg, Proc_Seg:
                  (Prog_Name: Segment_Name); {outer program/unit name}
         END {of Seg_Famly};
   Next_Dict: integer; {block number of next dictionary record}
   Filler: ARRAY [0..6] OF integer; {reserved for future use}
   Copy_Note: string[77]; {copyright notice}
   Sex: integer; {machine sex (Sex = 1)}
END {of SEG_DICT};
```

Disk_Info contains information about the segment's location within the file. Segment code always starts on a block boundary. Code_Addr is the number of the block where the segment code starts (relative to the start of the codefile). Code_Leng is the number of 16-bit words in the segment. This size includes the relocation list but does not include the segment reference list. All unused entries in this array should be zeroed.

Seg_Name contains the first 8 characters of the program, unit, segment, or assembly procedure name. Unused entries should be blank-filled.

Seg_Misc contains miscellaneous information about the segment. Seg_Type indicates the type of segment: Prog_Seg and Unit_Seg are outer segments of programs and units respectively; Proc_Seg is a segment routine within either a unit or a program outer segment; Seprt_Seg is an unlinked native code segment. Has_Link_Info indicates whether Linker information has been generated for this segment. Linker info resides in the blocks that directly follow the segment reference list. Linker info starts on a block boundary. The Boolean Relocatable specifies whether a code segment is statically or dynamically relocatable.

Dynamically relocatable code segments reside in the code pool; their position in memory may change many times during execution. Statically relocatable code segments are loaded only once, in a fixed position on the system heap: they remain position-locked and memory-locked throughout their lifetime.

All segments that contain only P-code are position-independent and thus dynamically relocatable. Segments that contain native code may be dynamically relocatable provided they make no assumptions about either the lifetime of any modifications made to the segment body itself, or the exact location of the segment body in memory across the execution of a single P-code.

Dynamically relocatable native code is generated by assembling routines using the RELPROC or RELFUNC assembler directives; a linked code segment containing assembly routines is dynamically relocatable only if all of its assembly routines were originally specified as dynamically relocatable. Note that the use of these assembler directives is an assertion by the programmer that the routines they declare behave properly; the System does not enforce this, so caution must be used. If a routine is to be dynamically relocatable, it cannot store information into the segment body, be self-modifying, or store any pointers to the code segment in data variables, and then assume that things will behave correctly the next time it is called.

The Boolean Relocatable is unaffected by the presence or absence of relocation lists, and is not relevant to concurrency considerations.

Seg_Text contains the starting block of the segment's INTERFACE text section, relative to the start of the codefile. The INTERFACE text section can appear

anywhere within the codefile that contains the code segment it describes. The Seg_Text array entry, in conjunction with the Text_Size field in the Seg_Family record, indicates the address and length of the INTERFACE section in blocks. The INTERFACE text section always starts on a block boundary and follows all of the conventions of a textfile, with the exception that the last page of the section may be either 1 or 2 blocks long. Only segments with a Seg_Type of Unit_Seg have INTERFACE sections. All other segments and unused entries should be zero-filled.

Seg_Info contains further information about the segment. Seg_Num is the segment number. M_Type tells what kind of object code is in the segment. If there is any native code in the segment, then M_Type will have one of the processor-specific M_Type's. If the segment consists exclusively of P-code, then its M_Type is M_Psuedo. Major_Version gives the version of the P-machine on which the codefile is intended to run.

Seg_Famly contains information about the code segment's compilation unit. The information contained in this array depends on whether Seg_Type indicates a principal or a subsidiary segment.

If the segment is a subsidiary segment, then Seg_Famly contains the first 8 characters of the parent compilation unit's name, stored in Prog_Name. If this name is not known at codefile generation time (as is the case with Seprt_Seg's), the field should be blank-filled.

If segment is a principal segment, then the information in Seg_Famly consists of four fields:

Data_Size is the number of words in this segment's base data segment. The variables of principal segments are referenced from any location, including their own outer routine bodies, via global loads and stores (rather than local operations). Therefore, the Data_Size field associated with the body of an outer routine in a code segment should be zero, so that no superfluous memory will be allocated in an unused local data area.

Seg_Refs is the size in words of the segment reference list for this segment.

Max_Seg_Num is the total number of segment numbers assigned to this compilation unit. Max_Seg_Num includes all segments with assigned numbers, regardless of whether the segment body is contained in this file or not.

Text_Size is the number of blocks of INTERFACE text within the compilation unit. Text_Size is used in conjunction with the Seg_Text array to specify the INTERFACE text for a compilation unit of type Unit_Seg; it is zero-filled for all other compilation unit types.

If the segment is unused (Seg_Type = No_Seg), then Seg_Famly should be zero-filled.

Next_Dict contains the block number of the next segment dictionary record, relative to the start of the codefile. In the last record of the segment dictionary, Next_Dict should be zero.

Filler is reserved for future use and should always be zero-filled.

Copy_Note is reserved for a copyright message, which can be created with either the LIBRARY utility or a Compiler directive.

Sex corresponds to the byte sex of the codefile. It is a full word that contains the value 1, with the same byte sex as the rest of the dictionary record. Thus, when this word is examined by a program running on a machine with the same byte sex as the codefile, it will appear as a 1; on a machine of opposite sex, it will appear as a 256. System programs use this word to detect the sex of the codefile, and if necessary, byte-swap the word-oriented fields of the dictionary.

II.2.2.2 Assembler-Generated Codefiles

Codefiles generated by an assembler have a slightly different structure from those generated by a compiler. A relocation list is generated for each procedure in an assembler-generated segment (instead of one relocation list for the whole segment). These are the only sort of lists that may contain ProcRel relocation. These lists are placed immediately after the body of the procedure they describe. The start or high end address of each list is pointed at by the seg-relative word pointer contained in the ExitIC field of each assembler-generated procedure.

An assembler-generated segment is also unique in that during the linking process, the code bodies of all its procedures and functions may be copied into one of the segments of the compilation unit it is being bound to. Further, the name of the segment or segments that the assembly code may be linked to is never known at assembly time. It is, however, always assumed that any number of assembly procedures or functions that communicate via REFs and DEFs are always bound into the same segment, regardless of whether they were assembled together.

The DataSize word generated by the assembler for each routine should have a value of -1 (0FFFF HEX): this indicates a data size of zero that is one's complemented, to signal that the first instruction of the code body is native code.

Finally, since the assembler-generated code segments cannot know what program or unit they are to be linked to, the Prog_Name entry in the Seg_Famly array of the segment dictionary should be blank-filled, and the DataSegNum field in the ListHeader record of all BaseRel relocation sublists should be zero-filled.

It is the Linker's responsibility, when linking assembler-generated segments, to convert all ProcRel relocation sublists into SegRel relocation lists, to correctly set the DataSegNum field in the ListHeader of all BaseRel relocation sublists, and to collect all relocation sublists and place them after the procedure dictionary of the code segment. The Linker should also update the Relocatable bit in the Seg_Misc array, depending on the information supplied in Linker info.

II.2.3 Code Segment Environments

II.2.3.1 Segment Information Blocks (SIBs)

A Segment Information Block (SIB) is a record that contains information about an "active" code segment. A code segment is active if it may be used by a program that is running. A SIB is allocated on the Heap, and remains there as long as the segment is active. There is only one SIB for each code segment, no matter how many other segments may be using it.

Note that a code segment need not be in memory to be active: an active code segment may be on disk or in the Codepool, but its SIB will always be on the Heap.

The following fragment of Pascal code describes a SIB:

```
SIB = RECORD
        Seg_Base: Mem_Ptr;      {segment's memory location}
        Ref_Count: integer;     {# of active calls to the seg}
        Activity: integer;      {memory swap activity}
        Link_Count: integer;    {number of links to the SIB}
        Residency: -1..maxint; {-1 = pos lock, 0 = swap, n = mem lock}
        Seg_Name: PACKED ARRAY [0..7] OF CHAR;
        Seg_Leng: integer;      {# of words in segment}
        Seg_Addr: integer;      {disk address of segment}
        Vol_Info: Vl_Ptr;       {pointer to disk drive info}
        Data_Size: integer;     {number of words in data segment}
        Res_SIBs: RECORD        {code pool management record}
                    Next_SIB: SIB_P;     {next SIB in list}
                    Prev_SIB: SIB_P;     {previous SIB in list}
                    CASE Boolean OF      {scratch area}
                      TRUE:  (Sort_SIB: SIB_P);     {next SIB in sort list}
                      FALSE: (New_Loc:  Mem_Ptr);       {temporary address}
                  END {of Res_SIBs};
      END {of SIB};
```

Seg_Base contains the current memory address of the code segment. If the code segment is not in memory, Seg_Base contains NIL.

Ref_Count contains the number of outstanding calls to the segment. It is incremented whenever a routine outside the segment executes a CXP to a routine within the segment. It is decremented whenever a RET from a routine within the segment returns to a routine outside the segment.

Activity contains a value based on the number of times a segment is used; it increases over time. It is incremented by 6 whenever a call is made to a routine outside the segment. It is also incremented by 6 whenever a routine within the segment returns to a routine outside the segment. Finally, it is incremented by 6 whenever a task switch suspends the segment that is currently executing.

Link_Count contains the number of links to the SIB from other Operating System data structures. When Link_Count becomes zero, the SIB is removed from the Heap (the space it occupied is available again).

Residency contains a value between -1 and maxint. A -1 indicates that the segment is Position_Locked (this occurs when the Boolean Relocatable in the segment dictionary is TRUE). A zero indicates that the segment is Swappable (that is, it can be removed from memory if necessary). A value greater than zero indicates that the segment is Memory_Locked. In this case, the value is a count of the number of memory lock operations that have been applied to that segment. Residency is incremented when a program declares the segment to be Memory_Locked, and decremented when a program declares it to be Swappable. It becomes actually Swappable when Residency is equal to zero (i.e., when no outstanding Mem_Lock operations remain). Programs can control the residency of segments by using the intrinsics MEMLOCK and MEMSWAP.

Seg_Name contains the first 8 characters of the segment's name.

Seg_Leng contains the number of words that the code segment occupies (including any relocation lists, but excluding segment reference lists).

Seg_Addr contains the segment's first block number on disk.

Vol_Info contains a pointer (Vl_Ptr) to a volume information record that contains the drive number and volume name of the disk on which the segment is resident.

Data_Size contains the number of words in the code segment's data segment. This only applies to principal segments: otherwise, Data_Size should be zero.

Res_SIBs is used to maintain the Code Pool. All SIBs of segments in the Code Pool are on a doubly-linked list formed by the Prev_SIB and Next_SIB pointers. The Sort_SIB and New_Loc fields are used for temporary values while managing the Code Pool.

The Operating System uses several data structures to manage code segments by maintaining active SIBs and managing the Code Pool. All of these data structures refer to SIBs through pointers.

When a program being prepared for execution requires a code segment that is not yet active, the appropriate SIB is allocated on the Heap and initialized. The

Operating System creates a pointer to the SIB, and the SIB's Link_Count is incremented. When the segment is no longer needed, the pointer is removed, and the Link_Count is decremented. When Link_Count becomes zero, the SIB is removed from the Heap.

II.2.3.2 Environment Records (E_RECs)

A code segment's "environment" is the mapping of segments it may access into local segment numbers. Segment numbers only have local meaning; a segment may only refer to segments that have been assigned local segment numbers. It may not refer to segments outside of this scope.

For each segment, there is an Environment Record (E_Rec). This record designates an Environment Vector (E_Vec) that describes the mapping of local segment numbers to actual code segments.

The following fragment of pseudo-Pascal describes environment records and vectors:

```
E_Vect_P = ^E_Vect;
E_Rec_P  = ^E_Rec;

E_Vect   = RECORD
             Vec_Length: integer;    {number of local segments}
             Map: ARRAY [1..Vec_Length] OF E_Rec_P;
                                     {local environment mapping}
           END {of E_Vect};

E_Rec    = RECORD
             Env_Data: Mem_Ptr;        {pointer to global data}
             Env_SIB:  SIB_P;          {pointer to SIB for seg number}
             Env_Vect: E_Vect_P;       {pointer to environment}
             CASE Boolean OF
               TRUE : (Link_Count: integer;  {number of links to E_Rec}
                       Next_Rec:   E_Rec_P); {next environment record}
           END {of E_Rec};
```

Env_Data points to the segment's global data. (The data segment is allocated on the Heap when the program is invoked.)

Env_SIB points to the segment's SIB. (Also placed on the Heap when the program is invoked.)

Env_Vect is an array of pointers to E_Rec's. It is indexed by a segment number: the pointer indicates an E_Rec that describes a code segment. In this way, a mapping from local segment numbers to actual segments is accomplished.

Link_Count indicates the number of active compilation units that are currently USE'ing the segment. This only applies to the principal E_Rec of a compilation unit. Link_Count is maintained in the same way a SIB's Link_Count is

maintained.

Next_Rec is a pointer on a chain of all active compilation units. This chain is called Unit_List. This field also applies only to the principal E_Rec's of a compilation unit.

In order to minimize index manipulations, the Map array in an E_Vect record starts at 1. Thus it may be indexed by local segment numbers (these must be 1 or greater). The Vec_Length field of the record may be considered to occupy the zero'th position of the map.

The Operating System uses a recursive routine to construct the environments of a program's USEd units, and then its subsidiary segments and principal segment (its "native segments"). The algorithm is roughly:

```
FUNCTION Build_Env (Seg_Dict): E_Rec_P;
  BEGIN
  IF outer block segment E_Rec exists in Unit_List THEN  BEGIN
     increment Link_Count;
     return existing E_Rec_P
  END  ELSE  BEGIN
     create E_Vect;
     create Env_Data for outer block data space;
     IF there are USEd units indicated in Seg_Dict THEN
        FOR all USEd units DO
           install Build_Env (New_Seg_Dict) into current E_Vect;
     FOR all native segments DO
     BEGIN
        create E_Rec and SIB for native segment;
        install E_Vect, SIB, and Env_Data in E_Rec;
        install E_Rec for native segments in E_Vect
     END;
     install E_Rec for outer block segment on Unit_List;
     return E_Rec_P for outer block segment
  END
  END
```

The Build_Env function returns a pointer to the E_Rec for the outer block of the program being executed. This pointer is installed into the Operating System's User_Program E_Vect entry.

After a program's execution, a recursive routine is used to de-link the environment for the program's outer block and all subsidiary units and segments. The algorithm is roughly:

```
PROCEDURE Dump_Env (E_Rec_P);
  BEGIN
  decrement Link_Count;
  IF Link_Count = 0 THEN
  BEGIN
     de-link from Unit_List;
     DISPOSE (Env_Data);
     FOR all E_Rec's on E_Vect whose Seg_Vect <> E_Rec.Seg_Vect DO
        Dump_Env (those E_Rec's);
     FOR all E_Rec's on E_Vect whose Seg_Vect = E_Rec.Sec_Vect DO
     BEGIN
        de_link E_REC^.SEG_SIB;
        DISPOSE (those E_RECs);
     END;
     DISPOSE (E_Rec.Seg_Vect);
  END
  END
```

The Operating System sets its E_Vect entry for the terminating program to NIL, and calls Dump_Env for the outer block's E_Rec. After Dump_Env returns, a pass is made through the Res_SIBs list to find all segments whose Link_Count = 1, and remove them from the Heap.

11.3 Task Environments

A task is a routine that is executed concurrently with other routines. task is implemented by three data structures: the body, the Task Information Block (TIB), and the task stack. In Pascal, a task is known as a PROCESS.

The "main task" of the p-System is the thread of execution that runs from Operating System initialization and all System utility or user program executions to the termination of the Operating System. A program may have subsidiary tasks.

During execution, each subsidiary task uses its own stack instead of the System Stack. The task's activation record is actually contained in the task stack: both are allocated on the Heap, along with an amount of free space into which the stack may grow.

The task body is a portion of a P-code segment. In structure it is no different from the body of a procedure or function.

The amount of space allocated to the task stack depends on the STACKSIZE parameter of the START intrinsic. The default is 200 words.

The main task uses the System Stack for expression evaluation and activation records. The Heap is shared by the main task and all subsidiary tasks.

The TIB of a subsidiary task is allocated on the Heap when the task is started. It contains information about a task's execution environment. This must be maintained, and restored whenever a task is restarted after having been idle.

At any given time, the P-machine may have:

one task running
several tasks ready to run, and
several tasks waiting for semaphores.

The tasks that are ready to run are organized into a queue. There is also a queue of waiting tasks for each semaphore (it may be empty). Tasks in queues are ordered by their priority.

The P-machine register CURTSK always points to the TIB of the currently executing task. The register READYQ points to the first in the list of tasks ready to run.

The following fragment of Pascal code describes a TIB:

```
TIB = RECORD  {Task Information Block}
        Regs: PACKED RECORD
                  Wait_Q: TIB_Ptr;
                  Prior: byte;
                  Flags: byte;
                  SP_Low: Mem_Ptr;
                  SP_Upr: Mem_Ptr;
                  SP: Mem_Ptr;
                  MP: MSCW_Ptr;
                  BP: MSCW_Ptr;
                  IPC: integer;
                  Env: ERec_Ptr;
                  ProcNum: byte;
                  TIBIOResult: byte;
                  Hang_Ptr: Sem_Ptr;
                  M_Depend: integer;
              END {of Regs}
        MainTask: Boolean;
        Start_MSCW: MSCW_Ptr;
      END {of TIB}
```

SP is the P-machine Stack Pointer. SP_Low and SP_Upr are the limits on SP for this task.

MP and BP designate (respectively) the local and global activation records for this task.

IPC is the P-code Instruction Counter (a seg-relative byte pointer), and ProcNum is the number of the executing routine.

Priority contains the task's priority. This is a number from 0..255. The lower the value, the more urgent the priority.

Wait_Q is used when the task is waiting to run, or waiting on a semaphore. Wait_Q is one link in a linked list of TIBs.

When a task is waiting on a semaphore, Hang_Ptr points to that semaphore. If the task is not waiting on a semaphore, Hang_Ptr is NIL. Hang_Ptr allows a task to be removed from a semaphore's wait queue if the task is being terminated.

Flags is reserved for future use.

Env is a pointer to the task's E_Rec. The task's SIB (Segment Information Block) may be found through the E_Rec.

TIBIOResult will in the future be used to save an IORESULT that is local to the task.

M_Depend contains machine-dependent data maintained by the Interpreter. It is initialized to 0.

MainTask, if TRUE, indicates that this is the TIB of a "root" ("parent") task.

StartMSCW points to the MSCW (Mark Stack Control Word) of the routine that START'ed this task.

Further information about tasks appears below in Chapter IV. Figure 4 shows the layout of main memory while the System is running, including the location of task stacks as discussed in this section.

The following fragment of Pascal code describes a TIB:

```
TIB = RECORD  {Task Information Block}
        Regs: PACKED RECORD
                  Wait_Q: TIB_Ptr;
                  Prior: byte;
                  Flags: byte;
                  SP_Low: Mem_Ptr;
                  SP_Upr: Mem_Ptr;
                  SP: Mem_Ptr;
                  MP: MSCW_Ptr;
                  BP: MSCW_Ptr;
                  IPC: integer;
                  Env: ERec_Ptr;
                  ProcNum: byte;
                  TIBIOResult: byte;
                  Hang_Ptr: Sem_Ptr;
                  M_Depend: integer;
                END {of Regs}
        MainTask: Boolean;
        Start_MSCW: MSCW_Ptr;
      END {of TIB}
```

SP is the P-machine Stack Pointer. SP_Low and SP_Upr are the limits on SP for this task.

MP and BP designate (respectively) the local and global activation records for this task.

IPC is the P-code Instruction Counter (a seg-relative byte pointer), and ProcNum is the number of the executing routine.

Priority contains the task's priority. This is a number from 0..255. The lower the value, the more urgent the priority.

Wait_Q is used when the task is waiting to run, or waiting on a semaphore. Wait_Q is one link in a linked list of TIBs.

When a task is waiting on a semaphore, Hang_Ptr points to that semaphore. If the task is not waiting on a semaphore, Hang_Ptr is NIL. Hang_Ptr allows a task to be removed from a semaphore's wait queue if the task is being terminated.

Flags is reserved for future use.

Env is a pointer to the task's E_Rec. The task's SIB (Segment Information Block) may be found through the E_Rec.

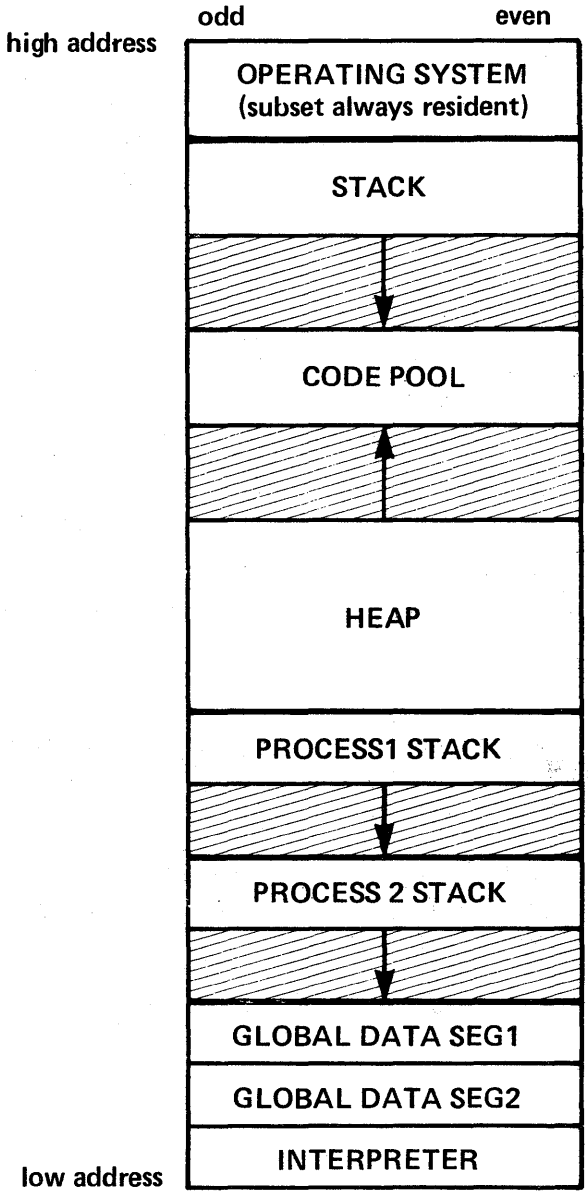
TIBIOResult will in the future be used to save an IORESULT that is local to the task.

M_Depend contains machine-dependent data maintained by the Interpreter. It is initialized to 0.

MainTask, if TRUE, indicates that this is the TIB of a "root" ("parent") task.

StartMSCW points to the MSCW (Mark Stack Control Word) of the routine that STARTed this task.

Further information about tasks appears below in Chapter IV. Figure 4 shows the layout of main memory while the System is running, including the location of task stacks as discussed in this section.



MAIN MEMORY USAGE

FIGURE 4

11.4 P-Machine Instructions

11.4.1 The Intrinsic P_MACHINE

A Pascal compilation unit may directly generate in-line P-code. This is done by calling the intrinsic procedure 'P_MACHINE'. Producing in-line P-code may be useful in very low-level system programming. **Absolutely no protection** is provided by this intrinsic or the System; it can only be used at the user's risk, and extreme caution should be exercised.

The form of a call to P_MACHINE may be sketched as follows:

P_MACHINE (<P-machine item> {, <P-machine item>})

... that is, the parameters to the procedure are a list of one or more <P-machine item>s. A <P-machine item> describes a portion of P-code, and causes one or more bytes to be generated.

There are three varieties of <P-machine item>:

1) P-code syllable: the simplest item is a (non-real) scalar constant. This item produces a single byte of P-code which is the least significant byte of the specified constant.

2) Expression value: if the item is an expression enclosed in parentheses, then a P-code sequence is generated which will compute the value of the expression and leave it on the stack.

3) Address Reference: if the first token of the item is '^', then the item is the specification of a variable, and P-code is generated which leaves the address of that variable on the stack.

... A <P-machine item> may not be a string constant.

EXAMPLE:

Given these declarations:

```
CONST STO = 196;


TYPE Records = RECORD
                    FirstField, SecondField: integer
                END;
     PRecords = ^Records;

VAR Vector: ARRAY [0..9] OF PRecord;
    i: integer;
```

... the following call to P_MACHINE ...

```
PMACHINE ( ^Vector[5]^.FirstField, (i*i), STO)
```

... would cause the square of i to be stored in the first field of the record designated by the sixth element of the array Vector.

11.4.2 P-Code Instruction Set

11.4.2.1 Operands and Notation

11.4.2.1.1 Instruction Parameters

The parameters to a P-code instruction contain information about the location and size of that instruction's operands. They are generated at compile time, and are therefore static. Each P-code uses some (fixed) combination of these parameters.

These are the five possible parameter formats (there are no others):

UB - Unsigned Byte

Represents a positive integer in the range 0..255. When converted to a 16-bit two's complement value, the most significant byte is zeroed.

SB - Signed Byte

Represents a two's complement 8-bit integer in the range -128..127. When converted to a 16-bit two's complement value, the most significant byte is a sign extension (all bits equal bit 7 of the low byte (SB)).

DB - Don't care Byte

Represents a positive integer in the range 0..127. It may thus be treated as either an SB or UB. Bit 7 is always 0.

B - Big

This is a parameter with variable length. If bit 7 of the first byte is 0, the remaining 7 bits represent a positive integer in the range 0..127. If bit 7 of the first byte is 1, then bit 7 should be cleared; the first byte is the high-order byte of a 16-bit word, and the following byte is the low-order byte of that word. The Big format may represent positive integers in the range 0..32767.

W - Word

This is a two-byte parameter. It is a 16-bit two's complement value that represents an integer in the range -32768..32767. The word is always least-significant-byte-first.

11.4.2.1.2 Dynamic Operands

In the P-machine instruction descriptions below, stack-oriented dynamic operands of the P-codes will be discussed. This section describes those operands.

Activation Record

See the following section.

Addr (address)

A 16-bit hardware word address (on byte-addressable processors, this is typically an even quantity).

Bool (Boolean)

A 16-bit quantity treated as a logical value.

Byte-ptr (byte pointer)

A 32-bit quantity. TOS is an index into an array of bytes. TOS-1 is the word address of the base of the byte array. Two words are used in a byte-ptr so that individual bytes may be specified even on word-addressed processors.

Int (integer)

A 16-bit two's complement integer.

Nil

A constant that references an invalid address. The actual value varies from processor to processor.

Offset

An offset into a code segment. This is either a word or a byte offset, depending on the natural addressing unit of the host processor.

Pack-ptr (packed array pointer)

Three words that designate a bit field within a 16-bit word. TOS is the number of the rightmost bit of the field, TOS-1 is the number of bits in the field, and TOS-2 is the address of the word.

Real

A 32-bit or 64-bit floating point quantity.

Set

A set is 0..255 words of bit flags, preceded by a word that contains the number of words in the set.

Word

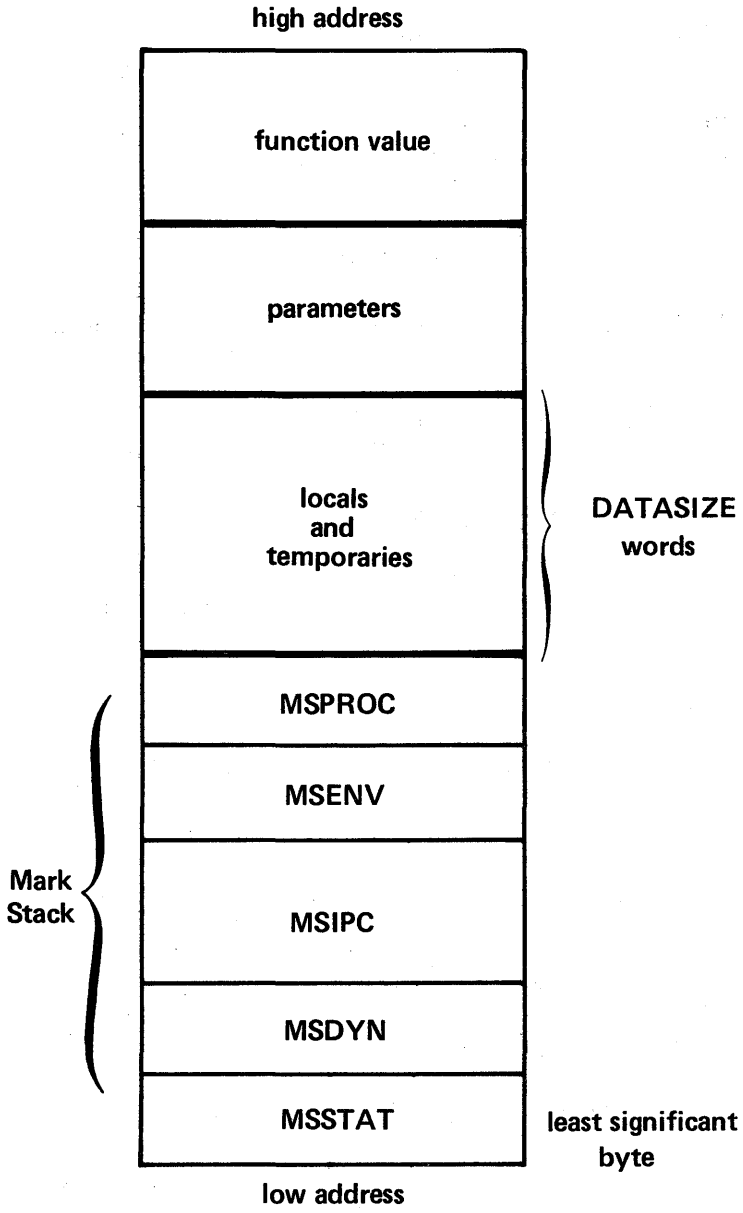
A 16-bit quantity that may be treated in any way: as an integer, Boolean, address, etc.

Word-block

A group of zero or more words.

11.4.2.1.3 Activation Records

An activation record is created for each invocation of an active routine. Figure 5 illustrates an activation record.



PROCEDURE ACTIVATION RECORD

FIGURE 5

The parts of an activation record are:

1) Mark Stack.
   Five (full) words of housekeeping information:
   a) MSSTAT - pointer to the activation record of the lexical parent.
   b) MSDYN - pointer to the activation record of the caller.
   c) MSIPC - seg-relative byte pointer to point of call in the caller.
   d) MSENV - E_Rec pointer of the caller
   e) MSPROC - procedure number of caller

2) Local and temporary variables. This area is DataSize words long.

3) Parameters.
   This area (which may be empty) contains:
   a) Addresses - for VAR parameters, and record and array value parameters.
   b) Values - for other value parameters.

4) Function value. This area is present only for functions, and is either one or two words (or four words, if reals are that size).

11.4.2.1.4 Conventions

Section 11.4.2.2 describes individual P-machine instructions, grouped by the nature of their operation.

On the left is the mnemonic for the instruction, followed by its value (all P-code instructions are represented by a single byte). This is followed by the format for the parameters, if any.

If the the instruction has more than one parameter of the same format, then they are distinguished by an underscore followed by a number (parameters of a given kind are numbered left to right, starting from 1).

On the right is a verbal description of the instruction.

Below the opcode value is a notational description of the P-machine Stack before and after the P-code's execution. Only the expression-evaluation portion (the top words of the stack) is shown.

On the left is a depiction of the Stack before the opcode is executed, followed by a colon (:), followed by a depiction of the stack after the opcode is executed. Each depiction of the Stack is enclosed in angle brackets (<>). Within the

brackets, the stack grows from left to right. Individual operands are separated by commas, and vertical bars represent exclusive alternatives (one or the other value, but not both). Thus the operand closest to the right bracket (>) is the top-of-stack (TOS). Brackets that do not enclose any operands represent an empty evaluation stack.

11.4.2.2 The Individual P-Code Instructions

11.4.2.2.1 Constant One-Word Loads.

SLDC	0..31 <>:<word>	Short Load Word Constant. Push the opcode, with the high byte zero.
LDCN	152 <>:<NIL>	Load Constant NIL. Push NIL. The value may vary across processors.
LDCB	128 UB <>:<word>	Load Constant Byte. Push UB, with high byte zero.
LDCI	129 W <>:<word>	Load Constant Word. Push W.
LCO	130 B <>:<offset>	Load Constant Offset. B is a word offset into the constant pool of the current segment. Convert B to a seg-relative word offset. If operating on a byte addressed machine, then convert to a byte offset. Push the offset on the Stack.

11.4.2.2.2 Local One-Word Loads and Stores

SLDL1 ... **SLDL16**	32 ... 47 <>:<word>	Short Load Local Word. SLDLx: fetch the word with offset x in the local activation record and push it.
LDL	135 B <>:<word>	Load Local Word. Fetch the word with offset B in the local activation record and push it.
SLLA1 ... **SLLA8**	96 ... 103 <>:<addr>	Short Load Local Address. Push the address of the indicated offset in the local activation record.
LLA	132 B <>:<addr>	Load Local Address. Calculate address of the word with offset B in the local activation record and push it.

SSTL1 ... **SSTL8**	104 ... 111 <word>:<>	Short Store Local Word. Store TOS in the indicated offset in the local activation record.
STL	164 B <word>:<>	Store Local Word. Store TOS into word with offset B in the local activation record.

11.4.2.2.3 Global One-Word Loads and Store

SLDO1 ... **SLDO16**	48 ... 63 <>:<word>	Short Load Global Word. SLDOx: fetch the word with offset x in the global data area of the current segment and push it.
LDO	133 B <>:<word>	Load Global Word. Fetch the word with offset B in the global data area of the of the current segment and push it.
LAO	134 B <>:<addr>	Load Global Address. Push the word address of the word with offset B in the global data area of the current segment.
SRO	165 B <word>:<>	Store Global Word. Store TOS into the word with offset B in global data area of the current segment.

11.4.2.2.4 Intermediate One-Word Loads and Store

SLOD1 **SLOD2**	173 B 174 B <>:<word>	Short Load Intermediate Word. Push the word at offset B in the activation record of the parent (LOD1) or grandparent (LOD2) of the local activation record.
LOD	137 DB,B <>:<word>	Load Intermediate Word. DB indicates the number of static links to traverse to find the activation record to use. Push the word at offset B in that activation record.

LDA	136 DB, B <>:<addr>	Load Intermediate Address. DB indicates the activation record as for LOD. Push the address of offset B in that record.
STR	166 DB, B <word>:<>	Store intermediate word. Store TOS at offset B in the activation record indicated by DB.

11.4.2.2.5 Extended One-Word Loads and Store

LDE	154 UB, B <>:<word>	Load Extended Word. Push the word at offset B in the global data area of local segment UB.
LAE	155 UB, B <>:<addr>	Load extended address. Push the address of the word at offset B in the global data area of local segment UB.
STE	217 UB, B <word>:<>	Store extended word. Store TOS at offset B in the global data area of local segment UB.

11.4.2.2.6 Indirect One-Word Loads and Store

SIND0 ... **SIND7**	120 ... 127 <addr>:<word>	Short Index and Load Word. TOS is the address of a record. SINDx: replace it with word x of the record.
IND	230 B <addr>:<word>	Index and Load Word. TOS is the address of a record. Replace it with the B'th word in the record.
STO	196 <addr,word>:<>	Store Indirect. Store TOS into the word pointed to by TOS-1.

11.4.2.2.7 Multiple-Word Loads and Stores

LDC	131 UB_1, B, UB_2 <>:<word-block>	Load Multiple Word Constant. B is a word offset into the constant pool of the current segment. Push the UB_2 words starting at that offset onto the evaluation Stack. If UB_1, the mode, is 2, and the current segment is of opposite byte sex from the host, swap the bytes of each word as it is pushed. If less than B+20 words available to the Stack, issue a Stack fault.
LDM	208 UB <addr>:<word-block>	Load Multiple Words. TOS is a pointer to the beginning of a block of UB words. Push the block onto the Stack, preserving the order of words in the block. If less than UB+20 words available to the Stack, issue a Stack fault.
STM	142 UB <addr,word-block>:<>	Store Multiple Words. TOS is a block of UB words. Transfer the block from the Stack to the destination block starting at the address TOS-1, and preserving the order of words in the block.
LDCRL	242 B <>:<real>	Load Real Constant. Push the real constant designated by the constant pool index B in the current segment. The constant is guaranteed to be in the native byte sex of the host, so no byte flipping is necessary during the load.
LDRL	243 <addr>:<real>	Load Real. TOS is the address of a real variable. Replace the address by the value of the variable.
STRL	244 <addr,real>:<>	Store Real. TOS is the value of a real variable. TOS-1 is an address. Store TOS at the address in TOS-1.

11.4.2.2.8 String and Packed Array of Char Parameter Copying

To copy value parameters of type string or packed array of char into the activation record of a called routine, the calling routine generates a "parameter descriptor." This descriptor is a 2-word record. The first (low address) word is either NIL, or a pointer to an E_Rec. If the first word is NIL, the second word is the address of the parameter. If the first word points to an E_Rec, the second word is an offset relative to the designated segment (the offset is generated by an LCO instruction).

The called routine uses a CAP or CSP instruction to copy the parameter into its activation record. CAP and CSP use the parameter descriptor to do this.

CAP	171 B <addr,addr>:<>	Copy Array Parameter. TOS is the address of the parameter descriptor for a packed array of characters. Cause a segment fault if the parameter descriptor designates a non-resident segment. Otherwise, copy the source (which is B words big) into the destination address at TOS-1.
CSP	172 UB <addr,addr>:<>	Copy String Parameter. TOS is the address of the parameter descriptor for a string. Cause a segment fault if the descriptor designates a non-resident segment. Otherwise, compare the dynamic length of the designated string to UB, the declared size (in bytes) of the destination formal parameter.
Cause		a string overflow fault if the length of the source is greater than the capacity of the destination. Otherwise, copy, for the length of the source, into the destination, whose address is in TOS-1.

II.4.2.2.9 Byte Load and Store

LDB	167 <byte-ptr>:<word>	Load Byte. TOS is a byte pointer. Pop it and push a word with the byte it designated in the least significant bits and a most signifant byte of zero.
STB	200 <byte-ptr,word>:<>	Store Byte. Store byte TOS into the location specified by byte pointer TOS-1.

II.4.2.2.10 Packed Field Load and Store

LDP	201 <pack-ptr>:<word>	Load a Packed Field. Replace the packed field pointer TOS with the field it designates. Before being pushed on the Stack, the field is right-justified and zero-filled.
STP	202 <pack-ptr,word>:<>	Store into a Packed Field. TOS is the right-justified data, TOS-1 a packed field pointer. Store TOS into the field described by TOS-1.

11.4.2.2.11 Record and Array Indexing and Assignment

MOV 197 UB, B
<addr,addr>:<>

Move. Move B words from the source designated by TOS to the destination designated by TOS-1. TOS is either the address of a word block (if UB is zero) of the offset of a constant word block in the current segment. If UB is 2, and the current segment has opposite byte sex from the host, swap the bytes of each word as it is moved.

INC 231 B
<addr>:<addr>

Increment Field Pointer. The word pointer TOS is indexed by B words and the resultant pointer is pushed.

IXA 215 B
<addr,word>:<addr>

Index Array. TOS is an integer index, TOS-1 is the array base word pointer, and B is the size (in words) of an array element. Push a word pointer to the indexed element.

IXP 216 UB_1, UB_2
<addr,word>:<pack-ptr>

Index Packed Array. TOS is an integer index, TOS-1 is the array base word pointer. UB_1 is the number of elements per word, and UB_2 is the field-width (in bits). Compute and push a packed field pointer.

11.4.2.2.12 Logical Operators

LAND	161 <word,word>:<word>	Logical And. AND TOS into TOS-1.
LOR	160 <word,word>:<word>	Logical Or. OR TOS into TOS-1.
LNOT	229 <word>:<word>	Logical Not. Take one's complement of TOS.
BNOT	159 <Bool>:<Bool>	Boolean Not. Complement the low bit and clear the remainder of TOS.
LEUSW	180 <word,word>:<Bool>	Less Than or Equal Unsigned. Push Boolean result of unsigned comparison TOS-1 <= TOS.
GEUSW	181 <word,word>:<Bool>	Greater Than or Equal Unsigned. Push Boolean result of unsigned comparison TOS-1 >= TOS.

11.4.2.2.13 Integer Arithmetic

ABI	224 <int>:<int>	Absolute Value Integer. Take absolute value of integer TOS. Result is undefined if TOS is initially -32768.
NGI	225 <int>:<int>	Negate Integer. Take the two's complement of TOS.
INCI	237 <int>:<int>	Increment Integer. Add 1 to TOS.
DECI	238 <int>:<int>	Decrement Integer. Subtract 1 from TOS.
ADI	162 <int,int>:<int>	Add Integers. Add TOS into TOS-1.
SBI	163 <int,int>:<int>	Subtract Integers. Subtract TOS from TOS-1.

MPI	140 <int,int>:<int>	Multiply Integers. Multiply TOS into TOS-1. This instruction may cause overflow if result is larger than 16 bits.
DVI	141 <int,int>:<int>	Divide Integers. Divide TOS-1 by TOS and push quotient. If TOS is 0, cause an execution error.
MODI	143 <int,int>:<int>	Modulo Integers. Divide TOS-1 by TOS and push the remainder.
CHK	203 <int,int,int>:<int>	Check Subrange Bounds. Insure that TOS-1 <= TOS-2 <= TOS, leaving TOS-2 on the Stack. If conditions are not satisfied, cause a runtime error.
EQUI	176 <int,int>:<Bool>	Equal Integer. Push Boolean result of integer comparison TOS-1 = TOS.
NEQI	177 <int,int>:<Bool>	Not Equal Integer. Push Boolean result of integer comparison TOS-1 <> TOS.
LEQI	178 <int,int>:<bool>	Less than or Equal Integer. Push Boolean result of integer comparison TOS-1 <= TOS.
GEQI	179 <int,int>:<bool>	Greater than or Equal Integer. Push Boolean result of integer comparison TOS-1 >= TOS.

11.4.2.2.14 Real Arithmetic

All overflows and underflows cause a runtime error.

FLT	204 <int>:<real>	Float Top-of-Stack. Convert the integer TOS to a floating point number.
TNC	190 <real>:<int>	Truncate Real. Convert the real TOS to an integer by truncating.

RND	191 <real>:<int>	Round Real. Convert the real TOS to an integer by rounding.
ABR	227 <real>:<real>	Absolute Value of Real. Take the absolute value of the real TOS.
NGR	228 <real>:<real>	Negate Real. Negate the real TOS.
ADR	192 <real,real>:<real>	Add Reals. Add TOS into TOS-1.
SBR	193 <real,real>:<real>	Subtract Reals. Subtract TOS from TOS-1.
MPR	194 <real,real>:<real>	Multiply Reals. Multiply TOS into TOS-1.
DVR	195 <real,real>:<real>	Divide Reals. Divide TOS into TOS-1. If TOS is 0, cause a runtime error.
EQREAL	205 <real,real>:<Bool>	Equal Real. Push Boolean result of real comparison TOS-1 = TOS.
LEREAL	206 <real,real>:<Bool>	Less than or Equal Real. Push Boolean result of real comparison TOS-1 <= TOS.
GEREAL	207 <real,real>:<Bool>	Greater than or Equal Real. Push Boolean result of real comparison TOS-1 <= TOS.

11.4.2.2.15 Set Operations

ADJ	199 UB <set>:<word-block>	Adjust Set. Force the set TOS to occupy UB words, either by expansion (adding zeroes "between" TOS and TOS-1) or compression (chopping of high words of set), and discard its length word. After this operation, if less than 20 words are available to the Stack, cause a Stack fault.

SRS	188 <int,int>:<set>	Build a Subrange Set. The integers TOS and TOS-1 must be in [0..4079]. If not, cause a runtime error, else push the set . If TOS-1 > TOS, push the empty set. Before this operation, if less than 20 words available to the Stack, cause a Stack fault.
INN	218 <int,set>:<Bool>	Set Membership. Push Boolean result of TOS-1 IN TOS.
UNI	219 <set,set>:<set>	Set Union. Push the union of sets TOS and TOS-1. (TOS OR TOS-1)
INT	220 <set,set>:<set>	Set Intersection. Push the intersection of sets TOS and TOS-1. (TOS AND TOS-1)
DIF	221 <set,set>:<set>	Set Difference. Push the difference of sets TOS and TOS-1. (TOS-1 AND NOT TOS)
EQPWR	182 <set,set>:<bool>	Equal Set. Push the Boolean result of set comparison TOS-1 = TOS.
LEPWR	183 <set,set>:<Bool>	Less than or Equal Set. Push true if TOS-1 is a subset of TOS, else push false.
GEPWR	184 <set,set>:<Bool>	Greater than or Equal Set. Push true if TOS is a superset of TOS, else push false.

II.4.2.2.16 Byte Array Comparisons

EQBYT 185 UB_1, UB_2, B
<addr|offset,addr|offset>:<Bool>

Equal Byte Array. TOS and TOS-1 are each a pointer to a byte array (if the corresponding UB is zero) or the offset of the constant byte array in the current segment. B is the size (in bytes) of that array. UB_1 and UB_2 are mode flags. They refer to TOS and TOS-1, respectively. If the byte sex of the segment is different from the host, and the corresponding mode is 2, swap the bytes of each word of that operand, before doing the comparison. Push the Boolean result of the byte array comparison TOS-1 = TOS.

LEBYT 186 UB_1, UB_2, B
<addr|offset,addr|offset>:<Bool>

Less than or Equal Byte Array. TOS and TOS-1 each point to a byte array (if the corresponding UB is zero) or the offset of the constant byte array in the current segment. B is the size (in bytes) of that array. UB_1 and UB_2 are mode flags. They refer to TOS and TOS-1, respectively. If the byte sex of the segment is opposite from the host, and the corresponding mode is 2, swap the bytes of each word of that operand, before doing the comparison. Push the Boolean result of the byte array comparison TOS-1 <= TOS.

GEBYT	187 UB_1, UB_2, B <addr|offset,addr|offset>:<Bool>	Greater than or Equal Byte Array. TOS and TOS-1 each point to a byte array (if the corresponding UB is zero) or the offset of a constant byte array in the current segment. B is the size (in bytes) of that array. UB_1 and UB_2 are mode flags. They refer to TOS and TOS-1, respectively. If the byte sex of the segment is opposite the host, and the corresponding mode is 2, swap the bytes of each word of that operand before doing the comparison. Push the Boolean result of the byte array comparison TOS-1 <= TOS.

11.4.2.2.17 Jumps

UJP	138 SB <>:<>	Unconditional Jump. Jump by byte offset SB.
FJP	212 SB <Bool>:<>	False Jump. Jump by byte offset SB if TOS is false.
TJP	241 SB <Bool>:<>	True Jump. Jump by byte offset SB if TOS is true.
EFJ	210 SB <int,int>:<>	Equal False Jump. Jump by byte offset SB if TOS <> TOS-1.
NFJ	211 SB <int,int>:<>	Not Equal False Jump. Jump by byte offset SB if TOS = TOS-1.
JPL	139 W <>:<>	Unconditional Long Jump. Jump W bytes from current location.
FJPL	213 W <Bool>:<>	False Long Jump. Jump W bytes from current location if TOS is false.

XJP	214 B <int>:<>	Case jump. The first word, W1, with word offset B in the constant pool of the current segment is word-aligned and is the minimum index of the table. The next word, W2, is the maximum index. The case table is the next (W2-W1)+1 words. If the byte sex of the segment is opposite to the host, any of these words must be byte-swapped before they are used. If TOS, the actual index, is in the range W1..W2, then jump W3 words from the current location, where W3 is the contents of the word pointed at by TOS. Otherwise do nothing.

II.4.2.2.18 Routine Calls and Returns

For all procedure call instructions, after the MSCW and Datasize words have been pushed on the Stack, a check is made to see that there are still at least 40 words available between the Stack and the Codepool. If there are not, a Stack fault is issued.

For all calls to external procedures, issue a segment fault if the desired segment is not already in memory.

CPL	144 UB <param>:<activation>	Call Local Procedure. Call procedure UB, which is an immediate child of the currently executing procedure and in the same segment. Static link of the new MSCW is set to old MP.
CPG	145 UB <param>:<activation>	Call Global Procedure. Call procedure UB, which is at lex level 1 and in the same segment. The static link of the MSCW is set to BASE.
SCPI1 **SCPI2**	239 UB 240 UB <param>:<activation>	Short Call Intermediate Procedure. Set the static chain to point to the lexical parent (CPI1) or grandparent (CPI2) of the calling environment. Call procedure UB.

CPI	146 DB, UB <param>:<activation>	Call Intermediate Procedure. Call procedure UB, which is at lex level DB less than the currently executing procedure and in the same segment. Use that activation record's static link as the static link of the new MSCW.
CXL	147 UB_1, UB_2 <param>:<activation>	Call Local External Procedure. Call procedure UB_2, which is an immediate child of the currently executing procedure and in the segment UB_1.
SCXG1 ... **SCXG8**	112 UB ... 119 UB <param>:<activation>	Short Call External Global Procedure. The segment number is indicated by the opcode (1-8) and UB is the procedure number. SCXG1 may refer to a procedure embedded in the Interpreter. If this is the case, an Interpreter table contains the procedure's location.
CXG	148 UB_1, UB_2 <param>:<activation>	Call Global External Procedure. Call procedure UB_2 which is at lex level 1 and in the segment UB_1. If the segment number is 1, then the procedure code may be embedded in the Interpreter; an Interpreter table contains its location.
CXI	149 UB_1, DB, UB_2 <param>:<activation>	Call Intermediate External Procedure. Call procedure UB_2 which is at lex level DB less than the currently executing procedure, and in the segment UB_1.
CPF	151 <param,proc-ptr> :<activation>	Call Formal Procedure. TOS contains a procedure number. TOS-1 contains an E_Rec pointer. TOS-2 contains a static link. Call the indicated procedure.

RPU	150 B <activation>:<func>	Return from Procedure. Restore state of calling procedure from MSCW and discard. Pop MSCW from Stack. Cut back an additional B words from Stack, leaving function value, if appropriate. If returning to different segment (Mark Stack E_Rec <> current E_Rec) then issue a segment fault if necessary. If procedure number in MSCW is < 0, return to EXITIC of procedure, not MSCW's IPC.
LSL	153 DB <>:<addr>	Load Static Link onto Stack. DB indicates the number of static links to traverse. Push the indicated static link.
BPT	158 <>:<activation>	Breakpoint. Unconditionally call execution error procedure.

11.4.2.2.19 Concurrency Support

SIGNAL	222 <addr>:<>	Signal. TOS is a semaphore address. Signal this semaphore.
WAIT	223 <addr>:<>	Wait. TOS is a semaphore address. Wait on this semaphore.

11.4.2.2.20 String Instructions

EQSTR	232 UB_1, UB_2 <addr|offset,addr|offset>:<Bool>	Equal String. TOS and TOS-1 each point to a string variable (if the corresponding UB is zero) or the offset of a constant string in the current segment. UB_1 and UB_2 refer to TOS and TOS-1, respectively. Push the Boolean result of the string comparison TOS-1 = TOS.
LESTR	233 UB_1, UB_2 <addr|offset,addr|offset>:<Bool>	Less or Equal String. TOS and TOS-1 each point to a string variable (if the corresponding UB is zero) or the offset of a constant string in the current segment. UB_1 and UB_2 refer to TOS and TOS-1, respectively. Push the Boolean result of the string comparison TOS-1 <= TOS.
GESTR	234 UB_1, UB_2 <addr|offset,addr|offset>:<Bool>	Greater or Equal String. TOS and TOS-1 each point to a string variable (if the corresponding UB is zero) or the offset of a constant string in the current segment. UB_1 and UB_2 refer to TOS and TOS-1, respectively. Push the Boolean result of the string comparisonT OS-1 >= TOS.

ASTR	235 UB_1, UB_2 <addr,addr|offset>:<>	Assign String. TOS-1 is the address of the destination string variable. UB_2 is the declared size of that string. TOS represents the source for the assignment. It is either the address of a string variable (if the mode, UB_1, is 0) or the offset of a string constant in the current segment. Cause a string overflow fault if the dynamic size of the source string is greater than the declared size of the destination. Otherwise, copy the source into the destination.
CSTR	236 <>:<>	Check String Index. TOS-1 is the address of a string variable. TOS is an index into that variable. Check that the index is between 1 and the current dynamic length of the variable. If not, cause a range-check execution error.

II.4.2.2.21 Miscellaneous Instructions

LPR	157 <int>:<word>	Load Processor Register. TOS is a register number. Push the contents of the register indicated in this fashion: (for SPR, also): a) register number is positive: it is a word index into the current TIB. b) register number is negative: -1 indicates the pointer to the TIB of the currently running task -2 indicates the current E_Vec_P -3 indicates the pointer to the TIB at the head of the ready queue
SPR	209 <int,word>:<>	Store Processor Register. TOS-1 is a register number (defined as for LPR). Store TOS in indicated register.
DUP1	226 <word>:<word,word>	Duplicate One Word. Duplicate one word on TOS.

DUPR	198 <word-block>:<word-block>	Duplicate Real. Duplicate the real on TOS.
SWAP	189 <word,word>:<word,word>	Swap. Swap TOS with TOS-1.
NOP	156 <>:<>	No Operation. Continue execution.
NAT	168 <>:<>	Native Code. Transfer control to native code that begins directly after this instruction. Details are machine-dependent.
NAT-INFO	169 B <>:<>	Native Code Information. Ignore the next B bytes in the P-code stream. This information is used in the generation of native code. Treat the instruction as a long form of NOP.
RESERVE1 ... **RESERVE6**	250 ... 255	These codes are reserved for use by the Compiler to identify embedded compiler directives. They must not be explicitly generated by programs.

III. LOW-LEVEL I/O

III.1 Introduction to the I/O Subsystem

Besides emulating the P-machine, each interpreter must contain some native code to perform certain time-critical operations, and deal with hardware dependencies such as I/O devices. The body of code that is not devoted to emulating P-code is called the Runtime Support Package (RSP). The portion of the RSP that is responsible for I/O is called the RSP/IO.

To make the System as portable as possible, the RSP/IO is machine-independent, except for a portion called the Basic Input/Output Subsystem (BIOS). The BIOS must vary depending on the hardware in use, but the interface between the BIOS and the RSP/IO is standard: calls to routines in the BIOS are clearly defined.

Thus, we have the I/O Hierarchy shown in Diagram 1.0: The user's I/O calls (e.g., READLN, WRITELN) are mapped by the Compiler and Operating System into calls to the RSP (i.e., UNITREAD, UNITWRITE). The RSP/IO itself calls the BIOS which controls the actual device operations. It is important for the reader to recognize that here we are discussing a synchronous I/O system. In other words, when an I/O request has been initiated by a user program, control does not return to that program until the I/O operation is completed.

This chapter describes the behavior and interfaces of the RSP/IO and BIOS. The SBIOS (Simplified BIOS) is described in detail in the Installation Guide. The easiest way to describe its relation to the BIOS and RSP/IO is to sketch the history of I/O support within the p-System.

The first implementation was for the PDP-11, which has well-established standard interfaces to peripheral devices (regardless of manufacturer). In this environment, there was no need for I/O adaptation.

When the p-System was adapted to the 8080 and Z80, the widespread availability of CP/M® was used: p-System I/O called CP/M BIOS routines. In this way, any hardware environment that CP/M already supported could then host the p-System.

As adaptations for additional processers (e.g., the 9900, 6502, and 6800) were begun, it became clear that the p-System needed some analog to the CP/M BIOS. It was at this point that the p-System BIOS, essentially as described in this chapter, was created and standardized.

The final step in this I/O development took place at SofTech Microsystems, where it was realized that:

1) The BIOS definition did not address the problem of standardizing bootstrap mechanisms, and

2) Implementing a BIOS was a difficult task, and virtually required the use of an already running p-System.

The Adaptable System was created to address these problems. The SBIOS is as simple a hardware interface as possible, so that it can be written by a relatively inexperienced programmer. It is called from a unit of "interface code" that accepts BIOS-style calls and emits SBIOS routine calls. This interface code allows the Interpreter/SBIOS interface to be simpler than the BIOS interface. The RSP/IO is essentially unchanged.

The Adaptable System also addresses the bootstrap problem by defining a hierarchy of bootstrap components, only some of which need to be implemented by the user installing a p-System.

A user who has access to a running p-System and the source code for the Interpreter and SBIOS interface code may wish to implement a BIOS-level I/O interface. This is potentially more efficient than an SBIOS-level adaptation, since the more elaborate BIOS interface allows the implementor to take advantage of such performance characteristics as DMA support in the disk interface.

Both BIOS and SBIOS I/O interfaces have been created as the System was adapted to new environments. Earlier adaptations (such as for the PDP-11) do not always use these conventions (though in the future they may).

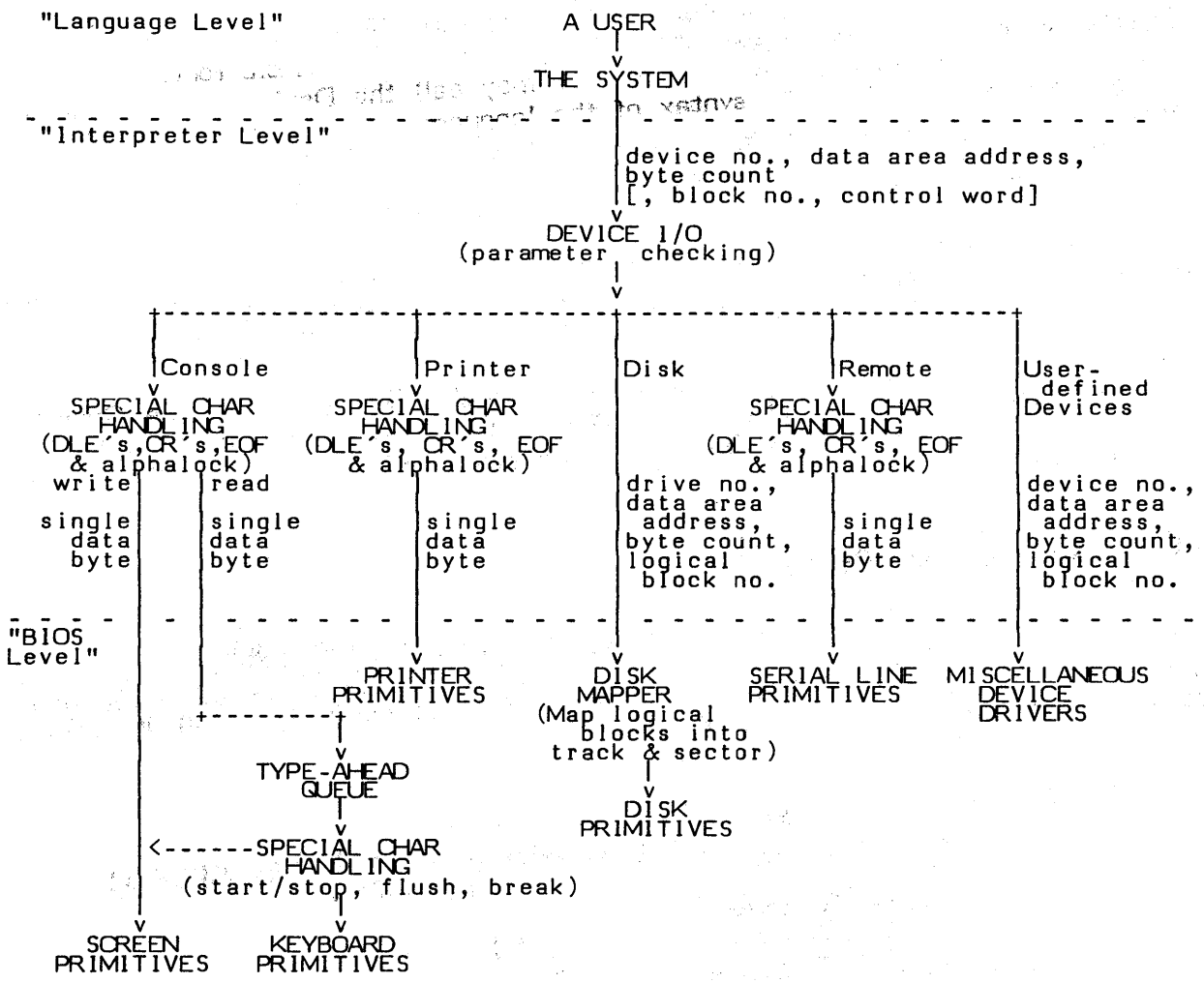


Diagram 1.0 ---- I/O Subsystem Hierarchy

III.2 The Language Level: Device I/O Routines

As mentioned above, all language-level I/O requests are eventually mapped by the Compiler and Operating System into calls to a group of intrinsic routines known as the Device I/O Routines. The programmer may call the Device Routines directly, or may use the standard I/O syntax of the language in use. The exact details of how this mapping is accomplished do not concern us here. The Device I/O Routines are not written in Pascal, but in fact are the native code procedures that comprise the RSP/IO. The way that these procedures are called is described next.

Throughout this chapter, it is assumed that all I/O support at or below the device I/O level is implemented in assembly language. If P-code is the native language of the host processor, these routines may in fact be implemented in Pascal.

The RSP/IO routines are implemented and accessed as routines of the Operating System's unit KERNEL. KERNEL is accessible as segment 1 of every compilation unit. The actual code for the routines may reside in the Interpreter itself, instead of in KERNEL.

III.2.1 Calling the RSP/IO

To the user making direct calls to Device I/O Routines, they look like any other intrinsic routine. If they actually were declared in Pascal, the declarations would have the following format (allowing a few illegitimate constructs such as optional parameters and variable-length arrays):

```
PROCEDURE UNITREAD( UNITNUMBER : INTEGER;
      VAR DATAAREA : PACKED ARRAY [0..BYTESTOTRANSFER-1]
                                          OF 0..255;
            BYTESTOTRANSFER :: INTEGER
            [; LOGICALBLOCK :: INTEGER]
            [; CONTROL : INTEGER] );

PROCEDURE UNITWRITE( <same as for UNITREAD> );

FUNCTION UNITBUSY( UNITNUMBER : INTEGER ) : BOOLEAN;

PROCEDURE UNITWAIT( UNITNUMBER : INTEGER );

PROCEDURE UNITCLEAR(UNITNUMBER : INTEGER );

PROCEDURE UNITSTATUS( UNITNUMBER : INTEGER;
      VAR STATUSWORDS :: ARRAY [0..29] OF INTEGER;
            CONTROL : INTEGER ));
```

Remember that no such declarations actually exist in the System. They are intended to model the parameters passed and returned by the native code RSP/IO routines.

III.2.1.1 Devices and Device Numbers

As described elsewhere, each device is referred to in the System by a given number. The formal parameter UNITNUMBER in the declarations above determines which physical unit the operation is intended for. Thus, the Device I/O Routines are device-transparent to the Pascal programmer; the same procedure will handle any physical unit. Diagram 2.0 is a list of the pre-defined unit numbers associated with each physical unit. The meaning of the other parameters is discussed later in this chapter.

Unitnumber	Volume name
0	<Reserved for the system>
1	CONSOLE
2	SYSTERM
3	<Reserved for the system>
4	disk0
5	disk1
6	PRINTER
7	REMIN
8	REMOUT
9	disk2
10	disk3
11	disk4
12	disk5
13-127	<Reserved for future expansion>

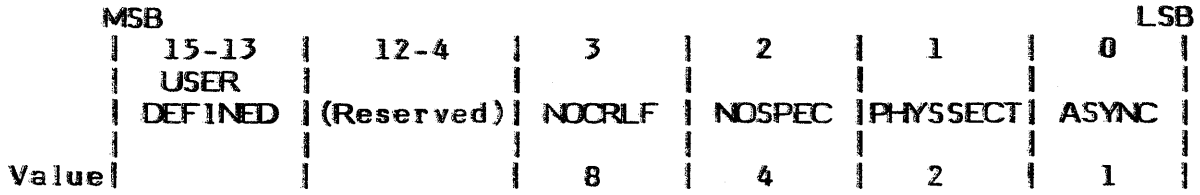
Diagram 2.0 -- Unitnumbers

III.2.1.1.1 User-Defined Devices

The System reserves all device numbers above 127 for user-defined devices. They have no pre-assigned names, yet can be accessed through the UNIT intrinsics just as devices with pre-assigned numbers.

III.2.1.2 CONTROL Parameters

The CONTROL parameter to UNITREAD, UNITWRITE and UNITSTATUS is a word used to pass special information to the RSP/IO and BIOS regarding the handling of the I/O request. The formats of the CONTROL words are shown in Diagrams 2.1 and 2.2.



Bit 0	ASYNC	Set (1) implies asynchronous I/O request. Reset (0) implies synchronous I/O request. (This bit should always be reset.)
Bit 1	PHYSSECT	Set implies "Physical Sector Mode" for disk I/O. Reset implies "Logical Block Mode" for disk I/O. (See section 2.3.1 for details.)
Bit 2	NOSPEC	Set implies "no special character handling". Reset implies "special character handling". (See sections 3.2.1 and 3.2.2 for details.)
Bit 3	NOCRLF	Set implies no LFs are to be appended CRs during non-disk I/O. Reset implies LFs are to be appended to CRs during non-disk I/O. (See sections 3.2.1.2 and 3.2.1.3 for details.)
Bits 4-12		Reserved for future expansion.
Bits 13-15		User-defined functions.

The default setting for all these bits is reset (0).

Diagram 2.1 - CONTROL word format for UNITREAD and UNITWRITE

	MSB	15-13	12-1	0	LSB
		USER DEFINED	(Reserved)	IODIR	
Value				1	

Bit 0 IODIR — Set (1) implies the status of the input channel is to be returned. Reset (0) implies the status of the output channel is to be returned.
Bits 1-12 — Reserved for future expansion.
Bits 13-15 — User-defined functions.

Diagram 2.2 - CONTROL word format for UNITSTATUS

III.2.2 IORESULT and Completion Codes

At times, an I/O request will terminate abnormally. To handle error conditions, a program may use the intrinsic IORESULT. The integer value returned by IORESULT describes the status of the last I/O request.

Each call to UNITREAD, UNITWRITE, UNITCLEAR or UNITSTATUS causes a "completion code" to be set in the SYSCOM data area (SYSCOM, for SYStem COMmunication area, is conventionally the only data space that may be directly accessed by both the Operating System and the Interpreter). Programmers may test the completion code by using IORESULT.

The standard completion codes are given in Diagram 2.3 below.

Code	Meaning
0	No error
1	Bad block, CRC error (parity)
2	Bad device number
3	Illegal I/O request
4	Data-com timeout
5	Volume is no longer on-line
6	File is no longer in directory
7	Illegal file name
8	No room; insufficient space on disk
9	No such volume on-line
10	No such filename in directory
11	Duplicate file
12	Not closed; attempt to open an open file
13	Not open; attempt to access a closed file
14	Bad format; error reading real or integer
15	Ring Buffer Overflow
16	Write attempt to protected disk
17	Illegal block number
18	Illegal buffer address
19 - 127	Reserved for future expansion

Codes 128 through 255 are available for non-predefined, device-dependent errors.

Diagram 2.3 - I/O Completion Codes

III.2.3 Logical Disk Structure

The System views a disk as a zero-based linear array of 512-byte logical blocks. All disks in the System have this logical structure, regardless of their physical format. The physical allocation units of a disk are commonly known as sectors; these may vary widely from one model of drive to another. The BIOS is responsible for mapping the logical structure of a System disk onto the physical structure of the device, i.e., mapping logical blocks onto physical sectors.

III.2.3.1 Physical Sector Addressing Mode

To provide enhanced flexibility for systems programming at a machine-specific level, a mechanism has been provided for directly accessing the physical sectors of the disk. When the PHYSSECT bit (bit 1, value 2) of the CONTROL word is set on a call to UNITREAD or UNITWRITE involving a disk unit, the I/O is performed in Physical Sector Mode. This has the following effects:

1) The parameter LOGICALBLOCK is interpreted by the BIOS as the physical sector number (PSN). (In the future, this may become the least significant 15 or 16 bits of the PSN.)

2) The parameter BYTESTOTRANSFER must be 0. (In the future, this may become the most significant 16 bits of the PSN.)

III.2.3.1.1 Physical Sector Numbers

Typically, the physical sectors of a disk are addressed by specifying both track and sector numbers. That is, the disk is viewed as an array of tracks where each track is an array of sectors. If this data structure were declared in Pascal, it would look like this:

```
type

BYTE = 0..255;

SECTOR = array [0..(BYTESperSECTOR-1)] of BYTE;

TRACK = array [1..SECTORSperTRACK] of SECTOR;

DISK = array [0..(TRACKSperDISK-1)] of TRACK;
```

(Note that here, we are using the convention that track numbers are zero-based but sector numbers start from one.)

We can convert the type DISK into a linear array of SECTOR as follows:

```
type

DISK = array [0..(TRACKSperDISK*SECTORSperTRACK)-1]  of SECTOR;
```

We use this linear representation for addressing the disk by physical sector number (PSN). The relations between the PSN, and track and sector numbers are:

```
PSN = (TRACKNUMBER*SECTORSperTRACK) + SECTORNUMBER-1;
TRACKNUMBER = PSN div SECTORSperTRACK;
SECTORNUMBER = (PSN mod SECTORSperTRACK) + 1;
```

III.2.3.1.2 Physical Sector Size

Any physical sector size may be accomodated. An I/O request in Physical Sector Mode simply causes a full sector to be transferred. The programmer is responsible for ensuring that the data area is at least large enough for one physical sector.

Programs written using physical sector mode are not expected to be portable to different disk hardware without some modification.

III.3 The Interpreter Level: The RSP/IO

This section details the design and operation of the Input/Output portion of the Runtime Support Package (RSP/IO). While the design itself is processor- and hardware-independent, it is intended to be realized in native code. Thus, the final product will be processor-specific but still independent of the exact peripherals used.

III.3.1 Calling Mechanisms

This section now discusses how each routine in the RSP/IO is called from the Pascal level (or the level of another compiled language). The level of detail is intended to be such that an implementor of the RSP will know how to pop parameters off the Stack when the RSP is called, and how the Stack should look when the RSP returns. The detailed semantics of each routine are discussed in Section III.3.2.

III.3.1.1 UNITREAD and UNITWRITE

```
PROCEDURE UNITREAD( UNITNUMBER : INTEGER;
      VAR DATAAREA : PACKED ARRAY [0..BYTESTOTRANSFER-1]
                                    OF 0..255;
            BYTESTOTRANSFER : INTEGER
            [; LOGICALBLOCK :INTEGER]
            [; CONTROL : INTEGER]  );

PROCEDURE UNITWRITE( <same as for UNITREAD> );
```

III.3.1.1.1 Parameter Description

UNITNUMBER has already been discussed.

DATAAREA is the user's buffer to or from which the data will be transferred. Describing it as a VAR parameter signifies that UNITREAD and UNITWRITE are passed a pointer to the start of the data area. This pointer is actually represented as an address couple, consisting of a word base and a byte offset. On processors which use byte addressing, the effective address is computed by simply adding the base and the offset, since both quantities are in bytes. For processors using word addressing, the effective address is computed by indexing byte-wise from the base address (always toward higher locations). Generally, the address of the start of the data area may or may not be on a word boundary. In the case of disk units, however, it is only defined in the case that it is on a word boundary; that is, a Pascal programmer must not allow actual parameters with odd

numbered indices (like A[3]) to occur when transferring to or from the disk. The reason for this inconsistency is to avoid restricting disk data to being moved byte-by-byte.

The third item in the parameter list, BYTESTOTRANSFER, contains the number of bytes to move between the user's data area and the physical unit.

Two optional parameters follow for UNITREAD and UNITWRITE: LOGICALBLOCK and CONTROL. These parameters are optional for the Pascal programmer; the compiler will assign them both the default value zero. LOGICALBLOCK is only relevant for disk reads or writes; as discussed in Section III.2.3, it specifies the Pascal logical block to be accessed. The CONTROL word has been discussed above in Section III.2.1.2.

III.3.1.1.2 Parameter Stack Format

UNITREAD and UNITWRITE receive their parameters on the evaluation stack in the following order (each box represents a 16-bit quantity):

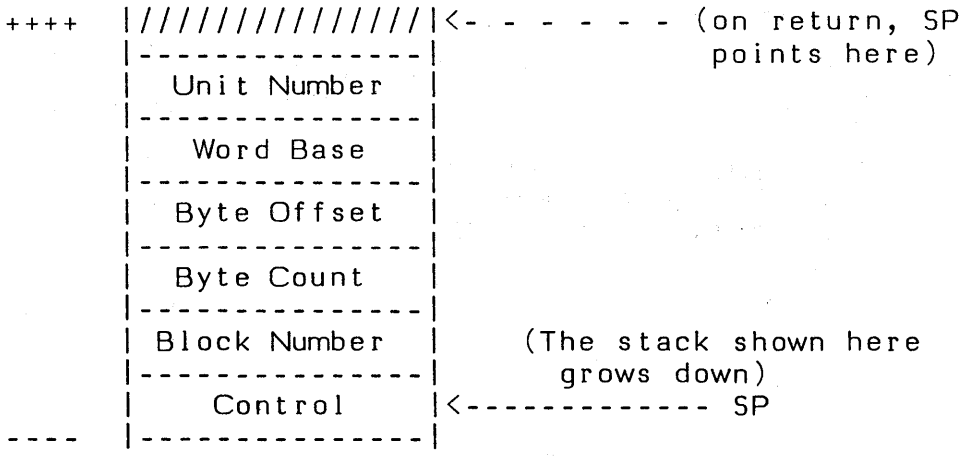


Diagram 3.0 - Stack state on entering UNITREAD or UNITWRITE

Like ordinary Pascal procedures, these RSP routines pop their parameters from the stack when they are finished.

III.3.1.2 UNITBUSY

FUNCTION UNITBUSY(UNITNUMBER : INTEGER) : BOOLEAN

The UNITBUSY function has meaning only in an asynchronous environment and thus will always return FALSE (0) for this synchronous specification. The use of the stack is illustrated in Diagram 3.1.

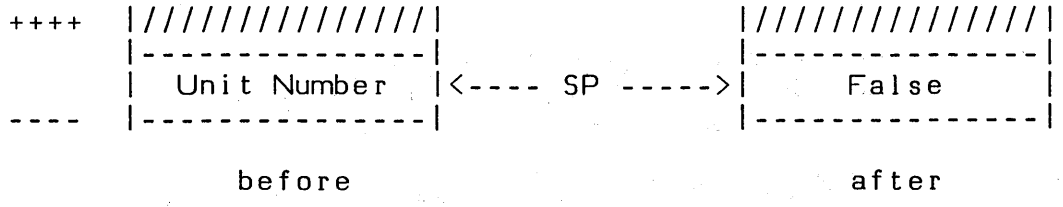


Diagram 3.1 - Stack state before and after UNITBUSY

III.3.1.3 UNITWAIT

PROCEDURE UNITWAIT(UNITNUMBER : INTEGER);

Like UNITBUSY, UNITWAIT is only useful in an asynchronous environment. In a synchronous system, as described here, UNITWAIT becomes essentially a no-op, since no unit will have a I/O request pending. A single parameter is on the top-of-stack when the procedure is called and is popped off before the procedure returns. The use of the stack is illustrated in Diagram 3.2.

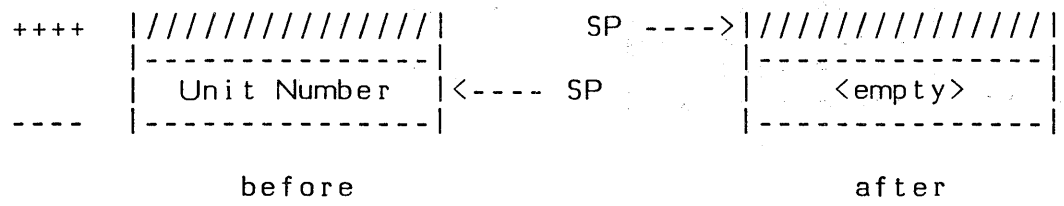


Diagram 3.2 - Stack state before and after UNITWAIT and UNITCLEAR

III.3.1.4 UNITCLEAR

```
PROCEDURE UNITCLEAR( UNITNUMBER : INTEGER );
```

The purpose of UNITCLEAR is to restore the specified unit to its "initial" state. At the RSP level, this would mean clearing any state flags pertaining to the specified unit (see sections III.3.2.1.1 and III.3.2.2.2). The "initial" state for each device at the BIOS level is defined in Section III.4.5. The stack format is identical to that of UNITWAIT (see Diagram 3.2 above).

III.3.1.5 UNITSTATUS

```
PROCEDURE UNITSTATUS( UNITNUMBER : INTEGER;
    VAR STATUSWORDS : ARRAY [0..29] OF INTEGER;
        CONTROL : INTEGER   );
```

The purpose of UNITSTATUS is to acquire various device dependent information from the specified UNIT. The procedure is passed a pointer to a status record (whose length is a maximum of 30 words) into which the status words are sequentially stored (Note: Users may define words starting at word 29 and allocating toward word 0, to allow for the system's use of the first words of the record) and a CONTROL word (see Section III.2.1.1).

UNITSTATUS receives its parameters on the evaluation stack in the following order (each box represents a 16-bit quantity):

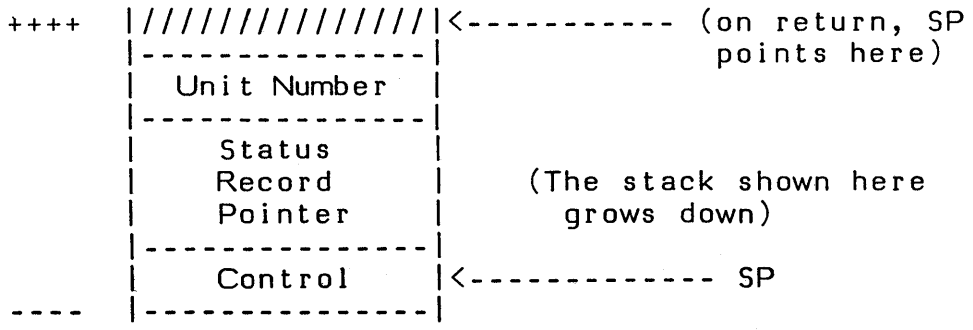


Diagram 3.3 - Stack state before and after UNITSTATUS

III.3.2 Semantics

This section will detail the processing to be performed by the RSP/IO. The primary function of the RSP/IO is to manage calls to the BIOS. Secondarily, the RSP/IO is responsible for handling certain special functions which shall be described here. Appendix A contains a Pascal realization of the RSP/IO which should be considered the most precise reference for the semantics.

III.3.2.1 Special Character Handling on Output

Output to the printer, console or remote units must properly handle Blank Compression Codes and Carriage Returns.

III.3.2.1.1 Blank Compression Code (DLE's)

The System supports textfiles that contain a two-byte blank compression code (only at the beginning of a line). It is the responsibility of the RSP/IO to decode the blank compression code and send an appropriate number of blanks. The first byte is an ASCII DLE (decimal 16) which signals that the next byte contains the excess-32 number of blanks to insert (i.e., it should be interpreted as being the <number of blanks to be sent>+32). Therefore, the next byte following the DLE should be processed by subtracting 32 from its value and sending that number of blanks. Note that negative results, obviously in error, are translated to a value of zero. Note also that the blank-count byte may not be the next input byte processed, due to device switching. This, therefore, requires the maintenance of a flag for each device to indicate that it is currently processing a DLE. The DLE character and the blank-count byte are not normally sent to the device (see Section III.3.2.3).

III.3.2.1.2 Carriage Return -- Line Feed

Textfiles contain ASCII CR's (decimal 13) at the end of lines. We define this character as meaning "New Line", i.e., a carriage return followed by a line feed. Thus, it is the responsibility of the RSP/IO to send an ASCII LF (decimal 10) after sending each CR (also see Section III.3.2.1.3).

III.3.2.1.3 NOCRLF Bit in CONTROL Parameter

When bit 3 (value 8) of the CONTROL parameter is set, the special handling accorded CR's is turned off, i.e., a LF is not automatically appended, and they are sent out like other characters.

III.3.2.2 Special Character Handling on Input

There are several characters which should receive special treatment when received from the console, the printer or the remote devices, in a complete implementation of this I/O system. All but two of them, however, are handled by the BIOS. Those which are handled in the RSP/IO are the EOF and ALPHALOCK characters.

III.3.2.2.1 EOF Character

The EOF character, when received from the console, printer or remote devices, signals that the "end-of-file" has been reached on that particular unit. Rather than being a fixed ASCII code, this is a "soft character". That is, the exact character code which will be interpreted as "End-Of-File" may be changed during system execution by the Pascal user. Further discussion of the soft characters used by the I/O Subsystem may be found in Section III.4.4. The EOF character is in the SYSCOM data area and must be accessed by the RSP/IO to determine what character to look for. When the EOF character is found in the input stream, the action to be taken depends somewhat upon which device was referenced. If we are reading from UNIT 1 (CONSOLE:), then the rest of the user's buffer, starting at the current position, is packed with nulls (decimal 0). For UNIT 2 (SYSTERM:), the printer and the remote, the EOF character is put into the user's buffer. In all cases, no further characters are transferred to the buffer and control returns immediately.

III.3.2.2.2 ALPHALOCK Character

The ALPHALOCK character, when received from a device by the RSP/IO, signals a default case change for all alphabetic characters. All lower case alphabetic characters (i.e., 'a' to 'z') received after the ALPHALOCK character will be converted to upper case. Receipt of another ALPHALOCK character will cause the case to revert back to non-converting mode (the default mode). As for DLE handling described above, a flag for each device to indicate that it is currently in the ALPHALOCK state should be maintained to ensure proper handling when devices are switched. The ALPHALOCK character is not normally returned in the buffer (see Section III.3.2.3).

III.3.2.2.3 BIOS Functions

The remaining special input characters BREAK, START/STOP and FLUSH are used only for input from the console, not from the printer or remote devices. They are handled by the BIOS and are described in Section III.4.5.1.4.

III.3.2.3 NOSPEC Bit in CONTROL Parameter

When bit 2 (value 4) of the CONTROL parameter is set, the special handling accorded DLE's, and the EOF and ALPHALOCK sensing functions described above are turned off. These characters should then be transferred as any other character. The BIOS functions are not affected.

III.4 The Machine Level: The BIOS

As explained above, the Basic Input/Output Subsystem is responsible for providing the actual access to I/O devices. Both the design and implementation of the BIOS are specific to a given processor and I/O configuration. In this section we will attempt to specify the nature of the BIOS in sufficient detail for an experienced programmer to write the code for a given processor and set of peripherals.

The general scheme discussed below uses vectors from the RSP/IO to the BIOS subroutines for reading, writing, initializing and controlling, and answering status requests. The exact vector scheme and means of passing parameters must be worked out separately for each processor. Arrangements that have already been worked out for certain processors are illustrated in Section III.6.2.

III.4.1 Design Goals

The speed of the BIOS code is fairly insignificant, compared to the (slow) speed of the I/O devices that it handles. When peripherals are changed, which may occur frequently, it often proves that only minor changes need to be made to an existing BIOS to service the new hardware. Also, since the BIOS always resides in main memory, each byte it occupies is one less available to the programmer. For these reasons, we suggest that major design goals (assuming correctness!) be (1) compactness and (2) clarity.

Like the rest of the Interpreter, the BIOS should be ROM-able. Obviously, it will also require access to some RAM. The addresses that the BIOS references should be specified in the assembly code by equates, so that it is a simple matter to change them and reassemble the BIOS whenever there is a change in the I/O ports or the memory configuration.

III.4.2 Completion Codes

All read, write, initialization and control, and status calls to the BIOS must return a byte to the RSP that contains status information about the I/O request just serviced. The value of this byte is the "completion code" discussed in Section III.2.2. Most of the standard completion codes are not relevant to the BIOS -- they are returned by the Operating System for file errors and the like. The following standard errors can be returned by the BIOS:

0	No error
1	CRC error
2	Illegal device number
3	Illegal operation on device
4	Undefined hardware error

9	Device not on line
15	Ring Buffer Overflow
16	Write protect; wrttempt to protected disk
17	Illegal block number
18	Illegal buffer address

All other errors are considered hardware-dependent. For these, the BIOS should return codes in the range 128..255. The selection of appropriate codes is left to the BIOS writer.

Note that any pre-defined devices not implemented must arrange to return a completion code of 9 ("Device not on line") when an attempt is made to initialize or use them.

Any user-defined devices not implemented should return a completion code of 2 ("Illegal device number") when an attempt is made to access them.

III.4.3 Calling Mechanisms

In this section we discuss the parameters required in the BIOS calls for each device. Each device has four BIOS calls associated with it: READ, WRITE, CONTROL (CTR L)and STATUS. Each device has varying needs for information associated with these functions. Remember that all calls must return a completion-code byte. For a summary of the BIOS calling requirements, see Section III.6.1.

III.4.3.1 Console

Only one parameter is needed for reading and writing: the data byte to be transferred. The status request requires two parameters: the CONTROL word and the pointer to the status record. For initialization and control of the console, the BIOS requires a number of special control characters. These are provided by passing the BIOS console initialization routine a pointer to the base of the SYSCOM data area, and a pointer to a break-handler routine.

III.4.3.2 Printer

To read and write to the printer, a single parameter is required: the byte that contains the data. To check the status, the CONTROL word and the pointer to the status record are required. For initialization and control, no parameters are needed.

III.4.3.3 Disks

Reading and writing with disk devices requires five parameters:

(1) a starting logical block number as described above
(2) a count of the number of bytes to transfer
(positive signed 16 bits, i.e., 0 to 32K-1)
(3) the address of the data area to transfer to or from
(4) a drive number (0..n-1, given n drives. Currently n=6 is assumed)
(5) the CONTROL parameter.

To check the status, the CONTROL word and a pointer to the status record are passed as parameters. For initialization and control, the drive number is passed.

III.4.3.4 Remote

The remote device requires a single parameter for reading and writing: a byte that contains the data being transferred. As with the devices just described, the status requires the CONTROL word and the pointer to the status record. Initialization and control of the remote device requires no parameters.

III.4.3.5 User-defined Devices

Reading and writing with a user-defined device requires five parameters:

(1) a starting logical block number as described above
(2) a count of the number of bytes to transfer
(positive signed 16 bits, i.e., 0 to 32K-1)
(3) the address of the data area to transfer to or from
(4) a device number (this will be the same as UNITNUMBER)
(5) the CONTROL parameter.

The native code in the BIOS may choose to ignore some of this information, of course.

When checking status, the CONTROL word, device number, and a pointer to the status record are passed. For initialization and control, the device number is passed. It is left up to the device handler to decode the specific device from its device number.

III.4.4 Character Codes

The System assumes that the printer and console devices will support the use of printable ASCII characters and a few standard control codes (CR, LF, SP, NUL and BEL). The remaining control codes that may be useful (such as cursor positioning and screen erasure) are "soft" characters that may be changed by the user (using the utility SETUP) to meet the requirements of some particular hardware.

These soft characters, along with all other information that is entered using SETUP, are stored in the file *SYSTEM.MISCINFO. SYSTEM.MISCINFO is read into a portion of the global data area SYSCOM whenever the System is booted or re-initialized.

The reason for keeping this hardware-dependent information at such a high level is that the control codes for terminals vary widely, and users change terminals fairly often, and so it was necessary to be able to change a terminal without creating a new BIOS. The basic issue is one of mapping logical control symbols into the control codes that are recognized by the hardware.

Suppose, for example, that there is a pre-declared procedure CURSORBACK which causes the cursor on a screen terminal to move left one column. Somewhere in the System, CURSORBACK must cause a control code to be sent to the terminal, which will cause the desired response: control-U, control-H, or an escape sequence. One way to do this would be for the Compiler to emit a standard code which the BIOS then translates into whatever is correct for the current terminal. This has the disadvantage of requiring a new BIOS for every slightly different terminal. The approach which we have taken sees to it that the correct code is sent to the BIOS for the terminal that is currently online. The details of how this is done are described elsewhere.

Since many devices can make use of eight-bit control codes, the System makes no assumptions about the relevance of the high-order bit, and transfers the whole byte unchanged. When using seven-bit ASCII, the value of the high- order bit is defined to be zero. This means that the BIOS must return ASCII codes with the high-order bit off for all standard characters received from the console. This is not required of any of the other devices that are driven by the BIOS.

The RSP sends both upper- and lower-case characters to the BIOS. If a device can handle only upper-case characters, the BIOS must map lower case into upper case.

III.4.5 Semantics

III.4.5.1 Console

In the following discussion, the console device is assumed to be a CRT terminal. A typewriter device may also be used for the console.

III.4.5.1.1 Output Requirements

As noted in above, we depend on the action of certain ASCII control codes. These are the minimum requirements for a console device:

CR <carriage return> (hex 0D) -- Move cursor to the beginning of the current line (column 0).

LF <line feed> (hex 0A) -- Move cursor down one line while the column position remains the same. Starting from any but the last line on the screen, the contents of the screen should remain the same while the cursor moves downward. If the cursor is on the last line when the LF is issued, it should remain in the same position while the rest of the display scrolls upward one line and the bottom line clears.

BEL <bell> (hex 07) -- If an audio signal is available, it should sound. If one is not available, the terminal should do nothing. The delay time required while doing nothing is immaterial.

SP <space> (hex 20) -- Write a space at the current cursor position (erasing whatever is there) and advance the cursor position by one column. If the cursor is already at the last position in a line, the position of the cursor after the SP is undefined. We prefer that the cursor remain in its prior position in this case. If the cursor is in the last column of the last line on the screen, not only is the position of the cursor undefined after the SP, but so is the state of the screen: maybe it scrolled and maybe it didn't. As above, we would prefer that the cursor remain where it was and that the screen not scroll.

NUL <null> (00) -- Delay for the time required to write one character. The state of the console should not change.

any printable character -- Same as the discussion for SP, except, of course, write the character.

Note that the effect of sending non-printable characters other than those described above is not defined, since it is known to vary from terminal to terminal.

III.4.5.1.2 Output Options

The following set of cursor and screen functions should be provided if possible. However, they are optional in the sense that almost all major functions of the System will still be available even if they are not provided. The control characters or sequences of characters which provide these functions are left unspecified (these are soft characters). If a standalone ASCII terminal is connected to the host system, these functions may be provided by the terminal itself. In this case, all the BIOS need do is pass the appropriate control characters.

Reverse Line Feed: Move the cursor to the next line higher on the screen without changing the column or the contents of the screen. If the cursor is already on the top line, the result is undefined. If possible, the screen should reverse-scroll in such a case, or if that is not feasible, the cursor and screen should just remain as they were.

Non-destructive Forward and Backward Space: Move the cursor in the direction indicated without changing the contents of the screen (i.e., move it non-destructively). The position of the cursor is undefined if an attempt is made to move it beyond the beginning or the end of a line. The preferred result is that cursor and screen remain unchanged in such a case.

Cursor HOME: Move the cursor to the upper left-hand corner of the screen without changing the contents of the screen.

Cursor X,Y Positioning: Move the cursor to some absolutely determined row and column without disturbing the contents of the screen. The result is undefined if an attempt is made to move the cursor to a non-existent position.

Erase to End of Screen: Erase from the cursor position to the end of the screen, leaving the cursor where it started and the other contents of the screen undisturbed.

Erase to End of Line: Erase from the cursor position to the end of the current line, leaving the cursor where it started and the rest of the screen undisturbed.

III.4.5.1.3 Input Requirements

Input from the console should not be echoed to the screen by the BIOS; this function is handled by RSP/IO. Keys which represent ASCII characters should generate 8-bit codes between 0 and 127. Other [non-ASCII, e.g., special function] keys can generate codes between 128 and 255, if desired.

III.4.5.1.4 Input Options

If possible, we recommend that the console input BIOS be responsible for the following special functions:

III.4.5.1.4.1 START/STOP

The START/STOP character is used to control console output. When START/STOP (a soft character) is received, console output is suspended until (a) another START/STOP character is received, (b) a FLUSH character is received, (c) the console BIOS is reinitialized, or (d) the BREAK character is received. The action to take in the last three cases is discussed below. Should another START/STOP character be received, the suspended activities should resume exactly as they left off. The chief benefit of this arrangement is that the user can suspend output processes which are proceeding too fast: e.g., a text file is scrolling across the screen at 9600 baud, or a printer must be brought online before the program starts sending it characters. The suspension process takes place wholly within the BIOS, and requires no communication to the RSP. (Note that the START/STOP character is never returned to the RSP. The queueing of keyboard input, if implemented, should continue during the suspension.)

IV.1.4.5.1.4.2 FLUSH

FLUSH is another soft control character; when FLUSH is typed, the console output BIOS discards all output characters (i.e., does not display them) until (a) FLUSH is typed again, (b) input is requested from the console BIOS, (c) the console BIOS is re-initialized or (d) the BREAK character is received. The FLUSH character is never returned to the RSP. If FLUSH is received while a START/STOP suspension is pending, the suspension is cancelled and FLUSH has its usual effect. This feature is useful when a long textfile is being displayed on the console and you're tired of looking at it. Push FLUSH and it terminates rather quickly. It is also useful when a process is generating console output that is irrelevant, but slows down the process. Note that FLUSH applies only to console output.

III.4.5.1.4.3 BREAK

When BREAK (also a soft character) is typed, the console input BIOS should immediately give control to a special Interpreter routine. The vector to this routine is passed at console initialization time. After execution of the BREAK routine, the BIOS should continue as before. The BREAK routine is responsible for notifying the Interpreter that a BREAK should be executed before the next P-code is interpreted. (Note that the BREAK character is never returned to the RSP. Receipt of BREAK should terminate any START/STOP or FLUSH suspension pending.)

III.4.5.1.4.4 Type-Ahead

When non-special characters (i.e., not described in the sections above) are received from the keyboard when no read request is pending, they should be queued until the next read request. The next read request should remove the first character from the queue. When characters in excess of the maximum queue size are received, they should be ignored; the queue should remain intact. While a type-ahead of even one character is better than none at all, we recommend a minimum queue size of about 20 characters. If possible, the bell should be sounded for each character entered from the keyboard after no room remains in the queue.

III.4.5.1.5 Initialization and Control

The initialization and control part of the console BIOS is responsible for the following tasks (and whatever else the BIOS implementor finds expedient):

Soft character recognition: The System stores the soft characters START/STOP, FLUSH and BREAK in a data area called SYSCOM. One parameter to console initialization and control is a pointer to the start of the SYSCOM area. The offsets to these characters from that pointer are (expressed as positive byte offsets):

FLUSH	- 83 decimal (53 hex and 123 octal)
BREAK	- 84 decimal (54 hex and 124 octal)
STOP/START	- 85 decimal (55 hex and 125 octal)

BREAK vector: Another initialization and control parameter is the address of the Interpreter routine which handles BREAK. The console initialization code is responsible for setting up a vector to this address via its private data area and calling this routine when the BREAK character is received.

Flags: Initialization should cause the START/STOP and FLUSH flags to be cleared (or whatever else may be required to return to normal).

Type-ahead queue: Initialization should cause any characters currently waiting in the type-ahead queue to be discarded.

III.4.5.1.6 Status

As described in Section III.2.1.2, bit 0 (value 1) of the CONTROL word defines the direction of the status request. The request should return, in the first word of the status record, the number of characters currently queued for the direction specified. If some form of buffering is being used, this will simply be the number of characters in the buffer. If no buffering is implemented, the output status will always return 0, but the input status will return 1 if a character is waiting to be read, or 0 if none is waiting.

III.4.5.2 Printer

The printer is conceived as being a line printer or other hardcopy device. In actual practice, any ASCII display device may be used.

III.4.5.2.1 Output Requirements

In order to serve the widest variety of hardcopy devices, the RSP/IO does not buffer a line of text and send it all at once. Rather, it sends the printer BIOS a single character at a time. Many line printers must buffer a line and then print it all at once: if this is the case, it is the BIOS that must do so. If this is the case, the BIOS must recognize the end of a line. EOLN is signalled by a certain character: the possibilities are listed below:

CR <carriage return> (hex 0D) -- Print the line and return the carriage to the first column. An automatic line feed should not be done.

LF <line feed> (hex 0A) -- In normal operation, the RSP/IO will only send an LF to the BIOS immediately after a CR. If the hardware allows a simple line feed to be performed (without a return) then this should be done. If a complete "new line" operation (i.e., return and line feed) is the only way your printer can print a line, then do so at an LF: don't do anything about a CR.

FF <form feed> (hex 0C) -- The printer should advance the paper to top-of-form, if possible, and perform a carriage return. If no such feature is available, the printer may execute a "new line" operation, i.e., a return followed by a line feed.

III.4.5.2.2 Input Requirements

There are no strict requirements for input from the printer device. If the printer device has the capability to transmit data, then the printer input BIOS should return all eight data bits unchanged. If not, then input should not be allowed and should return completion code 3 ("Illegal operation on device").

III.4.5.2.3 Initialization and Control

Initialization of the printer device should make it ready to print at the beginning of a blank line. A "new line" (carriage return and line feed) operation may be in order here. Any characters that have been buffered but not printed are lost. The printer does not need to do a form feed each time it is initialized.

III.4.5.2.4 Status

As described above, the number of bytes buffered for the direction specified in the CONTROL word should be returned in the first status word. If the printer has no form of self-checking, return 0.

III.4.5.3 Disk

III.4.5.3.1 Mapping Pascal Logical Blocks onto Physical Sectors

The disk device may be any type of disk drive (e.g., floppy or hard disk). The actual sectoring arrangements of the disk are immaterial. The System addresses the disk in terms of consecutive logical blocks of 512 bytes each. A primary function of the disk BIOS, therefore, is to provide an appropriate mapping scheme into the actual (physical) sectors used on the disk. The sector interleaving algorithm should be optimal for the hardware.

The System makes no assumptions about the interleaving method used by the BIOS (except that it works!).

III.4.5.3.1.1 Bootstrap Location

While bootstrap schemes vary, typical implementations make use of a hardware (usually ROM) bootstrap to load and execute a primary software bootstrap which, in turn, loads and executes a secondary software bootstrap. The secondary bootstrap then loads the Interpreter and Operating System, performs required initializations, and starts the System.

To be accessible to the hardware bootstrap, the primary software bootstrap must reside at a location on the disk which is predetermined by the hardware vendor. Since these locations can vary widely, it is necessary that the System's requirements for a physical disk format be flexible in this regard.

The primary bootstrap area must not overlap disk data structures maintained by the System (chiefly the directory and the bootstrap itself).

Logical blocks 0 and 1 of each disk are usually reserved for bootstrap code (a total of 1024 bytes). This is the most convenient alternative.

If 1024 bytes are not enough room, or if the interleaving format is unacceptable to the hardware bootstrap, the primary bootstrap area must be outside of the "Pascal disk". The Pascal logical blocks must be mapped onto the disk in such a way that

the hardware-defined bootstrap area is inaccessible to the Pascal system as a logical block. (It will still be accessible in Physical Sector Mode (see Section III.2.3.1)).

For Adaptable Systems, full details about bootstrap locations and the mechanisms of booting may be found in the Installation Guide.

III.4.5.3.1.2 Physical Sector Mode

When bit 1 (value 2) of the CONTROL word is set, disk access should be performed in Physical Sector Mode, as described in Section III.2.3.1.

III.4.5.3.2 Output Requirements

The disk device BIOS must transfer as many actual sectors as are needed to accommodate the data. To simplify a disk-write in which (BYTESTOTRANSFER) mod 512 is not equal to zero (i.e., a block is partially written to), the remaining contents of the last block are undefined. This makes it possible to write as much of whatever garbage remains in the buffer, if that is most convenient, to fill up a whole sector. Diagram 4.0 illustrates this situation. The language level is responsible for keeping track (in logical block numbers and byte counts) of where the good data is.

EXAMPLE: Write to disk.

Number of bytes to transfer = 1174
Starting logical block number = 72
Data area address = DATAAREA

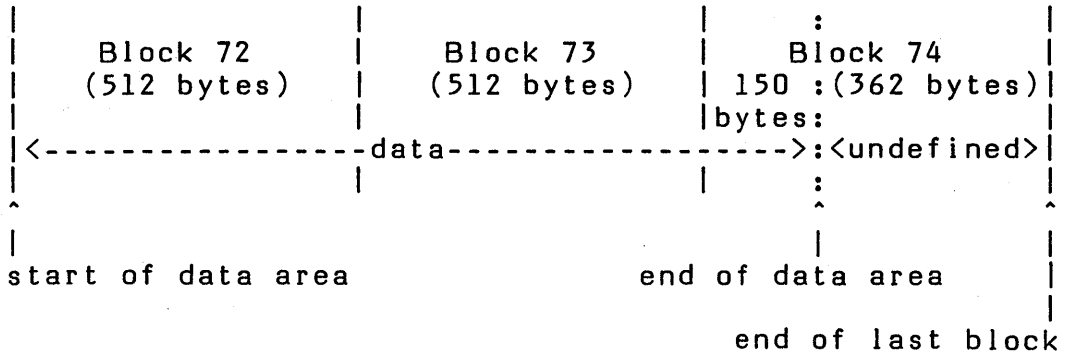


Diagram 4.0 -- State of blocks on Disk after being written

III.4.5.3.3 Input Requirements

On input from a disk device, it is not permissible to over-write the end of the assigned data area. Therefore, the BIOS is responsible for transferring no more than the number of bytes requested. One way to accomplish this is to buffer the last sector and then transfer only the requested part.

III.4.5.3.4 Initialization and Control

Initialization of a disk device should bring it to a state in which it is ready to read or write from any given track or sector. For some drives with simple controllers, the head may need to be stepped to track 0 to facilitate the BIOS disk driver's remembering the current track. Any buffered data is lost.

III.4.5.3.5 Status

Status requests from the disk will return the following words in the status record:

Word 1 - The number of bytes currently buffered for the direction specified in the CONTROL word, as described in Section IV.4.5.1.6 above. If no capability for checking is available, it should be set to 0.

Word 2 - The number of bytes per sector

Word 3 - The number of sectors per track

Word 4 - The number of tracks per disk

III.4.5.4 Remote

This unit is intended to be an RS-232 serial line for supporting various types of communication. It is important that it transfer raw data without changing it in any way. All eight bits of the transferred byte should be considered significant. The transfer rate is usually set to 9600 baud.

III.4.5.4.1 Output Requirements

As noted above, all eight bits of the data byte should be transmitted. The remote BIOS driver is sent one byte at a time.

III.4.5.4.2 Input Requirements

Input from a remote device should be buffered, if possible, by the scheme suggested in Section III.4.5.1.4.4. As noted above, all eight data bits must be returned.

III.4.5.4.3 Initialization and Control

Initialization of the remote device should bring it to a state in which it is ready to read or write.

III.4.5.4.4 Status

The number of bytes buffered for the direction specified in the CONTROL word should be returned in the first status word, as described in Section III.4.5.1.6 above. If no capability for checking is available, it should return 0.

III.4.5.5 User-Defined Devices

These devices are intended to allow the user the freedom to implement devices not specifically defined in this document. The actual implementation is left entirely to the user. The only requirement is that they return a completion code when finished and, if the UNITNUMBER is not defined, that it return code 2 ("Illegal unit number"). Users should use device numbers starting from 128 (see Section III.2.1.1.1).

III.4.6 Special BIOS Calls

These functions are provided by the BIOS to make configuration-specific functions accessible to the Interpreter. Although these functions are not related to Input/Output, they are put into the BIOS as the repository for configuration-specific code.

As with all other routines in the BIOS, each should return a completion code.

III.4.6.1 System Output

System Output is reserved for future expansion and, at this time, should cause the system to HALT. (Note that HALT may actually cause a reboot on some (few) implementations.)

III.4.6.2 System Input

System Input is also reserved for future use, and like System Output, should cause a HALT.

III.4.6.3 System Initialization and Control

The System Initialization and Control BIOS routine should initialize such things as the clock (reset it to 0) and the interrupt system, if either is to be used.

III.4.6.4 System Status

The System Status BIOS routine should return the following information in the status record:

Word 1 - The address of the last word in accessible contiguous RAM memory, e.g., on an 8080 system with 64K bytes of RAM, the last byte address may be 'FFFF', but the last word address is 'FFFE'.

Word 2 - The least significant part of the 32-bit word used by the system clock. If a clock is not present then this must be set to 0.

Word 3 - The most significant part of the 32-bit word used by the system clock. If a clock is not present then this must be set to 0.

Note: If a clock is used, the System assumes that the two words returned are representative of the time in 60ths of a second. It is the clock driver's responsibility to maintain the closest approximation to this time. The time is defined to be 0 at clock initialization. Currently the CONTROL word is ignored.

III.5 Appendices

III.5.1 Appendix A -- Summary of BIOS Calling Sequences

The following is a summary of the calling conventions described in Section III.4.3. Processor-specific protocols for certain machines are shown in the following section. All calls to the BIOS return a completion code.

Entry Point	Parameters
CONSOLEREAD	single data byte
CONSOLEWRITE	single data byte
CONSOLECTRL	BREAK vector
	SYSCOM pointer
CONSOLESTAT	STATREC pointer
	CONTROL word
PRINTERREAD	single data byte
PRINTERWRITE	single data byte
PRINTERCTRL	(none)
PRINTERSTAT	STATREC pointer
	CONTROL word
DISKREAD	block number
	byte count
	data area address
	drive number
	CONTROL word
DISKWRITE	(same as DISKREAD)
DISKCTRL	drive number
DISKSTAT	drive number
	STATREC pointer
	CONTROL word
REMOTEREAD	single data byte
REMOTEWRITE	single data byte
REMOTECTRL	(none)
REMOTESTAT	STATREC pointer
	CONTROL word

Entry Point	Parameters
USERREAD	block number
	byte count
	data area address
	device number
	CONTROL word
USERWRITE	(same as USERREAD)
USERCTRL	device number
USERSTAT	device number
	STATREC pointer
	CONTROL word
SYSREAD	block number
	byte count
	data area address
	device number
	CONTROL word
SYSWRITE	(same as SYSREAD)
SYSCTRL	device number
SYSSTAT	STATREC pointer
	CONTROL word

III.5.2 Appendix B -- Processor-Specific BIOS Calls

III.5.2.1 8080/Z-80

Entry Points: All BIOS entry points are given as positive offsets from the beginning of the BIOS code space. These locations should contain a JMP instruction to the appropriate address in the BIOS.

Parameters: When parameters are not being passed in a specified register, they are pushed on the stack. Offsets from top-of-stack are given, recognizing that the stack grows down.

Completion Code: Return in register A.

Calling Sequence: The RSP will use the CALL instruction to call the BIOS. Thus the return address is at (SP),(SP)+1. All registers are available for use by the BIOS. The BIOS should clean off the stack before returning to the RSP.

Entry Point	Offset(hex)	Parameters
CONSOLEREAD	00	return data byte in Reg C
CONSOLEWRITE	03	write data byte in Reg C
CONSOLECTRL	06	BREAK vector at (SP)+2,(SP)+3 SYSCOM pointer at (SP)+4,(SP)+5
CONSOLESTAT	09	STATREC pointer at (SP)+2,(SP)+3 CONTROL word at (SP)+4,(SP)+5
PRINTERREAD	0C	return data byte in Reg C
PRINTERWRITE	0F	write data byte in Reg C
PRINTERCTRL	12	(none)
PRINTERSTAT	15	STATREC pointer at (SP)+2,(SP)+3 CONTROL word at (SP)+4,(SP)+5
DISKREAD	18	block number at (SP)+2,(SP)+3 byte count at (SP)+4,(SP)+5 data area address at (SP)+6,(SP)+7 drive number at (SP)+8,(SP)+9 CONTROL word at (SP)+A,(SP)+B
DISKWRITE	1B	(same as DISKREAD)
DISKCTRL	1E	drive number in Reg C
DISKSTAT	21	drive number in Reg C STATREC pointer at (SP)+2,(SP)+3 CONTROL word at (SP)+4,(SP)+5
REMOTEREAD	24	return data byte in Reg C

REMOTEWRITE	27	write data byte in Reg C
REMOTECTRL	2A	(none)
REMOTESTAT	2D	STATREC pointer at (SP)+2,(SP)+3 CONTROL word at (SP)+4,(SP)+5
USERREAD	30	block number at (SP)+2,(SP)+3 byte count at (SP)+4,(SP)+5 data area address at (SP)+6,(SP)+7 device number at (SP)+8,(SP)+9 CONTROL word at (SP)+A,(SP)+B
USERWRITE	33	(same as USERREAD)
USERCTRL	36	device number in Reg C
USERSTAT	39	device number in Reg C STATREC pointer at (SP)+2,(SP)+3 CONTROL word at (SP)+4,(SP)+5
SYSREAD	3C	block number at (SP)+2,(SP)+3 byte count at (SP)+4,(SP)+5 data area address at (SP)+6,(SP)+7 device number at (SP)+8,(SP)+9 CONTROL word at (SP)+A,(SP)+B
SYSWRITE	3F	(same as SYSREAD)
SYSCTRL	42	device number in Reg C
SYSSTAT	45	device number in Reg C STATREC pointer at (SP)+2,(SP)+3 CONTROL word at (SP)+4,(SP)+5

III.5.2.2 6500 Series

Entry Points: All BIOS entry points are given as positive offsets from the beginning of the BIOS code space.These locations should contain a JMP instruction to the appropriate address in BIOS.

Parameters: When parameters are not being passed in a specified register, they are pushed on the stack. Offsets from the address pointed to by S (described as (S)) are given, recognizing that the stack grows down and that S normally points to the first available address below valid data.

Completion Code: Return in register X.

Calling Sequence: The RSP will use the JSR instruction to call the BIOS. Thus the return address is at (S)+1, (S)+2. All registers are available for use. The stack should be cleaned off by the BIOS before returning to the RSP.

Entry Point	Offset(hex)	Parameters
CONSOLEREAD	00	return data byte in Reg A
CONSOLEWRITE	03	write data byte in Reg A
CONSOLECTRL	06	BREAK vector at (S)+3,(S)+4 SYSCOM pointer at (S)+5,(S)+6
CONSOLESTAT	09	STATREC pointer at (S)+3,(S)+4 CONTROL word at (S)+5,(S)+6
PRINTERREAD	0C	return data byte in Reg A
PRINTERWRITE	0F	write data byte in Reg A
PRINTERCTRL	12	(none)
PRINTERSTAT	15	STATREC pointer at (S)+3,(S)+4 CONTROL word at (S)+5,(S)+6
DISKREAD	18	block number at (S)+3,(S)+4 byte count at (S)+5,(S)+6 data area address at (S)+7,(S)+8 drive number at (S)+9,(S)+A CONTROL word at (S)+B,(S)+C
DISKWRITE	1B	(same as DISKREAD)
DISKCTRL	1E	drive number in Reg A
DISKSTAT	21	drive number in Reg A STATREC pointer at (S)+3,(S)+4 CONTROL word at (S)+5,(S)+6
REMOTEREAD	24	return data byte in Reg A
REMOTEWRITE	27	write data byte in Reg A
REMOTECTRL	2A	(none)
REMOTESTAT	2D	STATREC pointer at (S)+3,(S)+4 CONTROL word at (S)+5,(S)+6
USERREAD	30	block number at (S)+3,(S)+4 byte count at (S)+5,(S)+6 data area address at (S)+7,(S)+8 device number at (S)+9,(S)+A CONTROL word at (S)+B,(S)+C
USERWRITE	33	(same as USERREAD)
USERCTRL	36	device number in Reg A
USERSTAT	39	device number in Reg A STATREC pointer at (S)+3,(S)+4 CONTROL word at (S)+5,(S)+6
SYSREAD	3C	block number at (S)+3,(S)+4 byte count at (S)+5,(S)+6

		data area address at (S)+7,(S)+8
		device number at (S)+9,(S)+A
		CONTROL word at (S)+B,(S)+C
SYSWRITE	3F	(same as SYSREAD)
SYSCTRL	42	device number in Reg A
SYSSTAT	45	device number in Reg A
		STATREC pointer at (S)+3,(S)+4
		CONTROL word at (S)+5,(S)+6

III.5.2.3 6809

Entry Points: All BIOS entry points are given as positive offsets from the beginning of the BIOS code space. These locations should contain a vector to the appropriate address in the BIOS.

Parameters: When parameters are not being passed in a specified register, they are pushed on the stack. Offsets from the address pointed to by SP (described as (SP)) are given, recognizing that the stack grows down and that SP normally points to the first available address below valid data.

Completion Code: Return in register B.

Calling Sequence: The RSP will use the JSR instruction to call the BIOS. Thus the return address will be at (SP)+0, (SP)+1. The U and Y registers contain interpreter information which must be preserved/restored by the BIOS prior to returning to the RSP. All other registers are available for use. The stack should be cleaned off by the BIOS before returning to the RSP.

Entry Point	Offset(hex)	Parameters
CONSOLEREAD	00	return data byte in Reg A
CONSOLEWRITE	02	write data byte in Reg A
CONSOLECTRL	04	BREAK vector at (SP)+2,(SP)+3
		SYSCOM pointer at (SP)+4,(SP)+5
CONSOLESTAT	06	STATREC pointer at (SP)+2,(SP)+3
		CONTROL word at (SP)+4,(SP)+5
PRINTERREAD	08	return data byte in Reg A
PRINTERWRITE	0A	write data byte in Reg A
PRINTERCTRL	0C	(none)
PRINTERSTAT	0E	STATREC pointer at (SP)+2,(SP)+3
		CONTROL word at (SP)+4,(SP)+5
DISKREAD	10	block number at (SP)+2,(SP)+3
		byte count at (SP)+4,(SP)+5

		data area address at (SP)+6,(SP)+7 drive number at (SP)+8,(SP)+9 CONTROL word at (SP)+A,(SP)+B
DISKWRITE	12	(same as DISKREAD)
DISKCTRL	14	drive number in Reg A
DISKSTAT	16	drive number in Reg A STATREC pointer at (SP)+2,(SP)+3 CONTROL word at (SP)+4,(SP)+5
REMOTEREAD	18	return data byte in Reg A
REMOTEWRITE	1A	write data byte in Reg A
REMOTECTRL	1C	(none)
REMOTESTAT	1E	STATREC pointer at (SP)+2,(SP)+3 CONTROL word at (SP)+4,(SP)+5
USERREAD	20	block number at (SP)+2,(SP)+3 byte count at (SP)+4,(SP)+5 data area address at (SP)+6,(SP)+7 device number at (SP)+8,(SP)+9 CONTROL word at (SP)+A,(SP)+B
USERWRITE	22	(same as USERREAD)
USERCTRL	24	device number in Reg A
USERSTAT	26	device number in Reg A STATREC pointer at (SP)+2,(SP)+3 CONTROL word at (SP)+4,(SP)+5
SYSREAD	28	block number at (SP)+2,(SP)+3 byte count at (SP)+4,(SP)+5 data area address at (SP)+6,(SP)+7 device number at (SP)+8,(SP)+9 CONTROL word at (SP)+A,(SP)+B
SYSWRITE	2A	(same as SYSREAD)
SYSCTRL	2C	device number in Reg A
SYSSTAT	2E	device number in Reg A STATREC pointer at (SP)+2,(SP)+3 CONTROL word at (SP)+4,(SP)+5

IV. THE OPERATING SYSTEM

IV.1 Organization

IV.1.1 Structured Overview of the System

The IV.0 Operating System is a collection of Pascal UNITs. The organization of UNITs in the Operating System was determined by three considerations: functional grouping, space and language restrictions, and necessary code-sharing with other portions of the System. Some UNITs (such as SCREENOPS) are intended to be accessible to user programs as well. The name of a UNIT in the Operating System generally reflects its function. This is a full list of Operating System UNITs:

Unit Name	Function
HEAPOPS EXTRAHEAP PERMHEAP	Heap operators
SCREENOPS	Screen control
FILEOPS	File and Directory operations
PASCALIO EXTRAIO SOFTOPS	File-level I/O
SMALLCOMMAND COMMANDIO	I/O redirection and chaining
STRINGOPS	String intrinsics
OSUTIL	Conversion utilities
CONCURRENCY	Concurrency
REALOPS	Floating Point Functions and Real Number I/O
LONGOPS	Long Integer operations
GOTOXY	Screen cursor control (may be user-supplied)
KERNEL	Nonswappable central facilities of Op. System (always resident in main memory)

GETCMD	Subsidiary segments of KERNEL
USERPROG	(swappable)
INITIALIZE	
PRINTERROR	

KERNEL contains the resident code necessary to maintain the codepool, handle faults, and read segments. The Kernel also contains four subsidiary segments, which are swappable:

GETCMD processes user input at the main command level, and builds a user program's runtime environment;

USERPROG is the reserved segment slot for the user's program (at bootstrap time it contains the Pascal-level code which builds the initial runtime environment for the Operating System);

INITIALIZE is called when the System is booted or re-initialized: it reads SYSTEM.MISCINFO, locates the System codefiles, and sets up the table of devices;

PRINTERROR prints runtime error messages.

The Operating System UNITs are compiled separately. They are bound together in a single codefile, SYSTEM.PASCAL, by using the utility LIBRARY.

Because of certain bootstrap restrictions, KERNEL must always reside in segment-slot 0 and USERPROG must always reside in slot 15. There are no other restrictions on the location of units within SYSTEM.PASCAL.

IV.2 P-Machine Support

IV.2.1 The Heap

IV.2.1.1 Overview

The Heap is an area in low memory used for the allocation of dynamically stored variables. The upper bound of the Heap depends upon the size of the Stack and the Codepool. The area between the Heap and the Codepool is provisionally available to the Heap: Stack faults and segment faults may change the size of this area. Heap faults are used by the Heap operators to request that more space be allocated to the heap.

The Heap is manipulated by a number of intrinsic routines. These either allocate or de-allocate Heap space in a particular way. The rest of this section is an introduction to these routines.

IV.2.1.1.1 MARK and RELEASE

MARK saves the location of the current top of the Heap. RELEASE cuts the Heap back to the location of the corresponding mark. Variables which were allocated between the time of the MARK and the time of the RELEASE are removed from the Heap, except for variables allocated by PERMNEW. MARK and RELEASE may be nested; the integrity of the Heap requires that they be correctly paired.

IV.2.1.1.2 NEW and VARNEW

NEW and VARNEW cause variables to be allocated on the Heap above the "topmost" mark. NEW(P), where variable P is a pointer to type T, causes the number of words in type T to be allocated. P is assigned the address of the first location allocated to P on the Heap. If T is a record with variants, space for the largest variant is allocated. In Pascal, a call to NEW may designate a particular variant, so that space for this particular variant is allocated (which may be less than the largest variant in that record).

VARNEW(P,NWords), where P is a pointer to type T, causes NWords to be allocated on the Heap. T would most commonly be an array. NWords (indirectly) determines how many elements of the array are actually available in this instance. P returns the address of the first location allocated on the Heap.

VARNEW is a function, and returns the number of words that actually were allocated: this should equal NWords; if it is 0, then there was less than NWords of available space, and if it is some other number, something went wrong.

IV.2.1.1.3 DISPOSE and VARDISPOSE

DISPOSE and VARDISPOSE de-allocate space reserved by NEW and VARNEW, respectively. DISPOSE(P) frees the number of words pointed to by P. VARDISPOSE(P,NWords) frees NWords words. In both cases, P is assigned the value NIL.

To avoid destroying important information that is on the Heap, extreme caution should be used with these intrinsics, which do little error-checking of their own. Heap space allocated by a VARNEW should be freed only by a VARDISPOSE with the same NWords parameter, and MARK/RELEASE pairs should always match. Furthermore, if the NEW is called for a specific variant, the same variant should be used to DISPOSE that area.

If these intrinsics are misused, the System is likely to crash: this is the least mysterious of the symptoms that may occur.

IV.2.1.1.4 PERMNEW and PERMDISPOSE

A variable can be allocated on the Heap by PERMNEW(P), where P is a pointer to the variable's type. A variable allocated by PERMNEW can only be de-allocated by PERMDISPOSE(P). Even a RELEASE cannot remove it. These routines are meant for System use, and are not user routines.

The Operating System uses these routines to allow variables to remain defined across MARK/RELEASE pairs. Program CHAIN commands are saved on the Heap with PERMNEW, so that even after the chaining program terminates, and its Heap space is released, these commands are still available to determine the further actions of the System.

IV.2.1.2 Heap Implementation

IV.2.1.2.1 Operating System Interface

IV.2.1.2.1.1 Unit Organization

Code for the Heap operators is contained in three units: HEAPOPS, EXTRAHEAP, and PERMHEAP. HEAPOPS contains MARK, RELEASE, and NEW. EXTRAHEAP contains DISPOSE, VARNEW, VARAVAIL, MEMLOCK, and MEMSWAP. PERMHEAP contains PERMNEW, PERMDISPOSE, and PERMRELEASE. (VARAVAIL, MEMLOCK, and MEMSWAP are for segment management and are discussed elsewhere.)

IV.2.1.2.1.2 Heap Globals

The Operating System uses several variables to manage the Heap. The Heap is maintained by a linked list of MARKs. The topmost MARK is indicated by HeapInfo.TopMark. A MARK (also called an HMR, for Heap Mark Record) has the following structure:

```
TYPE
  MemLink = RECORD
              Avail_list: MemPtr;
              NWords: integer;
              CASE Boolean OF
                true: (Last_Avail,
                       Prev_Mark: MemPtr);
            END;
```

In a MARK, NWords is always 0, and the variant is always TRUE. NWords is 0 because the MARK merely marks a location on the Heap, and does not reserve any space.

Each MARK points to an Avail_List, which is a list of records of type MemLink. These records are FALSE variants of MemLink, and NWords contains the number of words of available space (including the two words of the record itself). The Avail_List chain is ended by an Avail_List of NIL.

The first MARK on the Heap contains a Prev_Mark of NIL. All successive MARKs point back to their predecessor, so that the MARK chain can be traversed.

For each MARK, the first Avail_List record is the lowest unallocated space above the MARK. Last_Avail points to the last of the available space. This is typically bounded by allocated Heap space or by another MARK; if the MARK is TopMark, Last_Avail is bounded by the Codepool.

The Heap maintenance variables have the following structure:

```
VAR
  HeapInfo: RECORD
              Lock: semaphore;
              TopMark,
              HeapTop: MemPtr;
            END;
  PoolBase: MemPtr;
```

PermList: MemPtr;

The Lock semaphore guarantees that the Heap is modified by only one process at a time. TopMark points to the highest MARK. HeapTop points to the highest allocated space on the Heap. The fault handler uses HeapTop to determine how close the Codepool can be moved towards the Heap. PoolBase points to the base of the Codepool. PermList points to a linked list of PERMNEW'ed variables. The list is identical in structure to an Avail_List, but each NWords indicates the number of words allocated by a PERMNEW. If PermList is NIL, then no variables have been PERMNEW'ed.

IV.2.1.2.1.3 Tactics

In general, a request for Heap space through a MARK, NEW, VARNEW, or PERMNEW causes HeapTop to be set to the new top of the Heap. The fault handler always places the Codepool (located at PoolBase) above HeapTop; thus, HeapTop reserves space for the Heap as soon as a Heap operator requests it. This is necessary because of possible interactions between Stack fault handling and Heap space allocation.

The Operating System uses the global variable SysCom^.GDirP (global directory pointer) to allocate a disk directory on the Heap. The Operating System's use of this Heap space is meant to be invisible to the user. Therefore, before any Heap operation (except DISPOSE), SysCom^.GDirP is DISPOSE'd to make the space occupied by the directory available again.

IV.2.1.2.2 Runtime Environment

Since both the user and the Operating System use the Heap, the Operating System MARK's the Heap immediately before the execution of a user program by the call:

```
MARK (EMPTYHEAP);
```

... after the user program terminates, the Operating System calls:

```
RELEASE (EMPTYHEAP);
```

Thus, all user space is freed after the program terminates, unless space has been allocated by one or more calls to PERMNEW.

MARK (EMPTYHEAP) occurs after the runtime environment for the user program has been built. The program's runtime environment structures such as SIBs, E_Rec's, and E_Vec's, are for the use of the Operating System, and are allocated

space before EMPTYHEAP. Data that is global to the user program and any units it USES also appears before EMPTYHEAP. Heap space that follows EMPTYHEAP is intended only for the local use of the user program.

The Heap is shared by all tasks in the System.

IV.2.2 The Codepool

The Codepool resides in main memory between the Stack and the Heap. It contains executable code segments that may possibly be discarded, or swapped in from disk again. Thus, the contents, size, and position of the Codepool may change during a program's execution. The flexibility of the Codepool handling can provide a running program with more free memory space than in previous versions.

A segment in the Codepool must be either P-code or relocatable native code. Nonrelocatable native code segments reside on the Heap: they are placed there at associate time.

The Codepool is a contiguous block of code segments: whenever a segment is discarded, the surrounding segments are moved together. Segments being swapped in are given space at either end of the Codepool.

Segments in the Codepool are organized into a doubly-linked list by pointers in each segment's SIB (described in the previous chapter).

The routines that manage the Codepool are in the Operating System's KERNEL unit. They make use of the pointers within the SIB, and the following global values:

PoolHead: SIB_Ptr;	Points to the SIB of the segment at the base of the Codepool (next to the Heap).
PermSIB: SIB_Ptr;	Points to the SIB of the segment that is always resident in the Codepool (currently, GOTOXY).
PoolBase: Mem_Ptr;	Points to the memory location at the base of the Codepool.
SP_Low: Mem_Ptr;	The lowest possible bound of the Stack; this points to the address which is one word above the top of the Codepool.
HeapTop: Mem_Ptr;	Points to the top of the Heap.

When space is requested either for the Heap or the Stack, the Codepool management routines first attempt to re-position the Codepool without swapping out any segments.

The actual bounds of the Codepool are in Pool_Base, which points to the low end of the Codepool, and SP_Low, which points to one word above the top of the Codepool. The Codepool operators may move it all the way to HeapTop on the Heap side, or up to SP minus a 40-word margin on the Stack side.

The Codepool may be modified by any of the following circumstances:

(1) A Heap fault is detected, and the Codepool is moved up in memory toward the Stack to free the needed number of words for the Heap.

(2) A Stack fault is detected, and the Codepool is moved down in memory toward the Heap to free the needed number of words for the Stack.

(3) A Heap fault or Stack fault is detected, and the Codepool cannot be moved to allocate the space: one or more segments are swapped out, the remaining segments are moved together, and the Codepool is positioned to allow for the needed Heap or Stack space.

(4) A Heap or Stack fault is detected, and even after swappping out all of the swappable segments, not enough space is available: a STACK OVERFLOW is reported, and the System is re-initialized.

(5) A segment fault is detected. The Codepool management routines first try to read the segment in at either end of the Codepool without moving it. If this is impossible, they attempt to create more room by moving the Codepool toward either the Stack or the Heap, and then read the segment. If this too is impossible, segments are swapped out to make room, and the new segment is then read in. If this last effort also fails, a STACK OVERFLOW is reported, and the System is re-initialized.

The Codepool management routines are only called by the Faulthandler. Since the Faulthandler is a subsidiary task, its own stack is statically allocated. Thus, the Faulthandler can manipulate the Codepool freely, without fear of causing a Stack fault.

IV.2.3 Fault Handling

When memory space is required by the Stack or Heap, or entry into a non-resident segment is attempted, a fault is issued. The Faulthandler process is activated, and uses the Codepool management routines to rearrange main memory (as described in the previous section).

The Faulthandler is a process that is START'ed at bootstrap time. Most of the time it is idle, WAIT'ing on a semaphore. When the semaphore is SIGNAL'ed, the Faulthandler is activated and performs its memory management functions.

Faults can be SIGNAL'ed by the Interpreter (Stack and segment faults), or by the EXECERROR procedure in the Operating System (Heap faults and one segment fault).

The semaphore record used by the Faulthandler resides in SYSCOM. It is declared as follows:

```
Fault_Message = RECORD
                    Fault_TIB: TIB_Ptr;
                    Fault_E_Rec: E_Rec_Ptr;
                    Fault_Words: integer;
                    Fault_Type: Seg_Fault .. Heap_Fault;
                END;

Fault_Sem: RECORD
               Real_Sem, Message_Sem: semaphore;
               Message: Fault_Message;
           END;
```

The Interpreter detects only Stack and segment faults. When the Interpreter detects a fault, it places the appropriate information in Fault_Sem.Message and SIGNAL's Fault_Sem.Message_Sem. The SIGNAL causes a task switch to the Faulthandler, and the fault is processed. After it has dealt with the Codepool, Faulthandler WAIT's: this causes a task switch back to the previously running process. The instruction that caused the fault is re-executed.

The Operating System issues Heap faults, and in one instance, a segment fault. Heap faults are detected by the Heap operators when requests are made for Heap space by MARK, NEW, VARNEW, and PERMNEW. The one segment fault is issued by MEMLOCK if a segment to be locked in the Codepool is not already resident.

To issue a fault, the Operating System calls the execution error procedure (EXECERROR), and passes it the needed information. EXECERROR then performs a SIGNAL on Message_Sem.

The Faulthandler first ensures that the currently running segment is not swapped out, and then uses the Codepool management routines to adjust the main memory layout.

If a Stack fault is caused by a call to a routine in a different segment, Faulthandler must lock both calling and called segments into memory.

IV.2.4 Concurrency

Operating System routines support concurrency only by the activation and de-activation of processes: actual task switching is accomplished by the P-machine operations SIGNAL and WAIT.

Concurrency support in Version IV.0 is intended for low-level tasks. Most System-level facilities, particularly I/O, are synchronous. For instance, a READ or UNITREAD from the console does not return to the caller until a character is available. No task switch can occur during the waiting period.

The Operating System global variable Task_Info is used to keep track of some of the data for subsidiary processes. Its structure is as follows:

```
Task_Info: RECORD
                Lock,
                Task_Done: semaphore;
                N_Tasks: integer;
           END {of Task_Info};
```

Task_Info.Lock is used to ensure mutual exclusion while changing the values of other Task_Info fields. Task_Done is used to WAIT on the termination of any subsidiary processes. N_Tasks is the number of subsidiary tasks that have been START'ed.

The unit CONCURRENCY has three routines: START, STOP, and BLK_EXIT. For each process initiation, the Compiler emits initialization code that signals the semaphore passed to START. The Compiler also emits a call to STOP in the exit code of each process; a call to BLK_EXIT is part of the exit code of a main process.

START builds the data structures for a new task and sets it in execution. The task's TIB, activation record, and stack space are allocated on the Heap, and the Operating System forces a task switch by issuing a WAIT. Presumably, the new process starts executing, and switches back to START by doing a SIGNAL after its parameters have been copied. Actually, when START performs the WAIT, it is the process with the highest priority that begins executing.

STOP records the termination of a process. It decrements Task_Info.N_Tasks, SIGNAL's Task_Info.Task_Done, and then initializes and waits on a dummy semaphore in order to force a permanent task switch from the terminating process.

BLK_EXIT is called by a main task, and waits for the termination of all subsidiary tasks. It waits on Task_Done, and terminates the main task when N_Tasks equals zero.

IV.3 I/O Support

IV.3.1 FIBs

File I/O is controlled with a structure called a FIB (File Information Block). When a user declares a file, the Compiler emits code to initialize a FIB for that file. A FIB is declared as follows:

```
FIB = RECORD
        FWindow: Window_P;
        FEOF, FEOLN: Boolean;
        FState: (FJandW, FNeedChar, FGotChar);
        FRecSize: integer;
        FLock: semaphore;
        CASE FIsOpen: Boolean OF
          true: (FIsBlkd: Boolean;
                 FDev: DevNum;
                 FVolID: VolID;
                 FReptCnt,
                 FNxtBlk,
                 FMaxBlk: integer;
                 FModified: Boolean;
                 FHeader: DirEntry;
                 CASE FSoftBuf: Boolean OF
                   true: (FNxtByte, FMaxByte: integer;
                          FBufChngd: Boolean;
                          FBuffer: PACKED ARRAY [0..FBlkSize]
                                   OF CHAR))
    END {of FIB}
```

FWindow points to the current character in the file's buffer. FEOF and FEOLN are the EOF and EOLN flags. FState indicates that the file is either a standard (Jensen & Wirth) file, an INTERACTIVE file awaiting a character, or an INTERACTIVE file with a character. FRecSize is 0 for untyped files, 1 for INTERACTIVE files and textfiles; if it is larger than zero, it indicates the size (in bytes) of a record. FLock is used to ensure that only one process at a time may modify the file. FIsOpen is TRUE only when the file is open.

If FIsOpen is TRUE, then several other fields become relevant. FIsBlkd is TRUE if the file resides on a block-structured device. FDev is the number of that device, and FVolID the name of the volume. FReptCnt contains a count of the number of times the window value is valid before another GET is needed. FNxtBlk is the next (relative) block to access. FMaxBlk is the maximum (relative) block that can be accessed. FModified becomes TRUE if the file is modified: a new date is then set in the directory. FHeader is a copy of the file's directory

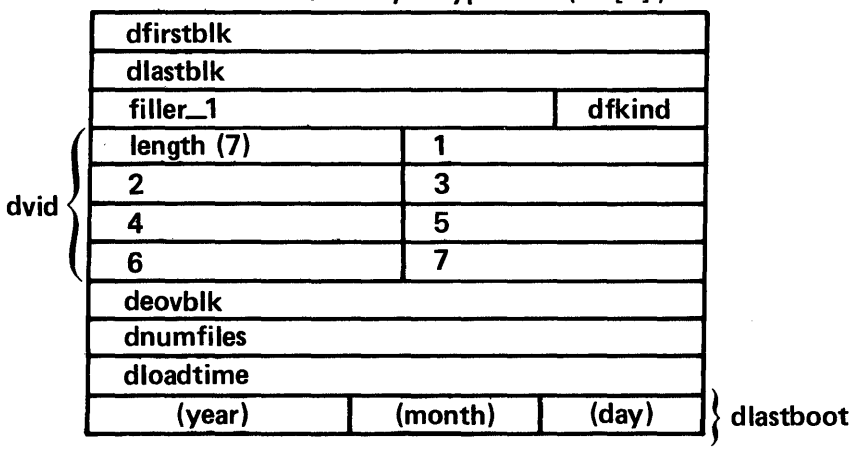
entry. FSoftBuf is TRUE if soft-buffered I/O is used: this is the case for all files on block-structured volumes, except untyped files.

If FSoftBuf is TRUE, then the last set of FIB fields are used: FNxtByte and FMaxByte are used for buffer handling, FBufChngd indicates that the buffer contents have been modified, and FBuffer is the buffer itself.

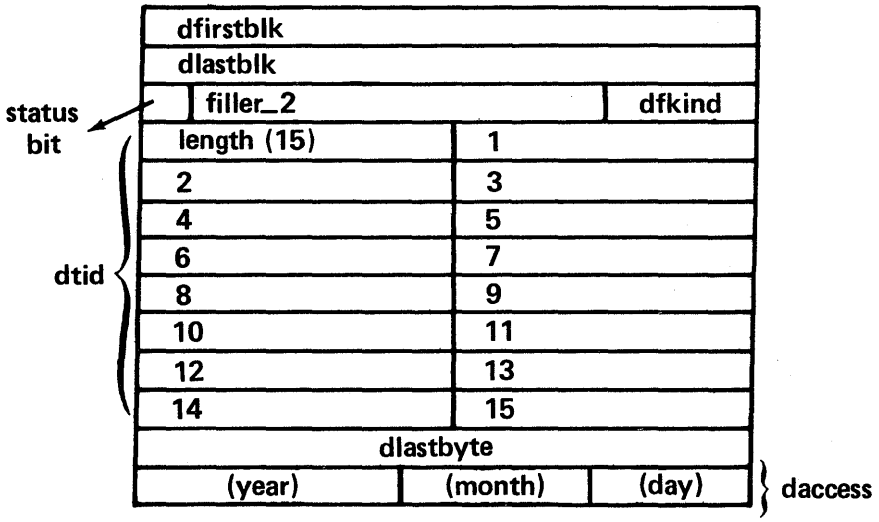
IV.3.2 Directories

Figure 6 illustrates the structure of a directory (as on a disk or other block-structured volume):

DIRENTRY RECORD (0)
for dfkind=securedir, untyped file (dir[0])



DIRENTRY RECORD (1-77)



DIRECTORY: array [0..77] of direntry;

DIRECTORY FORMAT

FIGURE 6

IV.3.3 Varieties of I/O

Record I/O

Record I/O applies to typed Pascal files, using the intrinsics GET and PUT.

Screen I/O

Screen I/O may be handled by the unit SCREENOPS, whose routines are described in the following section.

Input from the screen is accomplished by the procedure CHAR_DEV_GET, which uses SC_CHECK_CHAR (in SCREENOPS) and SYSCOM^.MISCINFO to determine whether any special handling needs to be done.

Output to the screen is accomplished by a simple UNITWRITE.

Block I/O

Block I/O applies to untyped files. The routines BLOCKREAD and BLOCKWRITE are used. These are part of the System routine FBLOCKIO in the EXTRAIO unit.

When a file is accessed as an untyped file, all other file formatting is disabled.

Text I/O

A textfile is a file of ASCII characters. It has a 2-block header that contains formatting information used by the Screen Oriented Editor. When a textfile is used by a System program other than the Editor, the Operating System ignores this header. When a new textfile is created, the Operating System writes a 2-block header filled with NULs.

Textfiles always have an even number of blocks. Thus, the smallest possible textfile is 4 blocks long. Any extra space is padded with NULs.

Each record in a textfile is one line of text, terminated by a <return> character. If the first character in a textfile record is a DLE (decimal 16), it is interpreted as a blank-compression code: the following byte is (32+n), where n is the number of leading blanks. This blank-compression code is generated by the Editor (chiefly for the purpose of saving space in indented program source).

User programs typically handle textfiles with READ, READLN, WRITE, and WRITELN. GET and PUT may be used, and follow the Jensen & Wirth standard for files of type TEXT.

IV.4 Using the Screen Control Unit

This section describes how the Screen Control Unit may be used to perform various CRT-related tasks.

In order to use the Screen Control Unit, the programmer must have a copy of SCREENOPS.CODE with its INTERFACE section. The program must contain the following USES declaration:

```
USES {$U SCREENOPS.CODE} SCREENOPS;
```

IV.4.1 Routines within the Screen Control Unit

All of the routines described in this section may be called from your program. The text ports mentioned below are rectangular portions of the screen which may be defined to be of a different size than the real screen. At present, this feature is not fully utilized by all of the UCSD p-System. Where text ports are mentioned in this section, the entire screen should be understood to be the default.

PROCEDURE SC_Init;

Usually this procedure is only called by the Operating System. It initializes all the Screen Control tables and variables.

PROCEDURE SC_Clr_Cur_Line;

Erases the current line.

PROCEDURE SC_Clr_Line (Y: integer);

Clears line number Y within the current text port.

PROCEDURE SC_Clr_Screen;

Clears the screen.

PROCEDURE SC_Erase_to_EOL (X, Line: integer);

Starting at position (X, Line) within the current text port, everything to the end of the line is erased.

PROCEDURE SC_Eras_EOS (X, Line: integer);

Starting at position (X, Line) within the current text port, everything to the end of the screen is erased.

PROCEDURE SC_Left;

Moves the cursor one character to the left.

PROCEDURE SC_Right;

Moves the cursor one character to the right.

PROCEDURE SC_Up;

Moves the cursor one line up (in the same column).

PROCEDURE SC_Down;

Moves the cursor one line down.

PROCEDURE SC_Home;

Moves the cursor to position 0,0 within the current text port.

PROCEDURE SC_GOTO_XY (X, Line: integer);

Moves the cursor to position (X, Line).

FUNCTION SC_Find_X: integer;

Returns the column position of the cursor, relative to the current text port.

FUNCTION SC_Find_Y: integer;

Returns the row position of the cursor, relative to the current text port.

PROCEDURE SC_GetC_CH (VAR CH: char;
Return_on_Match: SC_ChSet);

SC_ChSet is a SET OF CHAR. This procedure repeatedly reads from the keyboard into CH until CH is equal to a member of Return_on_Match. The characters that you pass in this set should all be capitals (if they are alphabetic). If a lower case alphabetic character is recieved from the keyboard, it will be translated into upper case before it is compared to the characters within Return_on_Match.

FUNCTION SC_Space_Wait (Flush: Boolean): Boolean;

This function repeatedly reads from the keyboard until a <space> or the ALTMODE character is recieved. Before doing this it does a UNITCLEAR(1) if Flush is TRUE, and writes 'Type <space> to continue'. It returns TRUE if a <space> was not read.

FUNCTION SC_Prompt (Line: SC_Long_String;
X_Cursor, Y_Cursor, X_Pos, Where: integer;
Return_on_Match: SC_ChSet;
No_Char_Back: Boolean;
Break_Char: char): char;

This function displays the promptline, Line (SC_Long_String is a STRING [255]) in the current text port at (X_Pos, Where). The cursor is placed at (X_Cursor, Y_Cursor) after the prompt is printed. If X_Cursor is less than 0, the cursor is placed at the end of the prompt. If the prompt is too large to fit within the current text port, it is broken up into several pieces, but only at the Break_Char -- the user can view different parts of the prompt (cycling through them) by typing '?'. If a character is being prompted for, No_Char_Back should be sent as false. The keyboard is repeatedly read until the character read matches one within Return_on_Match.

FUNCTION SC_Check_Char (VAR Buf: SC_Window;
VAR Buf_Index,
Bytes_Left: integer): Boolean;

While a string is being read, this function may be called to see if a <backspace> or a <rubout> (DEL) has been read. If so, the input buffer is altered accordingly, and TRUE is returned. Buf is a line on the screen, Buf_Index indicates the cursor position within Buf, and Bytes_Left is the number of characters to the right of the cursor.

FUNCTION SC_Map_CRT_Command (VAR K_CH: char): SC_Key_Command;

SC_Key_Command is a type consisting of the following elements: (SC_Backspace_Key, SC_DC1_Key, SC_EOF_Key, SC_ETX_Key, SC_Escape_Key, SC_DEL_Key, SC_Up_Key, SC_Down_Key, SC_Left_Key, SC_Right_Key, SC_Not_Legal). The character passed is mapped into one of these elements.

FUNCTION SC_Scrn_Has (What: SC_Scrn_Command): Boolean;

SC_Scrn_Command is a type consisting of the following elements: (SC_Home, SC_Eras_S, SC_Eras_EOL, SC_Clear_Lne, SC_Clear_Scn, SC_Up_Cursor, SC_Down_Cursor, SC_Left_Cursor, SC_Right_Cursor). This function returns TRUE if the CRT has the control character passed.

FUNCTION SC_Has_Key (What: SC_Key_Command): Boolean;

SC_Key_Command consists of the elements listed in the description of SC_Map_CRT_Command above. This function returns true if the CRT generates the keyboard character passed.

PROCEDURE SC_Use_Info (Do_What: SC_Choice;
VAR T_Info: SC_Info_Type);

This function is used to pass information back and forth between a program and the Screen Control Unit. Do_What may either be SC_Get or SC_Give, and indicates whether the program is getting or giving information to the Screen Control Unit. T_Info contains various items to be either passed or received. The following information is contained within T_Info:

```
SC_Version: string;
SC_Date: PACKED RECORD
                Month: 0..12;
                Day: 0..31;
                Year: 0..99;
              END;
Spec_Char: SET OF char; (* Characters not to echo *)
Misc_Info: PACKED RECORD
                 Height, Width: 0..255;
                 Can_Break, Slow, XY_CRT, LC_CRT,
                 Can_UpScroll, Can_DownScroll: Boolean;
               END;
```

PROCEDURE SC_Use_Port (Do_What: SC_Choice;
VAR T_Port: SC_TX_Port);

This function works like SC_Use_Info above. The contents of T_Port are either passed or recieved from the Screen Control Unit. T_Port contains the following information:

```
Row, Col,
Height, Width,
Cur_X, Cur_Y : integer;
```

V. PROGRAM EXECUTION

The runtime environment for a user program is created by the Operating System's GETCMD unit. GETCMD starts the execution of System programs such as the Compiler, Linker, Filer, etc., and user programs named in the eX(ecute command. In all such cases, GETCMD calls the procedure ASSOCIATE, which finds the appropriate codefile, and then calls BUILDENV. BUILDENV constructs a program's runtime environment, as outlined in Chapter II.

BUILDENV recursively traverses the segments used by a program. For each segment, it initializes an E_Vec, E_Rec, and SIB. As each E_Rec is created, it is linked to a chain of segments that are already active: in this way, the Operating System can keep track of all active segments. Before BUILDENV initializes segment information, it checks to see if that segment is already active, and if it is, it does nothing but initialize the proper pointers. Otherwise, the E_Vec, E_Rec, and SIB must be created from information present in the codefile.

SEGREFs are segment reference assignments emitted by the Compiler. Segment numbers are local to a code segment. The main program is segment 2, and subsidiary segments, if any, are numbered starting from 3. Segment 1 is always the Operating System's KERNEL unit. SEGREFs are emitted for any principal segments used by the compilation (such as a used unit). At associate time, BUILDENV uses the SEGREF list to find the segments that the program uses.

All runtime errors detected by the System cause the current program to halt. The System displays an error message, and when the user types a <space>, the System is re-initialized. The program's runtime environment is lost.

When a program terminates, control returns to GETCMD, which waits for further instructions. When a program terminates normally, its environment is not lost, and the program can be re-started with the U(ser restart command. The System may or may not need to call BUILDENV again.

VI. APPENDICES

VI.A Glossary

This is intended as an aid to readers who are unfamiliar with many "buzz words" used in this document, and is not meant to be either comprehensive or precise.

ASSOCIATE TIME - That part of a program's lifetime in which the segments and their various references to each other are associated by the Operating System. This occurs when the program is prepared for execution.

BLANK-FILLED - All 8-bit bytes within the specified region are filled with blanks (ASCII 32).

BLOCK - An area of memory (usually on a disk) with a fixed size of 512 contiguous 8-bit bytes (256 contiguous 16 bit-words).

BLOCK BOUNDARY - Byte zero of any block.

BYTE POINTER - A byte address (as opposed to a word address).

BYTE SEX - Some processors address 16-bit words with the most-significant-byte first, others with the least-significant-byte first. Byte sex refers to this difference in addressing; two machines with different addressing styles are said to have different (or opposite) byte sex.

COMPILATION UNIT - A program or portion of a program that can be compiled by itself: in other words, a program or a UNIT.

COMPILE TIME - That part of a program's lifetime in which it is being compiled (or assembled).

CONCURRENCY - The execution of two or more tasks or processes in parallel, i.e. at the same time. Synonymous with multitasking.

DYNAMIC - Information which changes during program execution (or is not known before runtime).

FILLER - A field in a data structure that is at present unused. If this area is described as "reserved for future use" then it usually should be zero-filled. This avoids confusion when future versions of the System make use of filler space.

INTER-SEGMENT - The data (or program) in question occupies more than one segment, or contains pointers to another segment.

LINK TIME - That part of a program's lifetime in which it is being operated on by the Linker.

MULTIPROGRAMMING - An environment that supports more than one user, where each user can perform multitasking. (The p-System does not support multiprogramming.)

MULTITASKING - The execution of two or more tasks in parallel, i.e. at the same time. A task is a PROCESS from the user's point of view; from the System's point of view it might be a program. (The p-System does support multitasking.)

MULTIWORD - Some positive integral number of words.

NATIVE CODE - Assembled code for some physical (as opposed to ideal) processor. Also called machine code or (sometimes) hard code.

ONE'S COMPLEMENT - All bits in the designated field are flipped.

P-CODE - Assembled code for an ideal processor. P-code stands for "pseudo-code." The p-System Interpreter implements a "pseudo-machine."

POSTPROCESSOR - A program which is executed after the completion of some other program, and uses as input the output of that previous program. A postprocessor that creates output which can be used by still another program is often called a "filter."

PRINCIPAL SEGMENT - A segment that has a segment reference list, i.e., a segment with a SEG_TYPE of PROG_SEG or UNIT_SEG. Corresponds to the outer segment of any compilation unit. UNITs, FORTRAN programs, and the outermost block of a Pascal program are all principal segments.

RECURSION - see RECURSION.

RELOCATABLE - A portion of object code that can be moved to different locations in memory without changing its meaning. P-code is relocatable. Native code may or may not be.

RUNTIME - That part of a program's lifetime in which it is being executed (or "run").

SELF-MODIFYING - Code which overwrites or modifies itself during execution, thus changing its meaning. This is not recommended!

SEG-RELATIVE - The address of an object is specified as an offset from the beginning of the code segment in which it resides.

STATIC - Information which does not change throughout program execution (it is known before runtime).

SUBSIDIARY SEGMENT - A segment that has no segment reference list, i.e., a segment with a SEG_TYPE of PROC_SEG or SEPRT_SEG. Corresponds to the object code of any segment whose source text is not separately compilable. Pascal segment procedures and segments produced by the UCSD Adaptable Assembler are subsidiary segments.

TOS - Short for "top of stack." The object that is on the top of the P-machine stack (which is the object that was most recently pushed).

UPWARD COMPATIBILITY - Code that runs on current versions of a system will run on future versions of that system. A more limited and more easily obtained version of upward compatibility requires source code to be recompiled on new versions, but ensures that it will run when recompiled.

WORD - 16 bits aligned on an even byte-address boundary. The byte which is most significant is determined by the byte sex of the machine for which it was generated.

WORD POINTER - A word address (as opposed to a byte address). The address of a word must be even.

ZERO-FILLED - A field of data that contains nothing but zeroes (all bits must be 0).

VI.B P-Codes

SLDC	0..31	Short Load Word Constant
LDCN	152	Load Constant NIL
LDCB	128	Load Constant Byte
LDCI	129	Load Constant Word
LCO	130	Load Contant Offset
SLDL1	32	Short Load Local Word
...	...	
SLDL16	47	
LDL	135	Load Local Word
SLLA1	96	Short Load Local Address
...	...	
SLLA8	103	
LLA	132	Load Local Address
SSTL1	104	Short Store Local Word
...	...	
SSTL8	111	
STL	164	Store Local Word
SLDO1	48	Short Load Global Word
...	...	
SLDO16	63	
LDO	133	Load Global Word
LAO	134	Load Global Address
SRO	165	Store Global Word
SLOD1	173	Short Load Intermediate Word
SLOD2	174	
LOD	137	Load Intermediate Word
LDA	136	Load Intermediate Address
STR	166	Store Intermediate Word
LDE	154	Load Extended Word

LAE	155	Load Extended Address
STE	217	Store Extended Word
SIND0	120	Short Index and Load Word
...	...	
SIND7	127	
IND	230	Index and Load Word
STO	196	Store Indirect
LDC	131	Load Multiple Word Constant
LDM	208	Load Multiple Words
STM	142	Store Multiple Words
LDCRL	242	Load Real Constant
LDRD	243	Load Real
STRL	244	Store Real
CAP	171	Copy Array Parameter
CSP	172	Copy String Parameter
LDB	167	Load Byte
STB	200	Store Byte
LDP	201	Load a Packed Field
STP	202	Store into a Packed Field
MOV	197	Move
INC	231	Increment Field Pointer
IXA	215	Index Array
IXP	216	Index Packed Array
LAND	161	Logical And
LOR	160	Logical Or
LNOT	229	Logical Not
BNOT	159	Boolean Not
LEUSW	180	Less Than or Equal Unsigned
GEUSW	181	Greater Than or Equal Unsigned

ABI	224	Absolute Value Integer
NGI	225	Negate Integer
INCI	237	Increment Integer
DECI	238	Decrement Integer
ADI	162	Add Integers
SBI	163	Subtract Integers
MPI	140	Multiply Integers
DVI	141	Divide Integers
MODI	143	Modulo Integers
CHK	203	Check Subrange Bounds
EQUI	176	Equal Integer
NEQI	177	Not Equal Integer
LEQI	178	Less Than or Equal Integer
GEQI	179	Greater Than or Equal Integer
FLT	204	Float Top-of-Stack
TNC	190	Truncate Real
RND	191	Round Real
ABR	227	Absolute Value of Real
NGR	228	Negate Real
ADR	192	Add Reals
SBR	193	Subtract Reals
MPR	194	Multiply Reals
DVR	195	Divide Reals
EQREAL	205	Equal Real
LEREAL	206	Less Than or Equal Real
GEREAL	207	Greater Than or Equal Real
ADJ	199	Adjust Set
SRS	188	Build a Subrange Set
INN	218	Set Membership
UNI	219	Set Union
INT	220	Set Intersection
DIF	221	Set Difference
EQPWR	182	Equal Set
LEPWR	183	Less Than or Equal Set
GEPWR	184	Greater Than or Equal Set
EQBYT	185	Equal Byte Array
LEBYT	186	Less Than or Equal Byte Array
GEBYT	187	Greater Than or Equal Byte Array

UJP	138	Unconditional Jump
FJP	212	False Jump
TJP	241	True Jump
EFJ	210	Equal False Jump
NFJ	211	Not Equal False Jump
JPL	139	Unconditional Long Jump
FJPL	213	False Long Jump
XJP	214	Case Jump
CPL	144	Call Local Procedure
CPG	145	Call Global Procedure
SCPI1	239	Short Call Intermediate Procedure
SCPI2	240	
CPI	146	Call Intermediate Procedure
CXL	147	Call Local External Procedure
SCXG1	112	Short Call External Global Procedure
...	...	
SCXG8	119	
CXG	148	Call Global External Procedure
CXI	149	Call Intermediate External Procedure
CPF	151	Call Formal Procedure
RPU	150	Return from Procedure
LSL	153	Load Static Link
BPT	158	Breakpoint
SIGNAL	222	Signal
WAIT	223	Wait
EQSTR	232	Equal String
LESTR	233	Less Than or Equal String
GESTR	234	Greater Than or Equal String
ASTR	235	Assign String
CSTR	236	Check String Index
LPR	157	Load Processor Register
SPR	209	Store Processor Register
DUP1	226	Duplicate One Word
DUPR	198	Duplicate Real

SWAP	189	Swap
NOP	156	No Operation
NAT	168	Native Code
NAT-INFO	169	Native Code Information
RESERVE1	250	reserved
...	...	
RESERVE6	255	

VI.C Appendix C -- American Standard Code for Information Interchange

0	000	00	NUL	32	040	20	SP	64	100	40	@	96	140	60	
1	001	01	SOH	33	041	21	!	65	101	41	A	97	141	61	a
2	002	02	STX	34	042	22	"	66	102	42	B	98	142	62	b
3	003	03	ETX	35	043	23	#	67	103	43	C	99	143	63	c
4	004	04	EOT	36	044	24	$	68	104	44	D	100	144	64	d
5	005	05	ENQ	37	045	25	%	69	105	45	E	101	145	65	e
6	006	06	ACK	38	046	26	&	70	106	46	F	102	146	66	f
7	007	07	BEL	39	047	27	´	71	107	47	G	103	147	67	g
8	010	08	BS	40	050	28	(	72	110	48	H	104	150	68	h
9	011	09	HT	41	051	29	)	73	111	49	I	105	151	69	i
10	012	0A	LF	42	052	2A	*	74	112	4A	J	106	152	6A	j
11	013	0B	VT	43	053	2B	+	75	113	4B	K	107	153	6B	k
12	014	0C	FF	44	054	2C	,	76	114	4C	L	108	154	6C	l
13	015	0D	CR	45	055	2D	-	77	115	4D	M	109	155	6D	m
14	016	0E	SO	46	056	2E	.	78	116	4E	N	110	156	6E	n
15	017	0F	SI	47	057	2F	/	79	117	4F	O	111	157	6F	o
16	020	10	DLE	48	060	30	0	80	120	50	P	112	160	70	p
17	021	11	DC1	49	061	31	1	81	121	51	Q	113	161	71	q
18	022	12	DC2	50	062	32	2	82	122	52	R	114	162	72	r
19	023	13	DC3	51	063	33	3	83	123	53	S	115	163	73	s
20	024	14	DC4	52	064	34	4	84	124	54	T	116	164	74	t
21	025	15	NAK	53	065	35	5	85	125	55	U	117	165	75	u
22	026	16	SYN	54	066	36	6	86	126	56	V	118	166	76	v
23	027	17	ETB	55	067	37	7	87	127	57	W	119	167	77	w
24	030	18	CAN	56	070	38	8	88	130	58	X	120	170	78	x
25	031	19	EM	57	071	39	9	89	131	59	Y	121	171	79	y
26	032	1A	SUB	58	072	3A	:	90	132	5A	Z	122	172	7A	z
27	033	1B	ESC	59	073	3B	;	91	133	5B	[	123	173	7B	{
28	034	1C	FS	60	074	3C	<	92	134	5C	\	124	174	7C	|
29	035	1D	GS	61	075	3D	=	93	135	5D	]	125	175	7D	}
30	036	1E	RS	62	076	3E	>	94	136	5E	^	126	176	7E	~
31	037	1F	US	63	077	3F	?	95	137	5F	_	127	177	7F	DEL

NOTES

NOTES

NOTES