P R O M A L

(PROgrammer's Micro Application Language)

LANGUAGE MANUAL

A PROMAL LANGUAGE DESCRIPTION AND REFERENCE

For Apple II and Commodore 64 Computers

SYSTEMS MANAGEMENT ASSOCIATES, INC.
3325 Executive Drive
Raleigh, North Carolina 27609
(919) 878-3600

Rev. C - Sep. 1986

PROMAL LANGUAGE MANUAL

CHAPTER 1: INTRODUCTION

This manual will introduce you to the PROMAL programming language, which we think will find to be the most enjoyable and creative language available for your computer. This manual will guide you step by step through a description of the PROMAL language, with examples along the way. It assumes that you already have a working knowledge of BASIC (or some other high-level language) and elementary computer concepts such as bits, bytes and memory addresses. Comparisons are often given between PROMAL programs and the equivalent BASIC program, so that you may draw on your previous experience.

You should study the manual carefully, because PROMAL is significantly different from BASIC. As a BASIC programmer, you may find some aspects of PROMAL a little strange at first reading. But if you give it a fair trial, we're sure you will soon want to do all of your programming in PROMAL.

We assume that you have already read the companion manual MEET PROMAL!, which provides a "hands-on" introduction to the PROMAL system as a whole. You will find the operational aspects of the PROMAL EXECUTIVE, EDITOR, COMPILER, and LIBRARY described in detail in the PROMAL USER'S MANUAL. This manual explains the heart of the PROMAL system, the PROMAL programming language, which you can use to create your own programs. PROMAL is especially well suited for:

* Text processing applications
* Scientific and Engineering applications
* Educational applications
* Interactive programming
* Small business programming
* Compilers, assemblers, editors or system software

Not only do PROMAL programs for these applications often run 20 to 100 times (or more) faster than BASIC, they are actually easier to program than BASIC! Programs that used to take weeks or months of assembly-language drudgery can now be quickly developed with PROMAL instead.

WHY USE PROMAL?

Why should you learn PROMAL when you already know BASIC? Why should you learn PROMAL instead of one of the older, structured languages such as PASCAL or C?

Perhaps the most important reason is that PROMAL is in many respects the **most structured** language available, because the PROMAL compiler reads indentation as part of the syntax of the language. As you will see, the fact that indentation **always** shows the true structure of your program will make your programs easier to write, and more importantly, **easier to maintain.**

Another important consideration is that PROMAL is the **only** compiled language available from a single vendor for the IBM PC, Apple II, and Commodore 64 - the three biggest-selling machines in history. If you plan to develop commercial software, or just think you might change computers some day, this will be important to you. And PROMAL gives you top performance on all machines.

CHAPTER 2: PROMAL PROGRAMMING LANGUAGE OVERVIEW

A PROMAL **source program** is a file of text composed of lines, normally created using the PROMAL EDITOR. Each line is called a **statement**. A program has a certain organization to it, which is similar to a recipe. A program starts by declaring its name, and then identifies what "ingredients" are used in the program. Ingredients are identified by the kind of data to be used, the name of the data, and the quantity required. The list of ingredients is called the **declarations** part of the program. After the declarations part of the program comes the actual instructions which tell how to manipulate the data. For clarity, the instructions are usually broken up into a number of **procedures**, each of which has a name suggestive of its function.

For example, consider the actual PROMAL program in the right column, below, and observe the similarities with the recipe at the left.

A Kitchen Recipe		A PROMAL Program
FRIED CHICKEN:	<-- Your recipe name -->	PROGRAM LONGESTLINE INCLUDE LIBRARY
2 lb. Chicken pcs. 1/4 lb. shortening	<-- Main ingredients and amounts needed -->	BYTE LINE [81] ;current line BYTE LONGEST ;longest length WORD IFILE ;input file
SEASONED FLOUR:	<-- Sub Procedure Name -->	PROC SIZELINE
1/2 cup flour 1 tsp. salt 1/4 tsp. paprika	<-- Ingredients for sub-procedure -->	BYTE LENGTH ;cur line length
Mix all ingredients for seasoned flour.	<-- Instructions for sub-procedure -->	BEGIN ; Procedure LENGTH=LENSTR(LINE) IF LENGTH > LONGEST LONGEST = LENGTH END
Heat oven to 450. Melt shortening.	<-- Main Process setup -->	BEGIN ; Main Program IFILE=OPEN("TESTFILE.T")
Coat chicken with seasoned flour.		LONGEST=0
Cook about 45 min. until golden brown.	<-- Loop waiting for a condition -->	WHILE GETLF(IFILE,LINE) SIZELINE ;test if biggest
Serve with gravy.	<-- How to serve up the results -->	OUTPUT "Longest = #I",LONGEST END

The program above reads a file and prints the length of the longest line in the file. It is useful to get the feel for what a complete (although very simple) PROMAL program looks like before delving into the details.

If you have programmed in BASIC, probably the first thing you will notice about the program above is that there are no line numbers. Line numbers are not used and not needed in PROMAL programs. You will soon discover that this makes PROMAL programs much easier to write and understand. **PROMAL statements normally start in column 1.** Let's look briefly at the statements that compose the program, just to get the general idea of what they do.

```
PROGRAM LONGESTLINE
```

This line starts the program. The name LONGESTLINE is the command you will eventually type from the EXECUTIVE when you want to run this program.

```
INCLUDE LIBRARY
```

This line tells the PROMAL COMPILER to include the definitions of all the built-in library routines (which are needed for input-output, etc.). You will normally have this statement near the start of every program.

```
BYTE LINE [81]      ;current line
```

This line declares that you will be using a variable called LINE which is an array of 81 BYTEs. One byte can store one character, so this array can hold an 80 character line plus a line terminator. The ";" indicates the start of a comment. The rest of a line after a ";" is ignored by the compiler.

```
BYTE LONGEST        ;longest length
```

This line declares a simple (non-array) variable called LONGEST. It is used to hold the number of characters in the longest line. In PROMAL, unlike BASIC, **all variables must be declared before they are used** (not just arrays).

```
WORD IFILE          ;input file
```

This line declares a variable of type WORD. Later you will learn that a WORD is usually used to hold an address. In this case, the address will be a "**file handle**" for the file of text to be read. You can think of a file handle as just a number identifying a particular file.

```
PROC SIZELINE
```

This line begins the definition of a PROMAL procedure, which is similar to a BASIC subroutine. It has been given the name SIZELINE by the programmer.

```
BYTE LENGTH         ;cur line length
```

This is a variable of type BYTE which is only used within the procedure. This will be explained further later on.

```
BEGIN
```

This line signals the beginning of actual executable instructions within the procedure.

```
LENGTH=LENSTR(LINE)
```


This is an **assignment statement** which uses the built in function LENSTR to determine the number of characters currently in the array LINE, and install that number into LENGTH.

```
IF LENGTH > LONGEST
  LONGEST = LENGTH
```

The IF statement tests if the current length is larger than the largest line length so far, and if so, updates the value of LONGEST to LENGTH. Otherwise, the second statement is simply skipped over.

```
END
```

The END statement indicates the end of the procedure (like a BASIC RETURN statement).

```
BEGIN
```

Since there are no more PROCEDURES, the BEGIN signals the beginning of the main program. In PROMAL, **the main program always comes last.** This may seem a little strange at first, but follows from the general rule that everything, including all subroutines, must be defined before being used.

```
IFILE=OPEN("TESTFILE.T")
```

This is actually the first statement which would be executed in the program. It tells the computer to "OPEN" the file called "TESTFILE.T" for reading, and installs the file handle into IFILE. Any subsequent input references to IFILE will read from "TESTFILE.T".

```
LONGEST=0
```

This statement works just like its BASIC equivalent, and initializes the value of LONGEST to 0.

```
WHILE GETLF(IFILE, LINE)
  SIZELINE   ;test if biggest
```

These two statements comprise a **loop.** The WHILE statement attempts to read one line from the file into the LINE array. If successful, the SIZELINE subroutine is called, and the WHILE statement is repeated again. This process is repeated until end-of-file is reached, in which case the GETLF function is unsuccessful, and control passes through without executing SIZELINE again. In PROMAL, **subroutines are called by merely typing their names**; no GOSUB is needed.

```
OUTPUT "Longest = #I", LONGEST
```

This statement is similar to a BASIC PRINT statement. It would show an answer on the screen, for instance:

```
Longest = 67
```


assuming the longest line was 67 characters. The "#I" in the OUTPUT statement is a code which tells the computer **how** to format the answer; in this case, telling it to print it as an integer number.

```
END
```

This line terminates the program.

It is not important to understand the details of the program at this point, but just to get the general idea of what a program looks like. The following sections will explain the rules for writing a program in detail.

CHAPTER 3: ELEMENTS OF THE PROMAL LANGUAGE

In the last chapter we got a quick "top down" view of how a simple complete PROMAL program looks. In this chapter, we will take a "bottom up" look at some of the elements of the PROMAL language in greater detail. Then we will learn how to combine these elements into statements and programs.

VOCABULARY

The following **reserved** words have special meaning in PROMAL programs, and form the basic vocabulary of the language:

AND	CON	EXT	INT	OWN	TO
ARG	CHOOSE	FALSE	LIST	PROC	TRUE
ASM	DATA	FOR	NEXT	PROGRAM	UNTIL
AT	END	FUNC	NOT	REAL	WHILE
BYTE	ELSE	IF	NOTHING	REFUGE	WORD
BEGIN	ESCAPE	IMPORT	OR	REPEAT	XOR
BREAK	EXPORT	INCLUDE	OVERLAY	RETURN	

The reserved words may be spelled with either upper or lower case letters, or a mix of both. Therefore BEGIN, begin, Begin, and BegIn are equivalent. These reserved words are also sometimes called **keywords**. In PROMAL (unlike BASIC) you **must** separate keywords from each other or from other names with blanks or other punctuation. This helps make programs readable and does not impose any speed or memory size penalty on the program. As a practical matter you may also wish to consider the standard library routine names listed at the start of the LIBRARY MANUAL as reserved words, although this is not strictly true because you do not have to use the LIBRARY. You may even change the names in the LIBRARY, although this is definitely not recommended (for reasons of consistency with other programmers).

NAMES

Names are used to identify constants, variables, data, functions, procedures and programs in PROMAL. You may choose names (also called identifiers) as you wish, following these rules:

1. A name may be from one to 31 characters in length.
2. The first character must be alphabetic.
3. The remaining characters must be alphabetic, numeric, or the underline character "_" (left-pointing arrow on the Commodore 64, which has no underline key).
4. Either upper or lower case alphabetic characters may be used. Both are considered equivalent. The PROMAL compiler treats all alphabetic characters as upper case in identifiers. Therefore XYZ and xYz are considered the same name.
5. A name may **not** duplicate one of the reserved words in the basic vocabulary above.

Unlike Commodore or Apple BASIC, which only looks at the first two characters of a name, **all** characters of a name are "significant" in PROMAL. For example, EXTRAPOLATEX1 and EXTRAPOLATEY1 will be considered as two different variables, even though the first eleven characters are identical.

Similarly, TON is a legal name, even though it contains the reserved word TO (which would make it illegal in BASIC). After compilation, programs using long names **do not** use any more memory or execute any slower than programs with short names, so you should select names which are meaningful. For example, AMOUNT_DUE is probably a better choice for a name than AD.

Some examples of legal names are:

A	ZERO	OldInventory	X_Y_Data
aBc	for4	S___	D200
C4	d2000	DearJohn	ET

Some examples of ILLEGAL names (for the reasons indicated) are:

B-4	(second character is not alphanumeric or _)
3D	(first character is not alphabetic)
LIST	(duplicates a reserved word)

Again, remember that you cannot run variable names and PROMAL keywords together the way you can in BASIC. For example,

```
IFID=MEORID=YOU
```

may be an acceptable way to start an IF statement in BASIC, but in PROMAL you would have to write:

```
IF ID=ME OR ID=YOU
```

instead.

DATA TYPES

A data type refers to the **kind** of data that a program can manipulate. PROMAL has four built-in data types, three of which are very simple and quite close to the data types that are used in machine language. This primitive simplicity greatly contributes to PROMAL's speed of execution. The four types are:

Type	Meaning
BYTE	An unsigned integer number between 0 and 255, **or** a single ASCII character, **or** the Boolean value TRUE or FALSE.
WORD	An unsigned integer number between 0 and 65,535.
INT	A signed integer number between -32,767 and +32,767.
REAL	A floating point number between approximately 1.E-37 and 1.E+37. Similar to BASIC's standard numeric data type.

The data type BYTE is a distinguishing characteristic of the PROMAL language. This is very important, because byte variables can be manipulated very rapidly and are frequently needed for the types of applications PROMAL is intended for. As the name implies, a BYTE variable occupies only one byte of memory. WORD and INT (integer) variables each occupy two bytes (16 bits). In memory, the low order 8 bits are stored in the first byte and the high order 8 bits are stored in the next higher address. This is the conventional way to store addresses for the 6502 family processor used in the Apple II and Commodore 64. BYTEs, WORDs, and INTegers **may not** have any fractional part; thus 11 and 12 are okay but 11.5 is not.

REAL variables occupy 6 bytes of memory each. They are similar to the numeric data type used in BASIC (5 bytes each), but are accurate to 11 significant digits instead of 9 significant digits like BASIC. REAL variables have the greatest flexibility because they can store very large and very small numbers, including a decimal fraction. However, they are manipulated **much** more slowly than the other data types (but not as slowly as in BASIC), and therefore should be used with discretion. PROMAL also provides facilities for formatted output, so that you can precisely control the number of digits and number of decimal places printed for REAL output.

BASIC programmers may note the absence of character strings as a standard data type. But **PROMAL can handle strings very well as an array of type BYTE.** String handling is not difficult and will be discussed in detail later.

LITERAL NUMBERS, CHARACTERS, AND STRINGS

Numbers may be written in the usual way. A number written without a decimal point is assumed to be of type BYTE, INT, or WORD, depending on its size and sign. Unsigned values less than 256 are assumed to be BYTE. Larger values are type WORD. Any negative number is assumed to be INT.

Examples of legal BYTE, INT or WORD type numbers are:

```
0       1       137       22340   65535    -78
```

The following are **illegal** as BYTE, INT or WORD type numbers (for the reasons indicated):

```
1,333     ; (Cannot have a comma)
120.6     ; (Cannot have a decimal point - OK for REAL numbers)
65539     ; (Out of range - must be less than 65536)
```

Literal numbers may also be specified in **hexadecimal**, by using a "**$**" prefix. Hexadecimal (base 16) numbers are often more convenient for specifying memory addresses or bit patterns. For example, it is easier to remember that the Commodore 64 VIC-2 video chip is at address $D000 than at its decimal equivalent, 53248. If you are not familiar with hexadecimal numbers, you may wish to consult your computer's reference manual. Examples of legal hex numbers are:

```
$0       $a       $2BD       $FFFF       $0012
```


The following are ILLEGAL hex numbers (for the reasons indicated):

```
$1B3.4    ; (Cannot have decimal point in hex number)
FFFF      ; (No $ prefix)
$102B0    ; (Out of range - must be less than $10000).
```

REAL numbers **must be specified with a decimal point**. In BASIC, you can write a real number without a decimal point, but not in PROMAL. If you forget to write the decimal point, PROMAL may accept the number as a valid byte, integer, or word value, without an error indication. However, if you pass this value to a function or procedure that is expecting a REAL value (such as OUTPUT using a #R format), the procedure or function will try to interpret your result as REAL, resulting in a garbage value. Therefore you should **always** be careful to specify a decimal point for a real constant. You may also write REAL literal numbers using the "E" format scientific notation, as in BASIC. Examples of legal REAL numbers are:

```
0.      .0      123.      3.1415926535      -.0000007     56.00
1.2e11  -.003E-10
```

The following are **illegal** real numbers (for the reasons indicated):

```
76000     ; (no decimal point - will be treated as out-of-range integer)
2,333.00  ; (cannot have a comma)
1.21E+50  ; (value out of range; must be less than 1E+37)
```

In specifying literal numbers, you should keep in mind the size limits for the various data types. Only REAL numbers may be larger than 65535 decimal ($FFFF) and may have a fractional part.

PROMAL programs often need to specify single ASCII characters for some operation (**Appendix A** contains a summary of the ASCII character set). To specify a single literal character, enclose it in **single** quotes, for example:

```
'a'     'Q'     '4'     '*'     ' '
```

The PROMAL compiler will substitute the numeric ASCII value of the character. For example, writing 'A' is equivalent to 65 or $41 (see table, **Appendix A**). If you need to show the single quote character itself (') as a literal character, you must double it ('''').

A **literal string** is a group of characters enclosed in **double quotes**. Examples of literal strings are:

```
"A"     "Hello There!"     "26"     "+-*/"     ""
```

When the compiler encounters a literal string in your source program, it generates the ASCII representation of the string, followed by a $00 byte terminator, in your object program. **Literal strings use one byte per character, plus a string terminator which is always a $00 byte.** Therefore the string "A" occupies **two** bytes of memory and the string "Hello There!" occupies 13 bytes in your compiled program. The last example ("") above is called the null string, and contains no characters. This is not the same as a string containing a blank. A blank is a character and occupies space in memory. The 0-byte terminator is always generated automatically by the compiler. A literal string

may contain 0 or more characters. As we will see later, character strings may be up to 254 bytes long, but as a practical matter, a literal character string is limited to the number of characters which will fit on a single line.

The most common use of a literal string is to output a message, which is just as easy as a BASIC PRINT statement:

```
PUT "Hello world!"
```

PUT is actually a built in procedure which should be followed by the address of a string which is to be printed. So for example when the PROMAL compiler sees:

```
PUT "Hello world!"
```

it actually generates a string for you in memory (terminated by a 0 byte), and generates a call to the PUT procedure, passing PUT the address of the string to print. The compiler uses the **address of the first character of the string as the "value" of the string.** If you don't understand this completely yet, don't worry about it. The importance and usefulness of this will be explained more fully later.

If you need to include the double-quote character itself (") in a literal string, it should be doubled. For example:

```
PUT  "She said, ""I'll be back."""
```

will actually cause the program to print:

```
She said, "I'll be back."
```

You can also embed unprintable codes (such as ASCII control characters or special characters, such as characters to trigger color changes on the Commodore 64) in a string by writing the character **** (**£** pounds sterling key on the Commodore 64, which has no backslash key) followed by **exactly two hex digits** giving the desired character code. For example:

```
PUT "New line \0Dstarting here"
```

will embed a $0D (ASCII carriage return) in mid-string. If you wish to include the **** itself in a string, you should double it, in the same manner as the quote. A particularly useful pair of embedded codes on the C-64 are \12 and \92, which start and stop reverse video output, respectively. On the Apple II, \0F and \0E will enable and disable reverse video.

It is important to remember the difference between a character and a string. A literal character is always a **single** character enclosed in **single quotes.** A literal **string** is zero or more characters enclosed in **double quotes.** This means that 'A' and "A" do not have the same meaning to the PROMAL compiler. 'A' occupies a single byte and has the value 65. "A" occupies two bytes, 65 followed by 0, and has the "value" of whatever address the PROMAL compiler assigns to the first character.

Note that you may use PUT only to print characters and strings on the screen. If you need to print the **value** of a variable, you will need to use OUTPUT instead, which is described later.

VARIABLES

PROMAL variables are used to hold values, in much the same way as BASIC variables. However, as we have already seen, PROMAL variables may have long names. PROMAL variables also have a "type" associated with them, which must be BYTE, INT (integer), WORD, or REAL. BASIC variables also have a type, but the type is implied by the name of the variable itself. For example, a % suffix in BASIC indicates an integer type variable and a $ suffix indicates a string type variable. When using PROMAL, however, you must **declare** the type and name of every variable explicitly instead. No special suffixes are used.

DECLARING VARIABLES

In PROMAL programs, all variables **must** be declared before they are used. A variable declaration tells the PROMAL compiler the name of the variable, what type of variable it is, and how much space it will need. A sample variable declaration might be:

```
INT SCORE ;     Game score
```

This declares that you will be using a variable with the name SCORE, and that it will be of type INT. Therefore the variable SCORE will be able to take on signed values between -32767 and +32767. **Only one variable may be declared on a line.** It is considered good programming practice to put a comment after the variable name explaining what it is used for, as is shown above.

In BASIC, you did not have to declare variables (except for the DIM statement, which is a declaration for arrays). Having to list all your variables at the top of the program may seem like a nuisance at first, but you will come to appreciate the value of it. When you pick up a PROMAL program, you can quickly find out the names of all the variables in the program and what they are used for by reading the declarations. If you want to add a new variable, you won't have to search the whole program to make sure the name you choose has not already been used for something else; you just look at the declarations. If you forget to declare a variable before you use it, the PROMAL compiler will flag the variable name with an error message saying "UNDEFINED" when you try to use it.

There is an even more important reason why variables need to be declared. This is best illustrated with an example from BASIC. Suppose you decide to modify an existing BASIC program which uses a variable called XO. You add a few lines to the program, using the variable XO, but the program mysteriously doesn't work. Eventually you discover that the reason is that you typed XO (X-"letter O") but the original variable was XO (X-"zero"). In this case, BASIC automatically creates a new variable, initialized with a value of zero, instead of using the existing variable XO which you really wanted. In PROMAL, you would not have this problem because the compiler would flag XO as UNDEFINED. As a matter of historical interest, one of the NASA space program's planetary probes was lost due to a navigational error caused by precisely this kind of bug in a FORTRAN program (like BASIC, you don't have to declare variables in FORTRAN).

As you learn the PROMAL language, you will find other instances like this where PROMAL imposes a certain structure on your programming to help improve the clarity and style of the program.

Unlike BASIC variables, which are automatically initialized to 0, PROMAL does **not** provide any initialization of variables. This means that you **cannot assume anything about the value of a variable until you have assigned some value to it.** The initial value of a variable is simply whatever happened to be "left over" in the memory location PROMAL assigns to the variable. **Chapter 7** describes a convenient method for initializing all variables to zero with a single statement.

CONSTANT DEFINITION

A constant is a name given to a numeric value which will not change throughout the program. A constant must be defined with a CON statement before it can be used. For example:

```
CON LF=10    ; ASCII linefeed character
```

defines the symbol LF to be 10. After this, anytime the PROMAL compiler encounters the name LF, it will substitute the value 10 instead. There are two differences between constants and variables. First, the value of the constant is permanent and is associated with the constant name at compile time. Second, no memory is set aside to save the value of the constant in the data area. Instead, any time the constant is referenced, the compiler generates the value of the constant (in the same manner as a literal constant) in the executable code of the program. Only one constant can be defined on a line. Again, it is considered good practice to add a comment to a constant definition explaining what the constant is. If you are an assembly language programmer, you may recognize that a PROMAL constant is equivalent to an assembly language "equate". You may also define the type of the constant explicitly, for example:

```
CON WORD STARTLOC=$40
```

defines STARTLOC to be of type WORD with a value of 40 hexadecimal. If you don't specify the type explicitly, PROMAL will assume type BYTE if the value is less than $100, INT if it has a minus sign, and type WORD otherwise. Later we will learn more about constant definitions, after we learn about operators and expressions.

You may **not** declare a REAL constant. Instead, you should use a DATA statement if you wish to associate a name with a permanent value of type REAL. Disallowing REAL constants saves memory and reduces the complexity of the compiler.

ARRAY VARIABLES

PROMAL allows arrays of any of the four data types, with up to eight subscripts. Subscripts for the array are enclosed in square brackets "[ ]", not in parentheses like BASIC. This makes it easy to tell the difference between an array element and a function call (where parentheses are used to enclose the arguments, as will be discussed later). Like all other variables, arrays must be declared before they can be used. An array variable declaration is similar to a simple variable declaration, but is followed by the number of elements of the array desired. For example:

```
BYTE BUFFER [81]
```

declares an array of type BYTE which can hold 81 elements (BUFFER[0] through BUFFER[80]). It is important to observe that if you define an array as X[N], then **the last element is X[N-1], not X[N]**, because X[0] is the first element.

It is considered good programming practice to define a constant which controls the size of a subsequently declared array. This will usually make it easier to alter the program later. For example:

```
CON BUFSIZE = 100
BYTE BUFFER1 [BUFSIZE]
BYTE BUFFER2 [BUFSIZE]
...
```

You may not use a variable as the dimension for an array, however. This is because the PROMAL compiler allocates memory for the array at compile time; the size of the array must be known at compile time, not when the program is actually run.

The **subscripts for an array of any type must always be of type WORD.** If an array subscript evaluates to type BYTE, it will be "promoted" to WORD automatically. A subscript which evaluates to type REAL will cause the compiler to generate an error message. The maximum subscript which can be used is dependent on the amount of free memory. **When you refer to an array name without subscripts (or backets), the address of the array will be used.** The importance of this will be illustrated later.

It is possible to define both simple variables and arrays at specified locations in memory. For example you can define the screen memory as an array starting at $0400 (1024). This is kind of declaration is called an **external** variable, described in Chapter 6, "Interfacing".

CAUTION: When array elements are referenced, PROMAL **does not perform any bounds checking** (because of the adverse affect on performance). Therefore a sequence like:

```
WORD I
BYTE BUF[10]
BYTE LINE[8]
...
I=12
...
BUF[I]=0
```


will not produce an error message and will move the 0 into part of the LINE array instead of the BUF array as was intended. You should always take care to insure that array indices stay in bounds, or strange and invariably unpleasant results will occur!

Multiple dimension arrays have the subscripts separated by commas. For example:

```
BYTE SCREENIMAGE [80,25]
REAL STIFFNESS [10,20,3]
...
IF SCREENIMAGE [0,I] = ' '
  SCREENIMAGE [0,I] = SCREENIMAGE [1,I]
...
BEND = TORQUE * STIFFNESS[I,J,K]
...
```

The amount of memory required to store an array is the product of its declared dimensions times the size of each element. The SCREENIMAGE array above uses 2000 bytes, and the STIFFNESS array uses 3,600 bytes. Multiple dimension arrays are mapped into memory such that incrementing the first subscript will address elements that are physically adjacent in memory. Or, another way to visualize this is to say that SCREENIMAGE is organized as 25 groups of 80 bytes each (**not** 80 groups of 25 bytes each). Therefore if you wish to have a two dimensional array of text, the column subscript should come first and the row subscript second, as was done for SCREENIMAGE above. This is discussed further in **Chapter 7**.

DATA DEFINITION

A data definition is similar to a variable but has a predefined initial value which is determined at compile time. For example:

```
DATA REAL PI = 3.1415926535
```

defines a data item of type REAL which will be predefined to the value of PI.

Unlike constants, data definitions can define arrays as well as simple variables. The DATA definition is most frequently used to define a table of values which will not be changed by the program. The DATA definition looks similar to a variable declaration, except that it starts with the word DATA and is followed by an "=" and the desired value (or values). For example:

```
DATA BYTE MYTABLE [] = 23, 12, 8, 4, 2, 1, 0
```

This line defines an array called MYTABLE of type BYTE having 7 elements. Notice that the size of the array is not given in the brackets; the PROMAL compiler counts the number of elements for you. You must explicitly define the value of all elements. The first element of the array will be MYTABLE[0] and will be initialized to 23. The last element will be MYTABLE[6] and have the value 0.

You may **not** define multiple-dimension DATA arrays. Only a single dimension is permitted for DATA declarations.

You **may not** change the value to a data item with an assignment statement. If a data name appears on the left side of an assignment statement, the compiler will generate a "Variable Expected" error. It is **possible** to force the data items to be altered with an assignment statement to a variable array which overlaps the data items, but this is considered poor programming practice (and will also cause your program to be reloaded from disk if you try to re-execute it, because data items are included in the checksum which the EXECUTIVE uses to determine if a program has been corrupted).

If you wish to use a table of data items to set the initial values of a variable array which will subsequently be altered, the correct procedure is to copy the data array to another variable array (using the BLKMOV procedure, described later), and then alter the variable array.

The data definition is the one statement in PROMAL which can consist of multiple lines. In order to continue the data definition on additional lines, either the = sign or a comma should be the last character of the preceding line. For example:

```
DATA WORD LIST [] =
0,45,13,27,
0,46,13,28,
1,46,14,28,
1,47,14,29
```

defines an array of 16 words.

DATA statements are frequently used to define an array of strings which can be used for messages, etc. during the program. For example:

```
DATA WORD ERRORMSG [] =
  "Function Successful.",                    ; 0
  "Illegal widgit.",                         ; 1
  "Widgit not found.",                       ; 2
  "You must specify a Widgit Number first." ; 3
```

This statement defines a table of four words, each initialized to point to a string. Later in your program, if you wanted to print the "Widgit not found." error message, you could simply write:

```
PUT ERRORMSG[2]
```

PUT is a built-in LIBRARY procedure which displays the string specified, in this case the third string in the table.

Please note that the type of the above data array is WORD, not BYTE. This is because each element of the array is a string. You may recall from our discussion of strings that the "value" of a string is the **address** of its first character; therefore a WORD is necessary to hold this address.

OPERATORS

An **operator** is a special symbol which indicates an action to be performed. PROMAL provides the following operators:

Op.	Description	Example	Result
+	Addition	3 + 5	8
-	Subtraction or negation	48 - 11	37
*	Multiplication	-10.32 * .034	-.35088
/	Division (fraction discarded except REAL)	200 / 30	6
%	Remainder (mod)	200 % 30	20
<<	Left shift	7 << 1	14
>>	Right shift	$A0 >> 4	$0A
<	Relational operator less than	4 < 9	TRUE
<=	Relational operator less than or equal	6 <= 6	TRUE
<>	Relational operator not equal	'A'<>'A'	FALSE
=	Relational operator equal	'A'=65	TRUE
>=	Relational operator greater than or equal	10 >= 'a'	FALSE
>	Relational operator greater than	3 > 8	FALSE
AND	Logical AND operator	3>1 AND 4<10	TRUE
OR	Logical OR operator	2<=1 OR 8>9	FALSE
XOR	Logical exclusive OR	$00 XOR $FF	$FF
NOT	Logical complement	NOT TRUE	FALSE
#	Address of variable	#X	addr of X
:<	Extract low byte of WORD or INT	$1234:<	$34
:>	Extract high byte of WORD or INT	$1234:>	$12
:+	Convert to WORD	$5A:+	$005A
:-	Convert to INT	$FF:-	+255
:.	Convert to REAL	45:.	45.0
@<	Indirect through pointer to BYTE	PTR@<	see text
@-	Indirect through pointer to INT	PTR @-	see text
@+	Indirect through pointer to WORD	(PTR+2)@+	see text
@.	Indirect through pointer to REAL	PTR @.	see text

Some of these operators may look familiar from your experience with BASIC; others are entirely new. These operators may be combined with operands, which may be numbers, characters, strings, constants, variables, data, or functions, to produce expressions. We shall now examine the most important of these operators in detail.

ARITHMETIC EXPRESSIONS

Like BASIC, arithmetic expressions are evaluated from left to right (in the absence of parentheses), with multiplication and division having a higher priority than addition and subtraction. Therefore the expression:

```
3 + 4 * 5
```

evaluates as 23, not 35. A summary of operator precedence is given below.

OPERATOR PRECEDENCE

(operators in the same row have equal precendece)

```
:<, :>, :+, :-, :., @<, @+, @-, @., #        Highest precedence
NOT
*, /, %, <<, >>
- (negative)
+, -
<, <=, <>, =, >=, >
AND OR XOR                                    Lowest precedence
```

The arithmetic operators, +, -, *, and /, work in the expected fashion, but with a few twists. First of all, remember that PROMAL deals with integers (whole numbers) as well as real numbers. The result of arithmetic on type BYTE, WORD or INT cannot have a fractional result. Therefore 5 / 2 evaluates as 2, not 2.5 (any fraction is always discarded). However, 5. / 2. evaluates as 2.5, because the presence of the decimal point tells the PROMAL compiler that the numbers are REAL.

Note for **Commodore 64**: Be careful not to type the shifted "+" character on the keyboard when you want a plus sign. It looks like a plus sign, but isn't (the same applies to BASIC).

Most operators take two operands. For most operators, these two operands do not have to be of the same type. In a mixed expression involving operands of different types, the operands are usually "promoted" to the "higher" type automatically, where BYTE is the "lowest" and REAL is the "highest" type. The table below summarizes the results of a partially evaluated expression of the type shown in the left column when an operator is encountered with a new operand of the type shown in the top row:

RESULT TYPE FOR MIXED MODE EXPRESSIONS

Present Type is... \ Next operand involved is...	BYTE	WORD	INT	REAL	
BYTE	BYTE	WORD	INT	REAL	←
WORD	WORD	WORD	INT	REAL	←
INT	INT	INT	INT	REAL	← Result type
REAL	REAL	REAL	REAL	REAL	←

The TYPE of the data being operated on must be considered. For example, adding two variables of type BYTE will always result in a value which is also of type BYTE, even if the result is too large to fit in a BYTE variable. For example, if X is a variable of type BYTE which has been previously assigned the value of 254, then the expression X+4 will NOT have a value of 258, but 2. This is because BYTE variables can only take on values between 0 and 255, so that when you add 4 to 254, the result is (258-256) = 2.

If you don't quite understand this, think of PROMAL BYTE, INT and WORD variables as being like the odometer on your car. Most odometers go up to 99,999.9. If your odometer reads 99,998.0 and you drive 4 more miles, the odometer will read 00,002.0, not 102,000.0. A PROMAL variable of type BYTE only goes up to 255 ($FF hex), and then "wraps around" again starting at 0. A numeric expression which overflows the maximum value representable simply "wraps around" like this with no indication of an error. Similarly, if you subtract a larger BYTE operand from a smaller BYTE operand, the result is "wrapped around" but still positive. For example, 3 - 4 evaluates to 255 (think of what happens if you turned back the odometer 4 miles when it had a reading of 3).

Since by definition a BYTE type variable is unsigned, you cannot apply the negation operator to it directly, so the byte is automatically promoted to type INT (integer) before the negation is performed. This "promotion" is only done in the temporary work area called the accumulator where PROMAL does its arithmetic; it does not change the type or size of the original variable.

An operand of type WORD also is always positive, but in this case the largest possible "odometer reading" is 65535 (FFFF hex). For example if Y is a variable of type WORD with a value of 1, then Y-3 is 65534 ($FFFE), not -2.

Only integers and reals may take on negative values. To understand how integers work, again consider your auto odometer. If you started out at 0 and turned the odometer back 1 mile it would read 99,999.0. Turn it back another mile and it would read 99,9998.0. If you wanted to use your odometer to measure both forward and backward movement from 0, you might define everything from 0 to 49,999.9 as positive, and everything from 50,000.0 and above as negative, effectively splitting the total number of representable numbers in two (half positive and half negative). This is exactly how INT variables work in PROMAL.

In two bytes there are 65,536 possible numbers, which we divide in two, with 0 to 32767 being considered positive ($0000 to $7FFF). The other half of the numbers represent negative numbers, with -1 represented by $FFFF. The most negative number possible is -32768, or $8000. However, since there is no +32768 number representable, the number -32768 is disallowed. This number scheme is called "two's complement" arithmetic, and is standard on almost all computers.

For example, consider the following fragment of a PROMAL program:

```
BYTE X
WORD Y
WORD ANSWER
...
X = 254
Y = 300
ANSWER = X + Y
```

This will produce the expected result of ANSWER=554. However, if you change the last line to read:

```
ANSWER = X + 3 + Y
```


then the result will be ANSWER=301, because X and 3 are both type BYTE, so X + 3 evaluates to 1; this is then promoted to a word and added to Y to give 301. If the order of the operands was changed to:

```
ANSWER = X + Y + 3
```

then the result would be 557, because X + Y would be evaluated first, with X being promoted to WORD before making the addition.

Most of the time you will not have to worry about mixing different types in an expression, but when you do you should bear in mind the order of evaluation. You can "force" an operand to be promoted (or "demoted") from one type to another with the "type cast" operators, which are:

```
:<   Extract low order byte from word or integer (or convert real to byte).
:>   Extract high order byte from word or integer.
:+   Convert to word (unsigned).
:-   Convert to integer (signed)
:.   Convert to real (floating point).
```

These operators are written immediately **after** the operand which they are to change. For example:

```
ANSWER = X:+ + 3 + Y
```

would result in ANSWER=557, because the :+ operator will promote or "cast" X to a word before performing the addition with Y. The expression X:+ is read as "X cast to a word".

There are four special cases for arithmetic operators.

1. The % operator (remainder) cannot be applied to REAL operands. The sign of the result is always considered positive for the % operator.
2. If you multiply or divide two operands of type BYTE, both operands will be promoted to WORD, and the result will be type WORD.
3. Taking the negative of a BYTE or WORD converts to an INT. No error is given if the result is out of range (result truncated to 16 bits).
4. Dividing by zero will produce a fatal run-time error. A "zero divide" error can be triggered by any of the following:

   a. Division by 0 (X / 0).
   b. Remainder by 0 (X % 0).
   c. A REAL result larger than the largest representable value (about 1.E+37).
   d. Conversion of a REAL to a BYTE, WORD, or INT which cannot be represented (e.g., 100000. :+).

RELATIONAL OPERATORS

The relational operators (<, <=, <>, =, >=, >) are the same as their BASIC counterparts, and return a value of TRUE or FALSE. In PROMAL, TRUE is represented by a byte of value 1 and FALSE by a byte of value 0. For purposes of comparison in a conditional statement such as an IF statement (which we will study later), **any non-zero value is considered TRUE.** The result of a comparison using a relational operator is always type BYTE. Promotion of operands in a comparison is the same as for the arithmetic operators, but the result is always type BYTE.

The fact that the result of a relational operation can be interpreted as 0 or 1 as well as FALSE or TRUE can be useful. For example, the two statements:

```
IF PHASORS > 100
  SCORE = SCORE + 1
```

can be replaced by the single equivalent statement:

```
SCORE = SCORE + (PHASORS > 100)
```

because the expression (PHASORS > 100) will evaluate as 1 if TRUE and 0 otherwise.

The relational operators all have equal priority of evaluation and are of lower priority than any arithmetic operators, so that "normal" comparisons will produce the expected result when written without parentheses. For example the expression:

```
3 * 3 > 3 + 3
```

evaluates as TRUE (1).

Please note that you may **not** compare two strings by simply using the relational operators on the variables involved, because this would merely compare the **addresses** of the strings, which has no relation to the **content** of the strings. To compare strings, use the CMPSTR function, described in the LIBRARY MANUAL.

LOGICAL OPERATORS

The logical operators (AND, OR, NOT, XOR) may be combined with relational operators or used for bit-by-bit Boolean operations. These operators may only be used on operands of type BYTE, which is normal if using them in conjunction with relational operators. All logical operators have an equal priority of evaluation which is lower than the arithmetic and relational operators, so that "normal" combinations of operators will produce the expected result without parentheses. For example the expression:

```
X > 100 AND  Y = 0
```

is equivalent to:

```
(X > 100) AND (Y = 0)
```


and will evaluate TRUE if X is greater than 100 and Y is 0.

AND, OR and XOR are useful in performing bit-by-bit Boolean operations and masking operations (on type BYTE operands only). For example:

```
PORT  AND  $0F
```

will "mask off" the high order 4 bits of PORT. As you may have already discovered, these masking operations are frequently needed to manipulate selected bits within a byte.

The operator NOT is a unary operator which converts any non-zero byte to 0, and 0 to 1. To perform a bit-by-bit complement, use XOR $FF instead.

SHIFT OPERATORS

The operators << and >> perform left and right shifts, respectively. The operand to be shifted appears on the left side of the operator, and the shift count on the right, for example:

```
XVAL << 4
```

shifts the value of XVAL left by four bits. Shifts may be applied to all data types except REAL; however, **the shift count must be of type BYTE.** The shift count should be in the range of 0-8 for BYTE operands and 0-16 for WORD or INT operands. Shift operators have the same precedence of evaluation as multiplication and division. One of the most frequent uses of shifts is to perform multiplications or divisions by powers of 2. For example:

```
COUNT << 3
```

will compute eight times the value of COUNT much faster than:

```
COUNT * 8
```

Right shifting by N is equivalent to (and much faster than) dividing by 2 to the Nth power. Shifts are also sometimes used in conjunction with the logical operators for manipulating data into specific bits of a register. Bits shifted out of a byte or word are lost; 0 bits are always shifted into the word (even if it is a negative integer). The result of a shift on type INT is type WORD. There is no built-in operator to perform bit rotations.

INDIRECT AND ADDRESS OPERATORS

The operator **#** is the **address** operator. It can only be applied to a variable or data name (not to a number, string, constant or function). The # operator returns the address of the variable which follows it. For example:

```
WORD PTR
REAL STRENGTH
...
PTR = #STRENGTH
```


sets the variable PTR to the address of the variable STRENGTH in memory. The # operator can also be used to find the address of a particular element in an array, for example:

```
WORD PTR
DATA BYTE COMDCHAR [] = 'D','X','P','A','E','Q'
...
PTR = #COMDCHAR[2]
```

will set PTR to the address of the character 'P'.

The operators @<, @-, @+ and @. are **indirect operators**. They are used to access data "pointed to" by some variable or expression. The expression to the left of the indirect operator should be of type WORD. If it is of type BYTE, it will be promoted to type WORD automatically. For example:

```
WORD POINTER
REAL VALUE[10]
...
POINTER = #VALUE[7]
...
IF POINTER@. > 0.5
  ...
```

Here POINTER is set to the address of a certain element of an array of REALs. Later, the expression POINTER@. can be used to test the value of that element. The expression "POINTER@." can be thought of as "the real number pointed to by POINTER."

One of the most common uses of the indirect operators is to extract characters from strings. For example, consider the following program fragment:

```
BYTE BUFFER [80]
WORD PTR
BYTE CHAR
...
PTR= BUFFER
...
CHAR = PTR @<
...
```

This sequence will set CHAR to the first character of the array BUFFER. Although this could also have been done with the more straightforward statement:

```
CHAR = BUFFER[0]
```

the use of PTR allows more versatility, since PTR could point to **any** array, not just the BUFFER array. Pointers and indirect operators are very useful in passing arrays and strings to subroutines to be operated on, as you will see in Chapters 5 and 7.

Note that you **may not** use the indirect operators to identify the destination variable for an assignment statement. Therefore


```
PTR@< = 10  ; ILLEGAL!
```

is not legal. You may use the predefined array M, which is defined in the Library as an array of bytes encompassing all of memory, to solve this problem. The above example could be correctly written as:

```
M[PTR] = 10  ; Right!
```

The use of pointers and the array M is discussed further in the section on subroutines and in Chapter 5 and 7.

GLOBAL VARIABLES

Variables are normally declared first in your program, before the executable statements. These variables are called **global** variables, because they can be accessed from anywhere in your program. Later another kind of variable will be introduced called a **local** variable. Local variables are defined **inside** subroutines, and are known only inside that subroutine. Global variables are defined before any subroutines (or between subroutines), and are known everywhere thereafter in the entire program, (including inside all subroutines). This distinction will be clarified in Chapter 5, where subroutines are discussed.

Now that you know how to declare variables and form expressions, you are ready to learn how to bring these pieces together with the reserved words to form statements, and then combine these statements into a complete working program.

CHAPTER 4: STATEMENTS

INTRODUCTION

In this chapter you will learn about PROMAL language statements. If you have only programmed in BASIC and not in another "high-level" language you should study this chapter very carefully. If you have programmed in Pascal or "C" this chapter will be important for understanding the differences as well as the similarities of PROMAL and other "structured" languages.

Some PROMAL statements are similar to statements in BASIC. For example,

```
XV = YV + 17
```

is an assignment statement, which is very similar to a BASIC LET statement. However, there are some important differences between BASIC statements and PROMAL statements, including:

1. Statements **do not** have line numbers.
2. Only **one** statement is permitted on a line.
3. A statement may not occupy more than one line (with the exception of the DATA statement).
4. Keywords and variables **must** be separated from each other by blanks or other punctuation marks as required by the statement.

SYNTAX DIAGRAMS

In many ways, PROMAL allows you a great deal more flexibility in constructing statements than BASIC. In order to help you determine exactly what makes up a legal statement, a set of syntax diagrams is included in **Appendix P**. These syntax diagrams tell you graphically how to construct a legal PROMAL statement. Syntax diagrams are not difficult to use, once you are familiar with them. If in the following descriptions you are unsure about a PROMAL statement's correct syntax, you may refer to the diagrams in **Appendix P**, and the accompanying discussion of how to read them.

PROGRAM STATEMENT

Every PROMAL program must start with a PROGRAM statement of the form:

PROGRAM Name [**OWN** [**EXPORT**]]
-or-
OVERLAY Name [**EXPORT**]

where **Name** is a legal PROMAL identifier not used for any other purpose. The PROGRAM line declares the command name by which you will execute the program when it is loaded into memory. You should always **make the PROGRAM name the same as the file name you COMPILE**. The OWN keyword is optional, and is normally not used. If specified, it will cause the compiled program to be loaded into memory with the global variables allocated immediately after the program, rather than being shared with other programs in high memory. This and the EXPORT and OVERLAY keywords are discussed further in Chapter 8 and in the optional Developer's Guide.

ASSIGNMENT STATEMENT

The assignment statement is the simplest and most fundamental statement in PROMAL (or in any other language). You are familiar with it in BASIC. Its form is:

```
variable = expression
```

where expression can be a constant, a variable, a function, or a combination of these in an arithmetic or relational expression. See the Syntax Diagrams in **Appendix P** for all the possibilities. The assignment statement assigns the contents of (or results of) the expression on the right side of the "=" sign to the variable on the left side.

The variable on the left cannot be a DATA item. Here are some sample assignment statements:

```
X=0
ENDPAGE = TRUE
SMALLX =MIN(X1,X2,X3)
VAL[I]=3.14159*RADIUS[I]*RADIUS[I]
YBIGGER = Y > X AND Y > Z
```

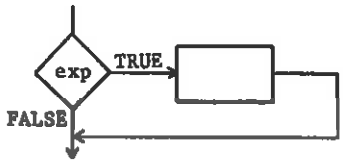
CONDITIONAL STATEMENTS

A conditional statement is a statement which alters the order of execution of statements based on evaluating a condition. In BASIC, the conditional statements are IF, FOR...NEXT, ON...GOTO, and ON...GOSUB. PROMAL has conditional statements which are more powerful and easier to read and understand than the related BASIC statements. The PROMAL conditional statements are the IF, WHILE, REPEAT, FOR, and CHOOSE statements.

IF STATEMENT

By far the most common conditional statement is the IF statement. It can take several forms. The simplest form is:

```
IF expression
  statement 1
  statement 2
  ...
statement n
```



In this form, the expression is tested, and if it is TRUE, then **all** the indented statements following it are executed. If it is FALSE, then control passes directly to statement n, on the same level of indentation as the IF.

For example:

```
IF X > 10
  OLDX = X
  X = 10
Z=X
```

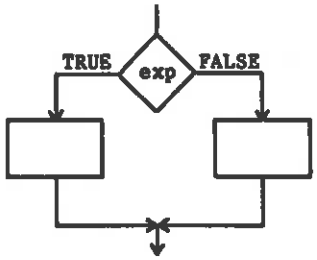

In this case, the conditional expression tests if X is greater than 10. If so, then OLDX is set to X and X is set to 10. If not, the two statements after the IF statement are skipped. In either event, the line Z = X is always executed.

BASIC programmers, please note that you **cannot** put THEN, GOTO, or anything else on the same line as the IF, after the condition! You **must** put the statements to be executed on the lines after the IF, and they **must be indented**. The indentation must be exactly two columns to the right of the IF. The proper indentation is easily obtained by using the TAB key (or CTRL I) in the PROMAL EDITOR.

If you have ever taken any courses in programming, you probably were told that indentation is a good way to show a program's structure. PROMAL simply enforces this concept. The indentation **does** show the structure of the program. **This is probably the most important feature of the PROMAL language.** By using indentation as a syntactical element of the language, PROMAL is able to do away with a host of confusing statement delimiters and begin-end brackets which pervade other structured languages.If you don't indent, you'll get an error message when you compile your program.

A second form of the IF statement has an ELSE clause:

```
IF expression
  statement 1
  ...
ELSE
  statement 2
  ...
statement n
```



In this form, the indented statements after the IF are executed if the expression is TRUE, and the statements after the ELSE are executed otherwise. This form is used to select one of two mutually exclusive paths. For example:

```
IF X > 100
  POINTS = 3
ELSE
  POINTS = 1
SCORE = SCORE + POINTS
```

If X is greater than 100, POINTS is set to 3 and control passes to the last line. If X is not greater than 100, POINTS is set to 1 and control passes to the last line.

The final form of the IF statement has one or more ELSE IF clauses before the final ELSE, for example:

```
IF CHAR = 'D'
  DRAW
ELSE IF CHAR = 'E'
  ERASE
ELSE IF CHAR = 'Q'
  EXIT
ELSE
  OUTPUT "ILLEGAL COMMAND."
```


This form is used to choose one of a number of mutually exclusive paths. Please note that the **only** thing that can follow an ELSE on the same line is an IF and a condition. The ELSE without an IF must be the last ELSE associated with the initial IF. Also be sure that ELSE and IF are typed as two words, not one.

IF statements may be "nested" to any depth needed. For example:

```
IF X > 100                ; 1
  IF Y > X                ; 2
    Z=3+X                 ; 3
    Y=0                   ; 4
  ELSE                    ; 5
    Y=1                   ; 6
    IF X > 200            ; 7
      Z=Y-100             ; 8
Q=Y+Z                     ; 9
```

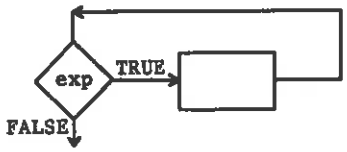
In this example, each IF controls all the statements with greater indentation. For example, if the first IF (statement 1) is false, then control will pass directly to statement 9. If statement 1 is true, then statement 2 decides if statements 3 and 4 should be executed or skipped. The only way statement 8 will ever be executed is if statement 1 is true, statement 2 is false, and statement 7 is true. You should have no doubt about which IF statement an ELSE "belongs to"; it is always the one with the same indentation.

Indentation plays a key role in making programs readable. You will soon be able to just scan over a PROMAL program or subroutine and immediately be able to understand its logic. Since PROMAL does not have a GOTO statement, there will be no mystery as to how you get to a certain statement. By just looking at the indentation, you will have a "picture" of the program organization.

WHILE STATEMENT

Next to IF, WHILE is the most commonly needed control statement in PROMAL. It has the following form:

```
WHILE expression
  statement 1
  ...
statement n
```



The WHILE statement evaluates the conditional expression. If it is TRUE, the indented statements are executed, as in an IF statement. After the last indented statement is executed, control returns to the WHILE statement and the condition is re-tested. The loop is repeated until the expression evaluates as FALSE; control then passes to **statement n**, which starts in the same column as the WHILE statement. The indented statements in a WHILE loop may be executed zero or more times. For example:

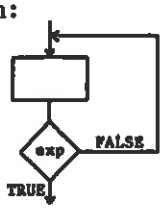

```
SUM = 0
X=0
WHILE X < XLIMIT
  SUM = SUM + X
  X = X + 1
Z=X
```

This program fragment forms the sum of the integers from 0 to XLIMIT. At the end of the loop, Z will be equal to XLIMIT.

REPEAT STATEMENT

A REPEAT statement is very similar to a WHILE statement, except that the condition is tested at the end of the loop instead of the top. The REPEAT statement has the following form:

```
REPEAT
  statement 1
  ...
UNTIL expression
```



The indented statements are executed one or more times. After the first execution of the indented statements, the conditional expression is evaluated. If the result is FALSE, control passes back to the top of the loop. If the statement is TRUE, control passes to the next statement after the UNTIL. For example:

```
REPEAT
  CHAR = GETC
UNTIL CHAR = 'A'
```

GETC is a standard LIBRARY function which returns a key from the keyboard. Therefore this loop waits for an 'A' to be typed, ignoring all other input.

FOR STATEMENT

The FOR statement is similar to a BASIC FOR-NEXT loop, but is more restrictive. A FOR loop has the form:

```
FOR Iter = Low TO Hi
  statement 1
  ...
statement n
```

Iter **must be a variable of type WORD**, and Low and Hi must be expressions which evaluate to the lower and upper bounds for the loop. For example:

```
WORD BUFFER [100]
WORD I
...
FOR I = 0 TO 99
  BUFFER [I] = 0
```


will initialize the array BUFFER to 0. Note that the iteration variable must be a simple variable, not an array element or expression, and must be type WORD. Also note that the loop must iterate upward, not downward (as is permitted in BASIC). There is no "STEP" size option as in BASIC; the step size is always 1. The indented block of a FOR loop is always executed at least once, even if Low is greater than Hi. These restrictions allow the FOR loop to execute very rapidly. If you need a FOR loop which doesn't meet these requirements, use a WHILE loop instead.

CHOOSE STATEMENT

The CHOOSE statement is a multi-way branch, somewhat similar to BASIC's ON-GOSUB statement, or the CASE statement of Pascal. It has the following form:

```
CHOOSE expression
choice 1
  statement 1
  ...
choice 2
  statement i
  ...
...
ELSE
  statement j
  ...
statement k
```

The CHOOSE statement works like a multiple-choice test. The expression is evaluated, and each of the choices listed below is compared to it in succession. When a match is found, the indented statements are executed. If no match is found, the indented statements after the ELSE are executed (think of the ELSE as "none of the above"). In any event, control always winds up at statement k, the first non-indented line after the ELSE. For example:

```
CHOOSE GETC
'B'
  X=0
  START
'C'
  CONTIN
'L'
  X=9999
  LASTLINE
ELSE
  PUT "Illegal key letter"
X=1
```

The program fragment above inputs a character from the keyboard (function GETC). If the character is 'B', then X is set to 0 and the START subroutine is called, and control transfers to the last line (X=1). If the character is 'C', the CONTIN subroutine is called instead, and control then passes to the last line. If the character does not match any of the choices, then an error message is output (the PUT does this), and control passes to the last line.

The choices for the CHOOSE must match the type of the expression in the CHOOSE line **exactly**, and the expression must **not** be type REAL. Note that this means if you have a CHOOSE with an expression of type WORD or INT, and your choices are small BYTE constants such as 0, 2, 100, etc., you must remember to promote the choices to type WORD or INT.

WRONG!

```
WORD NUM
...
CHOOSE NUM
1
  PROCESS_1
2
  PROCESS_2
ELSE
  PROCESS_OTHER
```

RIGHT

```
WORD NUM
...
CHOOSE NUM
1:+
  PROCESS_1
2:+
  PROCESS_2
ELSE
  PROCESS_OTHER
```

Any CHOOSE statement can be simulated with an IF statement with an appropriate number of ELSE IF clauses. However, CHOOSE will often be more efficient since you do not have to spell out each comparison explicitly.

The CHOOSE statement also has an alternative form where the word CHOOSE appears alone, for example:

```
CHOOSE
CHAR < ' '
  CONTROLCHAR
CHAR > $7F
  ILLEGALCHAR
ELSE
  NORMALCHAR
```

In this form, each of the choices is evaluated in succession until one evaluates TRUE. If all of the choices are FALSE, then the indented statements after the ELSE are executed. This is exactly equivalent to an IF with several ELSE IF clauses, except you do not have to write the ELSE IF's explicitly.

BASIC users should note that after the indented statements are executed for one of the choices, control automatically passes to the first non-indented statement after the ELSE; you do not need to put a GOTO after each like you do for a BASIC ON-GOSUB. Also note that the ELSE is mandatory, because it indicates the final choice ("none of the above").

BREAK STATEMENT

Sometimes it is desirable to "break out" of a loop at a point other than where the conditional test is done. The BREAK statement provides this capability for WHILE and REPEAT loops (but **not** for FOR loops!). For example:


```
WHILE TRUE     ; (do forever)
  IF I@< = '('
    IF (I+1)@< >= 'a' AND (I+1)@< <= 'z'
      IF (I+2)@< = ')'
        BREAK
  I=I+1
FOUND=I
```

This program segment will search all of memory for a single lower case alphabetic character enclosed in parentheses. Executing BREAK causes control to immediately pass to the statement after the **end** of the most recent WHILE loop (i.e., to the FOUND=I statement).

NEXT STATEMENT

The NEXT statement is used to cause an immediate jump to the **top** of the current WHILE or REPEAT loop (but **not** a FOR loop). For example:

```
INQUOTE=FALSE

COUNT=0
REPEAT
  CHAR = GETC
  IF CHAR='"'
    INQUOTE=NOT INQUOTE
    NEXT
  IF INQUOTE
    NEXT
  COUNT=COUNT+1
UNTIL CHAR=CR
```

This program segment counts the number of characters typed up to the next carriage return, excluding characters enclosed in quotes (including carriage returns in quotes). The NEXT statements pass control back to the top of the loop so as to ignore characters in between (and including) quotes. There may be better and easier ways to do this -- this is just for illustration.

NOTHING STATEMENT

The NOTHING statement does not perform any action, and the PROMAL COMPILER does not generate any object code for a NOTHING statement. This may seem of dubious merit, but is actually useful. For example:

```
REPEAT
  NOTHING
UNTIL GETC = CR
```

This loop simply waits for a carriage return from the keyboard, ignoring all other characters. The NOTHING statement fulfills the syntactical requirement that at least one indented statement must follow the REPEAT, but it performs no action. If you tried to leave out the NOTHING statement, you would get an error message from the compiler.

SHORTCUTS FOR CONDITIONAL STATEMENTS

You may recall that TRUE is represented by a byte with value 1 and FALSE by a byte with value 0. Several "shortcuts" can be used to take advantage of this fact to generate faster executing PROMAL statements. First of all, for a variable FLAG of type BYTE,

```
IF FLAG=TRUE
```

can be written equivalently but more economically as:

```
IF FLAG
```

Also the sequence:

```
IF X > 100
  FLAG = TRUE
ELSE
  FLAG = FALSE
```

can be more economically written as:

```
FLAG = X > 100
```

ESCAPE AND REFUGE STATEMENTS

The ESCAPE statement and REFUGE statement are unique to PROMAL and do not have a counterpart in other structured languages or BASIC. PROMAL, like many modern structured languages, does not have a GOTO statement, which results in much cleaner, more readable and more bug-free programs. There are occasions when you might wish you had a GOTO. This is best illustrated by an example.

Suppose you had a complex application program, with many layers of subroutines. Suppose further that at some low-level subroutine you come to a point where you need a piece of logic that could be paraphrased as:

```
IF Disaster
  Print error message
  Exit back up to the top level routine.
```

This is a common problem. Unfortunately, other languages do not provide a way to "exit back up to the top level routine". Instead, you must "unwind" all the CALLs with RETURNS. In other structured languages, you typically "solve" this problem by testing some global "Disaster" flag after returning from a lower level subroutine to short-circuit further processing, for example:

```
LOWERSUB    ; call lower subroutine
IF DISASTER ; if had problem in LOWERSUB
  RETURN    ; don't go any further
```

Each higher level subroutine would perform the same logic, until you "unwind" all the way back up to the desired routine. While this method works, it is unwieldy and dilutes the performance and clarity of the program with a lot of duplicate error checking.

PROMAL solves this problem a different way. The REFUGE statement can be thought of as an "executable label", and the ESCAPE statement as a GOTO which can exit back to a previously executed REFUGE.

The syntax of the ESCAPE and REFUGE statements is:

```
REFUGE n
ESCAPE n
```

where n is a constant between 0 and 2, allowing up to 3 different "refuges" to be defined concurrently in a single program. (Note: actually, there is also a REFUGE 3, but this is reserved for a special purpose and is described in the optional DEVELOPER'S GUIDE). Executing:

```
REFUGE 2
```

defines the statement after it as refuge number 2. Subsequently executing:

```
ESCAPE 2
```

will cause an immediate re-entry into the last subroutine (or main program) executing a REFUGE 2 at the line after the REFUGE statement, and will restore the context of the subroutine at that point. By restoring context, we mean that all intermediate variables, return addresses, etc., which would normally be "pending" when a RETURN is executed are discarded, up to the point where the refuge was executed. An ESCAPE is somewhat like a NEXT or BREAK statement, except that instead of just jumping to the beginning or end of a loop, you can **jump to anywhere you've been before.** It is the programmer's responsibility to insure that you do not try to ESCAPE to a REFUGE in a routine that has already returned (which will leave control in no-man's land!). On the next page is an example of fragments of a program using a REFUGE and ESCAPE:


```
PROC ERROR ; print error message and escape
 ARG WORD ERRNO
 ...
 BEGIN
 PUT NL, ERRORMSG[ERRNO],NL  ;display message
 ESCAPE 1
 END

 PROC CHECKCHAR
 BEGIN
 IF CHAR <> LEGAL
   ERROR 3
 ...
 END

 PROC PROCESSWORD
 ...
 CHECKCHAR
 END

 PROC DOPHRASE
 ...
 PROCESSWORD
 END

 PROC DOLINE
 ...
 REFUGE 1  ;Come here after error
 WHILE GETL(LINE)
   DOPHRASE
 ...
 END
```


CHAPTER 5: PROCEDURES AND FUNCTIONS

PROMAL provides a greatly enhanced subroutine capability compared with BASIC. Some of the most important characteristics of PROMAL subroutines are:

1. Subroutines may be either **PROC**edures or **FUNC**tions. Functions return a value which may be used in an expression. Procedures do not return a value.

2. Both procedures and functions must be defined (or "declared") before they can be called.

3. Functions and procedures are called by merely referencing their name in a statement.

4. Both procedures and functions may be passed **ARG**uments which they may operate on.

5. Both procedures and functions may have **local variables** which are known only within the scope of the subroutine. These local variables may duplicate other names outside the subroutine without interference.

6. Procedures and functions may be called recursively.

Let us now explain these concepts and show how to make effective use of subroutines.

BUILT-IN FUNCTIONS AND PROCEDURES

PROMAL does not have any built-in statements to do input and output, like BASIC PRINT and INPUT statements. Instead, PROMAL relies on a **LIBRARY** of pre-defined subroutines and functions to provide input and output. These routines are always resident in memory, and are used by the EDITOR, EXECUTIVE, and COMPILER as well as programs you write. When you use these subroutines, they could easily be mistaken for a special statement. For example:

```
OUTPUT "Hello World!"
```

appears just like a statement. There is no "CALL" or "GOSUB" keyword to reveal that this is really a subroutine call, with a passed argument of "Hello World!". This is no accident. A design intent of PROMAL is that subroutines should give you much of the power of adding your own statements to the language. You call subroutines of your own in the same way.

The built-in subroutines are described in detail in the LIBRARY MANUAL. At this point we would like to introduce you to just the most important of these routines, so that you can perform basic input and output operations.

Before you can call a subroutine, you must define it. For the LIBRARY subroutines, this is done by having the following statement near the top of your program:


```
INCLUDE LIBRARY
```

This defines all the standard LIBRARY routines to the PROMAL compiler. For now, it is sufficient for you to know that this LIBRARY gives the name and location of each of the built-in routines. You can display the Library with a **TYPE L** command from the EXECUTIVE.

SIMPLE OUTPUT

Probably the most fundamental of the standard procedures is called **PUT.** It outputs **single characters or strings** to the screen. It can have one or more arguments. For example:

```
PUT "Hello world!",NL
```

This statement calls the PUT procedure and passes it two arguments to be displayed. The first argument is the string "Hello world!", and the second argument is NL, the pre-defined "newline" character (which is the ASCII control character CR and has the value 13 for the Apple/Commodore version of PROMAL). Unlike a BASIC PRINT statement, **you must explicitly output an NL each time you want to start a new line.** This makes it easy to build up a composite line with several calls to PUT.

Please note that, unlike BASIC, you **cannot** print the numeric value of a variable with the PUT statement. You can only **print strings or characters.** To print a numeric value, you will want to use the OUTPUT procedure.

FORMATTED AND NUMERIC OUTPUT

OUTPUT is a procedure for performing formatted output to the screen. It accepts one or more arguments. **The first argument must be a string.** It is called a **format string,** because it tells the format in which any additional arguments should be printed. If you have ever used a BASIC version which supports PRINT USING, OUTPUT is similar. Actually, it is most similar to the PRINTF function in the C language.

The format string contains text to be printed on the screen as well as formatting information. The special lead-in character **#** is used to start a **field specification** (sometimes called a field descriptor), which tells **how** to print something. For example:

```
INT SECS
...
SECS = 673
OUTPUT "The answer is #I seconds.", SECS
```

These statements will display:

```
The answer is 673 seconds.
```

The value of the argument SECS replaces the format field specification #I. The "#I" indicates that the second argument should be displayed as an integer. The most commonly needed field specifications include:


```
#I      Print the argument as a signed integer number.
#W      Print the argument as an unsigned number (not for REAL variables!)
#H      Print the argument as a hexadecimal number.
#S      Print the argument as a string.
#C      Print a carriage return.
#E      Print the REAL argument in scientific notation.
#R      Print the REAL argument with a decimal point.
```

The OUTPUT statement can have more than two arguments. The format string **must have a field specification for each argument to be printed.** For example:

```
WORD N
...
N = 257
OUTPUT "#C#W decimal = #H hexadecimal.",N,N
```

will display:

```
257 decimal = 101 hexadecimal.
```

after a carriage return. Notice that the #C does not go with any argument; it just prints a carriage return.

You can also specify a "field width" in the format string (for example, to make columns of numbers line up). These options are fully described in the LIBRARY MANUAL. For REAL output, you normally specify both a field width and the number of decimal places to be displayed, in the form:

```
#w.dR
```

where w is the field width (from 3 to 12 characters), and d is the desired number of decimal places. For example:

```
REAL BUCKS
...
BUCKS = 276.10
OUTPUT "$#7.2R",BUCKS
```

will display:

```
$ 276.10
```

whereas BASIC would always print $ 276.1 instead.

SIMPLE INPUT

Now that you know how to output to the screen, let's see how you input from the keyboard. The procedure GETL is used to get one line from the keyboard and store it in an array **as a string.** GETL allows all editing features (backspace, insert, delete, CTRL-B, etc.) allowed by the EXECUTIVE. It returns when the RETURN key is pressed. For example:


```
BYTE LINE[81]
...
GETL LINE
```

reads one line from the keyboard and puts it in array LINE. After the call, the LINE array will be terminated by a 00 byte. It will not include the carriage return. Normally only one argument is present for the GETL procedure, and that argument is the **address** of where to put the line. Remember that **the name of an array without a subscript evaluates as the address of the array.** Optionally, you may include a second argument which is the maximum line length to accept (excluding the 0 byte terminator). For example:

```
GETL LINE, 20
```

will read a line from the keyboard up to 20 characters long. Additional characters on the line will be ignored. If the second argument is not specified for GETL, a maximum of 80 characters can be input.

The GETL statement is much more powerful than a BASIC INPUT statement because GETL supports a complete set of line-editing keys, as shown in **Table 1** of the USERS MANUAL. These keys are consistent with the editing keys used in the PROMAL EXECUTIVE and EDITOR.

One of the most useful features of GETL is the ability to recall prior lines by pressing CTRL-B. Another powerful feature for many applications is the use of function keys to "call up" pre-defined strings of up to 31 characters (much like many commercial "keyboard enhancers"). The LIBRARY MANUAL describes how to use FKEYSET to define a string to be substituted for a function key.

NUMERIC INPUT

How do you read in a numeric value from the keyboard? This is not quite as simple in PROMAL as in BASIC, because PROMAL does not have a built-in statement to read a number. Instead, you do it in two parts. First, you read a line into a buffer as described above. Then you convert the value represented by the string using function STRVAL or STRREAL. STRVAL converts a string to the numeric value it represents of type INT, or WORD. STRREAL is used to convert type REAL numbers. It is similar to the BASIC function VAL. For example, to read a number called HEIGHT from the keyboard, you could write:

```
BYTE BUF [81]
WORD HEIGHT
BYTE INDEX
...
GETL BUF
INDEX = STRVAL(BUF,#HEIGHT)
...
```

The STRVAL function expects at least two arguments. The first argument is the address of the string to be converted. The second argument is the **address** of the variable to receive the value. To specify the address of the variable (rather than the value), you need to specify the # operator, as shown above. Forgetting the # in front of the variable is a common error that results in the value being installed at whatever address is the current value of HEIGHT, so be careful! Also remember that the destination variable must be type WORD or INT,

not BYTE. Besides installing the value of the number into HEIGHT, function STRVAL will return an **index** of type BYTE. This index indicates the number of characters which were scanned in the string before the end of the number. If the INDEX is returned as 0, it indicates that no numeric digits were entered, probably representing an error condition. For example, if you typed

```
123
```

then INDEX would be returned as 3 and HEIGHT as 123. This method may seem a little ungainly and roundabout at first, but it allows a great deal of flexibility and programmer-defined error recovery, which is essential for serious programming. STRVAL also supports hexadecimal input, formatted input, and variable numbers of inputs on a line. These options are described in the LIBRARY MANUAL.

BASIC users accustomed to using the INPUT statement to prompt for a numeric input from the keyboard and input it may want to incorporate the following general purpose PROMAL routine. This INPUTR function will give a prompt for input and return the REAL value that the user enters from the keyboard. If an illegal input is entered from the keyboard, it repeats the prompt.

```
FUNC REAL INPUTR ; Prompt
  ; Prompt for numeric input from keyboard, return one REAL value.
ARG WORD PROMPT  ; Desired prompt
REAL TEMP        ; Value to be returned
BYTE INDEX       ; Index to # chars scanned
OWN BYTE BUF[21] ; Temp buffer for typed input line
BEGIN
REPEAT
  PUT NL,PROMPT  ; Display prompt
  GETL BUF,20    ; Get typed input
  INDEX=STRREAL(BUF,#TEMP)
  IF INDEX=0     ; No legal digits?
    PUT NL,"Please enter a numeric value"
UNTIL INDEX > 0
RETURN TEMP      ; Return value typed in
END
```

A sample program fragment using this routine for input might look like this:

```
...
REAL HEIGHT
REAL WIDTH
REAL AREA
...
HEIGHT=INPUTR ("Height of triangle? ")
WIDTH = INPUTR( "Base of triangle? ")
AREA=0.5*HEIGHT*WIDTH
OUTPUT "#CArea of triangle is #12.4R square units.#C", AREA
...
```


An example in the STRVAL section of the LIBRARY MANUAL contains a variation of the routine above for entering WORD or INT data instead of REAL values. For your convenience, both these functions are provided on disk as source files INPUTR.S and INPUTW.S, so you can easily include them in your programs or modify them to suit your individual needs.

The LIBRARY contains many more Input-Output routines, including file input and output. We will postpone a discussion of these routines until later.

USER-DEFINED SUBROUTINES

When you define your own PROMAL subroutine, you write it in the following general form:

{header}
{arguments}
{local variables}
BEGIN
{body}
END

The {header} is a single line that identifies the start of the subroutine. It has the form:

PROC name

or

FUNC Type Name

which defines whether the subroutine will be a procedure or a function. For example:

PROC SORT

declares the start of a procedure called SORT.

FUNC BYTE TESTPORT

declares the start of a function called TESTPORT which will return a value of type BYTE. The type returned may be BYTE, WORD, INT, or REAL.

The **{arguments}** and **{local variables}** will be discussed very shortly.

The **{body}** part of the procedure or function is contained between the **BEGIN** and **END** statements. It contains the executable statements of the procedure or function. When program control reaches the END statement, the subroutine will return to the calling program. Optionally, the RETURN statement can be used to return before the END statement.

For FUNCtions, a RETURN statement is required and **must** be followed by an expression which evaluates to the value to be returned by the function. For example:


```
RETURN YVAL+1
```

will return the value of YVAL+1 as the value of the function. Function values will be covered in more detail shortly.

PASSED ARGUMENTS

A powerful feature of PROMAL is the ability to use arguments passed to procedures and functions. BASIC does not support passed arguments (except in a very limited sense in simple function definitions using FNx, which is rarely used). To use passed arguments, the PROMAL subroutine definition should include one argument declaration line for each argument which is to be passed to the subroutine. An argument declaration looks like a simple variable declaration, with the word ARG in front:

```
ARG Type Name
```

Type is the desired data type which may be BYTE, INT, WORD, or REAL. Name is the desired name of the subroutine, formed in the same way as other variable names.

For example:

```
PROC SORT
ARG WORD N
ARG WORD PTR
```

declares two passed arguments, N and PTR, both of type WORD. The **order in which the arguments are declared is the same as the order in which the corresponding values will be passed.** For example, if the SORT procedure above was called with:

```
BYTE ARRAY[100]
...
SORT 26, ARRAY
```

then when SORT begins executing, N will have the value of 26 and PTR will have the address of ARRAY. As you can see, a procedure is called by simply writing the name of the procedure to be called. Arguments are passed by putting the arguments after the procedure name. Each argument can be an expression, and arguments are separated by commas. When you call a procedure or function which you have defined, the number of arguments must agree exactly with the number you declared, or you will get an error message from the compiler. The initial value of the arguments depends entirely on the values passed.

If the routine is later called with:

```
SORT CURSIZE+1, BUFFER
```

then N will have the value CURSIZE+1 and PTR will have the value BUFFER. As you might imagine, this substitution process makes subroutines very versatile.

A very important fact about arguments is that the names declared for passed arguments are **local** to the subroutine. This means that the name declared has meaning only within the subroutine where it is declared. It may duplicate a name used outside the subroutine for another purpose without harm. Technically, we say that the **scope** of the variable is local to the subroutine. This means that you can write a subroutine for one program and later copy it into another program without having to worry if the names you chose for argument variables will "collide" with some other variable names already in use. For example, suppose the following program fragment calls our sample routine:

```
N=11
SIZE=17
SORT SIZE,BUF
Z=N
```

After the SORT routine returns, what will be the value of N when it is assigned to Z? Will it still be 11 or will it be 17 because SIZE is 17 and was substituted for N in the SORT routine? The answer is that N will still be 11, because the N in the subroutine is only meaningful within the subroutine where it is declared.

PROMAL passes arguments on a "call by value" basis, with arguments passed on the microprocessor's hardware stack. This means that when you pass an argument to PROMAL, the argument is evaluated and this value is substituted for the local variable. Therefore, if the local variable's value is altered within the subroutine, it will not affect the value in the calling routine. For example, suppose that part of our SORT routine looks like this:

```
PROC SORT
ARG WORD N
ARG WORD PTR
BEGIN
...
N=0
```

Assuming we call the subroutine with:

```
SIZE=17
SORT SIZE, BUFFER
```

What will be the value of SIZE when the subroutine returns? Will it still be 17 or will it be 0? It will be 17, because the variable N is local to the SORT routine, and contains a copy of the value of SIZE, not the SIZE variable itself.

A passed argument need not have the same type as the type declared for the variable in the subroutine, although in general it is good practice to make them the same. If you pass a BYTE argument to a variable declared to be a WORD, the value will be converted to a WORD as it is passed. For example:

```
BYTE SIZE
...
SIZE=10
SORT SIZE, BUFFER
```


will work properly, even though SIZE is type BYTE and will be substituted for N inside the subroutine, which has a declared type of WORD. Technically, the declared variable is sometimes called a "formal parameter" and the value passed in the call to the subroutine is called an "actual parameter".

Although a BYTE actual parameter may be passed to a routine with a WORD formal parameter, you should be very careful to **only** pass a REAL argument to a REAL formal parameter. Passing REAL variables to a routine expecting BYTE, INT or WORD arguments will produce very strange results!

Sometimes you may want to modify a global variable which is passed as an argument to a subroutine. In this case, the usual procedure is to **pass the address** of the variable to be changed to the routine, and let the routine set the value using the globally pre-defined array **M**, which is defined in the library to be an array of bytes encompassing all of memory. For example, our subroutine SORT may wish to sort the array of bytes pointed to by PTR. To set the first value of this array to 0, for instance, we could write in the body of our subroutine:

```
M[PTR]=0
```

Arguments must be declared as simple (unsubscripted) variables. **You may not declare an argument which is an array.** This does not mean that you can't **access** a global array from inside a subroutine. You may do this freely. You cannot **declare** the array inside the subroutine. It is also possible for a subroutine to operate on an array whose address is passed as an argument. In our example procedure, SORT, the array BUFFER was given as the second argument. Remember that when an array name is used without a subscript, PROMAL generates the address of that array. Therefore our call will pass the address of the array to the subroutine, which is why PTR is declared to be a WORD. Since PTR contains the address of the start of the array, elements of the array can be accessed using the indirect operators, or by the M array as shown above. For example:

```
BYTE BUFFER[10]
...
PROC SORT
ARG WORD N
ARG WORD PTR
BEGIN
...
IF PTR@< > (PTR+1)@<
...
END

...
SORT 10,BUFFER
...
```

The IF statement above will compare the value of the first and second bytes of the array BUFFER.

Installing a REAL value into a variable whose address is passed as an argument can be accomplished by a block move of exactly 6 bytes from the address of the local variable to the desired destination. See BLKMOV in the LIBRARY MANUAL for information on block moves.

LOCAL VARIABLES

A local variable is similar to an argument, but has **no** initial value defined. Local variables are known only within the subroutine in which they are declared, and "disappear" when the routine returns. Local variables are most often used for temporary storage within the routine. Local variables should be declared **after** the last ARGument in the procedure or function. A local variable declaration appears the same as a simple global variable declaration. For example:

```
PROC SORT
ARG WORD N
ARG WORD PTR
WORD I
BYTE CHAR
...
```

declares two local variables, I and CHAR. The compiler knows these are local variables and not global variables because they are declared within the subroutine. Except for having no initial value, local variables behave identically to arguments. In particular, they are allocated on a stack and therefore must be simple variables, **not arrays.**

To illustrate the concepts of global and local variables, here is a complete, simple function which returns the number of blanks in a string. Remember that a PROMAL string is an array of bytes terminated by a $00 byte.

```
FUNC BYTE NUMBLANKS  ; string
;   return # blanks in string
ARG WORD STRINGPTR   ;address of string
BYTE N               ;counter
BEGIN
N=0
WHILE STRINGPTR@<
  IF STRINGPTR@< =' '
    N=N+1
  STRINGPTR=STRINGPTR+1
RETURN N
END
```

There are a number of important concepts here. First, the line

```
WHILE STRINGPTR@<
```

will evaluate TRUE as long as the byte pointed to by STRINGPTR is not 0; that is, not end-of-string. Second, the line

```
STRINGPTR=STRINGPTR+1
```


is perfectly legal and does not change the address of the original string passed to the subroutine. STRINGPTR is local to the subroutine and is initialized to point to the start of the string, and can be used to step through the string one character at a time.

Finally, notice that the variable N, which is used to count the number of blanks, must be initialized to 0 explicitly because PROMAL does not initialize local variables to anything. The result of the function is returned via the RETURN N statement.

If you call this function with the statement:

```
NB = NUMBLANKS("Hello there everybody!")
```

then NB will be set to two. Notice that functions, unlike procedures, must be called with the arguments enclosed in parentheses. This is because a function can be part of a larger expression, for example:

```
GETL MYMSG
NBP1 = NUMBLANKS(MYMSG) + 1
```

will set NBP1 to the number of blanks in MYMSG plus 1.

OWN VARIABLES

Local variables may not be arrays, and the value associated with a local variable "disappears" upon exit from the subroutine in which it is defined (because space for the variable is allocated on a stack). This meets the requirements of the vast majority of variables in subroutines. Sometimes though, you may want to have a variable known only within the subroutine, but which is an array or needs to preserve its value from call to call. This can be done by declaring an OWN variable. For example, the statements:

```
OWN BYTE TEMPBUF[8]
OWN WORD COUNT
```

declare two variables whose names are local to the subroutine in which they are declared, but which will maintain their values through multiple subroutine invocations. The most common use of OWN variables is to provide a scratch array needed for intermediate processing by a subroutine. OWN variables should be declared **after** all arguments and local variables, but before the BEGIN statement in a subroutine.

GOOD PROGRAMMING PRACTICE WITH SUBROUTINES

It is considered good programming practice to add a comment after the header line of a procedure or function definition which tells the function of the subroutine, what it expects for input, what it returns for output, etc. Many PROMAL programmers like to put a comment at the end of the header line listing the required arguments. If you do this, it will be easy to refer to the header line for a quick reminder of what arguments are expected. Finally, it is a good idea to put a comment on each argument declaration and local variable, identifying the purpose of the variable and any constraints on its use.

You should make frequent use of procedures. It is generally best to keep procedures short. Most PROMAL routines should have only a few lines of code. The PROMAL COMPILER, EDITOR, and EXECUTIVE contain hundreds of subroutines with less than twenty lines of code. If you have a procedure of more than about 50 to 100 lines, it should probably be broken up into lower-level subroutines. It is often a good idea to use procedures for various phases of processing, even if the procedures are only called in one place in the entire program. PROMAL subroutine calls require very little overhead time. Therefore in general you need not worry about extra procedure calls slowing down your program the way GOSUBS slow down BASIC. Remember, PROMAL doesn't have to search for your subroutines the way BASIC does. The PROMAL compiler generates the address of the routine during compilation, so calls are very fast, and take the same amount of time no matter where in the program the subroutine definition is located.

Subroutines are named the same way as variables. It is often a good idea to pick a verb which describes the main action of the subroutine for its name. Then when you call the subroutine with one or more arguments, the statement will be very readable, for example:

```
DISPLAY SPACESHIP
```

calls procedure DISPLAY with an argument of SPACESHIP.

You can learn a lot about procedures and functions by studying the sample programs on the PROMAL diskette.

RECURSION

PROMAL fully supports recursion. This means that it is permissable for a procedure or function to call itself, or for procedure A to call procedure B which in turn calls procedure A again. This capability is very important in certain programming disciplines, such as writing compilers, artificial intelligence applications, and in symbolic math. It is also possible to have forward references to procedures and functions. Techniques for recursive programming are described in **Appendix J.**

The ability to perform recursion on the Commodore 64 and Apple II is limited by the architecture of the 6502 processor, which only has a 256 byte stack. Although PROMAL has been carefully written to work around this limitation as much as is practical, you should not expect too many levels of nesting (or recursion) before you get a STACK ERROR message. You will use up more stack space as the number of local variables or passed arguments increases. A typical function with one passed argument and one local variable can call itself about 40 times before stack overflow occurs. This is why it is possible to get a stack overflow error while compiling a program with an expression that is very complicated and uses many levels of parentheses. The compiler uses recursion extensively to parse statements and can run out of stack space as repeated recursive subroutine calls are made to process complex statements.

USING THE INCLUDE STATEMENT FOR MULTIPLE SOURCE FILES

For large programs, it is not practical to edit the entire program at once. Instead, you should break up your source program into several files. Your main file can then have an INCLUDE statement for each of the sub-files. You have already seen how the INCLUDE statement is used to include the LIBRARY definitions in your program. You can do the same thing for your own programs. For example, the statement:

INCLUDE FILESUBS

will cause the compiler to pause at this point in the main file and compile all the lines in the file FILESUBS.S before continuing.

You may put an INCLUDE statement anywhere you can put a declaration. A ".S" extension will be assumed for the file name if one is not specified. The INCLUDE file can have a drive or directory prefix. For the Commodore 64, due to limitations of the Commodore 1541 disk drive, you cannot have nested include files (that is, a file INCLUDEd in the compilation cannot itself contain another INCLUDE statement). For the Apple II, INCLUDE files may be nested up to 3 deep. However, you may need to specify more than three buffers for ProDOS (see the BUFFERS command in the PROMAL USER'S GUIDE) in order to use nested INCLUDEs. INCLUDE statements can also be used to import definitions from separately compiled modules, as is described later in the LOADer section.

ENABLING AND DISABLING LISTING OUTPUT WITH THE LIST STATEMENT

The L option on the COMPILE command is used to enable listing output for the compiler. When making a listing, you can also disable the listing for parts of your program with the LIST statement. The LIST statement can appear anywhere a declaration can appear. It can have either of the following forms:

LIST Constant

or

LIST

The first form enables the listing if **Constant** evaluates to a non-zero value and disables the listing if it is zero. The second form restores the listing mode to whatever it was prior to the previous LIST (on or off). This form is useful at the end of a subroutine package which has the listing turned off, where it is not known if you will want the listing ON or OFF after the end of the subroutine package.

You may have any number of LIST statements in a program. If the L option is not specified on the COMPILE command, no listing will be made regardless of any embedded LIST statements. For an example, if you **TYPE L**, you will see how the listing of the LIBRARY is disabled.

CONDITIONAL COMPILATION

Sometimes you may have several versions of a program which vary only slightly. For example, the PROMAL EXECUTIVE is slightly different for the COMMODORE 64 version and the APPLE II version. In cases like this, you may wish to take advantage of PROMAL's **conditional compilation** capability. Conditional compilation allows you to generate several versions of a program from a single source file (perhaps with INCLUDEs), by specifying which version you wish to compile on the COMPILE command line. Here's how it works.

Inside your program source file, you can "bracket" the source lines which should only be compiled for a certain version. This is done by inserting a line above the first version-dependent line, containing a question mark in column one followed immediately by a single character representing the version for which the following lines are to be assembled. For example, you might choose 'A' for an Apple-dependent portion and 'C' for a Commodore-dependent section:

```
PROGRAM MYPROG
INCLUDE LIBRARY
...
?A
PUT NL,"The COLOR command is not supported on the Apple."
?

?C
COLOR=NUMVAL
?
...
```

In this example, there are two conditionally compiled blocks, each of a single line. The first block is started by the ?A and is only intended to be compiled if we want an Apple version. The ? by itself (exactly in column 1) terminates this block. The second block is started by ?C and terminated by the second plain ? character.

Selecting which (or neither) block should be compiled is selected by the V (version) option on the COMPILE command. For example,

```
COMPILE MYPROG V=A
```

will cause the Apple version to be compiled, and

```
COMPILE MYPROG V=C
```

will cause the statement COLOR=NUMVAL to be compiled instead. If you don't specify either V=C or V=A on the command line, then neither block will be compiled.

Conditional compilation is sort of like a simple IF statement, except that if the conditional block is skipped, the compiler does not generate any code at all for those statements; the result is equivalent to removing them with the EDITOR (or, more precisely, to "commenting them out" by putting a semi-colon in front of each).

You can specify a conditional block for either or two or more versions. For example:

```
?AC
```

starts a conditional block which will compile if either V=A or V=C is specified, but won't otherwise.

If no V option is specified, the compiler will compile a block which starts with

```
?*
```

if it appears. This is useful for embedding an error message to remind the user that a version must be specified on the command line. For example:

```
PROGRAM MYPROG
?*
*** YOU MUST SPECIFY V=A OR V=C TO COMPILE THIS PROGRAM! ***
?
INCLUDE LIBRARY
...
```

If you compile this program without the V option specified, the compiler will attempt to compile the warning line, giving an error message and displaying the line. If you specify a V option, the warning will not be compiled.

You may have any number of conditional blocks in a program. However, you may not nest conditional compilation blocks (that is, you can't have a conditional block inside another conditional block). The size of a conditional block is arbitrary, and may span INCLUDE files. It is your responsibility to insure that each block is terminated by a ? in column 1. You may have any number of single character version indicators following the ? character which begins a conditional block, but you may specify only one character on the V command line option (if you specify more, all but the first character will be ignored, so V=APPLE is equivalent to V=A).

The RELOCATE.S file on the PROMAL SYSTEM DISK illustrates the use of conditional compilation.

CHAPTER 6: INTERFACING

This chapter describes how your PROMAL program interfaces with its environment, including:

1. Disk files.
2. The printer.
2. The EXECUTIVE.
3. Your Apple II or Commodore 64 computer hardware.

FILES AND DEVICES

The PROMAL USERS GUIDE contains a section on the requirements for naming and using PROMAL files and devices on your computer. You may wish to review this material, particularly **Table 3** and **Table 4**, before proceeding with this section, which describes how to input and output to files and devices from within your PROMAL program. In particular, please remember that PROMAL file names normally have at least two characters, while device names have a single character. Also remember that the system will normally assume a default file extension of ".C" for file names if no file extension is specified.

PROMAL provides functions and procedures in the LIBRARY to input and output to files and devices. **The same routines may be used to access a file or device (such as the printer).**

OPENING AND CLOSING FILES

Before a file or device can be accessed, it must be **opened**. The library function OPEN performs this task. The OPEN function returns a **file handle** (sometimes called a file descriptor), which is a pointer to a table maintained in memory by PROMAL, used to control file I/O. This file handle should be assigned to a variable of type WORD. Once the file is open, the file handle can be used to direct subsequent I/O to the file desired. For example:

```
WORD INFILE
...
INFILE = OPEN("MYFILE.D",'R')
```

opens file MYFILE.D for reading. The second argument must be of type BYTE (not string!) and indicates the mode of operation, chosen from the following:

'R' (or omitted)	Open the file for read access.
'W'	Open a new file for write access.
'A'	Open an existing file for append access.
'B' (Not available on Commodore)	Open for both read and write access.

If the **file handle is returned as 0, then the open was not successful,** and an error code is available in a globally predefined variable called IOERROR. IOERROR will be one of the following:

IOERROR	Meaning (if file handle returned as 0)
0	No error, normal result
1	Illegal mode character
2	Illegal file or device name
3	Disk drive is not ready (or wrong volume name for ProDOS)
4	File not found
5	File already exists (for 'W' access attempt)
6	No free channels or buffers (too many open files)
7	Attempt to write on write-protected disk ('W' or 'A' access)
Other	Other error, see Commodore 64 disk manual or Apple ProDOS manual.

You should always test for an open failure before attempting I/O to the file or device. For example:

```
WORD INFILE   ;file handle for input file
...
INFILE = OPEN ("MYFILE.D",'R')
IF INFILE = 0
  IF IOERROR=4   ;the most likely error
    ABORT "MYFILE.D file not found."
  ABORT "#CDisk OPEN error #W",IOERROR
; Open was successful...
...
```

You can also open the devices for input or output in the same way, for example:

```
INFILE = OPEN ("W",'R')
```

opens the Workspace for read access. Recall that the Workspace is a single in-memory file with a fixed maximum size (variable size for the Commodore 64).

```
WORD PRINTER
...
PRINTER=OPEN("P",'W')
IF PRINTER=0
  PUT NL,"CANT OUTPUT TO PRINTER"
...
```

opens the printer device for output and checks the file handle for a successful open.

FUNCTIONS FOR FILE AND DEVICE I/O

Probably the most commonly used routines for accessing files are GETLF and PUTF. Function GETLF gets a line of text from a file or device, and procedure PUTF outputs characters or strings to a file or device. The first argument for all file-access routines **must** be the file handle of the previously-opened file or device. Function GETLF returns TRUE if it successfully got a line, and FALSE if end-of-file was encountered immediately. For example:

```
WHILE GETLF(INFILE, BUFFER)
  ...
```


reads a line from the file opened successfully with file handle INFILE and installs a line into the array BUFFER, which is assumed to have been previously declared as an array of bytes. The WHILE statement is frequently used in conjunction with this function to continue reading until end-of-file. The body of the WHILE loop contains whatever processing is to be done on the line. You may specify an optional third argument on function GETLF which specifies the maximum number of characters to be returned from the line. GETLF should only be used to read text files, not compiled programs.

Procedure PUTF is similar to the screen-output routine, PUT, except that the first argument must specify the file handle of a successfully-opened file or device. For example:

```
WORD OUTFILE
...
OUTFILE = OPEN ("MYFILE.T", 'W')
IF OUTFILE = 0
  PUT "Unable to open MYFILE.T for output."
  ABORT
PUTF OUTFILE,"This line goes to MYFILE.T",NL
PUTF OUTFILE,"So does this.",NL
...
```

Like PUT, PUTF can contain any number of strings or single character arguments to be output. It will not output a carriage return unless explicitly indicated. PUTF can put any kind of data byte out to a file, not just printable characters.

The OUTPUTF procedure is equivalent to the OUTPUT procedure for formatted output, except that the first argument must be the desired file handle. For example:

```
OUTPUTF OUTFILE, "#C#H   #S",LINENUM,LINE
```

outputs to the file previously opened.

Other functions are available for single character and block input-output to files and devices. These are described in the LIBRARY MANUAL.

STDIN AND STDOUT FILE HANDLES

When your PROMAL program begins, you already have two open file handles available for use. These are the globally predefined WORD variables **STDIN** and **STDOUT**. By default, these file handles normally point to the keyboard and screen, respectively. However, they can be **redirected** to any file or device when your program is executed by an EXECUTIVE command (See the MEET PROMAL and PROMAL USER'S MANUAL for details). Therefore if you input from STDIN and output to STDOUT, your program's output will be redirectable under EXECUTIVE control. For example:

```
PUTF STDOUT,"This goes to the screen or where I redirect it.", NL
PUTF STDOUT,"So does this."
```


You do not have to open STDIN or STDOUT. These variables already hold open file handles when your program starts. If you want some output to go to the screen no matter what, you simply use PUT and OUTPUT instead of PUTF and OUTPUTF. This will bypass I/O redirection set by the EXECUTIVE.

OUTPUT TO PRINTER

To output to the printer, simply OPEN the "P" device for output and use the file output procedures with the for the printer specified as the first argument. For example:

```
WORD PRT      ; Handle for printer
REAL X
...
PRT=OPEN("P",'W')    ; Open printer for writing
IF PRT = 0
  ABORT"#cUnable to open printer."
...
PUTF PRT, NL, "This line goes to the printer."
...
X=124.35
OUTPUTF PRT, "#cThe answer is #12.4R",X
...
```

Note that just because you were able to OPEN the printer successfully does not necessarily mean the printer is ready to receive output. If you OPEN the printer, send it output, and the system appears to hang, it may be that the printer is not on-line or ready to print.

For the **Commodore 64**, you must remember not to power on the printer while DYNODISK is on (also, for interfaces such as the CARDCO, the interface must be off too; for the CARDCO interface, this means that the single wire must be unplugged from the back of the computer while DYNODISK is on). You can turn DYNODISK on and off from within a program, if desired (see **Appendix G**).

When doing output to a printer, be sure to send a NL after the last line, since many printers keep the line in their internal memory until a CR is received to cause them to print.

PRINTER CONTROL

Printers vary considerably in terms of interface to the computer. To help reduce the difficulty in dealing with various printers and printer interfaces, PROMAL pre-defines several variables (in file PROSYS.S) to govern printer output.

For the **Apple II**, you can control whether or not PROMAL should automatically send a LF after every CR to your printer. See **APPENDIX E** for details. Also, if your computer is a IIc or is connected by a **serial** interface, you will need to set another variable to perform graphics or escape sequences. This is also described in **APPENDIX E**.

For the **Commodore 64**, printers (or printer interfaces) often have special modes selected on the basis of the "secondary address". The following three variables can be used to control your printer:

```
EXT ASM BYTE C64PSA AT $0DF3  ; Desired secondary address (default 7)
EXT ASM BYTE C64PUL AT $0DF4  ; Bit 7=1=flip case (default=$80=yes)
EXT ASM BYTE C64PDV AT $0DF5  ; C-64 printer device # (default 4)
```

These variables can be set by your program before opening the "P" device, or by direct commands from the EXECUTIVE or your BOOTSCRIPT.J file. See **APPENDIX E** for details.

PRINTER ESCAPE SEQUENCES

Most printers use ASCII control characters or escape sequences to select different attributes such as underlining, font selection, character size, etc. It is very easy to send these sequences to the printer using PROMAL PUTF statements, after you have set up your printer control options properly as described above. For example, if your printer manual tells you that the particular escape sequence you want is:

Escape sequence	Decimal form	BASIC form
ESC W 1	27, 87, 49	LPRINT CHR$(27);CHR$(87);CHR$(49)

then in PROMAL you could just write:

```
PUTF PRT, 27, 87, 49
```

assuming you have previously opened the "P" device with handle PRT.

For **Commodore 64** computers using interfaces such as the CARDCO, you may have to select some special mode before sending escape sequences to your printer. For example, the CARDCO model G+ needs to be opened with C64PSA=5 and C64PUL=0 (as described above) in order to select "transparent mode".

OUTPUT TO SCREEN AND PRINTER

Sometimes you may want to output the same text to the screen and the printer. This can be accomplished by executing the same PUTF or OUTPUTF statement twice, using different file handles. For example, the following program fragment supports selective output to either just the screen or to the screen and printer:

```
WORD SP [2]  ; File handles for screen, printer
WORD BOTHOUT ; =0 if just screen, 1 if screen + printer output wanted
WORD I
...
SP[0]=STDOUT ; screen file handle (already open)
BOTHOUT=0
PUT NL,"Do you wish output to printer too?"
IF TOUPPER(GETC)='Y'  ; yes?
  SP[1]=OPEN("P",'W') ; then open printer for writing
  BOTHOUT=1
```



```
...
FOR I=0 TO BOTHOUT
  PUTF SP[I],NL,"This will go to printer & screen if BOTHOUT=1",NL
...
```

ARGUMENT PASSING FROM THE EXECUTIVE

PROMAL provides a simple mechanism for passing command-line arguments from the EXECUTIVE to a program. The standard LIBRARY defines two globally predefined variables which are preset by the EXECUTIVE before control is passed to a program:

NCARG	is the number of arguments passed to the program.
CARG[1]	is a string containing the first argument, if present
CARG[2]	is a string containing the second argument, if present
...	
CARG[NCARG]	is the last argument.
CARG[0]	is a string containing the command which was executed (the command name)

For example, if your program is executed by the EXECUTIVE command:

```
DOIT  Myfile  2367
```

then on entry to the program,

NCARG	will be 2
CARG[1]	will be "MYFILE"
CARG[2]	will be "2367"
CARG[0]	will be "DOIT"

All the CARG array elements will be pointers to strings containing the arguments. The program should consider these strings as DATA and not modify them in place.

The EXECUTIVE normally treats blanks as the delimiters between arguments. Both leading and trailing blanks are stripped off the arguments, so any number of blanks may intervene between arguments. Also, the EXECUTIVE "folds" all lower case letters to upper case. However, if an argument is enclosed in quotes on the command line, then the entire quoted string is passed as a single argument, including blanks (if any), without folding alphabetic characters. The quotes themselves are stripped off. For example, if the command line was:

```
FIND "Now is the time for all good men"
```

then:	NCARG	will be 1
	CARG[1]	will be "Now is the time for all good men"

Command line arguments from the EXECUTIVE make a very useful way to pass file names or numeric values to a program. For example, here is a program segment which opens an input file specified on the command line for reading:


```
PROGRAM PROCESS
;   Program  segment to open a file passed as the first command line arg.

INCLUDE LIBRARY
WORD INFILE          ;Input file handle
BYTE LINE[81]        ;Buffer to hold a line from file
BEGIN
IF NCARG <> 1
  ABORT "#C***Error: PROCESS expects 1 argument which is a file name."
INFILE = OPEN(CARG[1])
IF INFILE = 0        ;open error?
  PUT NL, "*** Error: "
  CHOOSE IOERROR     ;error code from OPEN
  2
    PUT CARG[1]," is not a legal file name."
  3
    PUT "Disk drive not ready."
  4
    PUT CARG[1], " file not found."
  ELSE               ;unusual error of some kind
    PUT "Can't open ",CARG[1]
  ABORT
WHILE GETLF (INFILE,LINE)  ; read lines until end of file
  ...
```

Of course, you could make the error processing simpler if you wished, or make it more sophisticated (perhaps by giving the user a chance to try another file name), as is appropriate to the application.

EXTERNAL VARIABLES FOR ADDRESSING SPECIAL MEMORY LOCATIONS

In BASIC you use PEEK and POKE to examine or set special memory locations in your computer. With PROMAL, you can write the equivalent of PEEK and POKE as follows:

BASIC	PROMAL
X=PEEK(nnnn)	X=M[nnnn]
POKE nnnn,X	M[nnnn]=X

where nnnn is the address of interest. The array **M** is predefined in the LIBRARY to be an array of bytes encompassing all memory, so M[0] is the first byte of memory, and M[65535] is the last byte of memory.

However, there is an even better way to replace those PEEKS and POKES which is both more readable and more efficient. You can give those special memory locations a variable name of type BYTE, by declaring them to be **EXTernal** to your program. For example for the Apple II:

```
EXT BYTE HIRESON AT $C057
```

defines a variable named HIRESON of type BYTE which will be **assigned** the address $C057. This is the Apple "soft switch" for enabling graphics mode. Once defined, you can enable hi-res mode by merely saying,


```
HIRESON=TRUE
```

which conveys a lot more meaning than POKE -16297,1.

INTERFACING TO COMMODORE 64 SPECIAL MEMORY LOCATIONS

PROMAL is very well suited for taking advantage of the special hardware features of the Commodore 64, such as sprites, music synthesis, and color. It is far easier to program these fun-filled features with PROMAL than BASIC. In BASIC, you depended on a lot of incomprehensible PEEKS and POKES to access the special registers in the VIC-2 video chip and the SID sound synthesizer. With PROMAL, you can give these registers a variable name and manipulate them just like any other variable. This kind of variable is called an EXTERNAL variable, because it is located outside the PROMAL program.

For example, the BASIC statement,

```
POKE 53281,7
```

sets the screen background color to yellow. With PROMAL, you might choose to do the equivalent function like this:

```
CON YELLOW = 7
EXT BYTE BACKGROUND AT 53281  ;Screen Background color reg.
...
BACKGROUND = YELLOW
```

Once you have defined the address of the variable BACKGROUND, you can use it just like any other PROMAL variable.

Creating animation with sprites is much easier with PROMAL. For example, suppose you wanted to have a tank moving horizontally on the screen as one sprite and a bomb falling vertically as a second sprite. You might do this as follows:

```
EXT BYTE XCAR AT $D000     ;X position of sprite 0
EXT BYTE YCAR AT $D001     ;Y position of sprite 0
EXT BYTE XBOMB AT $D002    ;X position of bomb
EXT BYTE YBOMB AT $D003    ;Y position of bomb
...
XCAR = XCAR + CARSPPEED    ;move car to right
YBOMB = YBOMB + BOMBSPEED ;move bomb down (+ is down)
...
```

In this case the address of each external variable was specified in hexadecimal, which is frequently more convenient.

You can also directly manipulate screen memory or color memory as a PROMAL array. For example, suppose you wanted to clear the standard screen, and then "paint" 16 bars across the screen, each in a different color:


```
PROGRAM RAINBOW
INCLUDE LIBRARY
CON SCREENSIZE = 1000           ; # of bytes in screen memory
EXT BYTE SCREEN [] AT $0400     ; C-64 screen memory location
EXT BYTE COLOR [] AT $D800      ; Color RAM
WORD I
BEGIN
FILL SCREEN, SCREENSIZE, ' '    ; fill the screen with blanks
FOR I = 0 TO 15
  FILL SCREEN+40*I, 40, $A0     ; one line of reverse-video blanks
  FILL COLOR+40*I, 40, I        ; set corresponding color for line
END
```

Notice that an external array declaration does not specify the size of the array inside the brackets. This is because PROMAL does not need to reserve any space within the program for this array (and because PROMAL does not do any bounds-checking on array references because this would adversely affect execution speed). The procedure FILL is a built-in LIBRARY subroutine which fills a portion of memory with a specified byte. Its first argument is the starting address, the second is the number of bytes to fill, and the last argument is the fill character.

Just for fun, let's compare the above program segment to its equivalent BASIC program:

```
 90 SC=1024: SZ=1000: CO=55296
100 FOR I=SC TO SC+SZ: POKE I,32: NEXT
110 FOR I=0 TO 15
120 FOR J=40*I TO 40*I+39: POKE SC+J,160: NEXT
130 FOR J=40*I TO 40*I+39: POKE CO+J,I: NEXT
140 NEXT I
```

If you run the BASIC program and the PROMAL program above, and time how long each takes to clear and paint the screen, you will find:

```
BASIC....about 14 seconds.
PROMAL...about 0.1 seconds.
```

This is another reason why PROMAL is much better than BASIC for animated graphics. PROMAL is much faster. While not every PROMAL program will be 140 times faster than its BASIC counterpart as in this example, speed increases of 20 to 100 times or more are commonplace. Also, the larger and more complex the program, the greater will be the relative speed improvement compared with BASIC. Using the built-in LIBRARY subroutines wherever possible will speed up your PROMAL programs even more, as well as making them smaller and easier to debug.

Several PROMAL demonstration programs making extensive use of the graphics and sound capabilities of the Commodore 64 can be found on the Commodore disk. You can learn a lot about PROMAL from studying these samples and improving them or changing them to suit your own taste.

PROMAL INTERFACE TO MACHINE LANGUAGE

Many BASIC programs have to resort to calling machine language subroutines for some specialized functions. Usually you need machine language routines because (1) BASIC is too slow, or (2) BASIC can't do what you wanted. Because PROMAL is so much faster than BASIC, and because it provides bit-level operators, BYTE data types, and EXTernal variables, you may never need any machine language at all with PROMAL.

If you **do** decide you need to call a machine language routine, PROMAL makes it much easier to do than BASIC. PROMAL provides a clean interface to machine language, both for routines you write and for ROM resident routines in your computers operating system. You can call any machine language routine in ROM without writing any machine language interface code at all. You can call machine language routines by name, with passed arguments, just like regular PROMAL routines. You can even specify the contents of the hardware registers and test the results when the machine language routines return (including the flags). You can embed machine language routines inside PROMAL programs using DATA statements or load them from separate files under program control. You can also call all the built-in PROMAL LIBRARY routines from your machine language routines.

Appendix I describes how to use and write machine language routines, with examples.

CHAPTER 7: STRINGS AND ARRAYS REVISITED

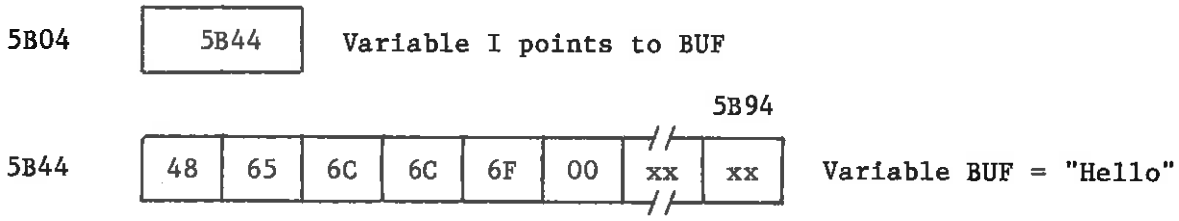
Earlier you saw how to declare and use strings and arrays. This section provides additional, more detailed information on using strings and arrays, especially multi-dimensional arrays and arrays of strings.

STRINGS

PROMAL always stores a character string as an array of bytes, one character per byte, plus a zero byte terminator indicating the end of the string. Strings are usually manipulated by specifying the address of the first character of the string. This is very convenient, since referring to an array name automatically generates a reference to the address of the first element. For example:

```
WORD I
BYTE BUF[81]  ; Input line string
...
GETL BUF      ; Input line from keyboard as string
I=BUF         ; I points to string
```

Assuming that this program was executed and that the user entered "Hello" from the keyboard, the memory for the variables might look like this, assuming some arbitrary addresses for the variables (shown in hex):



The "xx" in the diagram above means "don't care" or "undefined".

A big advantage of this representation of a string is that you can use the same array to either refer to the whole string, or to access single characters from the string. For example:

```
PUT BUF
```

will display "Hello", because the PUT procedure is passed the address of the string ($5B44 in the diagram above). Anytime you write the name of an array without any subscripts, the compiler uses the starting address of the array. You could extract a single character from BUF because it is an array of bytes. For example:

```
PUT BUF[1]
```


will display the character "e", the second character of the array (the first character is in BUF[0]). Alternatively, you could write **PUT (BUF+1)@<**, which would give the same result, because the expression will extract the value of the byte at $5B45 and print it. On the other hand, PUT BUF+1 would display "ello", because the value $5B45 would be passed to PUT instead of the contents of $5B45.

Remember that you can't use an assignment statement to copy a string from one variable to another (you need to use the MOVSTR procedure), but you can assign the address of a string to a word variable using an assignment statement. Therefore the the statements:

```
WORD I
BYTE BUF[81]
...
GETL BUF
I=BUF
PUT I
```

would cause whatever line was typed to be printed out. However, if these statements were followed by:

```
MOVSTR "Gone.", BUF
PUT I
```

then "Gone." would be printed, because I contains the address of BUF.

Similarly, you can't compare two strings with the ordinary comparison operators. For example:

```
DATA WORD BUF1 = "Hello"
DATA WORD BUF2 = "Hello"
...
IF BUF1 = BUF2  ; Wrong!
  PUT "Strings are the same"
```

will never print anything because the string given by BUF1 will never have the same address as the string given by BUF2, even though the contents of the strings are the same. This should be written as:

```
IF CMPSTR (BUF1, "=", BUF2)
  PUT "Strings are the same"
```

ADDRESS OF AN ARRAY ELEMENT VERSUS CONTENT OF AN ARRAY ELEMENT

It is very important to understand the difference between the address of an array and the value of an element in an array. Remember that almost all the built-in Library functions and procedures for string handling expect the **address** of the string for an argument (if the description of the routine says it expects a string, this means the **address** of the string). If you pass a single character where a string address is expected, you might create big problems! Here's an example:


```
BYTE BUF[81]
...
GETL BUF[0]     ; This is wrong!
...
PUT NL, BUF
```

Here the programmer simply intended to read in a line from the keyboard into the array BUF starting at the first element. The program compiles and appears to work, but always prints out garbage. In fact, sometimes it crashes the computer. Why?

The problem is that GETL expects the **address** of the buffer to receive the input, but the expression **BUF[0]** evaluates as the **value** of the first character of the buffer. In other words, whatever character is in BUF[0] when the call is made (some garbage value between 0 and 255) is passed to the GETL routine as the **address** of where you want the input line to go. GETL obliges by putting the input line someplace in the first 256 bytes of memory - but **not** in the BUF array. When BUF is printed by the PUT statement, it shows garbage, because it had never been set! Because the first 256 bytes of memory is "zero page" and holds critical operating system and PROMAL information, overwriting it may cause the computer to crash, necessitating a re-boot.

How do you fix this? In this case, the easiest way is simply to write:

```
GETL BUF
```

without the subscript, since PROMAL will always use the address of the array if you write its name without subscripts.

But what if you don't want the line to go right at the beginning of the array? Suppose, for example, you have a two dimensional array such as:

```
BYTE SCRN [81, 25] ; 25 lines of 80 characters each
```

Now suppose you want to input a line from the keyboard into the third row of the array. Here is how you do this:

```
GETL #SCRN[0, 2]  ; Read line from keyboard to 3rd row
```

The **#** operator tells the compiler to generate the **address** of the specified array element rather than the value of that element. The third element subscript is 2 instead of 3 because the first element is always 0, not 1. More importantly, remember that the last element is 24, not 25. If you try to read in the 25th line using:

```
GETL #SCRN[0, 25], 80  ; Wrong! Out of bounds!
```

GETL will oblige you by reading the line into memory over whatever happens to be in memory after the end of the SCRN array! This will have unpredictable and invariably unpleasant results.

To summarize, if a PROMAL Library routine (or any PROMAL subroutine for that matter) expects a string, you need to specify an address. If in doubt about how to make an address, place the **#** operator in front of the variable. If **X** is an array, then the following three statements are exactly equivalent:

```
PUT X
PUT #X
PUT #X[0]
```

All three statements will print the string which starts at the location of the X array (and is terminated by a zero byte). The following statement is **not** equivalent:

```
PUT X[0]
```

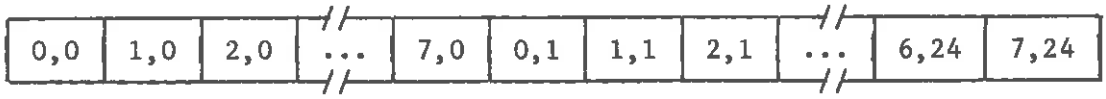
This statement prints only the first character of the string, because the expression will evaluate to the value of the first byte of the array X (which is assumed to be an ASCII character code). PUT is one of the few routines that can accept a single character or a string. If the argument is less than 256, PUT assumes the argument is a single ASCII character and prints it. If the argument is greater than 256, PUT assumes the argument is the address of a string to be printed.

SEQUENCE OF MULTI-DIMENSIONAL ARRAY ELEMENTS IN MEMORY

When using multi-dimensional arrays, note that the array elements with the **first** subscript will be adjacent in memory, so you should have the column subscript first and the row subscript second. For example:

```
BYTE PAGE [9,25]  ; Room for 25 lines of 8 chars each
...
PUT PAGE [0,5]  ; display single character on 6th line, first col
PUT #PAGE [0,5] ; display entire string of 6th line
```

Subscripts for the page array are allocated like this in memory:



Another way to look at this is to say that an array declared as:

```
WORD STUFF [10, 50]
```

declares 50 groups of 10 words each, **not** 10 groups of 50 words each. This distinction becomes important if you use BLKMOV to move part of the array or FILL to clear part of the array.

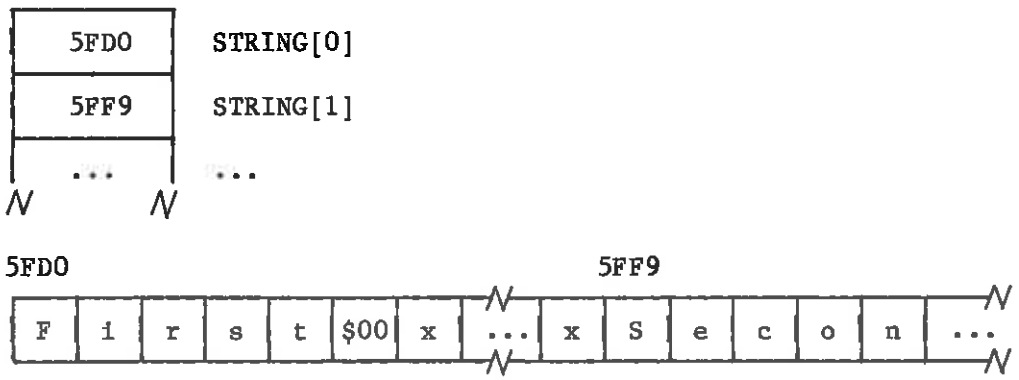
ARRAYS OF STRINGS

Sometimes, especially for sorting, you may wish to access an array of strings. Using an array of strings is usually more efficient than using a two dimensional array of bytes. The basic idea here is to use an array of type WORD which contains pointers into a singly-dimensioned array of type BYTE. This is especially true if you are simply going to sort the strings, because when a string is out of sequence, you can just exchange the pointers instead of exchanging entire strings. The sample program SORTSTRING.S provides a general purpose string sort routine which can sort an arbitrary array of strings passed as an argument.

Here is a program fragment showing how to develop an array of strings by reading them from the keyboard or a file. Input is terminated on end of file (CTRL-Z from the keyboard).

```
WORD STRING[100]  ; Array of strings (pointers into BUF array)
BYTE BUF[4100]    ; Storage for up to 100 strings of 40 char each
WORD I
WORD NSTRING      ; Actual number of strings (not exceeding 100)
...
NSTRING=0
I=BUF
WHILE GETLF(STDIN, I, 40) ; Read string to address I
  STRING[NSTRING]=I       ; Install pointer in string array
  I=I+41                  ; Starting addr of next string
  NSTRING=NSTRING+1
...
```

At the end of this program fragment, memory might look like this, assuming the lines read were "First", "Second", etc.



The SORTSTRING demo program uses a similar technique, but makes more efficient use of the BUF array. You may wish to use the sorting routine provided in SORTSTRING for your own programs.

PRESETTING GLOBAL VARIABLES TO ZERO

Unlike BASIC, PROMAL does not assign any initial value to variables declared in your program (except DATA, of course). Often you may wish to simply set all or a large number of variables to zero at the start of your program. Rather than writing assignment statments for each variable explicitly, here is a trick which will zero a block of variables. Assume you have a group of variables declared like this:

```
WORD FIRSTVAR
...
BYTE LASTVAR [1]
```

where FIRSTVAR is the first variable you want to zero and LASTVAR is a dummy variable you add after the last variable declared. At the start of your program, use:

```
FILL #FIRSTVAR, #LASTVAR-#FIRSTVAR, 0  ; Zero all variables
```

This will set all the variables from FIRSTVAR up to (but not including) LASTVAR to zero. Don't forget the # operators! Also note that LASTVAR is an array. This is necessary in the Commodore and Apple versions of PROMAL because the PROMAL compiler segregates scalar and array variables. It assigns addresses for all the scalar variables first (in the order declared), and then all the arrays (in the order declared). DATA variables are part of the code area of your program, not part of the data, so you don't have to worry about accidentally zeroing the value of any DATA identifiers.

CHAPTER 8: THE LOADER

INTRODUCTION TO THE PROMAL PROGRAM LOADER

PROMAL for Apple and Commodore 64 gives you, the programmer, the ability to control the loading and execution of programs. Your PROMAL program can load and run other PROMAL or machine language programs, or pieces of programs called **overlays**. Your program can also largely control **where** programs are loaded into memory, and specify what action should be taken when a program or overlay completes its task. Programs can call subroutines in other programs and use global variables in other programs previously loaded, and you can explicitly select which subroutines and variables can be used.

These capabilities are provided by the built in library procedure **LOAD**, which is used to control the loading, execution, and disposition of compiled programs and overlays. This procedure is extremely powerful, much more powerful than the simple "chaining" capability provided by BASIC and some other languages. To use it effectively requires an understanding of the loading and execution process as used by PROMAL, plus some new terminology. The remainder of this section deals with the LOADer. You may wish to skip over this section until you are familiar enough with PROMAL to be writing large programs.

DEFINITIONS

The following definitions are relevant to this section. The meaning of these terms will become clearer as the discussion develops.

A Module is the object file produced by the PROMAL COMPILER (with no error messages), with a .C extension, or a relocatable machine language program as generated by the RELOCATE program (discussed in **Appendix I**).

An Entry Point is the place where execution begins in a module. In a PROMAL source program, the entry point is represented by the BEGIN statement following the last procedure or function in the source program.

A Logical Program is a collection of one or more modules which, taken together, comprise a logically complete program for some purpose. A logical program may have several modules, each residing on disk in a separate file. As a minimum, a logical program has one module.

A Program is a compiled PROMAL program. Normally it performs a complete task by itself and is composed of an arbitrary number of procedures and functions, with exactly one entry point, which is at the BEGIN statement following the last procedure or function. The program source file begins with a PROGRAM statement, is compiled from one or more source files, and the resulting output module is contained in a single object file with a .C extension.

An Overlay is a piece of a complex logical program which is kept on disk until it is needed, and is then loaded into memory and executed under program control. The overlay source file begins with an OVERLAY statement, and is otherwise similar to a program. It has an arbitrary number of procedures and functions and exactly one entry point, which is at the BEGIN statement following the last procedure or function. It must be compiled separately from the rest of the logical program which is associated with it, from one or more source files, and the resulting output module is contained in a single object file with a .C extension.

Loading is the process of taking a PROMAL module from disk and copying it into memory, making any adjustments (called relocations) to the program which are needed to correct addresses in the program or interface to other modules, and transferring control of execution to the program, if desired.

Chaining is a special kind of loading where the program being loaded replaces the program which called the loader.

BREAKING UP A LOGICAL PROGRAM INTO MODULES

There are several reasons why you might want to have a logical program composed of several modules instead of a single, monolithic program:

1. The program is too large to fit in memory all at once.

2. The program is composed of logically separate modules (for instance, an accounting system might have a main menu with separate modules for receivables, payables, order processing, report generation, etc).

3. The program uses a logically-related group of subroutines which are not frequently changed and therefore do not need to be re-compiled (for example, the PROMAL graphics package or real functions).

4. The program takes too long to compile in its entirety.

5. The program uses large machine language routines (this is discussed in Appendix I).

HOW THE PROMAL LOADER WORKS

The PROMAL LOADer is a built-in procedure in the Library, called the same way as other library routines. You have already seen the PROMAL loader working, at least indirectly. When you type the name of a program you want executed from the EXECUTIVE, the EXECUTIVE calls the LOAD procedure to run your program. When your program finishes, it returns through the LOADer to the EXECUTIVE at the point from which it was called.

Your programs can in turn load and run other programs or overlays, by specifying the name of the program to run, and optionally some flags indicating **how** the program should be run. Briefly, the LOADer performs these tasks:

1. Looks to see if the specified program or overlay is already in memory, and if so, executes it beginning at the entry point. Otherwise, it:

2. Locates the specified module on disk and determines how much memory is needed for your program and its variables. If there is not enough room, it unloads other programs (unless the module is an overlay) until there is enough room.

3. The memory image of the module is copied from disk into the available memory space.

4. The loader then reads tables which follow the memory image on disk to determine what adjustments are necessary to the memory image. These adjustments are called relocations, and are needed to install the correct addresses for branch instructions and subroutine calls. The loader can also adjust address references to other modules already loaded (using EXPORTS and IMPORTS, discussed later).

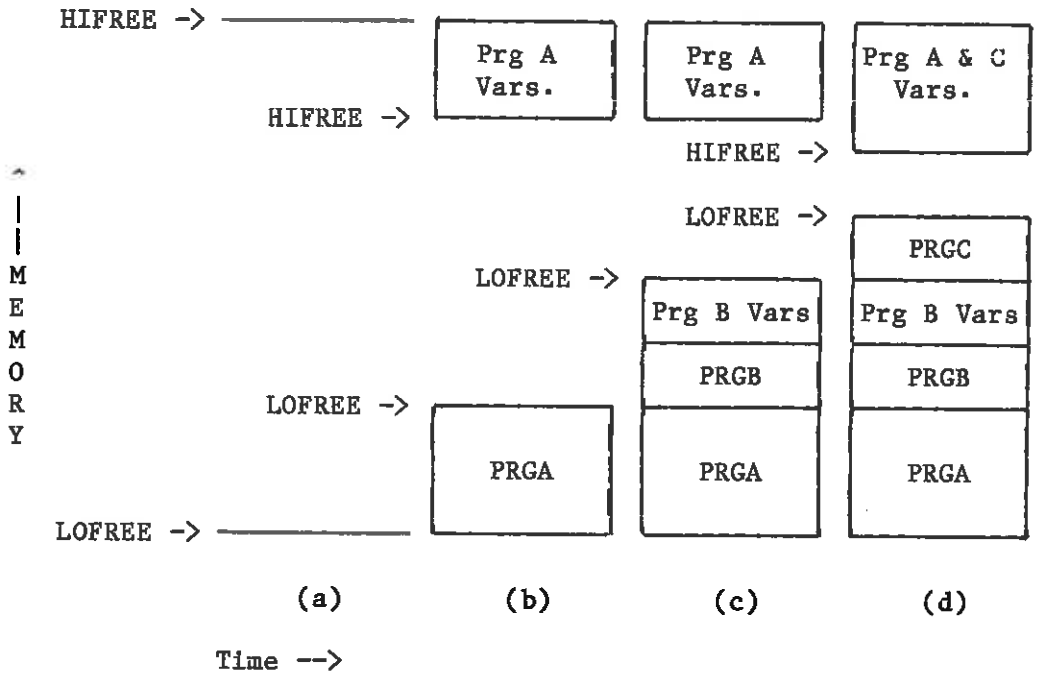
5. The loader then begins execution of your program at the entry point.

6. When your program completes (by coming to the END of the main program, calling EXIT or ABORT, or encountering a runtime error), the loader temporarily regains control. It normally transfers control back to the program which called the loader at the statement following the call to the loader. However, if your program called ABORT or encountered a runtime error, control is passed directly back to the EXECUTIVE instead.

The program which calls the LOADer to execute another program is called the **parent** of that program. The new program is called the **offspring** of the program that called the loader. The loader keeps track of the modules currently in memory by a series of tables. These tables have room enough for up to **six** modules to be resident in memory at once, plus the EXECUTIVE and EDITOR. These modules may be all part of one logical program, completely separate programs, or any combination.

The LOADer also uses several pointers for memory management. The most important of theses are called **LOFREE** and **HIFREE**. LOFREE always points to the first byte of unused memory, and HIFREE always points to the byte **after** the last unused byte of memory. These pointers always point to a page boundary in memory (that is, the address is of the form $XX00). Normally the LOADer allocates programs from the bottom of available memory up, and variables from the top down. Normally variables from one module can occupy the same memory as variables for another program since the variables have no initial value and the programs are not related. However, the key word OWN on the PROGRAM or an OVERLAY declaration of a program can be used to force the loader to allocate the variables immediately after the program, not shared with any other programs. OVERLAYs always have their variables allocated immediately after the overlay code.

The following **memory diagram** shows a series of programs being loaded from the EXECUTIVE:



Time is represented on the horizontal axis and memory on the vertical axis, with the highest addresses at the top. The diagram represents the memory configuration for the Apple. The Commodore 64 configuration is somewhat more complicated (see **Appendix G**), because the Workspace is also managed by the LOADer, but the principle is the same. In this diagram, there are initially no programs in memory (except the EXECUTIVE/EDITOR, not shown). Then PRGA is executed (part b of the figure), which is a normal module. PRGB is then executed. PRGB has OWN on its program line, so its variables are allocated after the code for PRGB instead of sharing its variable space with PRGA. Finally, PRGC is run, which is another normal program, and shares its variable space with PRGA. Since PRGC requires more variable space than PRGA, HIFREE is lowered by the loader. HIFREE will always point to the start of the variables for whatever program requires the largest block of shared variable space.

In this example, all the programs were small enough to fit in memory at once. If PRGC had been too large to fit, the loader would first unload PRGB and its variables and try again. If there was still not enough room, it would unload PRGA. When a program is unloaded, the LOADer simply deletes its table entry and moves the LOFREE pointer down (and the HIFREE pointer up, if possible), to recover the space. It does not clear the memory recovered.

HOW TO CALL THE LOADER

The declarations needed to access the LOADer are in a file called PROSYS.S on the PROMAL system disk. Therefore you should have:

```
INCLUDE PROSYS
```

near the top of any program which will be calling the loader (or, if you wish, you can extract the definitions from PROSYS and insert them directly into your program with the EDITor).

The loader is a built-in procedure in the PROMAL library, which you call with a statement of the following form:

LOAD Progname [,Bitflags]

where **Progname** is a string containing the desired module name (without the file extension), and **Bitflags** is an optional argument of type BYTE which contains several flags, described shortly. If the Bitflags argument is not specified, it defaults to 0, for a normal load-execute-return sequence. The loader also sets a special variable called LDERR (also defined in PROSYS), as follows:

LDERR #	Meaning
0	No error, module was successfully loaded/executed.
1	Module was not found in memory or on disk (or the name is illegal)
2	Not a valid PROMAL module (e.g., not a successfully compiled program).
3	Not enough free memory to load program.
4	Module required not loaded or relocation error (e.g., the module to be loaded calls a subroutine in another module which is not loaded).

For example,

```
PROGRAM MYPROG
INCLUDE LIBRARY
INCLUDE PROSYS
...
BEGIN
...
LOAD "YOURPROG"
IF LDERR <> 0
  ABORT "#C Unable to load YOURPROG"
...
END
```

will cause the module YOURPROG.C to be loaded into memory (if it is not already there) and executed. After YOURPROG ends, control will return to the IF statement following the call, which tests for a loader error (such as file not found). If, however, YOURPROG called ABORT or encountered a runtime error, control would never return to the IF statement above, but would return directly to the EXECUTIVE instead.

LOADER OPTIONS USING BIT FLAGS

The second, optional argument of type BYTE can be used to specify a variety of options which control the loading process. This byte is treated by the LOADer as several one-bit TRUE/FALSE flags. These flags are given names in PROSYS, defined as follows:

Name	Definition	Meaning to LOADer
LDCHAIN	$01	Chain to program. If TRUE (1), the calling module should be unloaded and replaced with the new module. When the new module ends, control should return to the **parent** of the calling module.
LDPRCLR	$02	Pre-clear memory. If TRUE (1), all programs in memory (including the caller) should be unloaded before loading the specified program. Control will be returned to the EXECUTIVE. This option is usually used to guarantee the maximum available memory for a program.
LDRELD	$04	Re-load module. If TRUE (1), the specified module should be reloaded from disk, even if it already is in memory. If FALSE (0), it will not be reloaded from disk unless it is not already loaded or the memory-resident copy has been corrupted. Note that specifying LDCHAIN=1 or LDPRCLR=1 also implies LDRELD=1 automatically.
LDRECLM	$08	Reclaim memory on exit. If TRUE (1), then the specified module should be unloaded from memory after it completes execution. This option is normally used for overlays to make room for other overlays in the same memory space. If 0, the module will remain loaded on completion and can be re-executed by a subsequent LOAD without having to access the disk.
LDNOGO	$10	Do not execute. If TRUE (1), then the specified module will be loaded into memory (if it is not already loaded) without executing it. This option is normally used to insure that a module is loaded and ready for later execution. It is also used to control the sequence of loading of multiple modules in a complex logical program. If 0, the specified module will be executed.
LDUNLD	$20	Unload. If TRUE(1), then the specified module is unloaded instead of loaded. No other action takes place and all other bit flags are ignored. Note that any program loaded above the specified module will also be unloaded. If the calling module is itself unloaded as a result of this process, control will return to the parent program instead. This option is normally used to free up additional memory.

The bit flags above can be combined in any sensible combination. For example:

```
LOAD "HISPROG", LDCHAIN + LDRECLM
```


will remove the calling program from memory, load and execute HISPROG, then remove HISPROG from memory and return control to the parent of the original caller (probably the EXECUTIVE).

```
LOAD "NEXTCOMD", LDPRCLR+LDNOGO
```

will unload all programs from memory, load NEXTCOMD without executing it, and return control to the EXECUTIVE (you might want to do this to setup the next program to be run in memory).

USING VARIABLES, PROCEDURES AND FUNCTIONS IN OTHER MODULES

One of the most powerful features of the PROMAL LOADer is that when a module is loaded and executed, it can call selected procedures and functions and access selected variables in other modules which are already loaded. It cannot, however, reference procedures, functions, or variables in modules which have not been loaded yet. This is a logical extension of the rule that functions, procedures and variables must be defined before they are referenced. It is up to you, the programmer, to determine which procedures, functions, and variables will be made available to other modules. This is done using EXPORTs and IMPORTs.

EXPORTS AND IMPORTS

In a PROMAL source program, the key word **EXPORT** can be used in front of any declaration of a constant, data declaration, variable, procedure, or function to designate an item which should be made available to other modules which wish to use it. If your program contains **any** EXPORTs, it **must** also have the keyword EXPORT on the PROGRAM (or OVERLAY) line. OWN should also be specified.

For purposes of illustration, let us assume we will have a logical program composed of two separate modules. The first module is a collection of subroutines which you frequently use, called SUBPKG, and the other module is a particular application program called MYPROG, which will use some routines in SUBPKG (and in addition has some procedures, functions and global variables of its own). Assume you wish to compile the subroutine package and MYPROG separately, because SUBPKG is already well-debugged and is fairly large. Therefore during development of MYPROG you will not have to re-compile SUBPKG each time you make a change to MYPROG, saving time. Here is a skeletal view of the source for SUBPKG:

```
PROGRAM SUBPKG OWN EXPORT
INCLUDE LIBRARY
...
WORD I
EXPORT WORD CLEARANCE
EXPORT CON WORD POOLSIZE=500
EXPORT REAL POOL[POOLSIZE]
REAL THRESHOLD
EXPORT DATA REAL PI = 3.1415926535
EXPORT DATA WORD ERRMSGS [] =
"Pool exhausted", "Undefined pool element", "Illegal pool element",0
...
```



```
...
EXPORT PROC ADDTOPOOL ; Item
ARG WORD ITEM
...
END
...
FUNC REAL BESTGUESS
...
END
...
EXPORT FUNC BYTE CHECKERROR
...
END
...
BEGIN
...
END
```

This example illustrates how a subroutine package might make selected identifiers available for use by other, separately compiled modules. In this case, the names CLEARANCE, POOLSIZE, POOL, PI, ERRMSGS, ADDTOPOOL, and CHECKERROR will be exported. The names I, THRESHOLD, and BESTGUESS will not be available for use by other modules. In other words, the subroutine BESTGUESS can be called by other routines in this module (PROGRAM SUBPKG), but not by other separately compiled modules.

When a program contains EXPORTs, the PROMAL COMPILER writes the definitions of **all** the exported items to a special text file at the completion of compilation. This text file will have the same name as is on the PROGRAM declaration in the source file, but with a .E extension. For example, the program above would cause the compiler to generate an export file called **SUBPKG.E**, which might look like this:

```
IMPORT SUBPKG ;10/17/85
  EXT FUNC BYTE CHECKERROR AT $0562
  EXT PROC ADDTOPOOL AT $0250
  EXT DATA WORD ERRMSGS [4] AT $000D
  EXT DATA REAL PI AT $0007
  EXT REAL POOL [500] AT $0000+$4
  CON WORD POOLSIZE = $01F4
  EXT WORD CLEARANCE AT $0002
```

This file tells the definitions of the exported identifiers, relative to the start of the SUBPKG module. It is not necessary to understand the exact meaning of the individual lines. The top line tells the name of the exporting program and the compilation date.

IMPORTING DEFINITIONS

Once the export file has been written by the COMPILER, you may INCLUDE it in the compilation of another, separate module, to import all the desired definitions. For example, another separately-compiled program which uses the SUBPKG module might look like this:


```
PROGRAM MYPROG
INCLUDE LIBRARY
...
INCLUDE SUBPKG.E
...
WORD K
...
PROC AJUSTPOOL
BEGIN
...
POOL[K] = PI/4.
ADDTOPOOL
...
END
...
BEGIN
...
IF K > POOLSIZE
  PUT NL,ERRMSGS[0]
...
END
```

Notice that this program uses the procedure ADDTOPOOL, the data items PI and ERRMSGS, and the constant POOLSIZE without ever explicitly declaring them. This is possible because the INCLUDE SUBPKG.E will cause the definitions to be imported. Please note that you must specify the .E extension on the INCLUDE statement; otherwise, the compiler will look for the file SUBPKG.S instead by default.

EXECUTING THE LOGICAL PROGRAM WITH SEPARATE MODULES

After compiling MYPROG, you will have two separate modules which work together: SUBPKG.C and MYPROG.C. If you attempt to execute MYPROG.C from the EXECUTIVE, you will get the message:

```
NOT LOADED OR RELOC ERROR: SUBPKG
```

This is because the SUBPKG module must be loaded before the MYPROG module which calls it, and you haven't loaded it. To solve this problem, you could type:

```
UNLOAD
GET SUBPKG
MYPROG
```

which would unload any existing programs (to make sure there's enough room for both modules), load the SUBPKG module, and then load and execute the MYPROG module. The LOADer is able to relocate all the references to routines in SUBPKG correctly because (1) it knows where it loaded SUBPKG into memory, and (2) it knows the definitions of the references to the exported items in SUBPKG as a result of the compilation of MYPROG.

In the example above, it is important to understand that the exported routines in SUBPKG can be called from the MYPROG module, but that no routines in MYPROG may be called from SUBPKG, even if you EXPORT them. This is because the module doing the exporting must always be loaded before the module doing the importing, and it is clearly impossible for both modules to be loaded first!

USING A BOOTSTRAP TO CONTROL LOADING

If you have an application program with several modules which need to be loaded in a certain order, you may want to write a **bootstrap** program, whose job is to load all the needed modules in the proper sequence and then run the main program. Typically a bootstrap program might do this:

1. Display a signon message and any information (such as a menu of choices) relevant to the program, that the user can read while the rest of the program is loading from disk.

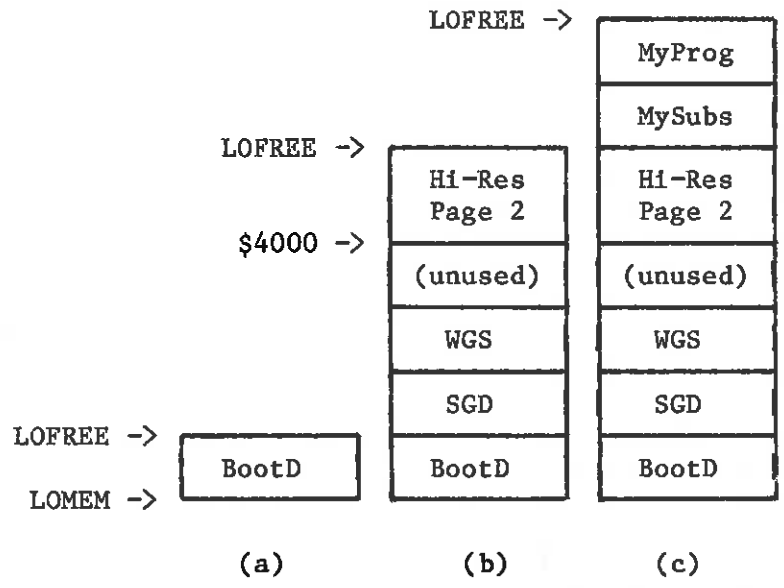
2. Load the modules needed in the desired order, using the LDNOGO option on each LOAD call to prevent execution.

3. Load and execute the main module.

In some cases, you may even want to use a two-stage bootstrap loader, where the first stage bootstrap loader signs on and then LOADs the second bootstrap stage loader with the LDPRCLR option to insure all possible memory is available for the application. The second bootstrap then loads all the modules needed to get the program going, in the correct order to resolve all the dependencies.

In some cases you may also use the bootstrap loader to directly manipulate the LOFREE pointer before LOADing some modules to reserve certain areas of memory. For example, on the Apple II using graphics you may wish to load part of your logical program below the 8K graphics page from $4000 to $6000, and part above it. The following memory diagram, with program fragment to the right, illustrates this (and other examples may be found in the PROMAL GRAPHICS TOOLBOX Manual).

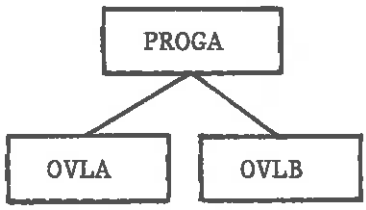
After programs BOOTD, MYSUBS, and MYPROG have been compiled (in that order), then executing BOOTD from the executive will cause BOOTD to load the SGD and WGS modules, as shown in (b) above, then sets LOFREE to $6000 under program control to reserve the Apple Hi-Res screen area, and finally loads the MYSUBS and MYPROG module to begins execution of MYPROG. Naturally you should adjust the LOFREE pointer only with a good understanding of what you are doing!




```
PROGRAM BootD
INCLUDE PROSYS
...
LOAD "SGD",LDNOGO
LOAD "WGS",LDNOGO
LOFREE=$6000 ;end hires
LOAD "MySubs",LDNOGO
LOAD "MyProg"
END

PROGRAM MySubs OWN EXPORT
...
INCLUDE SGD.E
INCLUDE WGS.E
EXPORT PROC MyProc
...
END

PROGRAM MyProg
...
INCLUDE SGD.E
INCLUDE WGS.E
INCLUDE MySubs.E
...
MYPROC
...
END
```

USING OVERLAYS

In the previous example, separate compilation was used primarily as a convenience. Sometimes, it is a necessity. This usually happens when a logical program is simply too large and complex to fit into the available memory space all at once. When this happens, the usual solution is to use overlays. A logical program which uses overlays will have one or more modules which remain resident in memory throughout execution, and will switch other modules in and out of memory as needed. A typical overlaid program might be organized like this:



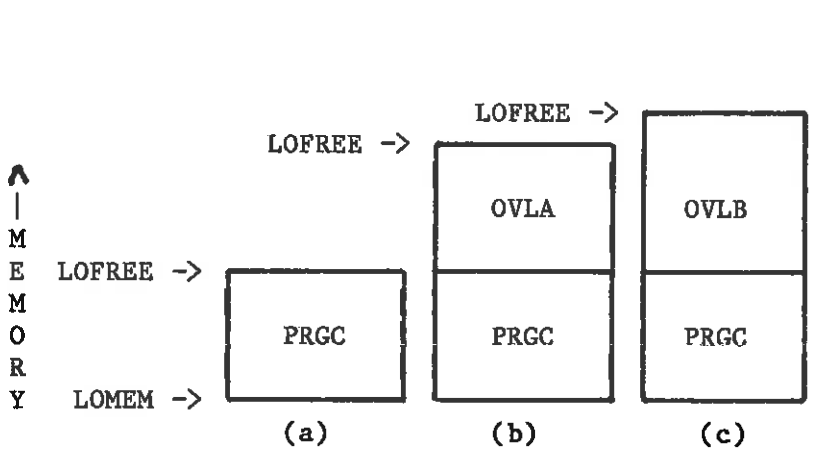
This tree diagram indicates that the logical program has a **root module**, PROGA, and two mutually-exclusive overlays, OVLA, and OVLB. By mutually exclusive, we mean that only one of the overlays will be in memory at any given time. Therefore the overlays can all share the same memory space, so that the total space needed will only be equal to the size of the largest overlay, instead of the sum of the overlays.

As far as the source code for an overlay is concerned, the only difference between a program and an overlay is that the first line of the program should contain the key word OVERLAY instead of PROGRAM, for example:

```
OVERLAY OVLA
```

The OVERLAY keyword also has the effect of including the OWN keyword on the program declaration line. Otherwise, the loader would allocate the variables belonging to the overlay right on top of the variables used by the rest of the logical program, probably producing a disaster.

As far as the loader is concerned, there is only one difference between a PROGRAM and an OVERLAY. The LOADer will **not** automatically unload any modules to make room for an overlay. If it did, there would always be the possibility that the loader would have to unload the calling module to make room for the overlay. This is contradictory to the normal use of overlays, which normally return to the parent module when completed.

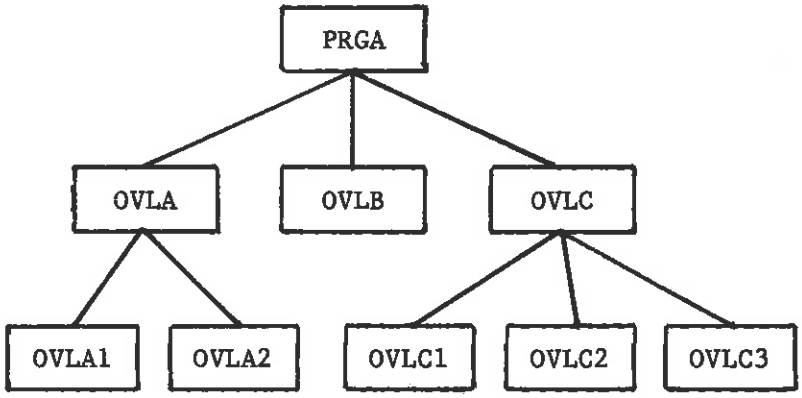



```
PROGRAM PRGC OWN EXPORT
INCLUDE PROSYS
...
LOAD "OVLA", LDRECLM
...
LOAD "OVLB"
...
END

OVERLAY OVLA OWN
INCLUDE PRGC.E
...
END

PROGRAM OVLB OWN
INCLUDE PRGC.E
...
END
```

In our example program with 2 overlays, the memory diagram might look as shown above, with the program skeleton to the right.

Note that the LDRECLM option was specified on the call to the LOADer. This is the normal way to load an overlay which will be replaced by another overlay later. Remember that the LOADer will not unload anything (including another overlay) to make room for an overlay; therefore you will probably want to specify LDRECLM to insure that the overlay is unloaded when it is completed. Of course, there is always the possibility that the root module might want to call the same overlay again. In this case, you might want to consider leaving the overlay in memory when it completes. If you need it again, the LOADer won't have to actually load it. If you need to replace it with a different overlay instead, you can unload it explicitly using the LDUNLD option, before loading the desired overlay.

The sample program fragment above had only two overlays. In a complex application, there might be several overlays, or even two layers of overlays, as shown by the tree below:



In this case, the overlays OVLA, OVLB and OVLC might export variables and subroutines to the five overlays at the bottom of the diagram. In this case, you would need to have the keyword EXPORT on the OVERLAY declaration:

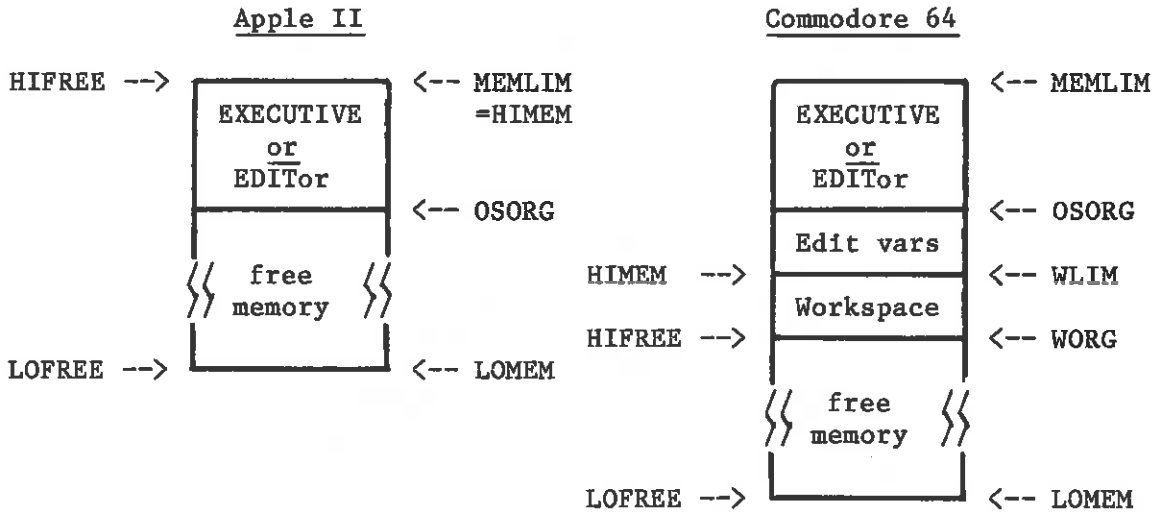
```
OVERLAY OVLA EXPORT
```

CONSIDERATIONS FOR THE EXECUTIVE AND EDITOR

This section tells you how it is possible to load larger programs under program control than it is possible to LOAD using the EXECUTIVE GET command, by overwriting the space usually reserved for the EDITor. The system pointers referred to below are defined in PROSYS.S. Further memory map information is included in Appendix G.

As was indicated before, the LOADer always considers the free, allocatable space to be between LOFREE and HIFREE. The EXECUTIVE and EDITor **both occupy the same address space**, which is OSORG through MEMLIM (about 11.5K bytes), but only one or the other of these programs occupies this space at any one time.

For the Apple II, a copy of the EDITor and the EXECUTIVE is kept in the extra 64K memory bank, and each is copied back into the normal address space at OSORG when needed (this does not apply for applications programs generated using the GENMASTER program in the Developer's package). Since the EXECUTIVE or EDITor are always copied into main memory when needed, all the space, including that used by the EXECUTIVE or EDITor, is available for allocation for your programs by the LOADer (about 25K). If you use the GET command to load a program which overlaps OSORG, however, it will be immediately destroyed when the EXECUTIVE is copied back into memory when the GET command is completed. The Workspace for the Apple II is kept in the extra 64K bank, so it is of no concern.



Memory with no programs loaded

For the Commodore 64, the situation is somewhat more complicated. The Editor is kept in the 12K byte section of RAM under the ROMs from $D000 to $FFFF when the EXECUTIVE is active. When any program is executed, the EXECUTIVE is swapped with the EDITOR. The EXECUTIVE is therefore "hidden" under the ROMs when your programs run, and the EDITOR occupies the space from OSORG to MEMLIM. When control returns to the EXECUTIVE, it is swapped with the EDITor again. For the Commodore, the Workspace is also allocated near the top of "free memory".

For large logical programs on the **Commodore 64**, you may want to let the LOADer use the space normally set aside for the EDITOR for your program(s). This can be done by using a bootstrap program (as described above), which should be the only program loaded (you can use a two-stage bootstrap to guarantee this). This bootstrap program might have the following form:

```
PROGRAM BOOTBIG OWN
INCLUDE LIBRARY
INCLUDE PROSYS
...
WPTR=WORG
WEOF=WORG       ; Make workspace empty
WSIZE=0         ; No workspace usable
EDRES=FALSE     ; EDITor will no longer be ther
HIFREE=MEMLIM   ; Reclaim Editor's space
...
```

It would then Load and execute your programs, described in the section on bootstrap programs, above.

When your program returns control to the EXECUTIVE, the EXECUTIVE will move back into memory at OSORG. If you subsequently use the EDITor, it will automatically be re-loaded (from disk, for the Commodore 64).

A GENERAL PURPOSE COMMODORE BOOTSTRAP FOR BIG PROGRAMS

Here is a bootstrap program which can be used to load and run a program which is too large to run on the Commodore 64 without overwriting the EDITOR. The program to be run **must** have OWN on the PROGRAM line. The Workspace will be cleared and the EDITor overwritten. To use the bootstrap, first give an **UNLOAD** command, and then type:

```
BOOTBIG Progname
```

from the EXECUTIVE, where Progname is your large, compiled program.

```
PROGRAM BOOTBIG OWN  ;Commodore 64 only boot big program
; Kills workspace and EDITor
INCLUDE LIBRARY
INCLUDE PROSYS
BEGIN
IF NCARG <> 1
  PUT NL,"BOOTBIG ABORTED: No name given"
  ABORT "#CUsage: BOOTBIG Progname"
HIFREE=MEMLIM  ;The max memory please
WORG=MEMLIM    ;No workspace
WPTR=MEMLIM
WEOF=MEMLIM
WLIM=MEMLIM
WSIZE=0
EDRES=FALSE    ;EDITor not resident
LOAD CARG[1]   ;Load & execute
IF LDERR <> 0
  ABORT "#cBOOTBIG LOAD ERROR $#H",LDERR
END
```

REMINDERS FOR SUCCESSFUL USE OF OVERLAYS AND SEPARATE COMPILATION

1. The root module or modules need to export any definitions needed by the overlays, and the overlays each need to INCLUDE the exports (don't forget the .E extension).

2. Each overlay must start with:

```
OVERLAY Name [EXPORT]
```

and must be compiled separately.

3. Overlays may call routines and use variables exported from the root module(s), or other overlays already loaded. The root module cannot contain calls to routines or reference variables declared in the overlays. The **only** way to enter an overlay is by a call to LOAD, which will transfer control to the entry point of the overlay.

4. Remember that **if you alter one module, no matter how trivial the change, you must re-compile all modules which access it**. This is entirely the programmer's responsibility; there is no way for PROMAL to check it for you. If you fail to do this, the LOADer will relocate the program incorrectly, probably resulting in mysterious crashes when your program runs. One way to

check for this is to look at the date which the compiler writes on the first line of the export file (.E extension). If this date is later than the compilation date on any of the modules which INCLUDE it, you need to recompile those modules.

5. If you manipulate LOFREE or HIFREE, remember that the low order byte must always be 0 (i.e., always points to a page boundary).

6. You should always check the value of LDERR after any call to LOAD, and print an appropriate diagnostic message if an error occurs.

7. Be sure to INCLUDE PROSYS (or copy the definitions from PROSYS.S directly into your source program) for any program using the loader.

8. The LOAD procedure depends on the underlying operating system, memory map and computer hardware for its operation. Therefore you should not expect programs using LOAD or EXPORT to necessarily be completely portable to other kinds of computers or operating systems, just because PROMAL is available on that computer.

9. If you EXPORT anything, you must have EXPORT on the PROGRAM (or OVERLAY) line, or you will get an "ILLEGAL EXPORT" error when the COMPILER encounters the first EXPORT declaration. A PROGRAM declaration should also have OWN specified (or else the variables will be assigned the same addresses as any other module not specifying OWN).

10. Exporting scalar variables may increase the memory usage of your program somewhat.

11. A maximum of six modules may be in memory at once (8 with a program generated with GENMASTER in the Developer's package).

12. For the Commodore 64, using LOAD with the LDPRCLR option, the LOADer will set HIFREE back to HIMEM after unloading all programs, to preserve the space normally occupied by the EDITor. If you want the EDITor space to be available for loading too, you need to set HIMEM to MEMLIM before calling the LOADer (Caution: this will cause the Workspace (if any) to be moved up to the top of memory too, and it will be clobbered when the EXECUTIVE swaps back in). Your program should restore HIMEM before exiting back to the EXECUTIVE.

13. If you use multiple modules and have an ESCAPE in one module to a REFUGE in a separate module, if you exit from the module with the ESCAPE via a normal END, the program will still return to the parent program of the original module.

