APPENDIX H

DYNAMIC MEMORY ALLOCATION

The PROMAL COMPILER and EDITOR use un-allocated memory for scratch buffer space, and under some circumstances use the Workspace and (in the case of the COMPILER "B" option on the Commodore 64) the space normally occupied by the EDITOR for buffer space. With care, user-written programs may also do this.

Refer to the Memory Map of the System Area in **Appendix G** to understand the variables referred to in this discussion, which are defined in file PROSYS.S. Additional information is contained in the section on the LOADer.

When a user program begins execution, it can safely use all memory between LOFREE and HIFREE for buffer space. This is the area shown as "FREE SPACE" by the MAP command.

On the Commodore 64, you may also safely use the Workspace for a buffer if it is empty. The Workspace exists between WORG and WLIM-1. If WEOF = WORG, then the Workspace is empty. In any event, the space between WEOF and WLIM-1 is unused (this is the "FREE WORKSPACE" area shown by the MAP command). Naturally these pointers will change if you write the Workspace in your program. They may also move if you use the LOADer to load a program or overlay which does not OWN its variables (not recommended). You can "force" the Workspace clear by setting WEOF and WPTR to WORG.

On the Commodore 64, if you want to use the space occupied by the EDITOR for a buffer, you may do so. This is the space between HIMEM and MEMLIM. If you specify "OWN" on the PROGRAM line of your program, for example,

```
PROGRAM MYPROG OWN
```

then you may use memory between WLIM and MEMLIM. This is because specifying "OWN" on the PROGRAM statement forces the PROMAL EXECUTIVE to allocate your global variables at the end of your program rather than at the high end of memory as it normally would. If using the EDITor space, you should set the EDRES flag to 0 (defined in PROSYS.S). The EDITor will be reloaded from disk if needed later.

Note: This applies **only** while in a user-program. This area of memory is absolutely vital to the EXECUTIVE when it is running.

By using the NOREAL command and using all the above techniques, it is possible to free up more than 34K bytes of contiguous space for a user program and data on a Commodore 64.

On the Apple II, about 28K can be made available for programs and variables by using a NOREAL command.

This page is intentionally left blank

APPENDIX I

CALLING MACHINE LANGUAGE ROUTINES FROM PROMAL

This Appendix describes how to call machine language subroutines from a PROMAL program. If you are not familiar with 6502 machine language programming, you may want to skip this section. Because PROMAL is functionally very close to machine language, it is normally not necessary to use any machine language programming at all with PROMAL. However, if you want to use machine language routines, a clean interface is provided. You can even pass arguments to a machine language routine, just like a PROMAL subroutine.

The way you call machine language routines depends on what you want to do. We might categorize the usual needs as follows, in order of increasing complexity:

1. Call a ROM routine that is built in to your computer.
2. Call a small routine you wish to embed in DATA statements as part of your PROMAL program.
3. Call a separate subroutine package, possibly with many routines and passed arguments.

These cases are well-supported with PROMAL. We will address each in order.

PROMAL has a very powerful way of calling machine language routines. It is especially useful for calling ROM-resident routines, such as the **Commodore Kernal** routines or **Apple II Monitor**. Virtually any 6502 machine language subroutine can be called directly from PROMAL with this method. This includes subroutines which expect arguments passed in registers, or return values to the caller in registers. This method can also be used to call machine language subroutines embedded in PROMAL DATA statements. The key to this extremely powerful and simple capability is the built-in JSR procedure, described below.

DECLARATION: **EXT ASM PROC JSR AT $0FB4** (defined in PROSYS.S)

USAGE: **JSR** [Address [,Areg [,Xreg [,Yreg [,Flags]]]]]

Procedure JSR calls a machine language subroutine at a specified address, optionally loading the 6502 processor's hardware registers with specified values before the call. **Address** is the address of the desired routine. **Areg**, **Xreg**, **Yreg**, and **Flags** are optional arguments which specify the desired values to be installed in the A, X, Y, and flags (processor status word) registers, respectively. All register arguments should be type BYTE. Naturally the address must be type WORD.

In order to use the JSR procedure, you will want to include the following declarations in your PROMAL source program (or **INCLUDE PROSYS**, since PROSYS.S contains all these definitions):


```
EXT ASM PROC JSR AT $0FB4  ; entry point
EXT WORD MLP  AT $0C51     ; Subrt. addr.
EXT BYTE REGA AT $0C53     ; A
EXT BYTE REGX AT $0C54     ; X
EXT BYTE REGY AT $0C55     ; Y
EXT BYTE REGF AT $0C56     ; Flags
```

These lines declare the location of the built-in procedure JSR, and the memory locations of copies of the processor register contents to be used. These will be explained presently.

Here is how JSR works. When your PROMAL program executes a JSR statement, the Address argument is copied into MLP, and any additional arguments are copied into REGA, REGX, REGY, and REGF, in that order. The 6502 registers are then loaded as follows:

REGA into the A register
REGX into the X register
REGY into the Y register
REGF into the flags (processor status word)

and a machine language jump to subroutine is performed to the address in MLP.

When the called machine language program returns (with an RTS instruction), the contents of the registers will be saved in REGA, REGX, REGY, and REGF before resuming execution of the next PROMAL statement. Your program can therefore examine the contents of the registers at the time of return. This is important since many machine language routines return values in the registers.

Any optional arguments on your JSR statement which are not specified are not changed. Therefore, for instance, if you JSR to one machine language routine and then JSR to a second routine with no registers specified, the registers will contain the values returned by the first routine.

Some examples should illustrate the simplicity of this method. The examples below refer to Commodore 64 "Kernal" machine language routines, as defined in The Commodore 64 Programmer's Reference Manual. All the examples assume you have added the declaration lines given above.

EXAMPLE 1:

```
; Call SCREEN Kernal routine - returns X=columns, Y=rows...
JSR $FFED ; Kernal "SCREEN" routine
OUTPUT "#C SCREEN IS #W COLUMNS BY #W ROWS",REGX,REGY
```

This program fragment calls the machine language routine at $FFED without specifying any registers. It then prints the contents of the X and Y registers which were returned by the subroutine.

EXAMPLE 2:

```
CON CHKIN = $FFC6  ; Kernal
CON CHRIN = $FFCF  ; I/O
CON CLRCHN = $FFCC ; routines
BYTE LINE[81]
WORD I
BEGIN
JSR CHKIN, 0,15    ; channel 15
I=0
REPEAT
  JSR CHRIN        ; input char
  LINE[I]=REGA     ; install char
  I=I+1            ; next location
UNTIL REGA=CR      ; end of line?
LINE[I-1]=0        ; replace CR with end of line
JSR CLRCHN         ; release channel
PUT NL,LINE,NL
END
```

This program fragment reads a line from the disk error channel and prints it on the screen. It will normally display:

```
00, OK,00,00
```

The JSR CHKIN calls the Kernal CHKIN routine, passing 0 in the A register and 15 in the X register to select channel 15. The loop repeatedly calls CHRIN, which returns the character read from the disk drive in register A. The value returned in register A is then installed in a string. When a CR is received from the drive, the channel is closed, the string terminator added, and the line displayed using an ordinary PUT call. Note that when using Commodore channels like this you should be careful not to mix normal PROMAL I/O calls at the same time your channel is open, because PROMAL uses Commodore channels to do its I/O, and only one channel can be selected at a time. Also, channel 15 is always open to the Command/Error channel of the disk drive. If you open your own channels, you should use channels and secondary addresses of 8 or 9 to avoid conflicts with normal PROMAL disk files. Be sure to close them, too!

NOTES ON JSR USAGE

1. When you call JSR, the address is also an optional argument, although it is usually specified. If no arguments are specified, a call is made to whatever address is in MLP. In this way, you can call one out of a number of possible subroutines selected from a table, by putting the desired address in MLP before each JSR call.

2. If a machine language routine executes a BRK (breakpoint) instruction, the address of the breakpoint and registers are stored in these same locations before control is returned to the Executive.

3. For the Commodore 64, Do not attempt to call any routines in the BASIC ROM because PROMAL switches the BASIC ROM out of the Commodore 64 memory map.

4. For the Commodore 64, the file REL_FILES.S on the PROMAL System disk and DISKETTE.S on the optional Developer's disk contain many illustrations of how to use JSR to perform I/O using the Kernal. For the Apple, PRODOSCALLS.S illustrates several examples of JSR usage.

5. The REGF register contains a copy of the processor flags. You can test the flags returned by using the following statements:

```
IF REGF AND $01     ; true if the Carry flag is set
IF REGF AND $02     ; true if the Zero flag is set
IF REGF AND $80     ; true if the Minus flag is set
```

CALLING A MACHINE LANGUAGE ROUTINE EMBEDDED IN DATA STATEMENTS

Many BASIC programs have machine language subroutines embedded in DATA statements. These instructions are READ and POKEd into some unused area of memory and then executed with a USR or SYS statement. You can also embed a machine language routine (or routines) in PROMAL DATA statements, and execute the code using JSR. It is quite a bit simpler than BASIC though, because you do not need to use a loop to READ it and POKE it first.

There are two ways to set up an embedded machine language routine, depending on whether your routine is address-dependent or address-independent. An **address independent** routine is one which will execute correctly regardless of the address at which it is loaded. An address-dependent routine will only execute properly at the address for which it was assembled. This distinction is important because, in general, your compiled PROMAL program (and therefore your data statements) will not be loaded into memory at the same location every time.

If your routine is address-independent (runs anywhere), then you can execute your machine language routine by simply using procedure JSR to call it by name. If your program is address-dependent, then you will have to insure that it is executed every time in a known location. The easiest way to do this is to use procedure BLKMOV to copy it to a known location and then JSR to this known location. This method is equivalent to the READ-and-POKE loop method used in BASIC.

A machine language routine will be address-dependent if it contains any references to addresses within the routine itself. For example, if your routine does JMPs or JSRs to labels that are part of the routine itself, it will not be address-independent. The same applies for a LDA of any data in the routine. Conditional branches are okay, though, because they are coded as displacements, not absolute addresses.

EXAMPLE:

```
BYTE LINE[81]
WORD I

DATA BYTE TOLOWER [] =
$C9,'A',       ;  TOLOWER CMP #'A'
$90,6,         ;          BCC SKIP
$C9,$5B,       ;          CMP #'Z'+1
$B0,2,         ;          BCS SKIP
$69,$20,       ;          ADC #'a'-'A'
$60            ;  SKIP    RTS

BEGIN
PUT "ENTER A LINE: "
GETL LINE
I=0
PUT NL,"IN LOWERCASE ONLY = "
WHILE LINE[I]           ; not end of string?
  JSR TOLOWER, LINE[I] ; convert char
  I=I+1                ; bump pointer to next char
  PUT REGA             ; show returned result
PUT NL
END
```

The program fragment above illustrates a call to an address-independent machine language subroutine embedded in data statements. For simplicity, a trivial routine was selected, which simply converts a character passed in the A register to lower case if it is upper case and returns it in A. The line,

```
JSR TOLOWER, LINE[I]
```

calls the embedded machine language routine, no matter where the program is loaded, passing the character desired in the A register. Of course, the actual conversion to lower case could be done much simpler with the PROMAL statements,

```
IF LINE[I] >= 'A' AND LINE[I] <= 'Z'
  LINE[I] = LINE[I] + $20
```

but this is, as we said, simply for illustration.

If your machine language routine is address-dependent, you will need to copy it to some unused memory area and then execute it, for example:

```
...
CON MYSUB = $0334          ; Where to put M/L sub
DATA BYTE MYSUBCODE [] =
... ; (put hex code for routine here)
DATA BYTE SUBEND [] = 0       ; dummy byte to compute loc. of end of code
...
BLKMOV MYSUBCODE,MYSUB,SUBEND-MYSUBCODE ; Copy routine to known loc.
...
JSR MYSUB
...
```


In this example you should assemble your routine for a starting address of $0334, of course.

WRITING MACHINE LANGUAGE EXTERNAL PROCEDURES AND FUNCTIONS

For medium or large assembly language packages, embedding machine language programs in data statements is not practical. For this situation, there are two more ways to interface PROMAL to your assembly-language routines. Both of these methods involve writing a separate assembler program and assembling it. The resulting machine language program is then loaded from disk by your PROMAL program and executed when needed. Your assembly package can have any number of subroutines, which may be either procedures or functions, and are called by name, just like a PROMAL routine.

There are two ways your machine language routine can be loaded into memory. The simpler but less powerful way is to use function MLGET, described below, to load your program at a specified address in memory. MLGET can load machine language programs generated by virtually any assembler for your computer. The only trouble is, you have to find a place to put your program. Since PROMAL allocates memory dynamically for programs, you will have to choose carefully to avoid assembling your program for a location which may be occupied by some other program. There are a few "holes" in the memory map, discussed below, where you can locate your machine language routine using this method. However, if your program is large, you should probably not use MLGET to load your program.

The second method is extremely powerful. This is to create a relocatable machine language PROMAL module, which can be executed by simply typing its name from the EXECUTIVE, or can be loaded under program control with the LOAD procedure. PROMAL 2.0 provides a utility program on the PROMAL diskette called RELOCATE, which has the ability to turn any assembly program into a relocatable program. Your program does not have to be address-independent. It can be used with virtually any assembler. If you don't have an assembler, one is provided that runs under PROMAL, on Volume 1 of the PROMAL Public Domain User Library (available from SMA). By using RELOCATE, you can have a machine language module which will run properly at any address which the PROMAL loader can find available for it. This technique is described at the end of this section.

NON-RELOCATABLE MACHINE LANGUAGE ROUTINES USING MLGET

The biggest problem of a standard machine language routine is where to put it. As you know, PROMAL programs are relocatable, and the EXECUTIVE automatically finds a spot for them in memory. Unfortunately, 6502 machine language programs are not generally relocatable, and will only work properly at the address they were assembled for. Although the PROMAL EXECUTIVE can load a non-relocatable machine language program into memory with the GET command, it won't keep track of where it is, and may allocate a PROMAL program (or variables) right over the top of it if care is not taken.

If your machine language routine is short, one best place to put it is at $0334 to $03FF. This area is available on both the Apple II and Commodore 64. However, the optional PROMAL HIRES GRAPHICS PACKAGE uses this area for global variables, so you should avoid using this area if you will be using hi-res graphics in your application. If your routine(s) take more than 200 bytes, you'll have to find another spot. If you are **certain** your program won't need

to do REAL operations, you can use the 256 bytes at $0800 for your machine language program (this area is used for allocating local REAL variables and performing REAL arithmetic).

For large machine language routines, you may want to pick a spot in the "unused" area shown by the EXECUTIVE MAP command. Be aware that this area expands and contracts as programs are loaded or unloaded, and that the EDITOR or COMPILER will use this area for buffer space. Therefore you will have to reload your machine language code from disk after you use the EDITOR or COMPILER. PROMAL allocates programs from the bottom of available memory up, and allocates global variables (for arrays and global REAL variables) from the top of available memory down. To find a safe spot, first UNLOAD any un-needed programs. Then GET your PROMAL program into memory. Then use the MAP command to determine where the "available space" starts. Round this address up to a nice round number to leave room for future growth of your program, and use this for the address of your machine language program segment. For example, if the free space goes from $5F00 to $7A00, you might want to pick $6000 as the starting address of your M/L code. **Appendix G** gives a PROMAL memory map.

Once you have decided where to assemble your program, the next problem is where to put your "zero page" variables. The only zero page locations you can use with complete safety are:

Available zero page for Commodore 64

$02 - $10 (not used by PROMAL, but used by BASIC)
$FB - $FE (the same space that is free for BASIC)

Available zero page for Apple II

$00 - $0E, $4A- $4D, $B0 - $FF

As you already know if you've done much 6502 machine language programming, the Commodore 64 system software uses up almost all of page 0. Unfortunately, this situation is not greatly improved with PROMAL.However, if you just need some scratch space for pointers and the like, you can use the following locations:

Scratch zero page locations for Commodore 64

```
$16 - $19   ; Used for scratch by PROMAL
$36 - $41   ; Used for scratch by the LIBRARY routines
$57 - $66   ; Used only for REAL arithmetic - free if no REALs needed
```

Once you have settled on where to put your program and zero-page variables, the hard part is over. Calling your machine language routine from a PROMAL program is very easy. All you have to do is declare the name of the routine and where its entry point is, for example:

```
EXT ASM PROC MYROUTINE AT $0334
EXT ASM FUNC BYTE TESTIT AT $0337
```


These declarations define two external assembly language (ASM) routines located at $0334 and $0337. It is not necessary to define what arguments (if any) will be passed to these routines. The compiler will accept any number of arguments when calling an EXT ASM routine.

PROMAL calls EXT ASM routines with a 6502 JSR instruction. If your routine is declared as a PROC and doesn't require any arguments, you can simply write it like any 6502 subroutine and just return when you are done via an RTS. More often, though, you will want to receive one or more arguments from the calling PROMAL routine.

PROMAL passes arguments on the hardware stack. All arguments are passed as 2-byte quantities, even if the argument evaluates as type BYTE (the high order byte will be 0 in this case). Passing REAL arguments to assembly language routines is not recommended. On entry to your routine, the Y register will contain the number of arguments passed on the stack. These arguments were pushed on the stack before the JSR, so they are logically "underneath" the return address. Generally you will want to pop off the return address and save it, then pull off the arguments (the last argument will be popped first) and save them in variables of your own. When the routine is done, you should push the saved return address back on the stack and return. You don't have to preserve any registers.

The following example shows how to write an assembly-language procedure with one argument expected to be passed from the PROMAL calling program:

```
;   Sample assembly language procedure MYPROC with 1 argument...

        *=$0334             ; in unused piece of memory...
MYPROC   PLA
         STA  RA             ; save return addr. low...
         PLA
         STA  RA+1           ; & hi byte
         PLA
         STA  ARG1+1         ; save passed argument hi
         PLA
         STA  ARG1           ; & low byte
         ...
;   Operate on ARG1 as desired here...then...
         ...
         LDA  RA+1
         PHA                 ; put return addr back on stack
         LDA  RA
         PHA
         RTS                 ; return to caller
RA       *=*+2               ; save for return address
ARG1     *=*=2
         .END
```

Here is the companion PROMAL declaration and a sample call:

```
EXT ASM PROC MYPROC AT $0334
...
MYPROC X+1     ; pass X+1 to m/l routine
...
```


If your machine language subroutine is to be a function, it should return its value on the top of the stack. If it is type BYTE, it should only return a byte on the stack, otherwise it should return two bytes.

To assemble your routine you can use any Assembler or Machine Language MONITOR which produces a standard Commodore machine language PRG file or Apple II BSAVE type file respectively as output. An assembler which runs under PROMAL is available in the PROMAL public domain library. Some of the small Machine Language Monitors such as the version of C64MON which loads at $8000 can be run directly from PROMAL for the Commodore 64. Others will have to be run from BASIC. Once you have saved your object file on disk, you can load it into memory from the PROMAL EXECUTIVE with the GET command or by using the MLGET function. When using GET, enclose the name of the file in quotes to indicate that it is a machine language file instead of a PROMAL program. Also be careful to type the name exactly as it is stored in the directory (usually with upper case letters). For example:

```
GET "MYPROG"
```

will load the machine language file "MYPROG" into memory at whatever address it was saved. Note that the MAP command will not show the location where this program is loaded. Alternatively, your application can load the machine language file itself, using the built-in function MLGET, described in below. This is the preferred method.

Let us now look at a slightly more complex example. This example illustrates a machine language function with one required argument of type WORD and one optional argument of type BYTE, defaulting to ' ' if not specified:

```
;   Assembly Language function MYFUNC (WORD [,BYTE])
          *=$0334
MYFUNC    PLA
          STA  RA              ; save return address
          PLA
          STA  RA+1
          LDA  #' '            ; default if only 1 arg specified
          CPY  #2
          BNE  MYFUNC2         ; branch if only 1 arg specified
          PLA                  ; else discard dummy hi byte
          PLA                  ; get byte argument specified
MYFUNC2   STA  ARG2            ; save default or specified 2nd arg
          PLA
          STA  ARG1+1          ; save hi byte of arg 1
          PLA
          STA  ARG1            ; save low byte of arg 1
          ...
;   operate on arguments as desired here...then...
          PHA                  ; push result to be returned to caller
          LDA  RA+1
          PHA                  ; push return address
          LDA  RA
          PHA
          RTS                  ; return to caller
```


Here is the companion PROMAL declaration and sample calls:

```
EXT ASM FUNC BYTE MYFUNC AT $0334
WORD WHERE
BYTE CHAR
...
CHAR = MYFUNC(WHERE)          ; call with default for 2nd arg
...
IF MYFUNC(WHERE-1,'A')        ; call with 2nd arg specified
  ...
```

CALLING LIBRARY ROUTINES FROM MACHINE LANGUAGE PROGRAMS

You may call LIBRARY routines from your machine-language subroutines (but you may not call subroutines written in PROMAL). The address of the desired routine can be obtained from the listing of the LIBRARY.S file in Appendix Q of this manual. Pass your arguments on the stack, remembering that passed arguments are always 2 bytes each. **Don't forget to set the Y register to the number of arguments you are passing.** The following example shows how to print a message on the screen from an assembly language routine by calling a library routine:

```
PUT     =    $0F15          ; Address of PUT routine (from Library)
        ...
;   Print an error message and then the character now in the X reg, then CR.

        LDA  #<ERRMSG
        PHA                 ; Push the address of the string to print (low)
        LDA  #>ERRMSG
        PHA                 ; then hi byte
        TXA                 ; push the character to print after the msg.
        PHA
        LDA  #0
        PHA                 ; push dummy hi byte (must be 0 for char.)
        LDA  #$0D           ; ASCII CR is third argument
        PHA
        LDA  #0
        PHA                 ; another 0 for the hi byte
        LDY  #3             ; we're passing 3 arguments
        JSR  PUT            ; display all three arguments
        ...
ERRMSG  DB   'Illegal character: ',0  ;0-byte terminates the string
```

From inspection of the program fragment above, you may have surmised how the PROMAL LIBRARY routine PUT tells the difference between a single character argument and a string argument. If the argument is less than 256 (high byte is 0), then it is a single character. If the argument is greater than 256, then it must be the address of the string to print.

LOADING NON-RELOCATABLE MACHINE LANGUAGE PROGRAMS FROM WITHIN A PROGRAM

Function MLGET can be used to load a stanndard Apple or Commodore format machine language program, such as would be generated by commercial assemblers. You can specify whether you want the program loaded at the same location it was saved at, or at another location. Function MLGET is described in the LIBRARY MANUAL.

MAKING YOUR ASSEMBLY PROGRAMS RELOCATABLE

The RELOCATE program supplied on the PROMAL disk is capable of converting virtually any assembly language program into a relocatable PROMAL module. The advantages of doing this instead of simply using MLGET to load a standard, non-relocatable program are:

1. The program can be executed by simply typing its name from the PROMAL executive, just like any other PROMAL program.

2. The PROMAL loader will find a free location in memory to run the program automatically.

3. The program can be loaded under program control using the LOADer.

4. You can import variables and subroutines from your machine language package to PROMAL programs which call it.

This makes using RELOCATE the most desirable method of preparing large assembly language modules for use with PROMAL.

To use RELOCATE, follow these steps:

1. Prepare your assembly language source program in the usual way, following the interfacing guidelines in the preceding section, and the organizational guidelines suggested in the following section.

2. Assemble your program twice, once with the origin set at some arbitrary page boundary (greater than $0200), and once with the origin set exactly $0100 bytes higher in memory. Save both resulting object programs. You may use virtually any assembler you wish. A public domain PROMAL assembler is available from SMA. The program should consist of a single, contiguous block of code. Your zero-page variables will not be relocatable, and must be assigned locations as described in the foregoing section.

3. Execute RELOCATE from the PROMAL EXECUTIVE by typing:

```
RELOCATE Object Object0100 Objmodule
```

where Object and Object0100 are the names of the two machine language object files saved from the previous step, and Objmodule is the name of the desired PROMAL module to be generated as output. No default extensions are assumed for the first two file names, which are normally "PRG" type files for the Commodore and "BIN" type files for the Apple. The last filename will have a .C extension by default. If you want Objmodule to be an overlay instead of a program (see the section on the PROMAL loader for more information), you can specify an optional fourth argument as the single character O (the letter "oh").

4. When RELOCATE finishes, your program is ready to run or load.

ORGANIZING YOUR RELOCATABLE PROGRAM

Your assembly language package can have multiple procedures and functions in it which can be called from your PROMAL program, complete with passed arguments. In order for the LOADer to be able to link up your PROMAL program correctly with your finished relocatable machine language package, we suggest you follow some simple conventions, which we will illustrate in a skeletal example program.

To organize your program, decide which routines you will want to call from your PROMAL program. These should be entered by a jump table at the very start of your program. These JMPs should be followed immediately by any non-zero page variables which you wish to make available to the calling PROMAL program (often none will be needed). For example, if you want to be able to call three routines, and have one variable which can be accessed by other PROMAL programs:

```
TEMP    =    $00FE      ; Temp 0-page variable used by this program

        *=$1000         ; Dummy origin (make it $1100 for 2nd assembly)
FUNCA   JMP  FUNCA1     ; Function exported to PROMAL program
PROCB   JMP  PROCB1     ; Procedures exported to PROMAL program
PROCC   JMP  PROCC1

ANSPTR  .WORD     0     ; Variable exported to PROMAL program

FUNCA1  PLA
        STA  RA         ; Save return address
        ...             ; etc.
        RTS

PROCB1  PLA
        STA  RA
        ...
        RTS

PROCC1  PLA
        STA  RA
        ...
        RTS
        END
```

Assume that this program has assembled successfully. Now you want to export the definitions of your routines and your variable ANSPTR to the PROMAL program(s) which will be using your machine language package. Since the assembler can't generate an export file automatically, you can generate a "fake" export file by hand using the PROMAL EDITor. Assuming our sample package will be called MLPKG, you could generate this text file with the file name MLPKG.E:


```
IMPORT MLPKG ;10/16/85
  EXT ASM FUNC FUNCA AT $0000
  EXT ASM PROC PROCB AT $0003
  EXT ASM PROC PROCF AT $0006
  EXT WORD ANSPTR AT $0009
```

Be sure to start the IMPORT line exactly in column 1 and to observe the indentation for all other lines. The addresses shown after "AT" in each of the lines should be relative to the start of your machine language program (each JMP instruction is 3 bytes long). If you use the jump table, you won't have to change this export file even if you make changes in the body of your machine language program later.

You can now INCLUDE MLPKG.E in any PROMAL programs that will call the machine language package, and compile them. Your PROMAL program should also have a "bootstrap" program to load the machine language package, as discussed in the section on the PROMAL LOADer. For example:

```
PROGRAM BOOTPROG OWN
INCLUDE LIBRARY
INCLUDE PROSYS
...
LOAD "MLPKG", LDNOGO
LOAD "CALLSML"
...
END

PROGRAM CALLSML  ; main module, calls MLPKG
INCLUDE LIBRARY
INCLUDE MLPKG.E  ; Export file from M/L package
...
WORD MYVAR[100]
WORD I
...
ANSPTR=MYVAR   ; Using variable imported from MLPKG
IF FUNCA(23-MYVAR[I]) < 100  ; Calling M/L function
  PROCB MYVAR[I+1], MYVAR[I+2] ; & M/L procedure
...
END
```

These two programs can then be separately compiled. The final step is to make our machine language package relocatable:

```
RELOCATE OBJECT OBJECT100 MLPKG.C
```

assuming OBJECT is the output file from assembling the program at $1000, and OBJECT100 is the object file resulting from assembling it at $1100.

To execute the program, type:

```
BOOTPROG
```


which will load the relocatable machine language module MLPKG into memory at some available location, load the main PROMAL program CALLSML into memory above it, link the function and procedure calls and variable references to the machine language package, and execute the program.

TECHNICAL NOTES ON RELOCATE

The source code for RELOCATE is provided on a PROMAL diskette. It uses conditional compilation for the Commodore and Apple II versions. You may therefore modify it to meet your needs if you have an assembler which produces object output files which are not compatible with RELOCATE.

The Commodore version assumes the object code files to be used as input to RELOCATE will be standard Commodore object files of type PRG, such as are generated by the BASIC SAVE command. This format consists of a word giving the starting address, followed by a memory image of the object program.

The Apple II version assumes a standard PRODOS object file with a file type of BIN, such as is generated with a BSAVE command. The Apple version of RELOCATE is more complex because the information about the starting location of the memory image is contained in the directory instead of the file itself. See the PRODOS Reference Manual for details.

The format of the PROMAL relocatable object module which is generated as output from RELOCATE is as follows:

Position	Field Name	Description
0	FHEAD	Header ID byte, set to $CE
1	FTYPE	Module type, $01 for M/L prog, $05 for M/L overlay.
2-3	FHCDBA	Nominal code base address (ORG where assembled)
4-5	-	Not used, set to 0000.
6-7	FHCDSZ	Code size in bytes of memory image. Do not include this header or relocation table in count.
8-D	-	Not used, set to 0.
E-10	FHDATE	Date of assembly, 1 byte each for day, month, year (year-1900 really), in that order.
11	-	Reserved. Set to $04.
12-1D	FHCOMD	Program name followed by $00 terminator, as it would appear on a PROGRAM line of a PROMAL program.
1E-1F	-	Not used, set to 0000.
20-n		The actual object code memory image. The size of this field is given by FHCDSZ above.
n+1-n+2		Reloc. Table header, set to 'R' followed by 'A'.
n+3-n+4		Count of number of bytes which follow.
n+5-end		List of words of addresses relative to the start of the memory image above, where relocations must be made. For example, If the memory image was assembled to start at $1000 and starts with a JMP $112B instruction, then the first entry in the list would be $0002 (indicating the high byte of the address portion of the JMP instruction will need to be modified when the program is loaded).

The RELOCATE utility can accept a fourth argument of O (the letter O, not zero), indicating that the output object file is an overlay instead of a program (the SIZE command will display a type of "AOV" in this case, for Assembly Overlay).

INTERRUPT SERVICE ROUTINES

Due to limitations imposed by the architecture of the 6502 processor, it is not practical to write interrupt service routines in PROMAL. However, you may write and use machine language service routines.

For the **Commodore 64**, your program should prepare for using interrupts as follows:

1. Turn off interrupts.
2. Save the contents of the interrupt vector at location $0314-0315 in another variable.
3. Install the address of your service routine in $0314-0315.
4. Enable interrupts.
5. Your service routine will be entered from initial Kernal interrupt processing via the vector at $0314. Your service routine may not use any library routines, Kernal routines, or any other software. It must preserve all the registers and the stack. If the interrupt is caused by the 1/60th second timer, you must do a jump indirect through the saved vector you extracted in step 2. Otherwise, you must restore the registers already pushed by the Commodore Kernal and do an RTI.

Because of the heavy usage of interrupts made by the Kernal, we recommend you avoid interrupts on the Commodore 64 unless absolutely essential.

For the **Apple II**, you may freely use interrupts in the normal manner. However, you may not call any Library routines in the service routine, because they (and the underlying PRODOS system) are not re-entrant. You should observe all the restrictions detailed in the Apple Reference manuals.

Correct operation of PROMAL with interrupt routines is entirely the responsibility of the programmer.

This page is intentionally left blank.

APPENDIX J

RECURSION AND FORWARD REFERENCES

The PROMAL Language fully supports recursion. In fact, the PROMAL COMPILER (which is a 2800 line PROMAL program) makes extensive use of recursion. To make full use of recursion, it is sometimes necessary to call a Procedure or Function before it is defined. This is permitted in PROMAL, as follows:

Prior to the first invocation of the routine to be forward referenced, declare it as an external (but not ASM), for example:

```
EXT FUNC BYTE EXP     ; Allow forward reference to Expression Parser
EXT PROC STATEMENT    ; Ditto for Statement processing routine.
```

You may then have forward references to the routine, by calling it in the normal manner, for example:

```
TYPE = EXP
STATEMENT ASSIGN, BYTETYPE
```

At the desired location, complete the normal declaration of the Procedure or Function, for example:

```
FUNC BYTE EXP
...
END

PROC STATEMENT
ARG WORD ASGNLOC
ARG BYTE RESULTTYPE
...
END
```

Additional calls to these routines may follow their definition in the normal fashion, if desired. Note that declaring a forward reference in this manner defeats the compiler's argument count checking and also its checking for undefined subroutines, so be careful.

NOTE: If you have the optional Developer's disk, file XREF.S illustrates an excellent example of the use recursion for searching a tree.

This page is intentionally left blank

APPENDIX K

REAL FUNCTION SUPPORT

A PROMAL Diskette includes a file called **REALFUNCS.S** which contains the complete source code for all of the following arithmetic functions:

Name	Description	Example
ATAN	Arctangent (returns angle in radians)	Y = ATAN(X)
COS	Trigonometric cosine (angle in radians)	Y = COS(X)
EXP	Exponential (e to the X power)	Y = EXP(X)
LOG	Natural logarithm (base e)	Y = LOG(X)
LOG10	Common logarithm (base 10)	Y = LOG10(X)
POWER	Power (X to the Y power)	Z = POWER(X,Y)
SIN	Trigonometric sine (angle in radians)	Y = SIN(X)
SQRT	Square root	Y = SQRT(X)
TAN	Trigonometric tangent (ang. in radians)	Y = TAN(X)

These functions all expect arguments of type REAL and return results of type REAL. They are provided in PROMAL source form instead of as built-in functions (as in BASIC) because:

1. Many programs do not need any of these functions. If your program doesn't need them, you do not have to have them in memory, which makes about 1.5 K bytes of additional memory available for things you do need.

2. If you do need these functions, you can simply put the statement

```
INCLUDE REALFUNCS
```

in your program, and they will be included in your compiled program (assuming you have copied the REALFUNCS.S to your Working diskette used for compilation). No other declarations are needed to use the functions.

3. If you only need one or two of the functions, you can use the Editor to extract just the functions you need and insert them into your program. This saves memory and decreases compilation time compared with including the entire REALFUNCS.S file. Note, however, that some of the functions call other functions internally. For example, SIN calls COS and LOG calls LOG2, so be sure to copy all needed routines.

4. You can examine and study how the source code works. The algorithms used depend heavily on Hart, et al, Computer Approximations, published by John Wiley and Sons in 1968 and reprinted in 1978 with corrections. Comments in the source code identify which algorithm was selected.

BASIC users will find most of these functions familiar, except for POWER, which replaces the BASIC operator "^". The POWER function is defined only for positive values of the first argument. All the functions are believed to give better precision than Commodore or Applesoft BASIC, often as much as two additional significant digits. Through normal range arguments, the functions can be relied on for about 9.5 significant digits (slightly less for POWER). Even though these functions provide greater precision and are written entirely in PROMAL, they usually still execute faster than their BASIC counterparts, which were implemented in hand-coded assembly language.

NOTE: PROMAL version 2.0 and earlier had function ABS in REALFUNCS.S. Version 2.1 has ABS in the standard LIBRARY for improved convenience and performance.

Also included on one of the PROMAL diskettes is a file called **FLOOR.S.** This contains the PROMAL function FLOOR, which has the form:

```
Realvar = FLOOR (X)
```

where **X** is a REAL value. FLOOR returns a REAL result which is equal to the largest integer less than or equal to the REAL argument. For example:

```
INCLUDE FLOOR.S
REAL X
...
X = FLOOR (100000.89)  ; Returns 100000.0
X = FLOOR (-3.8)       ;Returns -4.0
```


APPENDIX L

COMPATIBILITY ISSUES

One of the goals of the designers of PROMAL was to achieve a high degree of compatibility for PROMAL source programs on different kinds of computers while at the same time allowing users the freedom to take advantage of the special features of each supported computer. Obviously this entails some compromises. To achieve 100 percent compatibility, you can only support the "lowest common denominator" between machines. Clearly this is not a satisfactory approach. Instead, a standard Library of functions was developed, which is kept as similar as possible on all machines, but with additional system-dependent functions also provided in additional libraries.

This section describes the major differences between the Apple II/Commodore 64 versions of PROMAL (hereafter referred to jointly as "6502 PROMAL") and the IBM PC and compatibles version (hereafter referred to as "IBM PROMAL"). The information is oriented towards the software developer wishing to "port" an existing 6502 program to the IBM, but is also useful for going from the IBM to the 6502.

MAJOR DIFFERENCES BETWEEN 6502 AND IBM VERSIONS OF PROMAL

1. There is no EXECUTIVE in IBM PROMAL. 6502 PROMAL includes an EXECUTIVE program which is a command shell similar to DOS on the IBM PC. There is no EXECUTIVE in the IBM version because the DOS shell provides these functions. Users of 6502 PROMAL should have little difficulty adjusting to DOS, since the EXECUTIVE and DOS are fairly similar.

2. IBM PROMAL does not support multiple programs in memory at once, since DOS does not support it. This generally presents no problem.

3. IBM PROMAL file names are limited to 8 characters plus a three character extension, because this is the DOS standard. IBM PROMAL supports full DOS path names.

4. IBM text files (including PROMAL source files) have lines terminated by **CR, LF** pairs, whereas 6502 PROMAL uses only **CR** terminators, in keeping with the conventions of the respective computers. This may cause some initial problems when porting source files from one machine to the other. When moving source files from 6502 systems to the IBM, you will need to write a small "filter" program to insert a linefeed ($0A) after each CR ($0D). More significantly, if your 6502 program uses statements such as **PUT CR,...** to generate an end-line, you will need to edit your source file to change this to **PUT NL....** Using NL is preferred since it is portable between either machine; in the 6502 Library, NL is defined as a single character, CR. In the IBM Library, it is defined as the string CR,LF. When using a statement such as **OUTPUT "#C..."**, you do not have to change the **#C** since this is defined as the appropriate newline sequence on either machine.

5. In 6502 PROMAL, the file handle returned by OPEN always points to the name of the file. This is not true for IBM PROMAL, because standard DOS file handles are returned, which are small integers, not addresses. This is normally of no consequence. However, if your program depended on the file

handle pointing to the name you will need to change it.

6. IBM PROMAL does not support the W, L, S, or K devices. However, you can open a file named "W". If you manipulate WPTR, WEOF, etc. directly in your program, you will need to change this.

7. I-O redirection operates somewhat differently in IBM PROMAL. DOS provides the I-O redirection, not PROMAL. The REDIRECT procedure is not supported. I-O redirection, when enabled using the > operator on the DOS command line, affects all screen output, not just output to the STDOUT handle. Also, note that **GETLF (STDIN,...)** does not support the PROMAL line editing features from the keyboard, but only the DOS line editing keys.

8. The LOAD procedure is not supported in the present version 1.9 of IBM PROMAL. In most cases this should not pose a significant hardship since the IBM has a much larger memory space available for running your program, so programs needing overlays in the 6502 version will not need them in the IBM version. It is possible to have one PROMAL program chain to another program using the DOSCALL procedure.

9. Naturally, any machine language calls, memory mapped registers, etc. used in your programs will not be portable.

10. Applications using the T device may need to be altered for use on the IBM PC. The TMODE utility is not supported; the DOS MODE command replaces this program. IBM PROMAL supports interrupt driven serial I/O.

11. If your program uses special keys (such as function keys), you will need to adjust the key codes as specified in **Appendix B.** Function key string substitution is still supported in the IBM version, but not from the DOS shell.

12. If your program uses embedded control keys to select reverse video mode, you will have to change this since the IBM does not support a control sequence for reverse video (unless you use ANSI.SYS as described in the DOS documentation). Functions are provided for setting video attributes.

13. The DIR function displays file names in a different format on the IBM PC, consistent with the DOS **DIR** command (/W option).

14. The line editing keys for use with GETL, EDLINE, and INLINE are somewhat different for the IBM version, consistent with normal key conventions for the IBM.

15. CARG[0] is not defined in the IBM version.

16. OPEN for IBM PROMAL does not have a default file extension (it is .C on the 6502 version).

17. The RENAME function cannot have wildcards in the IBM version. Complete path names are supported, and you can rename into a different directory.

18. IBM PROMAL 2.1 reserves the following additional key words: LONG, STRUC, UNION, SIZEOF, SEARCHLIB.

APPENDIX M

RELATIVE FILE SUPPORT ROUTINES FOR COMMODORE 64

PROMAL on the Commodore 64 treats all files as Commodore sequential (SEQ) type files, including programs, text and data. For many database and business applications, another type of file structure may be more suitable for rapid access to data. The Commodore 1541 disk drive has an undocumented but fairly widely-known ability to create and access files by "relative records". Your local computer store can probably provide books with information on using relative records with BASIC, such as The Anatomy of the 1541 Disk Drive, by Abacus Software.

The PROMAL System Diskette contains a file called REL_FILE.S which provides a set of PROMAL routines for using relative files from your program. A totally complete discussion of relative files is beyond the scope of this manual, but here is a brief description.

A relative file is organized into a number of fixed-length records. The size of all records in the file is the same, and is established when the file is opened. The record size can be from 1 to 254 characters. Records of 20 to 100 characters or so are typically used. For a database application, each record might be subdivided into fixed-length fields; for example, a customer name field, address field, etc. Once you have opened the relative file on disk, you initialize the file. Initializing the file allocates space on the disk for the number of records you specify and sets each record to "empty".

Once you have opened and initialized the file, you may write and read records by specifying the relative record number desired. Typically this record number corresponds to a sequential customer number or some other "key" number by which the file is to be accessed. The first record on the file is number 1 (not 0), the last record has a relative record number equal to the highest record number specified at initialization.

The REL_FILE.S file has the source for routines to open, initialize, read, write and delete relative files. Due to internal format differences, you may not read or write relative files as ordinary sequential files, or by using the Executive or Editor (exception: you may DELETE or RENAME relative files). In particular, if you try to TYPE or COPY a relative file from the Executive, you will get a "FILE NOT FOUND" error because the type of the file is not sequential. Do not use DYNODISK with Relative files.

To use the relative file routines, put the following statement in your program before the first reference to the routines:

```
INCLUDE REL_FILES
```

The following subroutines are provided:

PROC REL_OPEN OPEN RELATIVE FILE

USAGE: **REL_OPEN** Filename, Recsize

Procedure REL_OPEN opens a relative record file. **Filename** is a string containing the desired file name. This may be any legal Commodore filename, but we suggest you use a legal PROMAL file name with a ".R" extension. **Recsize** is an argument of type BYTE specifying the size of each record, which may be 1 to 254. It is the programmer's responsibility to insure that the file is opened with the same record size every time.

In planning your record size, remember that the record size should be 1 greater than the actual maximum number of characters you plan to use in the record, to allow for the Carriage Return (CR) terminator which will be appended automatically to each record on disk. The 1541 drive only allows one relative file to be open at a time. REL_OPEN must be called prior to any other relative file routines.

EXAMPLE:

```
CON RECSIZE=81                  ; Up to 80 chars in a record
DATA BYTE FILE="INVENTORY.R"    ; Filename to be opened
...
REL_OPEN FILE,RECSIZE           ; Open relative file for I/O
...
```

PROC REL_INIT INITIALIZE RELATIVE FILE

USAGE: **REL_INIT** Numrecs

Procedure REL_INIT initializes a previously opened relative record file and specifies the maximum number of records to be allocated. Each record is initialized to "empty" (a null string). **Numrecs** (type WORD) is the desired maximum number of records. If this number is large, the initialization could take several minutes. It is only necessary to initialize a relative file when it is first created (after opening it) or when enlarging the maximum number of allowable records. It is not necessary (or desirable) to initialize it each time you open it. To enlarge the file for additional records, you can call **REL_INIT** again with **Numrecs** specifying the new maximum. Records previously written will not be affected.

EXAMPLE:

```
CON RECSIZE=81                  ; Up to 80 chars in a record
CON NUMRECS=200
DATA BYTE FILE="INVENTORY.R"
...
REL_OPEN FILE,RECSIZE           ; open relative file
REL_INIT NUMRECS                ; initialize file with null records
...
```

PROC REL_WRITE WRITE RECORD TO RELATIVE FILE

USAGE: **REL_WRITE** Recnum, Record

Procedure REL_WRITE is used to write a particular record in an open and initialized relative file. **Recnum** is the desired relative record number (type WORD), and **Record** is a string containing the text of the desired record. The string does not have to include a carriage return; one will be appended when the record is written to disk. The record written must not be longer than the record size which was specified when the file was open.

If the record was previously written, the new record replaces it in its entirety, even if the new record is shorter than the record it replaces. **Recnum** must be between 1 and the value specified for **Numrecs** when REL_INIT was called, inclusive. The string written should not contain a byte of $FF (255). Naturally it cannot contain any $00 bytes either since this is the string terminator in PROMAL.

EXAMPLE:

```
CON RECSIZE=50 ; Up to 49 chars in a record
CON NUMRECS=300
BYTE LINE[81]
BYTE INDEX
WORD RECNUM
DATA BYTE FILE="MYDATA.R"
...
REL_OPEN FILE,RECSIZE
...
PUT NL,"WRITE WHAT RECORD NUMBER ? "
GETL LINE
INDEX=STRVAL(LINE,#RECNUM)
PUT NL,"CONTENT OF RECORD ?"
GETL LINE
REL_WRITE RECNUM, LINE
...
```

The program fragment above prompts for entry of a record number and a line of text to be the desired record. It then writes the record specified.

PROC REL_READ READ RECORD FROM RELATIVE FILE

USAGE: **REL_READ** Recnum, #Buffer

Procedure REL_READ reads a specified record from an open relative record file and copies it to a specified buffer. **Recnum** is the desired record number (type WORD), between 1 and the value of **Numrecs** specified when the file was initialized. **#Buffer** is the address of the desired buffer to hold the record, which should be at least as large as the record size specified when the file was opened. The CR terminating the record on disk is not returned in the buffer; it is replaced with a $00 byte so the buffer can be treated as a standard PROMAL string. A record which has never been written will return a

null string without error.

EXAMPLE:

```
CON RECSIZE=50 ; Up to 49 chars in a record
CON NUMRECS=300
BYTE LINE[81]
BYTE INDEX
WORD RECNUM
DATA BYTE FILE="MYDATA.R"
...
REL_OPEN FILE,RECSIZE
...
PUT NL,"READ WHAT RECORD NUMBER ? "
GETL LINE
INDEX=STRVAL(LINE,#RECNUM)
REL_READ RECNUM, LINE
PUT LINE,NL
...
```

The program fragment above prompts for a record number and displays it on the screen, followed by a carriage return.

PROC REL_DELETE DELETE RELATIVE FILE

USAGE: **REL_DELETE** Filename

Procedure REL_DELETE is used to delete **an entire relative record file.** The file should be closed when REL_DELETE is called. All records will be discarded and the file space reclaimed for future use on the disk. **Filename** is a string containing the name of the file. The message

"01, FILES SCRATCHED, 01, 00"

will be displayed on the screen. This is not an error.

EXAMPLE:

```
...
REL_DELETE "MYDATA.R"
...
```

PROC REL_CLOSE CLOSE RELATIVE FILE

USAGE: **REL_CLOSE**

Procedure REL_CLOSE closes the previously-opened relative record file. No error occurs if the file is not open. This procedure should be called before exiting from any program which has opened a relative file, or when done with the file. Note that it is normal for the red light on the 1541 drive to be on the entire time a file is open. Because it is important to properly close the file, it is suggested that CTRL-STOP not be used to exit from a program which has opened a relative file.

EXAMPLE:

```
...
REL_OPEN "MYDATA",40
...
REL_CLOSE
...
```

Note that if you INCLUDE both RS_232 and REL_FILES in a single program, you will get some duplicate identifier errors when you compile, because both packages use and declare some of the same Kernal entry points. To correct this situation, simply copy whichever of these files is second in your program to another file, and edit it to delete the duplicate declarations.

The REL_FILE package requires version 1.1 or later of PROMAL.

RELDEMO PROGRAM

The PROMAL System disk contains a file called RELDEMO.S. This file is a simple demonstration program for relative files using REL_FILE.S support. It opens a relative file called "TEST_REL.R" for up to 20 records of 40 characters each, and prompts you to read or write selected records. The first time you run RELDEMO, you should select "initialize" from the menu before reading or writing records. A menu option is provided to delete the entire file if you no longer want it on your disk.

You can study the RELDEMO.S program for more information about using relative files. Since the REL_FILE.S support package is provided in PROMAL source form, you may also wish to study it to see how to use PROMAL to interface to the Commodore Kernal routines. Advanced users may even wish to use the same techniques to write their own direct-access disk routines. If you do decide to write your own disk-support routines in PROMAL, please note the following:

1. The PROMAL nucleus **always has channel 15 open to the disk command/error channel.** The routine REL_CHECK in REL_FILE.S provides a way to read the error channel.

2. PROMAL allocates C-64 channels in ascending order, with the secondary address the same as the channel number. You should pick "high" channels and secondary addresses (9 or 10 recommended) to keep out of PROMAL's way.

This page is intentionally left blank

APPENDIX N

OPERATING SYSTEM NOTES

COMMODORE 64

PROMAL Version 2.0 has full support for two disk drives. These can be either two 1541-type drives (one as device 8 and one as device 9), or dual diskette drives such as the SD-2 by Micro Systems Development (MSD), with drive numbers 0 and 1 on Commodore device 8. The default drive is drive 0. Files in drive 1 are designated by a "1:" prefix as part of the file name.

If a drive number is not specified as a prefix, the default drive is always drive 0 (device 8). A prefix of "1:" will access drive 1 (device 9). You always "boot up" PROMAL from drive 0. When compiling programs with INCLUDE files, the INCLUDE file name may have a drive prefix.

As shipped from SMA, PROMAL is set up to use one 1541 drive. If you wish to use a dual drive MSD system, you should disable DYNODISK permanently and set the device numbers for both drives to 8. This should be done from BASIC as follows:

```
LOAD "PROMAL",8
POKE 3553,8   :rem makes logical drive 1 device 8, not 9
POKE 3554,128 :rem defeats DYNODISK permanently
```

Insert a formatted disk in drive 0 and type:

```
SAVE "PROMAL",8,1
```

Then put the PROMAL diskette back in and type:

```
RUN
```

After the system is booted up, copy the rest of the files to your new disk. You can use a commercial copier to do this if you wish, or use the COPY command.

ELAPSED TIME FUNCTION

On the Commodore 64, you can read the "jiffy" clock using this function, TIME, which returns the clock reading in "jiffies" (1 jiffy = 1/60th of a second) as a REAL number:

```
EXT BYTE THI AT $A0
EXT BYTE TMED AT $A1
EXT BYTE TLO AT $A2
FUNC REAL TIME
BEGIN
RETURN TMED:+<<8 + TLO +65536.*THI
END
```


APPLE II

PROMAL uses the Apple ProDOS operating system. Users who are accustomed to operating under DOS 3.3 will find a utility program on the ProDOS Utilities Diskette (available at your Apple Dealer) which can convert your existing text and data files to ProDOS format so that you may use them with PROMAL.

Some non-PROMAL programs for the Apple produce text files with the high-order bit of each character set. These files may be converted to standard ASCII files (as expected by the PROMAL editor and other PROMAL programs) by using the CLEARBIT7 utility program on the PROMAL System diskette. The command syntax is:

CLEARBIT7 Oldfile Newfile

where **Oldfile** is the name of the file to be corrected and **Newfile** is the desired name for the corrected file to be written. The CLEARBIT7 utility also truncates any lines longer than 125 characters, so that the resulting **Newfile** will be acceptable to the PROMAL EDITOR.

SPECIAL PRODOS FUNCTIONS

The file PRODOSCALLS.S contains a source program fragment suitable for calling special ProDOS functions (such as testing or setting file attributes) which are not covered by built in LIBRARY routines. The ProDOS Technical Reference Manual contains all the neccessary details.

APPENDIX O

FORMATTING, BACKING UP, & MAKING WORKING DISKS

Apple II

One of the first things you should do with your all your PROMAL distribution disks is to **make at least one backup copy. Be sure to read the License agreement before opening your sealed diskette.** It is important to make a copy of your diskettes and only work with the copy, so that in the event that any files are accidentally deleted, you can always get a new copy from the original disk. Note that only the Demo disk is bootable. You may make backup copies for your own personal use subject to the license agreement. Making copies other than as permitted by the license agreement is a violation of Copyright Law and is a crime.

For the **Apple II**, you may back up your PROMAL diskettes using the ProDOS Utility Diskette which came with your Apple IIe or IIc (available from your Apple dealer). You should also use this program to format new blank diskettes before using them with PROMAL. It is a good idea to write the volume name of every diskette on the label using a soft marker. **All disks should have unique volume names.** To make a "working disk" which can boot up PROMAL and do development, use the ProDOS file copier or the PROMAL COPY or EXTCOPY commands to copy the following files from the PROMAL disk to your newly-formatted diskette:

```
PRODOS
PROMAL.SYSTEM
EDIT.C
EXECUTIVE.C
COMPILE.C       ; Either Demo compiler or full
LIBRARY.S
EXTDIR.C
COMPERRMSG.T
EXTCOPY.C       ; If desired
```

All the files above except the full compiler are on the Demo disk. COMPILE.C, EXTDIR.C, COMPERRMSG.T and EXTCOPY.C are not necessary to boot up the system, but you will usually want to have them on the disk if you will be developing any programs. Once your system is booted up, you can develop programs with a disk which only has COMPILE.C and COMPERRMSG.T (and perhaps EXTDIR.C) on it, to leave more room for your source and object programs. Remember to issue a PREFIX * command when changing disks. You may wish to copy other files such as PROSYS.S or REALFUNCS.S on an as-needed basis.

If you have a /RAM disk, you may want to set up a BOOTSCRIPT.J file to copy your working files to 0:. EXTCOPY is convenient for doing this.

It is a good practice to keep at least one formatted, blank diskette available at all times, with the Volume name clearly marked on the label. This will come in handy the first time you type in a big program with the Editor only to discover there's no room left on your working disk to save it! To extricate yourself, save to W and exit to the EXECUTIVE. Then use PREFIX * to select your blank disk and COPY W Filename to save your file on disk.

Commodore 64:

For the Commodore 64, the DISKETTE utility is a PROMAL program which provides the following disk and file maintenance services:

1. Duplicate an entire diskette using a single 1541/1571 disk drive, two 1541/1571 drives, or an MSD dual disk drive.
2. Format ("New") a diskette.
3. Copy a file to another diskette.
4. Erase files.
5. Rename a file.
6. Display file names (directory).
7. Change a diskette name or ID.

Several of these services such as copying, erasing, renaming and displaying the directory may also be done with built-in EXECUTIVE commands. The DISKETTE utility can copy, delete, and rename files of PRG type or SEQ type, with any legal Commodore name.

Probably your first use of the DISKETTE utility will be to make a backup copy of the PROMAL SYSTEM DISK or PROMAL DEVELOPER'S DISK. Be sure to read the License agreement before opening your sealed diskette(s). It is important to make a copy of all the PROMAL distribution diskettes and only work with the copy, so that in the event that any files are accidentally deleted, you can always get a new copy from the original disk. You may make backup copies for your own personal use subject to the license agreement. Making copies other than as permitted by the license agreement is a violation of US Copyright law and is a crime.

It is very easy to duplicate the PROMAL SYSTEM DISK, although rather time consuming because of the slow operation of the Commodore 1541 drive; it takes about 15-20 minutes to back up a PROMAL distribution disk. However, this is time well spent. If you are fortunate enough to have a dual drive system, it only takes 2 minutes. If you have a commercial fast-copier, you may use that. It is a good idea to put a write-protect tab on your PROMAL diskette before proceeding.

To run the DISKETTE utility, put the PROMAL Demo Disk (or a copy) in the drive and type this command from the EXECUTIVE:

```
--> DISKETTE
```

The screen will clear and a menu similar to this will be displayed:


```
PROMAL DISKETTE UTILITY 2.0

        MENU

Q = QUIT (TO EXECUTIVE)
D = DUPLICATE ENTIRE DISK
N = NEW (FORMAT) DISK
C = COPY A FILE
E = ERASE (DELETE) FILE(S)
F = FILE NAMES DISPLAY (DIRECTORY)
R = RENAME FILE
A = ALTER DISK NAME OR ID

YOUR SELECTION? _
```

Press D and RETURN to duplicate an entire disk. Then just follow the instructions. You will be asked if it is okay to unload the EDITOR to increase the size of the copy buffer; type Y and return. You may use the RETURN key by itself for a "yes" reply to questions needing a yes-or-no answer. You will be prompted when to swap disks if you have a single drive. When your duplicate disk is finished, the menu will be redisplayed. Press Q and RETURN to exit to the EXECUTIVE.

MAKING WORKING DISKETTES

For your normal operations, you will not need most of the files supplied on the PROMAL disks, but will want the PROMAL system so that you can "boot up" PROMAL from your working disk. The best way to do this is to use the DISKETTE utility to format a new disk and then copy only the files you want.

To format a disk using DISKETTE, select the N option from the menu and press RETURN. Again, just follow the directions. When prompted for a 2 character ID, pick any two characters that you have not used when formatting another disk. It is very important to use a different ID on each of your disks. This is how the 1541 disk drive DOS figures out when you have changed disks. If you have two diskettes with the same ID but different contents and swap them, the directories and files may be corrupted.

After formatting your new diskette, insert your copy of the PROMAL Demo disk in the drive and select the C option from the menu. Copy the following files one at a time using the C option:

```
PROMAL
EDIT.C
EXECUTIVE.C
COMPILE.C     ; Or the "full" compiler from the sealed disk
LIBRARY.S
COMPERRMSG.T
DATE.C
```

Note that unlike the EXECUTIVE COPY command, **you must type the ".C" extension explicitly** when copying files with DISKETTE. Also, since DISKETTE can copy files with any legal Commodore name (of type PRG or SEQ), **you must be careful to type in upper case letters if the file you want is in upper case** (you may wish to use CTRL-A to lock uppercase alphabetic characters).

The list of files above is the basic working set needed to boot PROMAL and develop software. You may also wish to copy DISKETTE.C, REALFUNCS.S or other programs. If you plan to boot up using another disk, then you only need to copy COMPILE.C and possibly EDIT.C if you will be compiling with the "B" (big program) option. It is possible to boot up without EDIT.C, COMPILE.C, COMPERRMSG.T, and DATE.C, but you will normally want these.

The size file which can be copied is limited to the size of the available buffer space, normally about 26K bytes. You can copy files of up to 64K bytes using the standard EXECUTIVE COPY command (more if you have 2 drives).

MISCELLANEOUS OPERATIONS

You can also delete and rename files with DISKETTE. When deleting, note that wildcards are acceptable when you are prompted for a file name to delete. Be very careful when using wildcards; there is no prompt for a chance to change your mind! When the deletion is completed, the standard Commodore disk message will indicate the number of files deleted ("scratched"). For example:

```
01, FILES SCRATCHED,02,00
```

indicates two files were deleted (the number after "FILES SCRATCHED", not before it!). If the message indicates 00 files scratched, you probably spelled the name wrong (don't forget you have to add .C explicitly for PROMAL programs and match upper and lower case exactly).

APPENDIX P

PROMAL SYNTAX DIAGRAMS

The syntax diagrams on the following pages provide definitive reference for statement construction in PROMAL. If you are not familiar with syntax diagrams, then study the narrative which follows while referring to the named diagrams.

HOW TO READ SYNTAX DIAGRAMS

Consider the first path of the **STMT** diagram, which is the syntax diagram which shows you how to construct an assignment statement.

The symbols shown inside ovals are keywords or punctuation which must be typed exactly as shown. The symbols shown in rectangles describe things which you, the programmer, must supply. The lines connecting the ovals and rectangles show all the legal paths which you may take. For example, to make an assignment statement, the first thing you need is a VAR. This is the variable name to receive the result of the assignment. Exiting from the right side of the VAR. rectangle, we see that we have a "fork in the road", meaning we can take any of the paths. If we go "straight ahead" on the middle track, we come to an oval with an equals sign in it. Since this is an oval, we would write the equals sign. Finally, we come to a rectangular box called EXP. This stands for "expression", which means we can put any kind of expression there.

We already know about forming expressions. Probably the simplest expression is just a literal number. Therefore a legal assignment statement could be:

```
X=0
```

which sets the variable X to 0. We also know that a variable can be used for an expression, so another legal assignment statement would be:

```
ZVAL=X
```

We are assuming that X and ZVAL have been declared previously. We also know that more complicated expressions can be formed with operators. For example:

```
CMIN=(ZVAL-1)/2
```

But how do we find out EXACTLY what is a legal expression on the right side of an assignment statement? For example, is this legal?

```
VB=X OR Y
```

To find out, we consult the syntax diagram which defines an expression. Since you already know that expressions can be quite complicated and involve many possibilities, you might expect that the syntax diagram for an expression is also complicated. In fact, it is the most complex element of the language. Let's look at the EXP syntax diagram, which appears deceptively simple.

The EXP starts with something called a **RELATION**. Consulting the syntax diagram for a RELATION, we find it in turn starts with a SIMPLEXP. This is getting complicated! A SIMPLEXP can start either with a - sign or a TERM. Since our case doesn't start with a - sign, it must be a TERM (if it's legal). A TERM starts with something called a FACTOR.

A **FACTOR** starts with a lot of choices immediately. One of these choices (on the eighth line down) is "VAR.", which stands for variable. Now we are getting some place! We know X is a variable, but can our variable X be followed by the keyword "OR"? Follow the path out of the VAR rectangle, up and out of the diagram for FACTOR. We have now completed FACTOR, but we remember that FACTOR was just the start of a TERM, from the diagram above it. Tracing the path from the FACTOR rectangle we see several choices (*, /, etc.), but none of them are OR, so we continue to the right, exiting the TERM diagram.

Also, we remember that TERM was encountered in the SIMPLEXP diagram, so we follow the arrow out of the SIMPLEXP diagram, since there are no paths there that lead to OR. Returning to the diagram for RELATION after the SIMPLEXP box, we again find no path leading to an OR, so we exit to the right. Finally, we come back to the EXP diagram, and following the RELATION rectangle, we find a path that leads to OR. Therefore we know we can have a variable followed by OR. Following the path from the OR rectangle, we see we must come to RELATION again. Therefore we must have another RELATION after the OR. But since we already know from working our way down to FACTOR before that a relation can be a variable, we know that our statement is legal since Y is a variable.

Before finally concluding that our statement is legal, though, we must make sure that nothing else is required to follow what we already have. To do this, we must trace a path from the exit of the RELATION box to the exit of the EXP box, and then from the exit of the EXP box in the ASSIGN STMT diagram to the end-of-line symbol. The end-of-line symbol is shown as a down-pointing arrow in a circle, symbolically representing a carriage return.

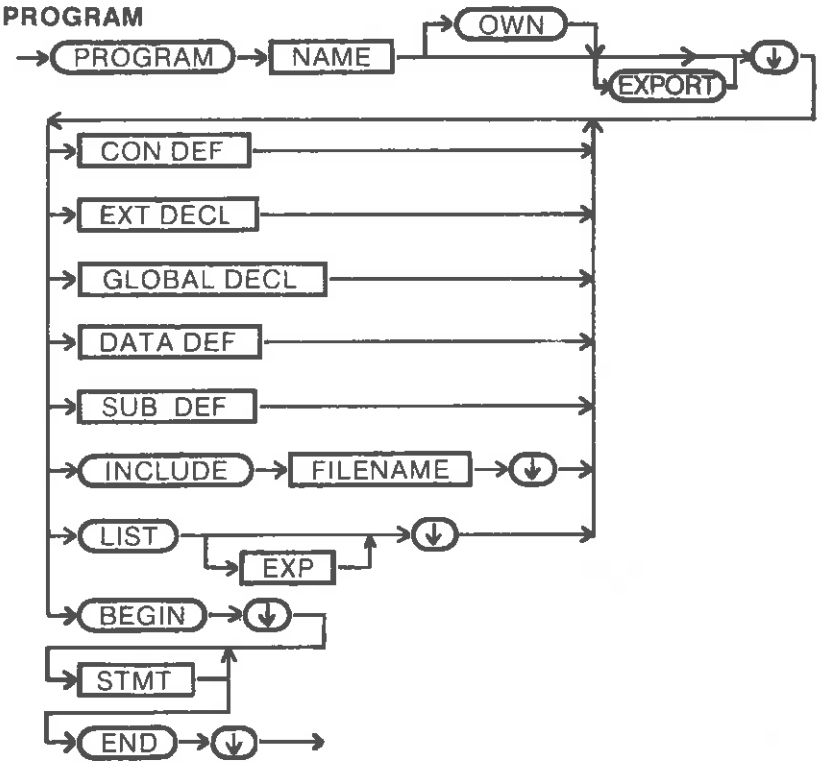
Naturally you won't consult the syntax diagrams every time you write a statement! But if in the process of writing a program, if the compiler gives you an error message, and it is not obvious what is wrong, you can always consult the syntax diagrams to help find out what the problem is.

As an aside for the technically curious, you might be interested to know that the PROMAL COMPILER is really just a PROMAL program which tries to match your source program to the syntax diagrams! Each syntax diagram in this Appendix is implemented as one subroutine in the compiler. For instance, the COMPILER contains procedures called EXP, SIMPLEXP, RELATION, TERM, and FACTOR. Each of the "forks in the road" in the syntax diagram corresponds to an IF statement in one of these routines. To "parse" your assignment statement, the ASSIGNSTMT routine calls the EXP routine which calls the SIMPLEXP routine, etc., in the same manner as we just traced through the syntax diagram. If the compiler gets to a point where the next thing in your program doesn't match any of the choices, it prints an error message.

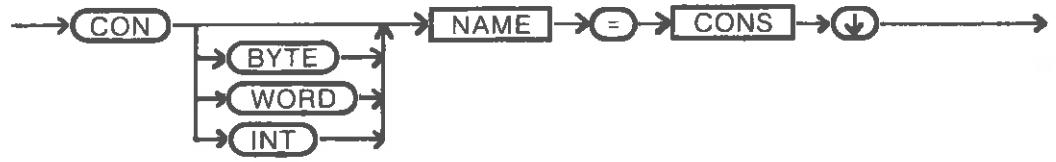
Now that you know how to read syntax diagrams, which are the "authority" on what is legal in PROMAL, you can refer to these diagrams whenever you have a question about the "legality" of a particular PROMAL statement.

Symbol	Meaning
	A syntactically legal path.
	A literal symbol, to be entered as shown.
	A user-supplied item based on another diagram.
	End-of-line (Carriage Return or comment).
	Indent one level.
	'Exdent' (opposite of indent) one level.

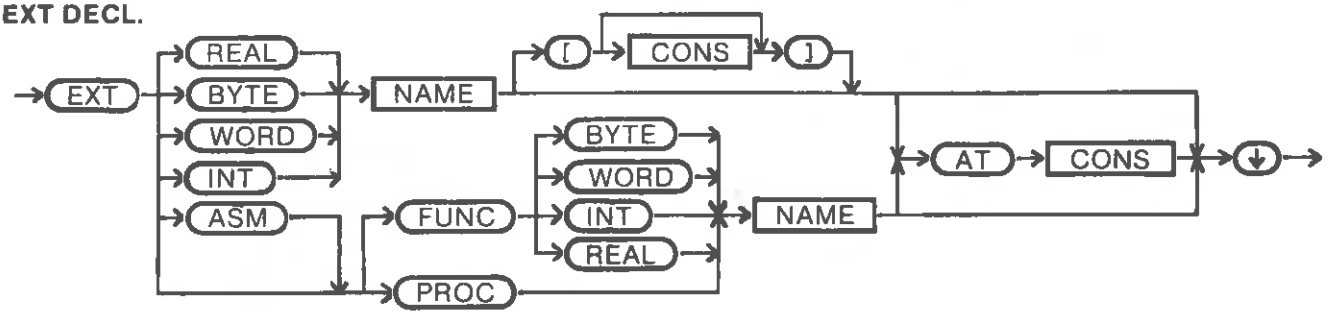
PROGRAM



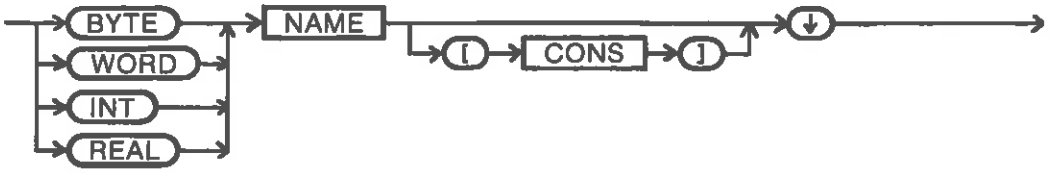
CON DEF.



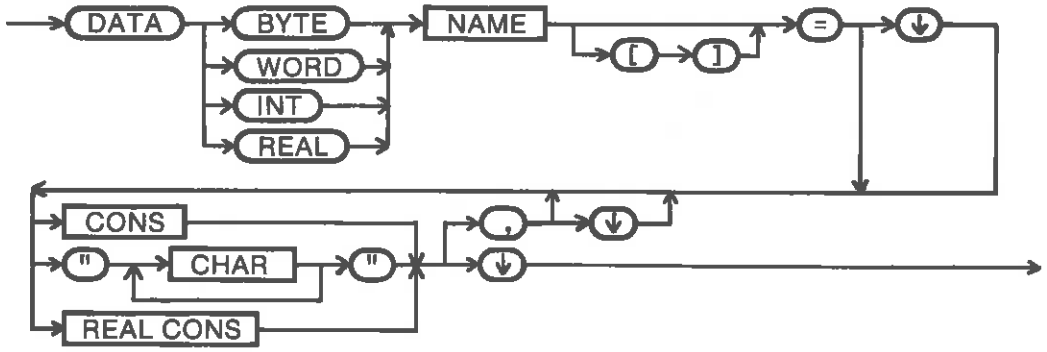
EXT DECL.



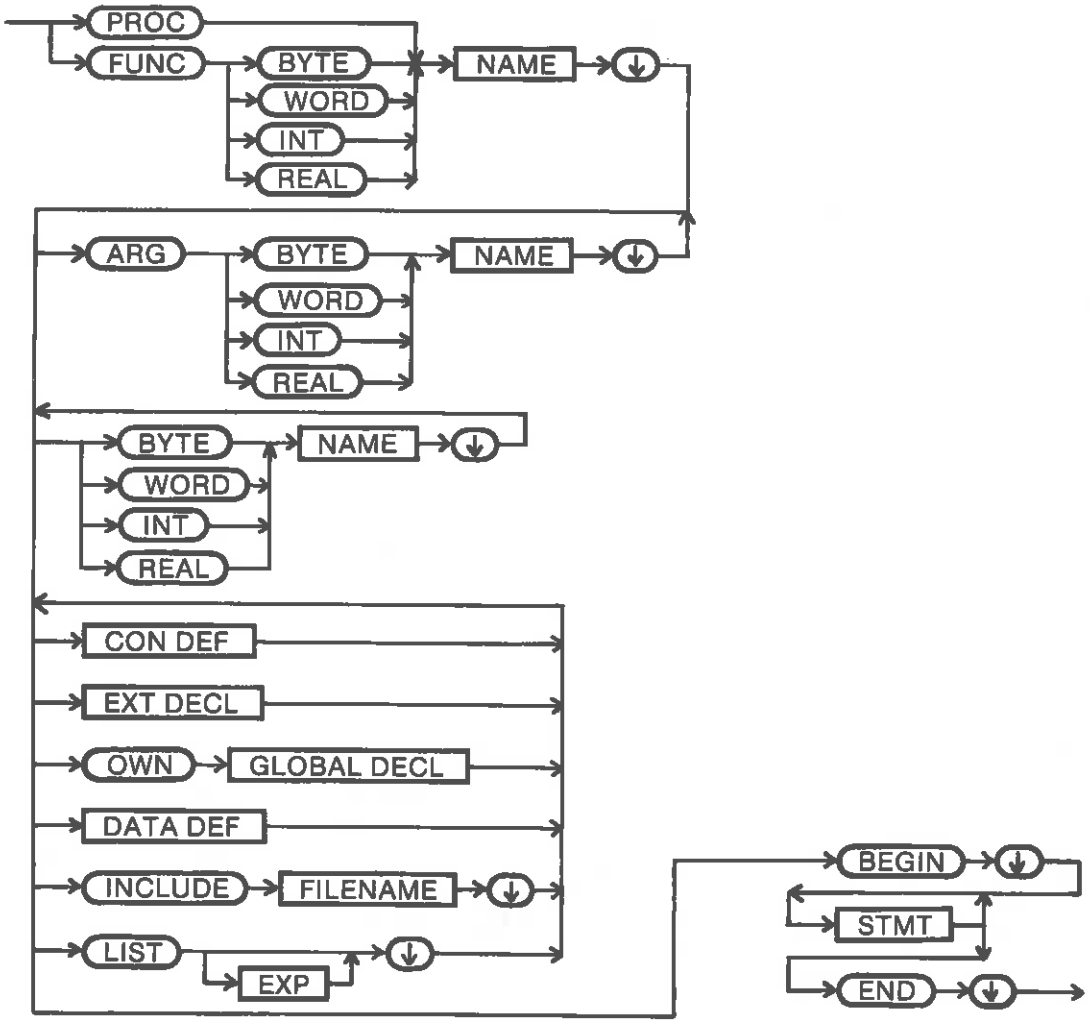
GLOBAL DECL.



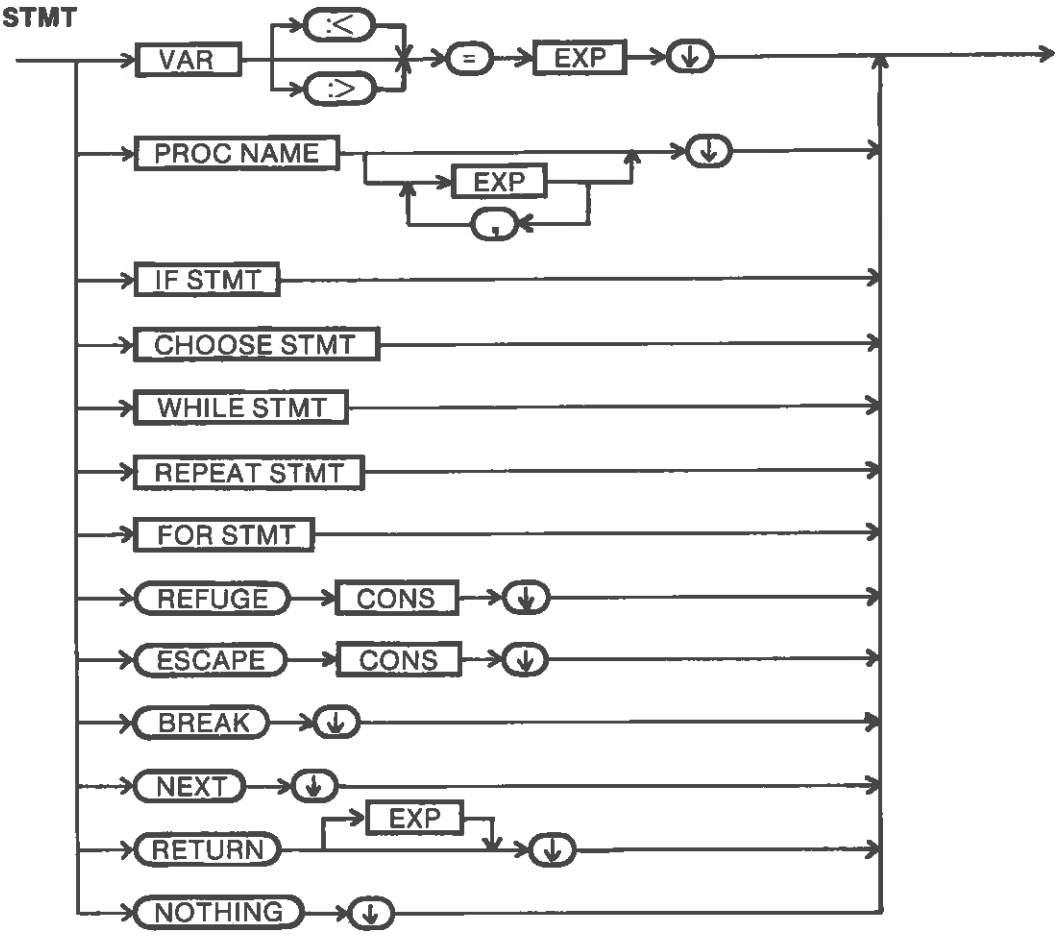
DATA DEF.



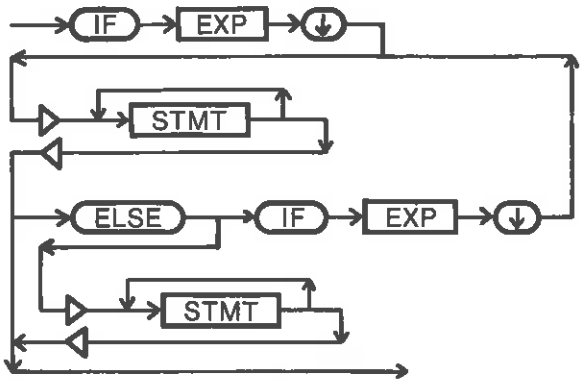
SUB. DEF.



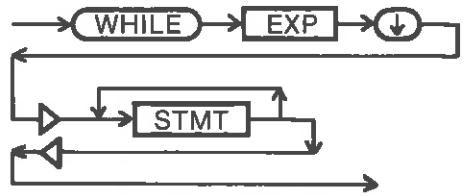
STMT



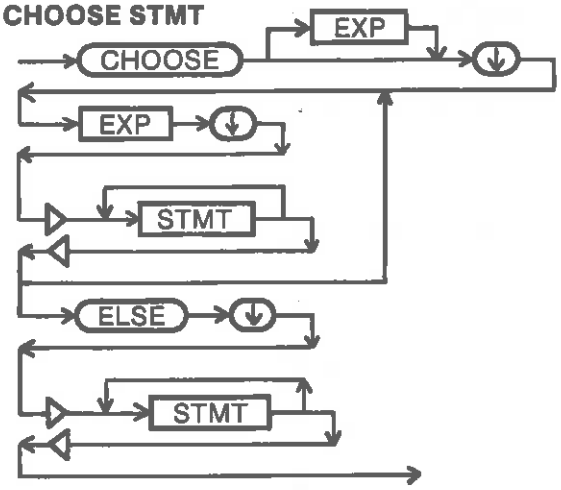
IF STMT



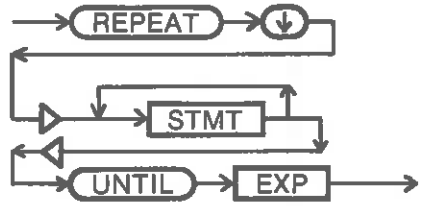
WHILE STMT



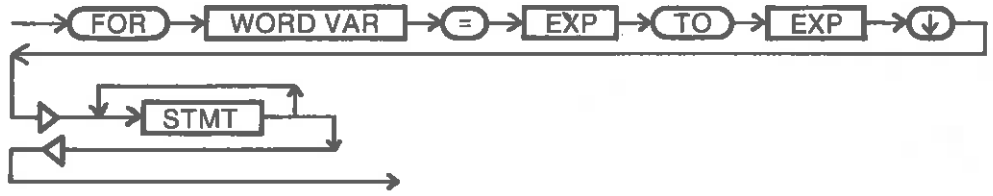
CHOOSE STMT



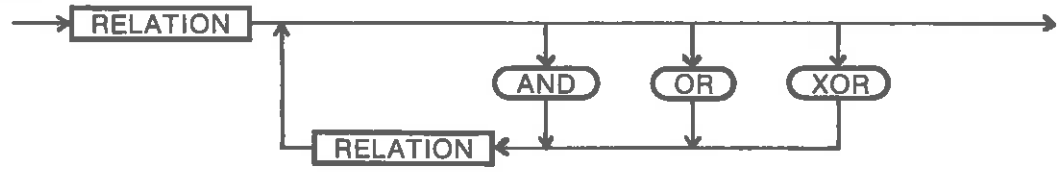
REPEAT STMT



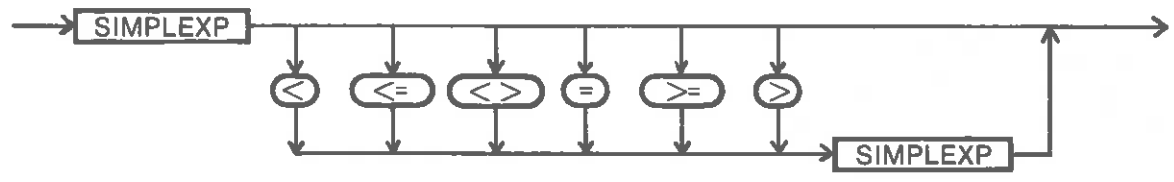
FOR STMT



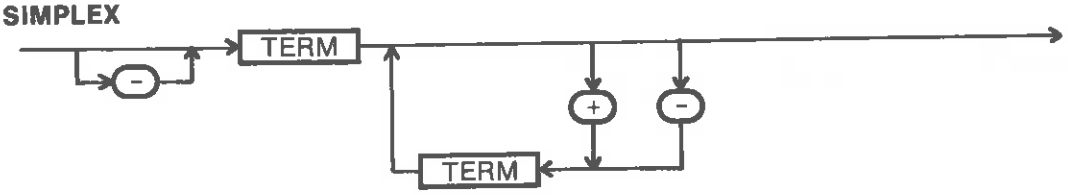
EXP



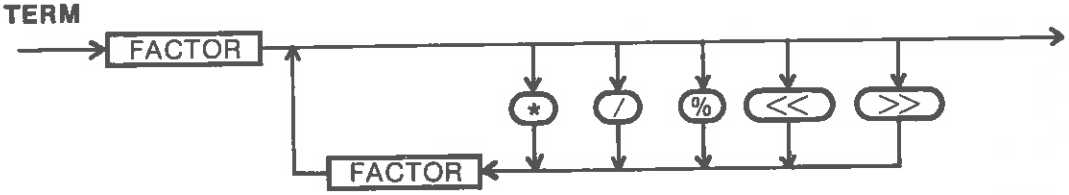
RELATION



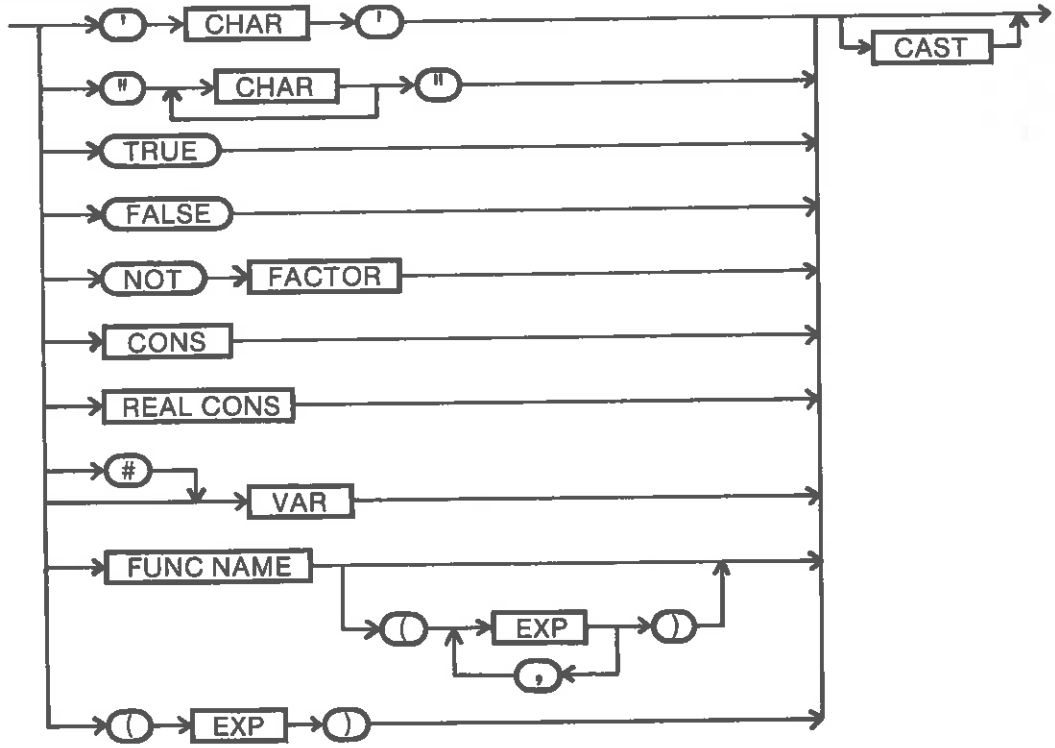
SIMPLEX



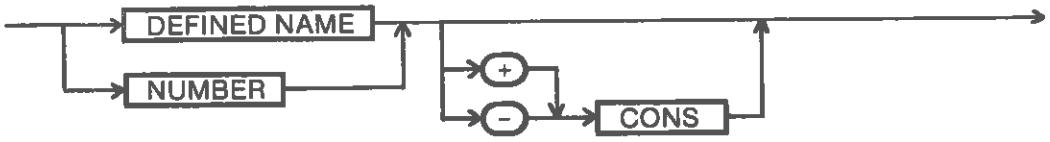
TERM



FACTOR

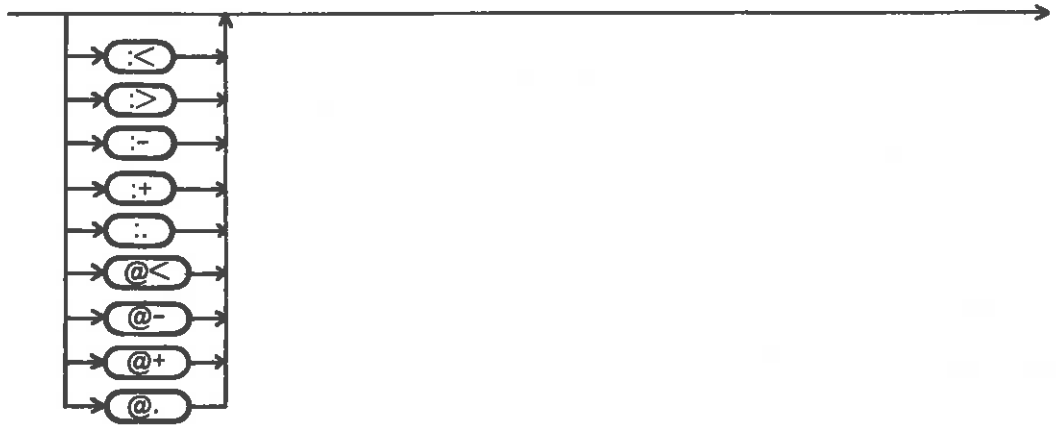


CONS.



VAR.

CAST



APPENDIX Q

PROMAL DEMO PROGRAMS

A number of demonstration programs are provided on the PROMAL System disk or optional Developer's disk, as well as on the Demo disk. Several demo programs were discussed in the **MEET PROMAL!** manual. You may compile and run these demonstrations, and you can use the EDITOR to extract parts of them to use in your own programs. By studying the programs you can learn many valuable techniques. Below is a summary of most of the demonstration programs provided.

BILLIARDS.S **(COMMODORE 64 ONLY)**

This program is discussed briefly in the **MEET PROMAL!** manual. It makes extensive use of sprites for animation of a billiards game. It also uses real math extensively for computing the motion of the balls, and has many conversions (type casts) between real and byte data types.

BUDGET.S

This is a very simple demonstration program which illustrates how to format real numbers for output. It uses the file BUDGETDATA.D for data. The file BUDGETDOC.T provides more information.

CALC.S

This program, when compiled, provides a demonstration program which simulates a four-function calculator with 26 memories (named A through Z). This program is discussed in **MEET PROMAL!**. To start the program, just type CALC, and follow the directions. The Demo disk has the source code for CALC, which illustrates how to write a recursive-descent expression evaluator in PROMAL. This program is only about 180 lines long (excluding comments), yet it can parse and evaluate arbitrary arithmetic expressions with nested parentheses.

CHECKSUM.S

This program computes the checksum of a specified block of memory. It is useful for determining if any bytes in a block of memory have changed. The comments provide further information on operation and theory.

CLEARBIT7.S **(APPLE II ONLY)**

This program will be found on the System Disk rather than on the demo disk. It is used to convert text files generated by other Apple II software which sets bit 7 of each byte to 1, to standard ASCII format for use with PROMAL. It also truncates lines longer than 125 columns, making files acceptable to the PROMAL EDITOR. It illustrates how to write a simple file filter in PROMAL.

DUMPFILE.S

This program is a useful utility which allows you to display the contents of any file in hex and ASCII, similar to the way the DUMP command displays memory. Any type of file can be dumped. The command syntax is:

```
DUMPFILE Filename [Type]
```

This will display the first 256 bytes of the file. The **Type** argument is needed only on the Commodore 64 for file types other than SEQ. No default file extension is used, so be sure to specify ".C" when dumping compiled programs. Pressing the RETURN key will display the next 256 bytes. Pressing any other key will terminate the program. DUMPFILE supports output redirection, so you can dump a file to the printer. The source and object code for DUMPFILE are on the Demo disk or the System disk. This program illustrates conditional compilation, so you must specify COMPILE DUMPFILE V=A on the Apple or V=C on the Commodore to compile the program. See the comments for more information.

FILECRC.S

This program computes the "Cyclic Redundancy Check" of a file and displays it. This is useful for comparing two files to see if they are identical. The comments in the source file explain the program operation and theory. It contains good examples of bit manipulation operators.

FIND.S

This program is discussed at length in the **MEET PROMAL!** manual. It searches a file for lines containing a specified string and displays these lines.

GRAPHDEMO.S **(COMMODORE 64 ONLY)**

This is a demonstration of high-resolution graphics using PROMAL. The program is self-explanatory when executed. The source code includes procedures for defining and clearing the hi-res screen, and drawing points and lines. These routines can be extracted using the editor for use with programs of your own design. NOTE: This program does **not** use or require the GRAPHICS TOOLBOX. The GRAPHICS TOOLBOX provides **much** higher performance and is much easier to use.

INFILTRATOR.S **(COMMODORE 64 ONLY)**

This is a fairly large and complex demonstration illustrating animation using screen scrolling, sprites, joystick input, and sound synthesis. You will need to issue a WS 0 command to edit this file and compile it using the B option (full compiler). This program is an excellent example of how to make good use of PROMAL procedures and functions to simplify a complex program.

RELDEMO.S **(COMMODORE 64 ONLY)**

This program is a simple demonstration of relative files. It is explained in **Appendix M.**

RELOCATE.S

This program converts machine language programs into relocatable form for use with the PROMAL loader. It is described in **Appendix I.** This program uses conditional compilation, so read the comments in the source before compiling. This program uses include files RELOCAPL.S or RELOCC64.S.

SORTDEMO.S **(APPLE II ONLY)**

This program is described briefly in **MEET PROMAL!.** It provides a demonstration of formatted output, file access, printer access, and general techniques.

SORTSTRING.S

This program provides a general shell sort routine for sorting arbitrary string arrays, and illustrates how to generate an array of strings read from a file. See the comments in the source file for usage.

SPLIT.S

This is a utility program which can be used to split a text file which is too large to edit into two smaller files. The file can be split after a specified number of lines, or before a line containing a specified string. The command syntax is:

```
SPLIT Sourcefile Firstfile Secondfile Count
or
SPLIT Sourcefile Firstfile Secondfile String
```

where **Sourcefile** is the file to be split, **Firstfile** is the name of the file to receive the first part of the file, and **Secondfile** is the name of the file to receive the other part of the file. No default file extensions are provided, so be sure to include ".S" for normal source files. **Count** is the number of lines to be copied from **Sourcefile** to **Firstfile**; the rest will go to **Secondfile**. **String** is a non-numeric string. The first line containing **String** anywhere in the line will be the first line sent to **Secondfile** when this form is specified. See the source program file for further information.

SSEND.S and SRECEIVE.S

This complementary pair of programs provides for error-free serial transmission of any type of files between Apple II computers at up to 9600 baud, and up to 600 baud for Commodore 64. Files can be exchanged between Apples and Commodores, too. The programs are described in **Appendix F.** The subroutines provided can be used as the basis for any kind of communications program.

TINYTERM.S

This program is a minimal implementation of a communications program for communicating over a modem to a time sharing service or bulletin board service. It is explained in **Appendix F.**

This page is intentionally left blank.

INDEX

A

ABORT (PROC).....4-5
ABS (FUNC).....4-5
ALPHA (FUNC).....4-6
AND operator.....3-17,3-21/22
ARG statement.....3-42
Arguments:
 -- cmmd line..1-14/15,1-21/22,3-56/57
 -- defining.....3-42
 -- passing.....3-42/45
 -- substituting in JOB file.....2-31
Arithmetic expressions.....3-17/20
Arithmetic operators.....3-17
Arrays:
 -- of DATA strings.....1-27,3-15/16
 -- of strings.....3-65
 -- declaring.....3-14/15
 -- multi-dimensional.....3-15,3-64
ASCII character set.....A-1
ASM routine, declaring..I-7/10,I-12/13
Assembly language subroutines:
 -- calling LIB routines from.....I-10
 -- interfacing to.....I-1/15
 -- relocatable.....I-11/15
Assignment statement.....1-19,3-26
AT keyword in EXT statement..1-29,3-58
ATAN (FUNC).....K-1
Audience, for PROMAL.....1-3

B

Backing up disks.....O-1/4
Batch job capability.....2-30/31
BEGIN statement.....3-41
BILLIARDS.S (sample program).....Q-1
Blank lines, as comments.....1-16
BLKMOV (PROC).....4-6
BOOTSCRIPT.J file.....2-30
Bootstrap, to control loading.....3-76
BREAK statement.....3-31/32
BUDGET.S (sample program).....1-24,Q-1
BUFFERS (APPLE II EXEC cmd).....2-15
Built-in functions & procedures...3-36
BYTE, data type.....1-18,3-8/9

C

CARG variable..1-14/15,1-21/22,3-56/57
CALC.S (sample program)....1-24/25,Q-1
Characters, literal.....3-10
Characters, string extraction.....3-23
Checksum.....1-13,4-7
Checksum, MAP display.....1-13
CHECKSUM.S (sample program).....Q-1
CHKSUM (FUNC).....4-7
CHOOSE statement.....3-30/31
CLEARBIT7.S (utility program)..N-2,Q-1
Clearing screen from program.4-43,4-45
CLOSE (PROC).....4-7
CMPSTR (FUNC).....4-8
COLOR (EXEC cmd).....2-16
Command line args.....1-14/15,3-56/57
Commands: (see EXECUTIVE, EDITOR)
 -- user defined.....2-4
 -- case insensitivity.....2-4
 -- line editing, (TABLE 1).....2-5/7
 -- notation conventions.....2-9
 -- system-dependent keys.....2-7
 -- EDITOR (TABLE 5).....2-47/49
 -- EXECUTIVE (TABLE 2).....2-8
Comment lines.....1-16
Compatibility with IBM PROMAL....L-1/2
COMPILE (EXEC cmd).....2-56/58
Compiler:.....2-2,2-56/60
 -- command options.....2-56/58
 -- cross reference utility.....2-60
 -- dialog.....2-58/60
 -- edit after error.....2-59
 -- introduction to.....2-56
 -- invoking.....1-11,2-56
 -- screen displays.....1-11
Compiling, conditional.....3-49/50
Compiling, sample program.....1-11
Compiling, very large prgrms...2-57/58
CON statement.....1-26,3-13
Conditional compilation.....3-49/50
Conditional stmts, short-cuts.....3-33
Constants, defining.....1-26,3-13
COPY (EXEC cmd).....2-17/19
Copyright Notice.....ii,1-2
COS (FUNC).....K-1
CS (EXEC cmd).....2-19
CTRL ^ (Adj. rt.-APPLE II)...1-28,2-48
CTRL \ (Clr. end -APPLE II)...2-5,2-48
CTRL [(Start of line -C64)...2-6,2-48
CTRL A (Alphalock).....1-9,2-6,2-48
CTRL B (Cmd recall).....1-9,2-6,2-49
CTRL C (Abort cmd. -APPLE II).....2-7
CTRL D (Del char.-APPLE II)...2-5,2-47
CTRL E (Insert -APPLE II).....2-5,2-47
CTRL F (Strt of ln -APPLE II).2-6,2-48
CTRL I (Indent).....2-48
CTRL J (Adj. rt. -C64).....1-28,2-48
CTRL K (Clr. end -C64).....2-5,2-48
CTRL L (End of ln -APPLE II)..2-5,2-48
CTRL N (Next page).....2-48
CTRL O (Adjust left).....1-28,2-49
CTRL P (Previous page).....2-48
CTRL Q (Un-indent -APPLE II).....2-48
CTRL RESET (Abort cmd.-APPLE II)...2-6
CTRL STOP (Abort cmd.-C64).....2-6

CTRL U (Un-indent -C64).....2-48
CTRL V (Normalize window -C64)....2-49
CTRL W (Set window -C64).....2-49
CTRL X (Clr line).....2-5,2-48
CTRL Y (End of line -C64).....2-5,2-48
CTRL Y (Home -APPLE II).....2-48
CTRL Z (End of file mark).....2-6
CTRL <-- (Del char. -C64).....2-5,2-47
CURCOL (FUNC).....4-9
CURLINE (FUNC).....4-9
CURSET (PROC).....4-10

D

DATA:
 -- arrays of strings.....1-27,3-16
 -- defining arrays of.....3-15/16
 -- definition.....3-15
 -- REAL.....3-15
 -- statement example.....1-27,3-15/16
Data communications support.....F-1/6
Data types.....3-8/9
Date, entering.....1-6
DATE (EXEC cmd).....2-20
DEL key (C64).....2-5,2-47
Delete key (APPLE II).....2-5,2-47
DELETE (EXEC cmd).....2-20/21
Demo diskette, limitations of.....1-3
Demo programs.....Q-1/3
Device names (TABLE 4).....2-13
Device numbers (C64).....E-1,N-1
DIR (FUNC).....4-10/11
DIROPEN (FUNC).....4-11
DISKCMD (EXEC cmd).....2-21/22
Disk drives, dual support of.....N-1
DISKETTE utility (C64).....O-2/4
DUMP command, example.....1-8
DUMP (EXEC cmd).....2-23
DUMPFILE.S (utility program).....Q-2
Dynamic memory allocation.....H-1
DYNO (EXEC cmd).....2-22/23
DYNODISK.....1-5,2-22/23,4-18

E

EDIT (EXEC cmd).....2-24,2-44
Editor:.....1-9,2-2,2-44/55
 -- CHANGE (F6).....1-26,2-51
 -- COPY (F7).....2-52
 -- char. sets & modes (C-64).....2-55
 -- cut & paste operations.....2-52
 -- DEL LN (F1).....2-49
 -- display format.....2-44/46
 -- EDIT (EXEC cmd).....2-24,2-44
 -- edit buffer & workspace....2-54/55
 -- editing keys (TABLE 5).....2-47/49
 -- entering from EXECUTIVE.....2-44
 -- features of.....2-44
 -- FIND (F5).....1-25/26,2-50/51
 -- FKEYs legend after MARK..1-27,2-52
 -- HELP (F7).....2-46
 -- initial screen display.....2-45
 -- inserting & deleting lines....2-49
 -- inserting block or file....2-52/53
 -- INS LN (F2).....2-49
 -- introduction to.....1-9/10
 -- invoking.....1-9,2-24,2-44
 -- MARK (F3).....1-27,2-52
 -- MOVE (F6).....2-52
 -- QUIT display.....1-10,2-53
 -- QUIT (F8).....2-53
 -- RECALL (F4).....2-49,2-53
 -- sample sessions.....1-9/10,1-25/29
 -- saving block to file.....2-52/53
 -- scrolling.....2-46
 -- search & replace.....1-26,2-51
 -- searching.....2-50/51
 -- status area.....2-46
 -- WRITE (F4).....2-52/53
EDLINE (FUNC).....4-12/14
Error messages.....C-1/12
Errors:
 -- from LOADer.....3-71
 -- after OPEN.....4-36
Executing sample program.....1-12
Executive:.....2-2/43
 -- arguments for commands.2-9,2-12/13
 -- arg passing fm cmd line....3-56/57
 -- commands, search order.....2-4
 -- commands, summary (TABLE 2)....2-8
 -- entering commands.....1-6/7
 -- guided tour of.....1-6/9
 -- HELP screen.....1-8
 -- line editing keys.....2-5/6
 -- user defined commands.....2-4
EXIT (PROC).....4-14
EXP (FUNC).....K-1
EXPORT Keyword.....3-73
EXPORT (.E) files.....3-74/75
Exporting, definition.....3-73/74
Expressions:
 -- arithmetic.....3-17/20
 -- logical.....3-21/22
 -- mixed mode.....3-18/20
 -- relational.....3-21
EXT keyword.....1-29,3-57/59
EXTCOPY (utility command).....2-18
EXTDIR: (utility command).....1-7,2-25
 -- example of.....2-25
Extensions, file name.....2-12

F

FALSE (0).....1-21,3-21
FILECRC.S (sample program).....Q-2
Field spec., formatted output.....4-41
File descriptor.....3-51
File name extensions (TABLE 3)....2-12
File names, rules for:
 -- Commodore 64.....2-10
 -- Apple IIe/IIc.....2-11/12
Files:
 -- converting APPLE II DOS 3.3....N-2
 -- COPY (EXEC cmd).....2-17/19
 -- DELETE (EXEC cmd).....2-20/21
 -- DISKETTE utility.....O-2/4
 -- handle, definition of....1-22,3-51
 -- JOB (.J).....2-30/31
 -- locked.....2-32
 -- opening.....3-51/52,4-36/40
 -- RENAME (EXEC cmd).....2-39
 -- TYPE (EXEC cmd).....2-41
FILES (EXEC cmd).....2-24/25
FILL (EXEC cmd).....2-26
FILL (PROC).....4-14/15
FIND.S (sample program)....1-15/23,Q-2
FKEY (EXEC cmd).....2-26/27
FKEYGET (PROC).....4-15/16
FKEYSET (PROC).....4-16
FOR statement.....3-29/30
Format string, output spec...3-37,4-41
Formatted output.....3-37/38,4-41/43
Formatting disks.....O-1/3
Forward references.....J-1
FUNC, function header.....3-41
Function keys:
 -- default setting....1-6,2-3,2-26/27
 -- editor's display.....1-9
 -- redefining.....2-26/27,4-16
 -- use of.....2-4
Functions & procedures:...1-18,3-36/47
 -- arguments in.....3-42/45
 -- built-in.....3-36/37
 -- intro to.....3-36
 -- REAL.....K-1/2

G

GET (EXEC cmd).....2-27/29
GETARGS (FUNC).....4-17
GETBLKF (FUNC).....4-18
GETC (FUNC).....4-19
GETCF (FUNC).....4-20
GETKEY (FUNC).....4-20/21,B-1/4
GETL (PROC).....4-21/22
GETLF (FUNC).....4-22/23
GETPOSF (FUNC) (APPLE II).....4-23
GETTST (FUNC).....4-24
GETVER (FUNC).....4-24/25
GO (EXEC cmd).....2-28/29
GRAPHDEMO.S (sample program).....Q-2
Graphics, Hi-Res.....1-33,2-15,Q-2

H

Hardware requirements.....1-2
HELP display screen.....1-8,2-29
HELP (EXEC cmd).....2-29/30
Hexadecimal, literal numbers....3-9/10
Hexadecimal, used in EXEC cmds....2-12
Hi-res graphics.....1-33,2-15,Q-2
HOME key (C64).....2-48

I

IBM PROMAL compatibility.....L-1/2
IF statement.....3-26/28
IMPORT keyword.....3-74
Importing, definitions.....3-74/75
INCLUDE statement..1-11,3-37,3-48,3-74
Indentation.....1-20,1-28,3-27
Indirect operators.....3-23/24
INFILTRATOR (sample program)..1-32,Q-2
Initialization, PROMAL system....2-3/4
INLINE (FUNC).....4-25
INLIST (FUNC).....4-25/27
Input, numeric.....3-39/41
Input, simple.....3-38/39
INSET (FUNC).....4-27/28
INST key (C64).....2-5,2-47
INT, data type.....3-8/9
Interfacing:.....3-51/60
 -- to C64 graphics & sound....3-58/59
Interrupt service routines.....I-15
INTSTR (PROC).....4-28
IOERROR:
 -- error code variable...3-51/52,4-36
 -- errors, codes for.....4-36
I/O:
 -- functions GETLF, PUTF.....3-52/53
 -- redirection.....2-14
 -- redirection, example of..1-23,2-14
 -- with files STDIN/STDOUT....3-53/54

J

JOB (EXEC cmd).....2-30/31,2-33
JOB files (.J):.....2-30/31
 -- substitution arguments in.....2-31
JSR (PROC).....4-29,I-1/4

K

Keyboard (K) device.....2-13
Key codes.....B-1/4

L

LENSTR (FUNC).....4-29/30
LIBRARY:.....1-11,2-3,3-51,4-2
-- (L) device.....2-13
-- routine description notation...4-4
-- summary of routines.....4-2/3
Line editing, keys (TABLE 1).....2-5/6
LIST statement.....3-48
Literals:.....3-9/12
-- characters.....3-10
-- numbers.....3-9/10
-- strings.....3-10/11
LOAD (PROC).....3-70/73,4-30
LOADER:.....3-67/82
-- bootstrap program with.....3-76/77
-- calling.....3-70/73
-- definitions used.....3-67/68
-- errors from.....3-71
-- EXPORTing definitions.....3-73/74
-- IMPORTing definitions.....3-74/75
-- memory diagram.....3-70,3-78
-- operation of.....3-68/70
-- options for.....3-72
-- overlays.....3-77/79
-- separate compilation.....3-75/76
Loading, PROMAL diskette.....1-4/6
Loading, programs.....2-27/28,4-32/33
Local variables.....1-19,3-45/46
LOCK (EXEC cmd).....2-32
Locked file.....2-32,2-42
LOG (FUNC).....K-1
LOG10 (FUNC).....K-1
Logical operators.....3-21/22
LOOKSTR (FUNC).....4-30

M

M array.....3-24,3-44
Machine language programs:
-- calling LIB routines from.....I-10
-- calling with JSR.....4-29,I-1/4
-- embedded in DATA.....I-4/6
-- effect of BRK.....I-3
-- executing with GO.....2-28/29
-- interfacing to.....I-1/15
-- interrupt service with.....I-15
-- loading with GET.....2-27/28
-- loading with MLGET.....4-32/33
-- passing arguments to.....I-8/10
-- relocatable.....I-11/15
MACRO (EXEC cmd).....2-32/33
MAP (EXEC cmd):.....1-12,2-33/35
-- display screens...1-12/13,2-33/35
Memory allocation:
-- MAP display definitions....2-33/36
-- dynamic.....H-1
Memory map, PROMAL internal.....G-1/6
MAX (FUNC).....4-31
MIN (FUNC).....4-32
MLGET (FUNC).....4-32/33
MOVSTR (PROC).....4-33/34
MSD dual disk support (C64).....N-1

N

Names, rules for.....3-7/8
NCARG variable.....1-21/22,3-56/57
NEWDIR (EXEC cmd).....2-36
NEXT statement.....3-32
NOREAL (EXEC cmd).....2-36/37
NOT operator.....3-17,3-21/22
Notation, conventions.....2-9
NOTHING statement.....3-32
Null (N) device.....2-13/14
Numbers, literal.....3-9/10
NUMERIC (FUNC).....4-34
Numeric input.....3-39/41

O

O, compiler option.....2-56/57
Object program.....1-4
ONLINE (FUNC) (APPLE II).....4-35
OPEN (FUNC).....1-22,3-51/55,4-36/40
OPEN, error codes.....4-36
Opening files.....3-51/52,4-36/40
Operating system notes.....N-1/2
Operators:
-- arithmetic.....3-17/18
-- indirect & address.....3-22/23
-- list of all.....3-17
-- logical.....3-21/22
-- relational.....3-21
-- shift.....3-22
Options for loader.....3-72
OR operator.....3-17,3-21/22
Output, field descriptors.....4-41
OUTPUT (PROC).....4-41/43
OUTPUTF (PROC).....4-43/44
Output:
-- formatted numeric..3-37/38,4-41/44
-- PROC, example.....3-37/38
-- simple.....3-37
OVERLAY statement.....3-25,3-77/79
Overlays:
-- definition of.....3-68

-- using.....3-77/79
-- guidelines for.....3-81/82
-- memory map.....3-78
-- sample program.....3-78
OWN variables.....3-46,H-1

P

Pathnames.....2-11
PAUSE (EXEC cmd).....2-37
Pointers.....3-22/24
POWER (FUNC).....K-1
PREFIX (EXEC cmd).....2-11,2-38
Printer (P) device.....2-13,3-54/55
Printer support.....E-1/2
PROC, procedure header.....3-41
Procedures & functions:...1-18,3-36/47
-- introduction to.....3-36
-- passing arguments to.....3-42/45
-- local variables in.....3-45/46
ProDOS, special functions.....N-2
Program authors.....ii
PROGRAM statement.....3-4,3-25
Programs, demo:.....Q-1/3
-- BUDGET.....1-24
-- CALC.....1-25
-- FIND.....1-16/23
PROMAL:
-- definition of.....1-3
-- initialization.....2-3/4
-- loading.....1-4/6
-- signon screen.....1-6
-- special capabilities.....1-29
-- system components.....2-2
-- vs. BASIC.....1-4
PROMAL language:
-- applications of.....3-2
-- data types.....3-8/9
-- introduction to.....3-2
-- overview.....3-3/6
-- names.....3-7/8
-- reserved words.....3-7,L-2
-- rules for.....1-18
-- syntax diagrams.....P-3/8
PROQUIT (PROC).....4-44
PUT (PROC).....1-10,3-37,4-44/45
PUTBLKF (PROC).....4-45/46
PUTF (PROC).....1-22,4-47

Q

QUIT (EXEC cmd).....2-38/39

R

RANDOM (FUNC).....4-47/48
REAL:
-- constants disallowed.....3-13
-- DATA.....3-15
-- data type.....3-8/9
-- literals.....3-10
-- numbers.....1-24
-- variables, internal format.....D-2
REALSTR (PROC).....4-48/49
Recursion & forward references.....J-1
REDIRECT (PROC).....4-49/50
Redirection, I/O.....1-23,2-14
REFUGE statement.....3-33/34
Relative file support (C64).....M-1/5
RELDEMO.S (sample program).....M-5,Q-2
RELOCATE utility.....I-11/15
RELOCATE.S (utility program).....Q-3
RENAME (EXEC cmd).....2-39
RENAME (FUNC).....4-50/51
REPEAT statement.....3-29
Reserved words.....3-7,L-2
RETURN key.....2-5,2-46
RETURN statement.....3-41/42
Reverse video.....3-11,4-43,4-45
RS-232 support.....F-1/6
Runtime errors.....C-2/3
Runtime errors, locating.....D-1

S

Screen (S) device.....2-13
Scrolling, left & right.....1-20,2-47
Scrolling, up & down.....1-16,2-47
SET (EXEC cmd).....2-39/40
SETPOSF (PROC) (APPLE II).....4-51
SETPREFIX (FUNC) (APPLE II).....4-52
Shift operators.....3-22
Signon display.....1-6
SIN (FUNC).....K-1
SIZE (EXEC cmd).....2-40/41
SORTDEMO.S (sample prog.)..1-30/31,Q-3
SORTSTRING.S (sample program).....Q-3
SPLIT.S (utility program).....Q-3
SRECEIVE.S (utility prog.)...F-4/5,Q-3
SSEND.S (utility program)....F-4/5,Q-3
SQRT (FUNC).....K-1
Stack overflow.....3-47
Starting, system.....1-4/6,2-3/4
Statements:.....3-25/34
-- (=) assignment.....3-26
-- BREAK.....3-31/32
-- CHOOSE.....3-30/31
-- ESCAPE & REFUGE.....3-33/34
-- FOR.....3-29/30

-- IF.....1-21/22,3-26/28
-- NEXT.....3-32
-- NOTHING.....3-32
-- PROGRAM.....3-25
-- REPEAT.....3-29
-- RETURN.....3-41/42
-- WHILE.....1-20,3-28/29
STDIN, SDTOUT file handles.....3-53/54
STOP key (C64).....2-6/7
String operations:.....3-45/46,3-23
-- arrays of.....3-65
-- compare.....1-20/21,4-8
-- conversion.....4-28,4-52/53,4-57
-- editing.....4-12/14,4-25
-- length.....4-29/30
-- move.....4-33/34
-- searching.....4-25/28,4-30,4-55
STRREAL (FUNC).....4-52/53
STRVAL (FUNC).....4-53/54
Subroutines:.....1-18,3-41/42,3-46/47
-- passed arguments.....3-42/45
-- user defined.....3-41/47
Subscripts, arrays.....3-14/15,3-64
SUBSTR (FUNC).....4-55
Syntax diagrams, how to read.....P-1/2
Syntax diagrams.....P-3/8
System data areas.....G-5/6

T

TAN (FUNC).....K-1
Telephone (T) device.....2-13,F-1/6
TINYTERM.S (sample program)....F-5,Q-3
TO keyword.....3-29
TESTKEY (FUNC).....4-55/56,B-1/4
TOUPPER (FUNC).....4-56
Trademarks.....ii,1-2
TRUE (1).....1-21,3-21
TYPE (EXEC cmd).....2-41

U

UNLOAD (EXEC cmd).....2-42
UNLOCK (EXEC cmd).....2-42
UNTIL keyword.....3-29
Unprintable codes, embedding.....3-11
Upper & lower case mode (C64).....2-55
Upper case & graphics mode (C64)..2-55
User-defined commands.....2-4
User-defined subroutines.....3-41/47

V

Variables:
-- arrays, declaring.....3-14/15
-- command line argument.....3-56/57
-- declarations.....3-12/13
-- external (EXT).....1-29,3-57/59
-- global.....1-19,3-44
-- initializing all to zero.....3-66
-- introduction to rules.....1-18
-- local.....1-19,3-45/46
-- locating in memory.....D-1/2
-- non-initialization of.....3-13
-- OWN.....3-46
-- types supported.....1-18,3-8/9
Volume names (APPLE II).....2-11

W

WHILE statement:.....3-28/29
-- example of.....1-20
WORD data type.....1-18,3-8/9
WORDSTR (PROC).....4-57
Workspace:
-- and edit buffer.....2-54/55
-- auto update after edit.....2-54
-- changing size of.....2-43
-- clearing of.....2-43
-- writing to.....1-10
-- (W) device.....2-13
WS (EXEC cmd).....2-43

X

XOR operator.....3-17,3-21/22
XREF (utility program).....2-60

Z

ZAPFILE (FUNC).....4-57/58