M E E T   P R O M A L !

(**PRO**grammer's **M**icro **A**pplication **L**anguage)

AN INTRODUCTION TO THE PROMAL PROGRAMMING SYSTEM

For APPLE IIe, IIc and COMMODORE 64 Computers

SYSTEMS MANAGEMENT ASSOCIATES, INC.
3325 Executive Drive
Raleigh NC 27609

Rev. C - Sep. 1986

MINIMUM HARDWARE REQUIREMENTS

Apple II: IIe or enhanced IIe, with Extended 80 column card (128K) and one floppy disk, or Apple IIc. PROMAL supports ProDOS, not DOS 3.3. Sorry, PROMAL does not work on an Apple II **Plus**. PROMAL works with (but does not require) RAMWORKS or equivalent memory expansion for /RAM disk.

Commodore 64 or 128: One 1541/1571 or true compatible disk drive, or MSD drive. Note: Most "Turbo" or other fast disk load accessories are incompatible with PROMAL, and should be removed before running PROMAL. However, PROMAL has DYNODISK built-in, which doubles the reading speed of 1541 or 1571 disk drives. This feature can be disabled if desired. PROMAL is compatible with the Skyles 1541 Flash and IEEE Flash.

COPYRIGHT NOTICE

TRADEMARKS

DISCLAIMER

NOTICE FROM APPLE COMPUTER INC. FOR APPLE VERSION

MEET PROMAL!

Congratulations! You're about to meet PROMAL, the most powerful integrated programming environment you can buy for your computer. This manual can be used in conjunction with the PROMAL Demo Diskette to help you get acquainted with PROMAL.

About the PROMAL Demo Diskette... Don't let the name "Demo" fool you. This disk contains a fully functional PROMAL system. It contains all of the features of the sealed System Disk, **except the ability to compile large programs** (400 lines, maximum, excluding comments). Before opening your sealed System Diskette you should use the Demo Diskette for evaluation. **PROMAL IS NOT COPY-PROTECTED AND ONCE THE SEAL IS BROKEN ON THE SYSTEM DISKETTE IT CANNOT BE RETURNED FOR A REFUND.** All the files referred to in this MEET PROMAL manual will be found on the PROMAL Demo Diskette. This way you can satisfy yourself that PROMAL is everything we say it is. We want you to be 100 percent satisfied with our product.

AUDIENCE

We do not claim that PROMAL is for everyone. PROMAL is for the person who has a working familiarity with the computer and BASIC (or another high-level language), and wants to get the most from his or her computer. If you are frustrated by the limitations of BASIC but want to avoid the complexity and low productivity of machine-language programming, then PROMAL is for you. If you want to realize the full potential of your personal computer, you need the power, the speed, and the capabilities of PROMAL.

ABOUT THIS MANUAL

Your PROMAL system includes four manuals plus Appendices and an Index. The first manual, "MEET PROMAL!", which you are reading, will not make you a proficient PROMAL programmer, but will provide enough information for you to evaluate the PROMAL system. The "MEET PROMAL!" Manual is required reading for the owner of the PROMAL System; please do not attempt to study the USER'S GUIDE or LANGUAGE MANUAL until you have read and executed the examples in "MEET PROMAL!". This approach will help even the advanced programmer get the most from PROMAL.

WHAT IS PROMAL?

PROMAL (PROgrammer's Micro Application Language) is a high-performance programming system. It includes:

* An Interactive Command EXECUTIVE
* A Full-Screen EDITOR
* A Structured Language COMPILER
* An on-line LIBRARY of predefined subroutines

The **EXECUTIVE** lets you enter commands to load and execute programs, to manage files and memory, to display files and directories, and to do much more. Many built-in commands are provided, and you can add as many of your own commands as you want.

The full-screen **EDITOR** makes it easy for you to create or modify PROMAL programs or other text files. It has powerful features similar to many word-processors, but is designed specifically to simplify program generation.

The **COMPILER** converts your program into a very compact, very fast-executing command (or "object program") which you can run by just typing its name. Compiled PROMAL programs will often run 20 to 100 (or more!) times faster than BASIC, and occupy much less memory.

The **LIBRARY** provides an extensive base of built-in, pre-programmed subroutines which are always available to your programs. The library greatly simplifies programming by providing a nucleus of commonly-needed functions.

ARE PROMAL PROGRAMS LIKE BASIC PROGRAMS?

Although your previous experience with BASIC will be helpful in understanding PROMAL, you will find very much that is different. Some aspects of PROMAL may seem strange at first, but you will soon appreciate its simple elegance. Once you have mastered a few basic concepts, you will find PROMAL much easier to program than BASIC for any non-trivial application.

LET'S GET STARTED!

If you're ready to begin your guided tour, get your PROMAL Demo Diskette, your computer, and read on ...

DUPLICATING THE DEMO DISK

Because the Demo Diskette contains important files which are **not** duplicated on the sealed System Diskette, you should make a copy of the Demo Diskette before proceeding. The examples in MEET PROMAL will write on the demo disk. Therefore you should only use the copy, so that if you make a mistake or accidentally overwrite or delete a file, you will still have the original Demo Disk. You can copy the Demo disk with any kind of disk copy program. Commodore owners who don't have a copy program can use the DISKETTE utility, as described in **Appendix O.** Apple users can use the standard ProDOS disk copier furnished on the System Utilities Disk which came with your computer.

LOADING PROMAL

For Apple II:

Put your **copy** of the PROMAL Demo Diskette into the disk drive, close the door, and turn on your computer. If your computer is already on, you can hold down the CTRL and open-Apple keys and press RESET.

If you have a hard disk and wish to boot from hard disk, create a new directory, and copy all the files from the Demo disk onto it. The volume name for the Demo Disk is /PROMAL/. Then execute PROMAL.SYSTEM.

If you have a Ramworks II or similar /RAM disk and want to use it with PROMAL, first install your /RAM device as usual. Then execute PROMAL.SYSTEM. This can be done from BASIC with the command **-/PROMAL/PROMAL.SYSTEM.**

Please note that **only the Demo disk** contains a bootable version of PROMAL. The System disk (and optional Developer's disk) in the sealed envelope are crammed full of additional sample programs, utilities, etc., so there isn't enough room for duplicate copies of ProDOS, PROMAL.SYSTEM, etc. on each disk. Always boot up with a copy of the Demo disk; then you can change disks (using the **PREFIX *** command described later) to access files on the other disk(s).

For Commodore 64/128:

Please follow the following steps **carefully**.

1. Turn on your computer system in the usual manner.

2. If you have a Commodore **128**, you must type **GO 64**, and reply **Y** to the "Are you sure?" prompt. PROMAL works only in Commodore 64 mode. Do not attempt to use 128 mode or non-1541 compatible disk features of the 1571 disk (such as double-sided mode or "fast" mode).

3. Insert your copy of the Demo Disk in the drive, close the door and type:

LOAD "PROMAL",8 <RETURN>

4. If the **only** peripheral on your system is a single 1541 or 1571 disk drive, proceed to step 5. The Commodore 64 version of PROMAL includes DYNODISK, which doubles the reading speed of the Commodore 1541 or 1571 disk drive. This feature is comparable to many other disk speedup products which you may have or may have seen. PROMAL is compatible with Skyles Flash products provided you disable DYNODISK as described below. Do not use PROMAL with any other disk speedup cartridge or software package unless you are **sure** it does not use **any** memory in the Commodore 64 and is completely 1541-compatible; otherwise, PROMAL will not work properly. DYNODISK works automatically with all PROMAL programs; there is nothing to install. DYNODISK has the same limitations as other disk speedup products, namely:

(a) You can't use DYNODISK if you have any other device (such as a printer or second disk drive) attached to the computer on the serial bus. Therefore if you have a printer, you should either turn it off while reading from disk, or disable DYNODISK as described below. If you decide to leave DYNODISK enabled and turn off your printer, and you have a printer interface (such as a CARDCO), you will need to turn it off too (in the case of the CARDCO G+, this means disconnecting the single wire from the back of the computer). Failure to observe this precaution may cause the computer to "hang up", necessitating reloading PROMAL. DYNODISK can be re-enabled by a simple command when PROMAL is running.

(b) If you don't have a 1541 or 1571 disk drive, see **APPENDIX N** before continuing.

If you need to **disable** DYNODISK, type:

POKE 3555,0 <RETURN>

5. Type:

RUN <RETURN>

For Either computer:

The system will then load the EDITOR, EXECUTIVE, and LIBRARY. On the screen you will see:

```
LOADING EDITOR
LOADING EXECUTIVE
LOADING LIBRARY...
```

The screen will then clear and display:

```
PROMAL Development System EXECUTIVE
             Version 2.1
     Copyright (C) 1986 SMA Inc.

        PROMAL is a Trademark of
  Systems Management Associates, Inc.
```

At the end of the signon information, you will see (unless you have a Thunderclock card in an Apple II system, which sets the date automatically):

```
PLEASE ENTER TODAY'S DATE
(in the form MM/DD/YY):
```

Type in the correct date, for example 09/30/86, and press RETURN. PROMAL uses this date to automatically "tag" all compiled programs with their creation date, so if you make revisions to a program, you can easily see which is the current version. You will now see:

Apple II	Commodore 64
F1 = EDIT	F1 = EDIT
F2 = PREFIX *	F2 = DUMP
F3 = COMPILE	F3 = COMPILE
F4 = GET	F4 = GET
F5 = FILES	F5 = FILES
F6 = EXTDIR *	F6 = MAP
F7 = HELP	F7 = HELP
F8 = COPY	F8 = COPY

These lines tell what the default function keys do. On the Apple II, the letter "F" may be replaced by an "Apple" icon. **On the Apple II, a function key is activated by holding down either Apple key and pressing the desired number.** You may use function keys to quickly enter commands when using the EXECUTIVE. Now lets begin our guided tour of the PROMAL EXECUTIVE.

USING THE PROMAL EXECUTIVE

The "-->" is the PROMAL EXECUTIVE's prompt, which indicates that the EXECUTIVE is ready for you to type in a command. Try typing this:

TYPE README.T

followed by <RETURN>. All commands need to be terminated by the <RETURN> key. This command will display additional information about your system not included in this manual. If instead you get a "FILE NOT FOUND" error, just go on.

Now try typing:

```
hello
```

followed of course by <RETURN>. You should see:

```
--> hello

PROGRAM OR OVERLAY NOT FOUND: HELLO

--> _
```

Here's what happened. When you typed "hello", PROMAL tried to execute the command called "hello". Since there is no built-in command called "hello", the EXECUTIVE searched for a user-defined command file on disk. Since it didn't find the command file "hello", it gave you an error, and is now ready for another command. This is how PROMAL executes commands. First it looks for the command in memory, and then on disk. (You can create new commands by compiling a PROMAL program.) Unlike BASIC, you can have several programs in memory at once, and execute any of them by just typing the name. In fact, the EXECUTIVE and EDITOR are just compiled PROMAL programs that are already in memory!

Now press function key **F5** (for Apple II, hold down either Apple key and press 5). You will see "FILES" appear. This is a built-in command. Press <RETURN> and the EXECUTIVE will execute the FILES command. Try it. You should see a list of file names. On the Apple these names will be displayed in four columns. On the Commodore, the display will be similar to a directory listing made from BASIC.

These are the file names on your demo diskette. Notice that most of the filenames end in ".C" or ".S". This is because PROMAL file names have a single character file extension after the name which tells what **kind** of file it is. ".C" files are compiled command programs and ".S" files are PROMAL language source (text) programs which can be edited. Therefore FIND.S is a source program and FIND.C is the compiled (executable) form of the same program.

For the Apple II only, you can get more information about the files on the disk by using the EXTDIR command (EXTended DIRectory). Press F6 (Apple key with 6), **or** type **EXTDIR** * and press <RETURN>. Be sure to type a space before the asterisk. You should see a list of all the files, with information about each file, and a summary showing how many blocks are free on disk. Unlike the FILES command, the EXTDIR command is not built-in to the EXECUTIVE. Instead, EXTDIR is actually a PROMAL program which is loaded from disk and executed when you type EXTDIR.

IMPORTANT: For the Apple II only, please note that if you **change disks** you will need to press **F2** and <RETURN> after changing disks to issue a **PREFIX** * command. This tells ProDOS what the new volume name is; otherwise, it will still look for your old diskette, probably generating DEVICE NOT READY or FILE NOT FOUND errors.

Now press **F7** and <RETURN> (or just type in **HELP** - the result is the same). You will see the screen scroll up to display a "help" page similar to:

```
                PROMAL HELP
    CTRL-                    CTRL-
    E Enter insert mode      D Delete char.
    B Recall Prior Line      F Cursor to first
    \ Clear To End Line      L Cursor to last
           PARTIAL COMMAND SUMMARY
    COMPILE [File [L[=List]][O=Object]
    COPY File [Dest.][#Drvs]
    DELETE File             Function Keys
    DUMP From [To]           f1 = EDIT
    EDIT [File]              f2 = PREFIX *
    FILES [Dir]              f3 = COMPILE
    FILL From To Value       f4 = GET
    FKEY [Number String]     f5 = FILES
    GET Commandfile          f6 = EXTDIR *
    JOB File.J [Arg...]      f7 = HELP
    MAP                      f8 = COPY
    PREFIX [/Path/ or *]
    RENAME File Newname
    SET Addr Val [Val...]
    TYPE File
    UNLOAD [Command]
    --> _
```

The display above is for the Apple II; the Commodore display will differ somewhat. The top four lines are clues to using some control keys for line-editing while you are in the EXECUTIVE. We'll get to them in a minute. The rest of the lines summarize many of the most commonly-needed commands. Many commands need or may have **arguments** after the command name. These arguments are usually file names or numbers. Arguments shown in square brackets ([]) on the help screen are optional. For example, DUMP needs one argument, "From", and can optionally have a second argument, "To". Try typing this:

```
    dump 1100
```

You will see a display similar to:

```
    1100  72 65 7A 85 72 A5 71 65   rez.r.qe
```

The actual numbers and letters shown will be different. This is the contents of memory locations 1100 hex through 1107, displayed in hex and ASCII. A "." is shown for any byte which doesn't represent a printable ASCII character. If you're not familiar with hex notation, don't be concerned; later you may want to consult the reference manual for your computer to learn about hexadecimal. DUMP with one argument will display 8 bytes starting at the FROM address.

Now suppose you wanted to dump from 1100 to 1180. Wait! If you think you know, don't type yet! Instead press **CTRL**, and while holding it down, press **B**. What happened? You should see:

```
    --> dump 1100_
```

You can recall your last command by using CTRL-B! Now you can "edit" your prior command by just typing " 1180" at the end of the line. Don't forget the space between 1100 and 1180; PROMAL expects a space between arguments. Press <RETURN>. You should see a display of memory from 1100 to 1180.

You will find CTRL-B very useful for correcting mistakes in commands. You can recall the prior command and edit it again, rather than re-typing all of it. There are several keys you can use to edit command lines, but we'll save these until we discuss the EDITOR (unless you want to experiment right now using the clues given by HELP).

One more thing about CTRL-B. Try pressing it several times. What happens? You can "backtrack" through prior commands, one at a time! After you get all the way back to the start of the session (before you typed in the date), the next CTRL-B will "wrap around" again to the most recent command.

WRITING A PROGRAM WITH THE EDITOR

Now that you know a little about using the EXECUTIVE, let's try writing a program using the EDITOR! Press **F1** and <RETURN> (or type **EDIT**). Almost instantly the screen will clear, except for the bottom 5 lines which show:

```
 ---------------------------------------------------------------------------


 LINE =    1
1=DEL LN  2=INS LN  3=MARK   4=RECALL  5=FIND   6=CHANGE  7=HELP   8=QUIT
```

The top 20 blank lines form your text area, and the bottom 5 lines are a status area, which tells you that the function keys have now been redefined. On the Commodore 64, these function key legends occupy two 40-column lines instead of one as shown above. For example, F1 now means "delete line". You won't need any of these function keys (except F8=QUIT) to edit your first PROMAL program.

The blinking cursor is at the top of the screen. It always indicates where the next character you type will appear. Try typing in the following program, exactly as shown below. Be sure to start each line in the first column, and use <RETURN> to end each line.

```
PROGRAM HELLO
INCLUDE LIBRARY
BEGIN
PUT "HELLO YOURSELF!",NL
END
```

If you make a mistake, use the DELETE key to delete characters. You can use the cursor keys to backup and over-type any corrections too. You can type in upper or lower case or a mix - it doesn't matter. If you prefer upper case, you may want to press CTRL-A, which will cause ALPHALOCK to be displayed in the status area. After this, any alphabetic keys you type will automatically be in upper case.

One thing you probably noticed right away is that this program doesn't have any line numbers. PROMAL programs don't need them. You'll see later how you make branches and subroutine calls without line numbers. Another thing that's different about PROMAL programs is that you can only put one statement on a line. This makes programs more readable and easier to change. If you think about it, the main reason you put multiple statements on one line in a BASIC program was to avoid using another line number. Since PROMAL has no line numbers, there's no need for several statements on one line.

The first line of your program simply gives it a name (HELLO). Every PROMAL program starts this way. The next line tells PROMAL to include all the subroutines in the built in LIBRARY. You will normally use the INCLUDE LIBRARY statement in every program you write. The "BEGIN" line merely indicates the start of the main program. The next line is the substance of your little program. PUT is similar to a BASIC PRINT, and prints strings or characters. Unlike BASIC, you have to tell PROMAL explicitly when you want to start a new line. The **NL** does this (it stands for New Line). You can also use **CR** (Carriage Return) in place of NL; the result is the same.

When you're done press **F8.** You will see a display similar to that below. The numbers on your screen may be different (especially on a Commodore 64). This display shows your choices of what to do. "WORKSPACE", is a small, in-memory file with the name "W". It is useful for temporarily storing a small source program you are experimenting with, without saving it on disk.

Press W and <RETURN>. Immediately, the cursor will re-appear below the "SELECTION?" prompt. This indicates that your text has been written to the workspace (it's fast!), and you may now make another selection.

```
        PROMAL EDITOR 2.1
     Copyright (C) 1986 SMA, Inc.

 BUFFER SIZE = 9862
 FILE NAME = W (AUTO UPDATE ON QUIT)
 FILE SIZE = 66

              OPTIONS

      R = REPLACE ORIGINAL FILE
      N = WRITE TO NEW FILE
      W = WRITE TO WORKSPACE
      C = CONTINUE EDITING
      Q = QUIT EDITOR

  SELECTION?
 _
```

Type **Q** and <RETURN>, and the EXECUTIVE prompt reappears:

```
-->  _
```

COMPILING YOUR FIRST PROGRAM

Unlike BASIC, you must COMPILE your program before you can run it. Compiling it does not destroy your source (text) program, but generates a separate ("object") program which is executable.

Press **F3**, (or type **compile**) and press <RETURN>. After a pause while the COMPILER loads from disk, you will see it "sign on", followed by a rapidly changing display as your program compiles. This will only take a few seconds. The compiler will display READING LINE XX, where the XX's change rapidly to show you what line number it's working on. The final display should look similar to this.

```
      PROMAL DEMO COMPILER 2.1
     Copyright (C) 1986 SMA Inc.

READING LINE 2
INCLUDING LIBRARY
READING LINE 67
RESUMING FILE W
READING LINE 68
COMPILING <MAIN PROGRAM>
READING LINE 70
HELLO compiled:
  70 Source lines
  $0025 (37) bytes Obj.
  $0 Scalar, $0000 (0) tot. Vars.
Table usage:
  Symbols: 36%  Fwd. Refs: 1%
  Strings: 1%

--> _
```

The display may differ slightly, especially for the Commodore 64. The summary at the end shows information about the **object file** which the compiler wrote to disk. This object file is your executable program, with the default name HELLO.C.

You may be wondering why the compiler said it compiled 70 source lines (less for the Commodore 64). Your program was only 5 lines long! The answer is that the INCLUDE LIBRARY line of the program caused the COMPILER to read another file called the LIBRARY at that point in your program. Later when you start writing big programs, you can tell it to include other files of your own. For example, you can tell it to INCLUDE some file of subroutines. In this way, you can share commonly-used routines between many programs without having to re-type them or "paste" them into your program. The LIBRARY file contains about 63 lines of standard definitions (59 for the Commodore) that you will want in nearly every PROMAL program. Like the Workspace, the LIBRARY is a memory-resident file.

The summary also tells how big your compiled program is (37 bytes), how many bytes are needed for its variables (none in this case), and some information about how much of the compiler's internal tables were used. This is described in the PROMAL USER'S MANUAL. Don't be concerned that your little program took up 36 percent of the symbol table; the demo compiler has a small symbol table. The full compiler can compile programs with thousands of lines.

EXECUTING YOUR PROGRAM

Now that the compiler is done, your program is ready to execute. Just type:

```
    hello
```

And your program will display:

```
    HELLO YOURSELF!

    --> _
```

When your program finished, it returned control to the EXECUTIVE. You have now created a new, user-defined command for the EXECUTIVE! Your **object** program is saved on disk as file HELLO.C, and will be run anytime you type HELLO from the EXECUTIVE. You can save as many commands as you want on as many diskettes as you like. To execute a command, just type its name. If it's been executed before and is in memory, it will execute instantly. If not, the EXECUTIVE will fetch it from disk and execute it.

Note that you have not yet saved your **source** program (the text file you can edit) on disk; it's only in the temporary workspace. Normally, you will want to save your source program on disk when you exit from the EDITor. If you want to, you can still save it now, from the EXECUTIVE, by typing **COPY W HELLO.S** which copies the Workspace to a new disk file called HELLO.S.

WHERE DOES PROMAL PUT THE PROGRAM?

You might be curious to know where your program is in memory. Type:

```
    MAP
```

and you will see a "map" of memory, which tells where PROMAL put your program and how the remaining memory is allocated. The top half shows information about what program(s) you have in memory, and the bottom half shows a summary of available space. We won't go into all the details of the MAP display here. It is fully explained in the PROMAL USERS GUIDE.

<u>Apple II</u>

```
    HELLO         (PRO.) 09/30/86 CHKSUM 49CB
      CODE $2900-29FF

    OBJECT PROGRAMS  $2900-29FF (256)
    FREE SPACE       $2A00-8DFF (25600)
    SHARED VARIABLES $8E00-     (0)
    EXEC./EDIT SPACE $6100-8DFF (11520)
    TOTAL SPACE      $2900-8DFF (25856)

    ACTIVE WORKSPACE $1200-1241 (66)
    FREE WORKSPACE   $1242-5AFF (18622)
```

Commodore 64

```
COMPILE     (PRO.) 11/ 4/85 CHKSUM 1465
  CODE $4F00-81FF, VARIABLES $8200-84FF
HELLO       (PRO.) 09/30/86 CHKSUM 4A86
  CODE $8500-85FF

OBJECT PROGRAMS  $4F00-85FF (14080)
FREE SPACE       $8600-98FF (4864)
ACTIVE WORKSPACE $9900-9941 (66)
FREE WORKSPACE   $9942-A0FF (1982)
SHARED VARIABLES $A100      (0)
EXEC./EDIT SPACE $A200-CFFF (11776)
TOTAL SPACE      $4F00-CFFF (33024)
```

For the Apple version, you may have two programs in memory (EXTDIR and HELLO), or just one if you didn't try EXTDIR before. The Commodore version shows two programs in memory, COMPILE and your HELLO program. Since the Apple has somewhat less memory than the Commodore, but has a relatively fast disk, the Compiler is always unloaded automatically when it finishes, leaving more room for other programs. Since the Commodore has a little more available memory but a slow disk, PROMAL normally keeps the compiler in memory when it finishes so you won't have to wait for it to load when you need it again. The number after "CODE" shows the address where your program was loaded.

If you executed another program, a part of PROMAL called the **loader** would put it right above where HELLO ends. PROMAL always allocates programs and data on exact "page" boundaries in memory (that is, the starting hexadecimal address will always end in 00). Technically, PROMAL programs are known as "relocatable" object code, which means that they can be run **anywhere** in memory. In the event that you use up so much memory that there isn't enough room to load the specified command program, PROMAL will automatically "unload" programs to make enough room. The MAP also tells other information about your HELLO program. It tells you it is a PROMAL program, that it was compiled on 11/04/85 (or whatever date you used), and what its "checksum" is. If your program uses any variables (your HELLO program doesn't), it shows what space is allocated for variables.

The checksum requires more explanation. When PROMAL loads a program into memory, it computes a 16-bit sum of all the bytes loaded. This is the checksum shown, in hex. Anytime you execute that program, PROMAL recomputes the checksum and compares it to the saved value. If the two values don't match, the loader knows something has corrupted the program in memory (such as another program POKEing around where it shouldn't!). The loader then automatically reloads the program from disk. This provides an integrity check for your programs.

The meanings of the rest of the summary are described in the PROMAL USER GUIDE. One thing you might be interested in now though is the line labelled TOTAL SPACE. This is the maximum amount of memory that you can use for PROMAL programs; about 25K bytes for the Apple or 33K bytes for the Commodore. Although this amount is somewhat less than is allowed by BASIC, you can still have PROMAL programs that are **much** larger and more complex than is possible using BASIC, because PROMAL programs are much more compact. For example, the

COMPILEr itself is a PROMAL program of over 4000 lines, yet occupies only about 14K bytes. In addition, PROMAL allows you to use **overlays** to run programs that are larger than memory. This is described in the PROMAL LANGUAGE MANUAL.

You won't concern yourself often with how PROMAL allocates memory, because it is automatic. We have examined it briefly here to give you an idea of how the PROMAL program loader really works, and to touch on some of the technical concepts behind the power of the PROMAL System.

REVISING YOUR PROGRAM

Let's try making a small change to your program. Re-enter the EDITOR by pressing **F1** and <RETURN>. Your program will reappear in the text area of the display automatically.

Use the cursor keys to position the cursor over the "Y" in "YOURSELF" in the fourth line. Now press **CTRL-D** if you have an Apple II, or **CTRL-backarrow** (the key to the left of the "1" key) if you have a Commodore 64. What happened? This key deletes a character, but unlike the DELETE key, it pulls all the text on the line to the right over to "fill in the hole". Press this key 8 more times to get rid of "YOURSELF!". The line should look like this:

```
PUT "HELLO ",NL
```

Now put the cursor over the N in "NL" and press **CTRL-E** if you have an Apple or **SHIFT-INST** if you have a Commodore. The highlighted word INSERT appears in the status area at the bottom of the screen, indicating you are now in insert mode. In insert mode, anything you type will "push over" the text to the right of the cursor. With insert mode on, type:

CARG[1],

from your cursor position. To exit from insert mode, press any cursor key or <RETURN>. Your line should now look like this:

```
PUT "HELLO ",CARG[1],NL
```

In PROMAL, square brackets are used to enclose array subscripts instead of parentheses as in BASIC, so you can tell an array from a function easily. CARG is an array which is pre-defined (in the LIBRARY) for a special purpose. CARG is short for "Command Argument". The CARG array is an array of strings, which is automatically available to your program when it starts. CARG[1] is the first argument on the command line, CARG[2] is the second argument, etc.

Now press **F8** to exit, and select **W** and then **Q** to resave your new version to the workspace and exit to the EXECUTIVE. Now re-compile your program by pressing **F3** and <RETURN>. After your program compiles, type:

hello PROMAL

What happened? Your program displayed

```
HELLO PROMAL
```

Now try typing:

```
hello everybody
```

Whatever you type as the first argument gets passed to your program by the EXECUTIVE in the variable CARG[1], and your program prints it. Now try:

```
hello editor and executive
```

Your program will display:

```
HELLO EDITOR
```

What happened to "and executive"? Why didn't it display? Remember that a blank separates arguments on a command. Therefore "AND" got put in CARG[2], and "EXECUTIVE" got put in CARG[3], because the EXECUTIVE thinks they're each a separate argument. Your program only prints CARG[1].

You can force the EXECUTIVE to accept a string with blanks in it as a single argument by enclosing it in quotes. Try this:

```
hello "editor and executive"
```

You will now see:

```
HELLO editor and executive
```

If you were observant and if you were typing in lower case, you might have noticed another difference. Ordinarily the EXECUTIVE converts lower case commands and arguments to upper case as it reads them in. But if you enclose an argument in quotes, it won't convert the argument to upper case.

Naturally you can do more with command arguments than just print them. For example, you can use a command argument as a file name for your program to read or write. Our next example will demonstrate how to do this, and how you can do even more powerful "I/O redirection" on a command line.

Now that you've written a trivial PROMAL program, you may want to see a program that really does something useful. In the next section we'll take a look at a fairly short but useful program that illustrates a text-processing application, and illustrates many key PROMAL features, including using the LIBRARY.

If you want to postpone your exploration of these programs until later, you can simply turn off the computer.

A PROMAL TEXT-PROCESSING PROGRAM

Suppose you had a mailing list on a disk file called MAILLIST.T, which you had prepared with the PROMAL Editor. For simplicity's sake, suppose you had one entry per line, for example:

```
Board, Kim O., 6502 Processor St, Santa Clara, CA 95050
```

Suppose you had this file, and now you wanted to see all the customers in zip code 95050 of California. You could go buy a data base manager, right? Well, that's one way, but with PROMAL, a simple program may solve the problem. What you need is a command to find and display (or print) all the lines containing "CA 95050", for example:

```
FIND "CA 95050" MAILLIST.T
```

The FIND program to do this is already on your PROMAL Demo Diskette. To see the source program, type:

EDIT FIND.S

The EDITOR will "sign on", load the file into memory, and then display the first 20 lines of the program. The entire text of the program is reproduced on the following page for convenience.

After the first line, you will notice that there are a number of lines which start with a semicolon (;). These are comments. Any line which starts with ";" is a comment. Completely blank lines are considered comments, too. You can also put a ";" and a comment at the end of a statement. It is considered good programming practice to put enough comments in your program to adequately document it. It is important to realize that because PROMAL compiles your program, adding comments will not make your program use any more memory or execute slower. The compiler simply ignores all comment lines. This is an important difference from BASIC, where comments eat up valuable memory and increase execution time. Naturally adding comments to your source files will make them bigger, but this has no relation to the size of your executable object program. So feel free to comment!

Hold down the **"cursor down"** key and the EDITOR will "scroll" the text upward till you reach the end of the program. You can use the "cursor up" key to back up in the same way. Now scroll the screen until the line

```
INCLUDE LIBRARY
```

is on the top line of the display.

Now let's take a look at this program. Don't expect to understand all of it after this explanation; you just want to get the general idea of what a PROMAL program looks like and what some of the main concepts are. You'll need to study the PROMAL LANGUAGE MANUAL before you will actually be able to write or fully understand a complete program.

SOURCE PROGRAM "FIND.S"

```
PROGRAM FIND

;  by B. Carbrey 5/22/84
;  Program to print all lines in a text file which match a specified string.
;  Command syntax:  FIND <string> <file>
;  Examples:

;  FIND JANUARY MYDATA.T

;  will display all lines in the file MYDATA.T which contain the
;  word JANUARY.

;  FIND "New Jersey" MAILLIST.T > TEMP.T

;  will output all lines with New Jersey to the file TEMP.T from MAILLIST.T.
;  (quotes are needed if the string sought includes blanks).

INCLUDE LIBRARY

BYTE LINE[81]      ;Input/output buffer
WORD INFILE        ;Input file handle

FUNC BYTE HASSTRING  ; STRING
;  Returns true if LINE contains the desired STRING
ARG WORD STRING    ;desired string
WORD I             ;index to line
BEGIN
I=0
WHILE I < LENSTR(LINE)
  IF CMPSTR(STRING,"=",LINE+I,TRUE,LENSTR(STRING))
    RETURN TRUE
  I=I+1
RETURN FALSE
END

BEGIN ; main program...
IF NCARG <> 2  ; wrong # of arguments?
  PUT NL,"FIND error: 2 args. needed."
  PUT NL,"Usage: FIND string file"
  ABORT
INFILE=OPEN(CARG[2]) ; open data file
IF INFILE=0  ;open error?
  PUT NL,"FIND error: cant open ",CARG[2]
  ABORT
WHILE GETLF(INFILE,LINE) ; read a line
  IF HASSTRING(CARG[1])  ; string match?
    PUTF STDOUT,LINE,NL  ; yes, show
END   ;thats all folks!
```

First we need a few rules:

1. All variables must be defined (or "declared") before they are used.

2. Variables may have up to 31 characters. Unlike BASIC, which only looks at the first two characters, PROMAL uses all the characters to distinguish between variables. Also unlike BASIC, variables can contain PROMAL key words without causing trouble. Long variable names do not use any more memory than short variable names in your compiled program.

3. A variable definition tells the kind of data the variable represents. PROMAL supports the following data types:

BYTE: a single character, or an unsigned number from 0 to 255.
WORD: an unsigned number from 0 to 65,535, often used as an address.
INTEGER: a signed whole number between -32,767 and +32,767.
REAL: a "floating point" (decimal) number between -1.0E-37 and +1.0E+37.

4. Key words and variables must be separated from each other by blanks. They can't be run together as in BASIC.

5. Subroutines are given names (not line numbers), and can be either PROCedures or FUNCtions. A function starts with the key word FUNC, and returns a value to the calling program (much like BASIC). The function definition tells the type of data the function returns. Procedures start with PROC and do not return a value. They are similar to BASIC subroutines. Both procedures and functions are called by just putting the name in a statement. Functions and Procedures must be defined before they are called. In PROMAL, therefore, the main program always comes last, after all functions and subroutines.

With these rules in mind, let's look at few lines of the program:

```
BYTE LINE[81]             ;Input/output buffer
WORD INFILE               ;Input file handle
```

These two lines define two variables used by the program. The first variable is called LINE, and is of type BYTE. It is declared to have a dimension of 81. This variable will be used to hold a line of text read from the data file, up to 80 characters long. Note that PROMAL, unlike BASIC, does not have a primitive STRING data type. Instead, strings are treated as an array of bytes. This may seem like a shortcoming at first, but you will soon discover that PROMAL can actually manipulate strings very easily and much more efficiently than BASIC (experts in BASIC take note: this is because PROMAL never needs time-consuming "garbage collection"). By convention, a PROMAL string is an array of bytes terminated by a 0 byte (which is why LINE is dimensioned 81 to hold up to 80 characters).

INFILE is declared to be a variable of type WORD. Note that you still have to declare it, even though it is not an array. The comment says that INFILE is a "file handle". A file handle is a variable that is used to represent an active file. We'll demonstrate this later.

Now study the 13 lines beginning with:

```
    FUNC BYTE HASSTRING   ; STRING
```

These 13 lines define a function called HASSTRING. The function ends with the END line. In BASIC you could only define functions with one argument, and only on one line. In PROMAL, a function can have any number of arguments passed to it and can have any number of lines. In our case, the line,

```
    ARG WORD STRING   ;desired string
```

tells us that the function will expect one argument (ARG stands for "argument"), and it will be of type WORD and will be given the name STRING inside this function.

If BASIC is the only language you've ever used, this next part may be a little hard to grasp, so don't be concerned if you don't get it. Actually, the variable STRING will hold the **address** of whatever string the calling routine passes to it. If function HASSTRING is called like:

```
    HASSTRING("CA 95050")
```

then the STRING variable will hold the address of a string of bytes in memory containing "CA 95050", terminated by a zero byte. This whole string can be manipulated using the STRING variable.

The line:

```
    WORD I
```

declares a working variable for use within the function. Because this variable and STRING are both declared within the HASSTRING function, they are called **local** variables and only have meaning within the function. You may define other variables with the same names in other subroutines which would be completely different. This is an important concept. You may define and use variables in a subroutine without having to worry if you have already used the name for something different elsewhere. Variables declared before the first subroutine (such as LINE and INFILE) are **global** and may be used by all subroutines.

The BEGIN line starts the "action" part of the function. The next line:

```
    I=0
```

will be the first statement executed when function HASSTRING is called. It sets I to 0. This is an assignment statement, just like BASIC. Note that variables are not automatically initialized to 0, as in BASIC, but contain "garbage" until you assign something to them.

Now let's take a look at the rest of the function definition:

```
    WHILE I < LENSTR(LINE)
      IF CMPSTR(STRING,"=",LINE+I,TRUE,LENSTR(STRING))
        RETURN TRUE
      I=I+1
    RETURN FALSE
    END
```

There's a lot going on in these few lines! First of all, if you are looking at the lines on the 40 column Commodore display, one of the lines is only partially visible because it is longer than 40 characters. The last visible character is highlighted in reverse video so you can tell there is more to the line off-screen. To see the rest, just put the cursor on the line and move the cursor right. When it reaches the last column of the display, the whole line will scroll left to let you see the rest. Just press RETURN to restore the line to its normal position (Note: it is also possible to move the whole "window" to the right to view long lines - this is described in the PROMAL USER'S GUIDE).

The WHILE statement is one of several kinds of loops in PROMAL. It has no direct counterpart in BASIC, but is something like a combination IF and FOR-NEXT loop. A WHILE loop has the form:

```
WHILE condition
  statement 1
  statement 2
  ...
next statement
```

A WHILE statement tests the condition (like a BASIC IF statement). If the condition is TRUE, then <u>all</u> the <u>indented statements</u> (statement 1, statement 2, ...) are executed. After the last indented statement (...) is executed, control passes back to the top of the loop and the condition is tested again. This is repeated until the condition is false. Control then passes directly to the next (non-indented) statement.

Now you may see why PROMAL does not need statement numbers. **The structure of a PROMAL program is given by its indentation.** By the way, it is very easy to generate indented lines with the EDITOR. The TAB key (or CTRL-I) moves the margin in by one level of indentation (two spaces), and CTRL-Q (Apple) or CTRL-U (Commodore) moves it back out.

In our case, the WHILE tests to see if I is less than the length of the current line of interest, LINE. LENSTR is a built-in LIBRARY function which returns the length of a string, much like the BASIC function LEN.

Another built-in library function is CMPSTR, which is used in the next line:

```
IF CMPSTR(STRING,"=",LINE+I,TRUE,LENSTR(STRING))
```

CMPSTR compares two strings. It has the following form:

```
CMPSTR(string1,operator,string2,fold,limit)
```

Here string1 and string2 are the addresses of the two strings to be compared, and **operator** is the kind of comparison desired, chosen from:

```
"<", "<=", "<>", "=", ">=", ">"
```

which are the same as for BASIC. "Fold" should be TRUE if lower case letters are to be considered the same as the equivalent upper case letters. "Limit" is the maximum number of characters to compare.

The string comparison is done in an IF statement, which is much like an IF-THEN statement in BASIC. If the condition after the IF is true, then all the indented statements after the IF statement are executed. If the condition is false, then the indented statements are skipped. In our case there is only one indented statement. The IF statement above is roughly equivalent to the BASIC statement:

```
    IF STRING$ = MID$(LINE$,I,LEN(STRING$)) THEN...
```

except that there is no provision in BASIC for treating equivalent upper and lower case letters as equal during a string comparison.

If our comparison is true, then the desired string (for example, "CA 95050") has been found somewhere in the line, so

```
    RETURN TRUE
```

is executed, which exits from the function and returns the value TRUE to the calling routine. In PROMAL, FALSE is defined as a byte with the value 0, and TRUE is 1.

If the desired string is not found then:

```
    I=I+1
```

is executed, which advances to the next character of the line, and the WHILE loop is repeated. If the desired string is not found anywhere in the line, then control falls out of the WHILE loop into:

```
    RETURN FALSE
```

which simply exits back to the caller with the value FALSE returned. Therefore our function will return TRUE if LINE contains the string passed to it and FALSE if it doesn't.

Again, if some of this discussion seems unclear, don't worry. We are covering a lot of ground very fast and superficially, to try and give you "the big picture". Keeping this in mind, let's move on to the main program.

The main program starts after the last subroutine (there was only one in our case), and after the BEGIN. The first four lines are:

```
    IF NCARG <> 2  ;wrong # of arguments?
      PUT NL,"FIND error: 2 args. needed."
      PUT NL,"Usage: FIND string file"
      ABORT
```

The variable NCARG is a companion to CARG, and is predefined in the LIBRARY. It tells the number of command-line arguments passed from the EXECUTIVE to the program. In our case, the FIND program needs two arguments, so we check to see if two were given when FIND was executed. If not, we print two error messages and then ABORT. ABORT is a built-in procedure which returns control to the EXECUTIVE.

Assuming NCARG was 2 as it should be, the indented lines above would be skipped, and this line would be executed:

```
    INFILE=OPEN(CARG[2])
```

OPEN is another built-in function, which opens a file. It expects the name of the file as its argument. In our case, we want to open whatever file name is the second argument on the command line. OPEN returns 0 if the open was **not** successful, and the "file handle" otherwise. We store this handle in INFILE. From now on, anytime we want to access the file, we use this file handle. You may have multiple files open at once.

First we must make sure the file was opened successfully. If not (for instance, if the file was not found), we just print an error message and quit:

```
    IF INFILE=0     ;open error?
      PUT NL,"FIND error: can't open ",CARG[2]
      ABORT
```

Assuming the OPEN succeeded, we are now ready to search the file for lines containing our string:

```
    WHILE GETLF(INFILE,LINE)    ; read a line
```

GETLF is another built-in function. It stands for "GET Line from File". The first argument is the file handle and the second argument is the address where we want the line in memory. In our case, we will read the line into the array LINE. GETLF returns TRUE if it was successful and FALSE if end-of-file was encountered before any data could be read. Since we are testing this returned value in a WHILE loop, the indented statements after the WHILE will be repeated until end-of-file is reached. These indented lines are:

```
    IF HASSTRING(CARG[1]) ;string match?
      PUTF STDOUT,LINE,NL  ;yes, show
```

The IF statement calls our function, HASSTRING, passing it our desired string, which is the first command argument. IF HASSTRING returns TRUE, then the line contains the desired string and we should print it.

PUTF is another built-in procedure which is very similar to PUT. The only difference is that PUTF can output to a file, not just to the screen, like PUT. The first argument of PUTF is the file handle to write to. The remaining arguments are the same as for PUT.

Why do we want to use PUTF instead of PUT, and what is STDOUT? Well, we could have just used:

```
    PUT LINE,NL  ;yes, show
```

instead, but then we would only be able to output to the screen. You might like to be able to output to the printer or a file, without having to change the program. This re-directing of output can be done using a PROMAL feature called **I/O Redirection**.

The LIBRARY pre-defines a file handle called **STDOUT** (STandarD OUTput). This file handle is always open. By default, it is opened to the screen. However, you can redirect it from the EXECUTIVE to the printer or to a file. For example, consider this EXECUTIVE command:

```
    --> FIND "CA 95050" MAILLIST.T >P
```

This would redirect all the output to the printer ("P" is the name of the printer in PROMAL). This I/O redirection is done automatically by the EXECUTIVE. All your program has to do is output to STDOUT, and the output will go wherever the command line redirects it. It is not necessary to open this file, because the EXECUTIVE has already done it. The ">" is the output redirection operator of the EXECUTIVE and should follow the last argument. For example:

```
    --> FIND "CA 95050" MAILLIST.T > CALLIST.T
```

would output the list of lines to a file called CALLIST.T

This completes our discussion of the FIND sample program. If you want to try the FIND program, you can exit the editor (f8 key followed by Q and <RETURN>), and then execute it from the EXECUTIVE. You don't have to compile it because the object file FIND.C is already provided (but you can if you want to). Since you don't actually have a mailing list on disk to try it on, you might want to try this example instead:

```
    FIND "WHILE" FIND.S
```

What does this do?

If you'd like to try out the I/O redirection feature, but don't have a printer, try this:

```
    FIND "NL" FIND.S >W
    TYPE W
```

This will write all lines containing NL in the file FIND.S to the Workspace (deleting whatever was there before). The TYPE command will display the contents of the workspace on the screen. If you find the I/O redirection interesting, you might want to try this too:

```
    DUMP 1100 1180 >W
    TYPE W
```

What does this tell you about the built-in EXECUTIVE commands?

If you have persisted this far, you will have little difficulty learning to program your own applications in PROMAL. So far, you have learned how to EDIT and COMPILE a PROMAL program for handling text files.

REAL (FLOATING POINT) NUMBERS

So far the sample programs have dealt mostly with integer numbers, which is all that is needed for many applications. PROMAL also provides the data type REAL for floating point arithmetic. This is the kind of numeric data you know from BASIC, except that:

1. PROMAL real arithmetic is accurate to 11 significant digits instead of only 9 like BASIC.

2. You can precisely specify the output format for PROMAL real data (for example, how many decimal places you wish to use). This is very important for business applications, where decimal point alignment is expected:

```
BASIC OUTPUT                PROMAL OUTPUT

$100                        $100.00
 23.21                        23.21
 6.66666667                    6.67
 11.1                         11.10
 ------                      ------
 140.9766667                 140.98
```

3. PROMAL real arithmetic is usually faster than BASIC (but not nearly so fast as arithmetic on BYTE, WORD, or INT data types).

4. The PROMAL LIBRARY does not include built-in functions for square root, trig functions, exponentials, log functions, etc. Instead, these functions are provided in source form which you can include easily in your programs as needed. **APPENDIX K** describes these functions.

A SIMPLE BUSINESS PROGRAM

A simple program called BUDGET, illustrating the use of REAL data is included on the PROMAL Diskette. To run the BUDGET program, type:

```
BUDGET
```

from the EXECUTIVE. This program displays a hypothetical budget report showing expenditures for a month. You can study the source file, BUDGET.S, with the EDITOR to see how to format REAL numeric output. We won't cover this program here, but the file BUDGETDOC.T is a text file which describes the program. The PROMAL LANGUAGE MANUAL provides full information on REAL data.

AN ADVANCED PROGRAM

Have you ever wondered how BASIC, PROMAL, or any language evaluates an arbitrary arithmetic expression with variables, constants, parentheses and operators? If so, you may have imagined that it takes a very complex program to do so. Well, the file CALC.S on your Demo disk contains a PROMAL program of about 180 lines (excluding comments) which simulates a four-function calculator with 26 "memories". This program can evaluate any arithmetic expression of arbitrary complexity, using the four operators +, -, *, and / plus parentheses. To try the program, type:

```
   CALC
```

from the executive and just follow the directions. You may type in an expression, for example:

```
   3.26+(1-.82)/3
```

CALC will display the answer (showing two digits after the decimal point by default):

```
    =         3.32
```

You may also type an assignment statement to one of the 26 variables, A through Z. For example:

```
   p = 3.14
   x = 2*p
```

You may also change the number of decimal places displayed for answers. To change to 6 decimal places, type:

```
   #6
```

When you have finished experimenting with CALC, you can exit back to the EXECUTIVE by just pressing <RETURN> by itself. Now let's take a look at this program. Having a good-sized program will give us a chance to try out some of the more advanced editing features of PROMAL, too.

ADVANCED EDITOR FEATURES

From the EXECUTIVE, type:

```
   UNLOAD
   EDIT CALC.S
```

This will unload the programs we now have in memory and EDIT the source program for our four-function calculator. First, let's zip all the way down to the main program. Press the **F5** function key. The word FIND will appear in the status area below the working display area. Complete the FIND command like this:

```
   FIND 'MAIN'
```

and press RETURN. Be sure to remember the quotes (either " or ' will work, as long as they're the same on both ends of the string). The screen will almost instantly change to show the 20 lines beginning with:

```
   BEGIN ; Main Program
```

The status line will show that this is line 244. The cursor will be on the **M** in **Main**. This is how the FIND command works. You can also search for a particular line by specifying the line number instead of the quoted string. For example, FIND 1 will put you back at the top of the program, and FIND 9999 will put you at the end of the program (because there are less than 9999 lines). You can also back up or go forward from where you are by using a signed number. For example, FIND -100 will back up 100 lines from your present cursor position and then re-display.

Now go back to the beginning of the program by using a **FIND 1** command. Do a **CTRL-N** to see the next 20 lines. The status line should indicate that the top of the screen is now showing line 21. The lines starting with CON declare some constants in the program. For example, the line:

```
   CON LINESZ = 80 ; Max line size
```

defines the constant LINESZ to have the value 80 throughout the program. A constant is similar to a variable, but cannot have its value changed during execution. Below the constants are declarations for several variables. Press **CTRL-N** to advance to the next screen (starting with line 41). Notice the line:

```
   REAL VAR[27]  ; Variables A-Z cur value
```

This array of type REAL will hold the current value for each of our simulated calculator's "memories". Suppose you decide you want to change the variable VAR to be called MEMS instead, throughout the program. Use the **F6** function key, and complete the command as follows:

CHANGE **100 'VAR' 'MEMS'**

and press RETURN. This tells the EDITOR to change 100 occurrences of VAR to MEMS. You will see the first occurrence of VAR highlighted (in a comment) and the status area will show the prompt:

```
   CHANGE THIS STRING (Y/N/C=CANCEL)?
```

You can press Y to change the string and advance to the next occurrence, N to advance to the next occurrence without changing this one, or C to cancel the command at this point. Press **Y** and watch what happens. The next occurrence is highlighted, and you are again asked if you want to change it. However, this occurrence of VAR occurs in the word VARIABLES in a comment, so you don't want to change it. This is why you get a chance to "veto" each occurrence! Otherwise you might get some surprises. Press **N** to skip this occurrence. Continue pressing Y or N at each prompt, as appropriate (most will be N). When all occurrences have been found and presented to you, the status area will show:

```
   11 CHANGES MADE. PRESS RETURN.
```

to indicate the total number of changes made out of the 100 you specified. Press <RETURN>. Now press **F5** to issue a **FIND 72** command. You should see a DATA statement that looks similar to:

```
   DATA WORD HELP [] = ; Instructions...
   "Please enter an arithmetic expression",
   "such as \OF 3.14 * (20.25-8.5) \OE OR",
   "an assignment statement to A through Z",
   "such as \OF X=3-a/.55 \OE OR",
   "#n to select n decimal places (0-8)",
   "in answer (for example \OF #4 \OE) OR",
   "just press RETURN to exit the program.",
   " ", ; blank line
   0  ; end of list
```

This DATA statement declares an array of strings called HELP. PROMAL DATA statements are somewhat like BASIC DATA statements, except that you don't have to READ the data into a variable; its already there. DATA statements are the only statements that may take up multiple lines. The brackets after the variable name indicate an array. It is not necessary to put a dimension inside the brackets for a DATA declaration, because PROMAL will figure out how big to make the array. The first subscript of a PROMAL array is always 0, not 1. Therefore if we later have a statement:

```
   PUT HELP[2]
```

it will display:

```
   an assignment statement to A through Z
```

If you have a Commodore 64 instead of an Apple, the "\OF" and "\OE" in the DATA statements will be replaced by **£**12 and **£**92. The Commodore does not have a backslash key, so the "pounds sterling" key is used instead. The \ or **£** symbol is used in a string to embed non-printable characters. The \ is followed by exactly two hexadecimal digits which give the code for the desired embedded control character. In this case, the \OF and \OE turn reverse video on and off respectively on the Apple (and **£**12 and **£**92 do the same function on the Commodore 64).

Note that the array HELP is declared to have a type of **WORD** rather than **BYTE** as you might have expected. This is because the array is actually an array of **pointers** to the strings, and each pointer is a word (this is explained in the PROMAL LANGUAGE MANUAL).

Getting back to the EDIT session, suppose you decide that you want to move the lines containing the DATA statement for the HELP array closer to the top of the program for some reason. Put the cursor on the first line of the DATA statement (line 73) and press the MARK function key (**F3**). The line will be highlighted. Now move the cursor to the last line of the DATA statement (line 82) and press **F3** again. The lines will be shown in reverse video, indicating they are now "marked" for some action. The function key legends in the status now show the choices for what to do with the marked lines:

```
   1=DELETE 2=       3=MARK   4=WRITE  5=FIND   6=MOVE   7=COPY   8=CANCEL
```

COPY makes a copy of the marked lines at the new cursor location. WRITE lets you write out the marked lines to a file, the printer, or the Workspace. MOVE cuts the lines from where they are and inserts them at the cursor location.

DELETE simply deletes the marked lines. CANCEL lets you back out of the command without doing anything. Move the cursor up, scrolling the screen, until the cursor is on line 32, right below INCLUDE LIBRARY. Then press the MOVE function key (**F6**). Instantly you will see the screen re-displayed with the DATA statement moved to the new location, and the function key legends restored to normal.

You have now used all the function keys except F1 (DELETE LINE), F2 (INSERT LINE), and F4 (RECALL). You can experiment with F1 and F2 to see how they work. The RECALL function key (F4) is used to insert another file into your program at the cursor location. You can "cut" from one program and "paste" into another using WRITE and RECALL. These commands are described in the EDITOR section of the PROMAL USERS GUIDE.

Let's try one more thing with the EDITOR. **FIND** line **258**. The second line will begin a WHILE loop. Now suppose that for some reason you decide you need to add another loop of some kind that would encompass all the lines in the existing WHILE loop. This means that you need to indent all these lines by another level. Here's an easy way to do this. Put the cursor on the first line to be indented (the WHILE statement) and press **CTRL-^** (control key with shift and 6) if you have an Apple or **CTRL-J** if you have a Commodore 64. The line jumps to the right by two spaces. Press the same key again and the next line will indent. Just repeat this for as many lines as desired. To get rid of an unwanted level of indentation, press CTRL-O (the letter O). If you forget what control keys do what, use the HELP function key (F7).

You've now seen most of the major features of the EDITOR. If you wish you may study the rest of the program before exiting back to the EXECUTIVE. The comments should give you an idea what is going on. An important routine is procedure GETTOKEN. This routine reads characters from the line until it has a complete "token". A token is a complete number, a variable name, or an operator. When examining this routine, it will be helpful to know that the PROMAL operator @< is called an indirect operator. It follows a variable name that is being used as a pointer. LPTR@< therefore gets the character at the address given by LPTR. PROMAL supports other indirect operators as well, and can perform very powerful operations using pointers.

This routine uses several new functions defined in the LIBRARY. The standard function TOUPPER converts lower case characters to upper case. The standard function INSET determines if a character is in a string or set of characters. The function NUMERIC tests if a character is a numeric digit. The function STRREAL converts a string to a real value (somewhat like the BASIC function VAL). All these functions are detailed in the PROMAL LIBRARY MANUAL.

The actual parsing of the input to CALC is done by a technique known as "recursive descent". This is a goal-oriented technique which tries to match the input line to a known pattern, where each part of the pattern is processed by a subroutine. The theory involved is fairly advanced, and there is no need for you to understand it. If you are interested you can follow through how the CALC program processes a sample expression "by hand", which will be very instructive to understanding it. Also, Appendix P of the PROMAL LANGUAGE MANUAL has some further discussion of recursive descent parsing and syntax diagrams. This will only be of interest to the very advanced programmer, however. Our main purpose in examining this program is to show how to use the advanced editor features, and introduce some new statements. Also, we wanted to give credence to our claim that PROMAL is a suitable tool for developing compilers, assemblers, and other system programs.

Exit the EDITOR with function key **F8**. Don't save the modified program to workspace or to disk. Instead, simply exit back to the EXECUTIVE by pressing Q and <RETURN>. If you wish you may compile the original CALC program directly from disk, by typing the EXECUTIVE command:

```
COMPILE CALC
```

The COMPILER will ask you if you want to replace the existing CALC.C file when it finishes. You can reply either Y or N, since the result will be the same because you haven't changed the program.

SOME SPECIAL CAPABILITIES

For advanced programmers, here are some additional capabilites which may be important to you. These are described in the PROMAL LANGUAGE MANUAL:

1. You can assign the address of an **external** variable anywhere in memory. This allows you to give meaningful names to those "special" addresses. For example on an Apple system you might use:

```
EXT BYTE HIRES_ON AT $C057
```

which assigns the name HIRES_ON to the Apple Softswitch controlling hi-res graphics. The statement HIRES_ON=1 will therefore enable hi-res mode. A Commodore 64 example would be:

```
CON YELLOW=7
EXT BYTE BACKGROUND AT $D021
...
BACKGROUND = YELLOW
```

This sequence lets you give a meaningful name to the VIC background color register and manipulate it like any other variable. Isn't BACKGROUND=YELLOW much clearer than its BASIC equivalent of POKE 53281,7?

2. You can perform bit-level operations with PROMAL such as AND, OR, EXCLUSIVE OR, and SHIFTS. This often eliminates the need for machine language programming for I-O or special needs.

3. If you ever do need machine language routines, PROMAL provides a clean interface. You can call machine language routines from PROMAL with passed arguments (you can even set the hardware registers if you want). You can embed machine language routines in DATA statements, or load larger programs from disk using a built-in library function, or directly from the EXECUTIVE.

SOME SPECIAL SYSTEM-DEPENDENT DEMO PROGRAMS

Your PROMAL Demo disk has some other demonstration programs which exploit the special capabilities of your computer. You may wish to try these programs, which are described briefly below.

FOR THE APPLE II ONLY:

The following section applies only to the Apple II version of the PROMAL Demo diskette. If you have a Commodore computer, you may wish to skip down to the section, "FOR THE COMMODORE 64 ONLY".

A DATABASE APPLICATION PROGRAM

Now that you've seen some of the features of the PROMAL system in action, let's move on to a more complex application program. This will give you the chance to see more capabilities of the PROMAL language and to use some advanced editing features. This example takes advantage of the 80 column screen on the Apple IIe or IIc.

A hypothetical record store (Pete Promal's Record Shop) keeps a database of information about what albums are in stock on disk. The records are kept as ordinary sequential text files, with each record having the following format:

Field offset (size)

0 (19)	20 (10)	30 (20)	50 (10)	60 (2)	63 (4)	68 (4)
Artist name	(First)	Album name	Label	Year	Quantity	Bin #

For example, a typical record from the database file looks like this:

```
Jackson             Michael   Thriller            Epic      82 0040 0108
```

The last two columns tell the quantity on hand and the bin number, which is the physical location of the albums in the store.

A small segment of this hypothetical database is the file RECORDDATA.T on the demo disk. To see what it looks like, type:

TYPE RECORDDATA.T

The SORTDEMO program lets you sort this database by any of the fields, in ascending or descending order. Type:

SORTDEMO

from the EXECUTIVE and you should see:

```
                           PETE PROMAL'S RECORD SHOP

                                 SORT UTILITY

   Please select the sort options below.

   <space>   changes the highlighted option.
   <RETURN>  accepts the highlighted option.
   <ESC>     exits the program.

   Sort order is [ASCENDING]
```

If you press <space>, the highlighted word [ASCENDING] turns to [DESCENDING]. Pressing <space> again changes it back. Press <RETURN> to accept an ascending sort. In a similar manner, you can pick the remaining options to choose which field of the record to sort on and what output device to select (screen, disk, or printer).

After selecting the output device, the program will read the data file, sort the records in the manner you specified, and display the records in sorted order. The program then exits to the EXECUTIVE.

We will not go over this source file for this program, but if you wish to study it, the comments will help explain the operation of the program.

FOR THE COMMODORE 64 ONLY:

The following section applies only to the Commodore 64. If you don't have a Commodore 64, you may wish to skip down to the section, "IN CONCLUSION".

SPRITES, ANIMATION, AND SOUND SYNTHESIS WITH PROMAL

On the Commodore 64 PROMAL Diskette is a moderately complex program called BILLIARDS. This illustrates how PROMAL can be used to program sprites and sound in real time. Although the program is not a complete game program, it could be upgraded to be one. The program was deliberately chosen because it involves a lot of computation, a great deal more than most animated games. The program simulates the motion of three balls on a billiard table. It actually approximates the true behavior of the balls by the physical equations of motion. It solves collisions between balls and the rails using transfer of momentum, and includes coefficients of drag so the balls appear to slow down realistically. We're not saying you have to be a physicist to use PROMAL; but rather, that if you **do** need to do something complex, PROMAL can handle it a lot better than BASIC.

If you write this program in BASIC, there simply won't be any animation to speak of, because the balls will move slowly and jerkily. With PROMAL, the balls bound around the table fairly realistically, complete with sound effects. The billiards program has already been compiled for you on the PROMAL Diskette. To try it from the EXECUTIVE, type:

```
UNLOAD
BILLIARDS
```

After the program loads, you will see the table and three balls. One of the balls will zip diagonally across the table, colliding with the others. When the balls almost stop, the program exits back to the EXECUTIVE. You can use CTRL-B if you want to run it again. After you've learned more about PROMAL, you might like to modify this program to accept different angles and velocities for the "hit" on the cue ball. Or, if the balls on the screen look egg-shaped instead of round (due to the fact that pixels are not the same width as their height), you may want to change the sprite data to make the balls look more round.

We won't go over the source program for BILLIARDS, but you may examine it with the EDITOR if you wish. Although the comments will give you a good idea what is going on, you will need to study the PROMAL LANGUAGE MANUAL to fully understand the details of this program.

If you liked the BILLIARDS program, try this:

UNLOAD
INFILTRATOR

This will execute a fairly simple arcade-type game written entirely in PROMAL. The source code to this program is included on one of the PROMAL disks, if you'd like to examine or improve it. This game features smooth horizontal scrolling (impossible from BASIC), multiple multi-color sprites, and some "phasor" and explosion sound effects. Have fun!

Note: Because this program's source file is larger than 400 lines, you will not be able to compile it with the Demo compiler. The standard compiler can compile it easily with the B option. This program uses an INCLUDE file.

IN CONCLUSION

As we said earlier, the purpose of this manual was not to teach you how to program in PROMAL but to give you a good idea about what it is like to program in PROMAL. You have now gone through the mechanics of editing, compiling and running small and large programs. You have used a few EXECUTIVE commands and know how the EXECUTIVE can run programs and pass command arguments to programs. We hope you now have a good idea of what PROMAL is all about.

There are many powerful PROMAL features we have not even touched on yet. For example, you can use the EDITOR to prepare a "script" of commands for the EXECUTIVE to execute using the JOB command. This can be used to run a whole series of programs or commands as a "batch". Also, we have only seen a very few of the routines in the LIBRARY. There are many routines in it to do everything from a block-move to searching a linked list.

The PROMAL LANGUAGE MANUAL contains a full explanation of how to program in PROMAL, with plenty of examples provided. The LIBRARY has a reference manual of its own. The PROMAL USER'S GUIDE explains all the built-in EXECUTIVE commands in detail, as well as all EDITOR commands and compiler options. If you've liked what you've seen of PROMAL, but don't quite understand everything we've covered, you'll still have very little difficulty becoming a proficient PROMAL programmer.

MAKING WORKING DISKS

To start using PROMAL for writing your own programs, you will want to make a "working disk" with just the files you need on it plus your own programs. **Appendix O** tells you what files you need and how to copy them.

END USER VERSION AND DEVELOPER'S VERSION

The PROMAL system is available in two versions. The Standard or "End User" version gives you everything you need to develop programs for use on your computer, or for use on other people's computers which also have PROMAL. The Developer's Version includes all of the Standard System, but in addition contains a special Utility which will let you generate stand-alone PROMAL applications which will run on computers that do not have PROMAL. The developer's Version includes an Unlimited License for you to distribute these programs without payment or royalties of any kind to SMA. It also includes another small manual, the PROMAL DEVELOPER'S GUIDE. If you have purchased the End User system, you can upgrade later to the Developer's Version by just paying the difference in price (call for details).

SOURCE CODE FOR THE PROMAL SYSTEM

As an option, you can purchase the source code for the PROMAL EXECUTIVE, EDITor, and a source listing of the assembly language Runtime Package and Library. This is a unique benefit to PROMAL programmers, which is not available for any other commercial programming system. Contact SMA for pricing and availability.

GRAPHICS TOOLBOX

A Graphics Toolbox is available as an option. This provides very fast, high resolution drawing subroutines you can use from your PROMAL program. It can be used with either the End User or Developer's version of PROMAL.

With this package you can easily write programs to draw bar graphics, pie charts, function plots, and other graphic images. The Graphics Toolbox for the Commodore 64 draws in 16 colors using the 320 by 200 high resolution mode. On the Apple, graphics are done in 280 by 192 high resolution monochrome, since the Apple does not have a high-res color mode. Despite substantial underlying differences in hardware, graphics applications written for the Commodore are highly portable to the Apple and visa versa.

Call SMA for ordering information on the low-cost Graphics Toolbox.

REGISTERING YOUR PROMAL SYSTEM

After you've opened your sealed System diskette(s), be sure and fill out and send in your "END USER AGREEMENT, LICENSE and REGISTRATION FORM". This is the **only** way you will be able to get on our mailing list, so you can receive important upgrade notices, product announcements and the PROMAL NEWSLETTER.

CUSTOMER SERVICE

If you run into a problem with PROMAL you can't resolve by carefully reading the manual (and the trouble shooting guide in Appendix C), please call our Customer Service line at **(919) 878-3600**. Please be prepared to tell us what your computer hardware is, your version of PROMAL, serial number (on sealed disk if you've opened it), and an **exact** description of what actions you took and what symptoms you observe. Please get the exact wording of any error messages. This will make it much easier for us to help you.

Thank you for purchasing PROMAL. We are sure you will find it to be an indispensible addition to your software collection.