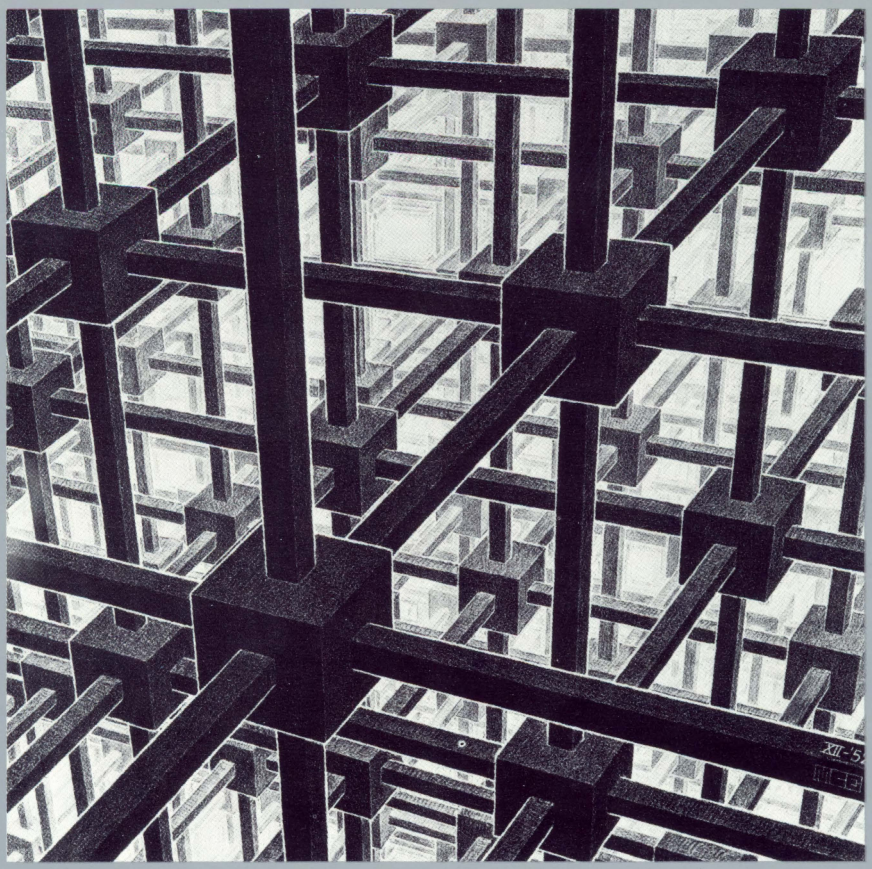
Apple III

Pascal

Program Preparation Tools



Notice

Apple Computer reserves the right to make improvements in the product described in this manual at any time and without notice.

Disclaimer of All Warranties And Liabilities

Apple Computer makes no warranties, either express or implied, with respect to this manual or with respect to the software described in this manual, its quality, performance, merchantability, or fitness for any particular purpose. Apple Computer software is sold or licensed "as is." The entire risk as to its quality and performance is with the buyer. Should the programs prove defective following their purchase, the buyer (and not Apple Computer, its distributor, or its retailer) assumes the entire cost of all necessary servicing, repair, or correction and any incidental or consequential damages. In no event will Apple Computer be liable for direct, indirect, incidental, or consequential damages resulting from any defect in the software, even if Apple Computer has been advised of the possibility of such damages. Some states do not allow the exclusion or limitation of implied warranties or liability for incidental or consequential damages, so the above limitation or exclusion may not apply to you.

© 1981 by Apple Computer
10260 Bandley Drive
Cupertino, California 95014
(408) 996-1010

© BEELDRECHT, Amsterdam/VEGA, NY
Collection Haags Gemeentemuseum

The word Apple and the Apple logo are registered trademarks of Apple Computer.

Reorder Apple Product #A3L0005

Apple /// Pascal

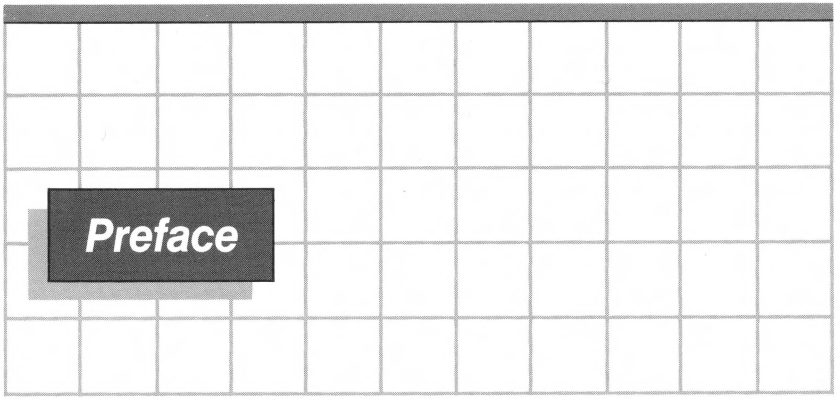
Program Preparation Tools

Acknowledgements

The Apple III Pascal system is based on UCSD Pascal. "UCSD PASCAL" is a trademark of the Regents of the University of California. Use thereof in conjunction with any goods or services is authorized by specific license only and is an indication that the associated product or service has met quality assurance standards prescribed by the University. Any unauthorized use thereof is contrary to the laws of the State of California.

Contents

Appendices



The Apple III Pascal system is described in three manuals:

Apple III Pascal: Introduction, Filer, and Editor
Apple III Pascal Program Preparation Tools
Apple III Pascal Programmer's Manual (Volumes 1 and 2)

Before using the Apple III Pascal system, or reading its manuals, you should be familiar with starting up the Apple III as described in the Apple III Owner's Guide.

When you are familiar with the contents of that manual, begin reading the Apple III Pascal Introduction, Filer, and Editor manual. The Filer and the Editor described in this manual are needed by everyone who uses the Pascal system. If you are familiar with the Apple II Pascal system, this manual will also show you the differences in operation between the two systems.

Apple III Program Preparation Tools is the next manual that you should read before you start to develop Pascal and assembly-language programs to run on the Apple III. The components of the Apple III Pascal system covered in this manual include:

The Linker, used to combine separately-developed program segments stored in libraries with your application program;

The Apple III Pascal 6502 Assembler, used to translate assembly-language source files produced by the Pascal Editor into machine-language code files; and

The Librarian, used to put commonly-used routines into libraries for use with application programs.

Your main source of information while developing Pascal programs will be the two volumes of the Apple III Pascal Programmer's Manual, which contain a complete description of the Pascal language on the Apple III and the use of the Apple III Pascal Compiler.

The Contents of this Manual

This manual tells how to use the 6502 Assembler, the Linker, and the Librarian in the preparation of your programs.

Chapter 1 tells you how to start using the program preparation tools and describes the steps in preparing a short Pascal program that includes assembly-language subroutines.

Chapter 2 tells you how to use the Apple III 6502 Assembler. It contains a brief explanation of parameter passing between Pascal and assembly language subroutines.

Chapter 3 tells you how to use the Linker to combine code files, either compiled P-code or assembled machine code, into a single code file for execution.

Chapter 4 tells you how to put separately compiled or assembled routines into a library file. Routines that you put into the system library or into a program library will be included automatically when you execute your program.

Appendix A is a brief description of a complex sample program.

Appendix B describes some special memory locations that will be useful if you are using assembly language routines.

Appendix C contains tables of the system files, the Pascal I/O devices, and the ASCII characters.

Appendix D is a handy summary of the commands used in the Assembler, the Linker, and the Librarian.

Appendix E contains notes on making turnkey diskettes and using Exec files.

Appendix F is a short bibliography of books on 6502 assembly language programming.

Appendix G contains lists of error messages.

Symbols Used in this Manual

The following symbols are used throughout this manual:

The pointing finger indicates an especially useful or noteworthy piece of information.

The eye means "watch out." It indicates a warning of an unusual situation to which you should be alert.

The stop sign warns you of a situation which might cause a problem, such as loss of data.

1

First Steps in Program Preparation

Introduction

This chapter uses a sample program to introduce the Apple III Pascal System's program preparation tools. Using the sample, this chapter shows you the sequence of steps involved in preparing programs to run with the Apple III Pascal System. Only the basic procedures are demonstrated in this introduction; once you have seen the way these procedures are used, you should refer to the other chapters of this manual for complete descriptions of the Assembler, the Linker, the Librarian, and the Library Mapper.

Before beginning this manual, you should be familiar with certain basic information, including:

- how to operate the Apple III,
- the nature of disk files,
- how to use the Filer, and
- how to use the Editor.

An introduction to the operation of the Apple III is given in the Apple III Owner's Guide. Information about disk files and about the use of the Filer and the Editor is found in Apple III Pascal: Introduction, Filer, and Editor.

The demonstration program in this chapter is written in Pascal and assembly language. For a description of the Pascal language and of the compiler that is used in the preparation of Pascal programs for the Apple III, refer to the Apple III Pascal Programmer's Manual.

This manual tells how to use parts of the Apple III Pascal system. It is not intended to teach you how to program in assembly language. If you are not already familiar with 65Ø2 assembly language, you should study one of the books listed in the Bibliography.

Pascal System Software Tools

The Apple III Pascal System is made up of a number of programs. These system programs are the tools you use in creating your own programs.

You should already be acquainted with the two most basic tools, the Filer and the Editor. You use the Editor to create a textfile that contains your program. If your program turns out to contain errors, you also use the Editor to make the necessary corrections.

The Apple III cannot run your program in the form you originally wrote it with the Editor. Before you can run your program, you must run the appropriate compiler or assembler, which converts the program textfile into an executable codefile.

One way to compile and run a Pascal program is to:

(1) Use the Editor to create a textfile of your program,
(2) Store the program textfile as the system workfile, then
(3) Type R for Run.

The Run command causes the system to compile the Pascal program in the workfile into a codefile and immediately execute it. After you have run your program, if you decide that you want to correct errors or make changes, you simply type E to invoke the Editor, which will automatically load your program textfile. After you have made the desired changes, you exit from the Editor and use the Run command again to compile and run the modified program.

You may want to break your program up into sections that are compiled separately and then combined. In this case, you will not normally use the Run command, since it causes automatic execution of the program or program section that has just been compiled. Instead, the sequence of operations you will use is:

(1) Use the Editor to create a separate textfile for each program section;
(2) Use the Compiler to make a codefile from each of these program sections;
(3) Use the Linker to combine the separate codefiles into the complete program; then
(4) Use the Execute command to run the program.

Some of your applications programs may require the use of assembly language routines to obtain more processing speed or to interface to custom hardware. The Apple III Pascal Assembler is a tool for creating assembly language routines

that can be called by a Pascal host program. The Assembler also provides a means for your assembly language routines to access data structures that you have defined in the Pascal host program.

Once you have assembled your assembly language routine, you use the Linker to attach the resulting codefile to your host program codefile.

The Pascal System includes a library of program routines (Units, Procedures, and Functions) that can be used in many other programs. Routines stored in this library are automatically loaded when you run a program that uses them. You can use the Librarian to add your own routines to this library or you can set up a separate library for your routines.

Getting Started

To use the Apple III Pascal System to write and execute programs, you need an Apple III with its video monitor and at least one external disk drive, in addition to the Apple III's built-in drive. The Apple III Owner's Guide tells how to connect the monitor and the external disk drive.

Program Preparation Diskettes

Several diskettes are supplied with your Apple III Pascal System. The programs you need for program development are on the following system diskettes:

PASCAL1
PASCAL2
PASCAL3

As supplied, these diskettes are arranged so that all of the files needed to boot the Pascal system are on a single diskette. If you are doing program preparation on an Apple III with only one external Disk III drive (or Disk II for the Apple III), you'll have to put your program files on these diskettes along with the system files, so that you can have all the necessary files in disk drives at one time. This can cause a space problem, because there is not much room left on the diskettes you will have in the drives most of the time. The solution to this problem is to make a copy set of system diskettes with the system files rearranged.

You can rearrange the system files in several different ways. For example, if you don't mind swapping diskettes each time you want to do a different task, you can make several system diskettes with only a few of the system files on each one. One of these diskettes might have only the Compiler and the SOS files needed for booting, leaving lots of room for program files to be compiled.

Note that you can move the system files around any way you wish. For example, if a larger disk is available, you might arrange the system files quite differently. If you want to devise your own file arrangement, refer to the TABLES appendix to find out which files are needed for each step in program development.

Another way to get more room on the system diskettes is to move the Apple III SOS files onto another diskette. With this arrangement, booting the system takes two steps, as described below and in the appendix USER NOTES. If you are just beginning to use the Pascal system, this might be a good arrangement for you, since it makes room on the Pascal system diskette for the system workfile.

The workfile is used in the creation of the sample program later in this chapter. If you want to use the sample as a practice exercise, this would be a good time for you to make a set of diskettes with the system files arranged this way.

Here is the procedure for rearranging the system files so that you can use the workfile and create your program with only two diskettes in drives at one time:

(1) Boot with the System Utilities diskette and use the Format Diskette option to format three blank diskettes, naming them NEWPASCAL1, NEWPASCAL2, and NEWPASCAL3.

(2) Invoke the Filer from diskette PASCAL1. Once the Filer is running, you can put different diskettes into the drives and use the Transfer command to move files around.

(3) Transfer files SOS.KERNEL, SOS.DRIVER, and SOS.INTERP from PASCAL1 to NEWPASCAL1. Transfer files LIBRARY.CODE, LIBMAP.CODE, SETUP.CODE, and AIIFORMAT.CODE from PASCAL3 to NEWPASCAL1.

(4) Transfer files SYSTEM.PASCAL, SYSTEM.MISCINFO, SYSTEM.LIBRARY and SYSTEM.FILER from PASCAL1 to NEWPASCAL2. Transfer file SYSTEM.EDITOR from PASCAL2 to NEWPASCAL2.

(5) Transfer all the files on diskette PASCAL2 to NEWPASCAL3, then Remove file SYSTEM.EDITOR from NEWPASCAL3.

Here is a list of the files on your program preparation diskettes. The order of the files on each diskette doesn't matter.

NEWPASCAL1	NEWPASCAL2	NEWPASCAL3
SOS.KERNEL	SYSTEM.PASCAL	SYSTEM.ASSMBLER
SOS.DRIVER	SYSTEM.MISCINFO	OPCODES.65Ø2
SOS.INTERP	SYSTEM.LIBRARY	ERRORS.65Ø2
SETUP.CODE	SYSTEM.EDITOR	SYSTEM.COMPILER
AIIFORMAT.CODE	SYSTEM.FILER	SYSTEM.SYNTAX
LIBRARY.CODE		SYSTEM.LINKER
LIBMAP.CODE		

NEWPASCAL1 contains the three SOS files needed for starting a cold boot. When the first stage of the boot is finished, the system will prompt you to remove this diskette and insert a Pascal system diskette into the built-in drive. This diskette also contains seldom-used system programs there isn't room for on the other diskettes.

NEWPASCAL2 contains SYSTEM.PASCAL , so it will be your Pascal system diskette for program development. It must be in the built-in drive whenever the system returns to the command level. The Filer and the Editor are both on this diskette, so you can use it for creating files on a user diskette in the external disk drive. If you are using the workfile, it will be on this diskette.

NEWPASCAL3 contains the major program development tools, that is, the Compiler, Assembler, and Linker, along with their associated error-message files.

If you have only one external disk drive on your Apple III, you can do your program development with NEWPASCAL2 in the built-in drive and NEWPASCAL3 in the external drive. You'll either have to use the workfile or store your program files on one of these diskettes for compiling, assembling, and linking.

Startup for Program Preparation

When you start up the system with the rearranged system diskettes, you have to use a two-step bootstrap procedure involving two diskettes, NEWPASCAL1 and NEWPASCAL2 . To boot the system, put diskette NEWPASCAL1 in the built-in disk drive, close the door of the drive, and press CONTROL-RESET. When the disk drive stops spinning and the message

```
Put system disk in built-in drive. Press RETURN
```

appears on the screen, remove NEWPASCAL1 from the built-in drive, insert NEWPASCAL2 , close the door of the drive, and press the RETURN key. The bootstrap operation will finish, then the Apple III will display the command prompt line and wait for you to type a command.

Starting the system with the two-stage boot requires only that you place the proper diskettes in the built-in disk drive. You can leave the other drive empty.

Make sure you never put two diskettes with the same name into the disk drives at the same time. Doing so may cause the directories of those diskettes to get scrambled, making the data they contain inaccessible.

Two disk drives are the practical minimum for doing program development on an Apple III. If your Apple III has more than two disk drives, you will usually find it convenient to leave NEWPASCAL2 in its normal location in the built-in drive and do all your file copying and transferring between diskettes in the other drives.

A Multi-part Sample Program

The sample program that follows is deliberately trivial: it is only used for demonstrating the steps in program development. Here is what the program does:

(1) create an array with the integers Ø through 9,
(2) display the array on the monitor, then
(3) display the array with each of its elements increased by one, and finally
(4) display the incremented array with each of its elements doubled.

This sample program is simple enough that it could be written, edited, compiled, and run as a single, simple object. Real programs are larger and more complex, and it is often advantageous to break them up into sections that reflect the structure of the program. With the program in sections, you need to edit and re-compile or re-assemble only the affected section in order to correct an error or modify a program step.

In the Pascal system, you create your program sections in the form of subroutines called procedures and functions. The program section that calls the subroutines, but is not a subroutine itself, is called the host program. A description of the techniques used for writing Pascal programs with subroutines is given in the chapter PROCEDURES AND FUNCTIONS in the Apple III Pascal Programmer's Manual.

To demonstrate the difference between a procedure and a function, the sample program includes one of each. To demonstrate the Assembler, both the procedure and the function are written in assembly language. The Pascal host program shown here and the assembly language routines that follow are discussed in more detail in Chapter 2, THE ASSEMBLER.

When writing the program, you should have a copy of your Pascal system diskette (NEWPASCAL2) in the built-in disk drive. This copy should not be write-protected, since you are going to write the sample program on it as the system workfile. If you already have a workfile on this diskette, you should make sure you have saved a copy of it on some other diskette, then use

the Filer's New command to remove the old workfile and initialize a new workfile.

For this sample program, you will be using the system workfile for your program text and code files, but you don't have to use the workfile for your program development. When you are creating more complex programs than this sample, you may prefer to ignore the workfile and do all your program development in named files. This approach is used with the sample program described in the chapter THE ASSEMBLER.

With the Command prompt line showing, select the Editor, then use the Insert command and type the sample host program below. Be careful to type all punctuation exactly as shown.

```
program CALLASM;

{       sample Pascal host program with calls       }
{       to an external function and procedure       }

type list = packed array[Ø..9] of Ø..255;

var i,k: integer; aa: list;

procedure INCARRAY(size:integer; var data: list);
external;

function TIMES2(data:integer):integer;
external;

begin
   writeln('initial array:');
   for i := Ø to 9 do
      begin
      aa[i] := i;
      write(aa[i],' ');
      end;
   writeln;
```

```
    writeln('array, incremented:');
    INCARRAY(1Ø,aa);
    for i := Ø to 9 do write(aa[i],' ');
    writeln;
    writeln('incremented array, times 2:');
    for i := Ø to 9 do write(TIMES2(aa[i]),' ');
end.
```

Compiling the Host Program

Once you are satisfied that you have the sample program typed correctly, type Q for Quit, then type U to update the system workfile with your program. Your program is now in a file on your Pascal system diskette called SYSTEM.WRK.TEXT .

If you were to try to use the Run command to compile and run this program, the system would detect the external references in it and use the Linker to try to link the external routines. That linking can't be done, because the assembly language external routines aren't available yet. To compile the sample host program without attempting to link or execute it, you use the Compile command.

With the Command prompt line present and diskette NEWPASCAL3 in a drive, type C for Compile. The disk drive with diskette NEWPASCAL3 in it whirrs and the message

```
Compiling...
```

appears on the display. As the Compiler creates the codefile version of the program, it displays messages like the ones below to keep you informed of its progress. For a description of these messages, refer to the appendix THE APPLE III PASCAL COMPILER in the Apple III Pascal Programmer's Manual.

```
Apple /// Pascal Compiler [A3/1.Ø]
<   Ø>.......
INCARRAY     [2ØØ9 words]
<  13>...
TIMES2       [198Ø words]
<  16>...
CALLASM      [ 1991 words]
<  19>...
34 lines
Smallest available space = 1991 words
```

Correcting Errors

If the Compiler discovers an error in the program, it will display a message pointing out the error. For example, if you misspell the keyword PROGRAM, the message is:

```
PRORAM <<<<
Line 2, Error 18: <sp>(continue), <esc>(terminate), E(dit
```

If you get such a message, just type E for Edit. The workfile will be automatically read back into the Editor for repairs. The error message will appear at the top of the screen and the cursor will indicate the location of the error. When you press the spacebar, the error message will disappear and you can move the cursor and correct the error. Then exit from the Editor with the Quit command, type U to update the workfile, and type C to compile the program again.

Saving the Sample Program

The workfile on NEWPASCAL2 now consists of the text version of the sample host program, named SYSTEM.WRK.TEXT, and the compiled codefile version of the sample program, named SYSTEM.WRK.CODE . Once the program compiles correctly, you should save these files.

To do this, you use the Filer's Save command. When the Command prompt line is on the screen, type F to invoke the Filer. When the Filer prompt line appears, type S for Save. The system will ask

```
Save as what file?
```

You should respond by typing the pathname you want your program saved under. For this sample program, type:

```
/NEWPASCAL3/CALLASM
```

When you press the RETURN key, the system will save both parts of the Pascal system diskette's workfile, SYSTEM.WRK.TEXT and SYSTEM.WRK.CODE, on NEWPASCAL3 under the local filenames CALLASM.TEXT and CALLASM.CODE . These messages will be displayed to tell you what has happened:

```
/NEWPASCAL2/SYSTEM.WRK.TEXT  --> /NEWPASCAL3/CALLASM.TEXT
/NEWPASCAL2/SYSTEM.WRK.CODE  --> /NEWPASCAL3/CALLASM.CODE
```

Sample Assembly Language Routines

The assembly language part of the sample program includes the procedure INCARRAY and the function TIMES2, which are called by the Pascal host program given above. You can usually write all of your program subroutines in Pascal; these routines are only written in assembly language to demonstrate the use of the Apple III Pascal Assembler.

Once you have saved your Pascal host program files, use the Filer's New command to remove the workfile so you can make a new workfile with the assembly language routines listed below.

Type Q to get out of the Filer, then use the Editor and the Insert command to type the following assembly language sample. Note: all text to the right of a semicolon (;) is commentary. The remaining text--the part that is capitalized--is the actual instructions, which must be typed exactly as shown. If some parts of these routines don't seem to make sense, don't worry about it now; explanations of these routines are given in the chapter THE ASSEMBLER.

```
;
;       sample macro pops word from eval. stack
;
        .MACRO  POP
        PLA
        STA     %1
        PLA
        STA     %1+1
        .ENDM
;
;       sample macro pushes word to eval. stack
;
        .MACRO  PUSH
        LDA     %1+1
        PHA
        LDA     %1
        PHA
        .ENDM
```

```
;
; sample function for Pascal, declared:
; function TIMES2(data:integer):integer;
;
        .FUNC   TIMES2,1        ;one word of parameters
RETURN  .EQU    Ø               ;temp store retn addr

        POP     RETURN          ;save Pascal retn addr
        PLA                     ;discard 4 bytes
        PLA                     ;of stack bias
        PLA                     ;(only need to do this
        PLA                     ;for .func)

        PLA                     ;lsb of data
        ASL     A               ;times 2
        TAX                     ;save in x
        PLA                     ;msb of data
        ROL     A               ;times 2, with carry
        PHA                     ;move msb to
                                ;evaluation stack
        TXA                     ;restore lsb to acc
        PHA                     ;move lsb to
                                ;evaluation stack

        PUSH    RETURN          ;restore Pascal ret add
        RTS                     ;RETURN to Pascal

;
; sample procedure for Pascal, declared:
; procedure INCARRAY(size:integer; var data: list);
;
        .PROC   INCARRAY,2      ;2 words of parameters
RETURN  .EQU    Ø               ;temp store retn addr
SIZE    .EQU    2               ;temp store SIZE param
PSEUDO  .EQU    ØEØ             ;Pascal pseudo reg.

        POP     RETURN          ;save Pascal retn addr
        PLA                     ;lsb of addr of array
        STA     PSEUDO
        PLA                     ;msb of addr of array
        STA     PSEUDO+1
        PLA                     ;lsb of SIZE param.
        STA     SIZE
        PLA                     ;msb of SIZE--discard
```

```
        LDY     #Ø              ;initialize array index
ALOOP   CLC                     ;clear for add
        LDA     @PSEUDO,Y       ;load array byte
        ADC     #1              ;increment
        STA     @PSEUDO,Y       ;store inc'd array byte
        INY                     ;increment array index
        CPY     SIZE            ;test versus array SIZE
        BMI     ALOOP           ;do while less than

        PUSH    RETURN          ;restore Pascal rtn adr
        RTS                     ;return to Pascal

        .END                    ;end of assembly
```

Once you are sure you have typed the assembly language routines correctly, use the Quit command to leave the Editor and type U for Update to save this text as the system workfile.

The file that contains the Assembler is named SYSTEM.ASSMBLER and is present along with the Pascal Compiler on diskette NEWPASCAL3 . With this diskette in any disk drive, type A from the Command level to start the Assembler. Messages like these will appear on the display:

```
Assembling...
65Ø2 Assembler [A3/1.Ø]
Output file for Assembler listing (<CR> for none):
```

The system is waiting for you to tell it what to do with the listing file the Assembler will produce. If you have a printer connected to your Apple III, you can direct the listing to the printer by typing .PRINTER . If you want to save the listing as a diskette file, you should type the pathname for it now. If you simply type a carriage return, the Assembler will not produce a listing.

After you have typed your response to this prompt, the assembly begins. The Assembler displays messages to keep you informed of its progress. For an explanation of these messages, see the chapter THE ASSEMBLER in this manual.

```
[10768]<    0>.....................
2 blocks for procedure code   9642 words left
[ 9635]<   26>.............
[ 9528]<   50>.........
Current minimum space is  9595 words
[ 9616]<   60>.............................
Current minimum space is  9571 words
[ 9592]<   94>
Assembly complete:          94 lines
     0   Errors flagged on this assembly
```

When the Assembler finds an error, it stops and displays an error message, then gives you the option of using the Editor to correct the error.

Once the routines have assembled correctly, you should use the Filer's Save command to save both parts of the workfile on NEWPASCAL3 under the local filename ASMSUBS.

Linking the Sample Program

Now that you have codefiles both for the host program and for the external routines that it calls, you can use the Linker to combine them into an executable codefile. From the Command level, type L for Linker. The system will display

```
Linking...

Apple /// Pascal Linker [A3/1.0]
Host file?
```

Type the pathname of the host file--in this case, /NEWPASCAL3/CALLASM . (You need not type the suffix .CODE .) The Linker will continue with:

```
Opening CALLASM.CODE
Lib file?
```

Lib stands for library: a codefile that contains external routines called by the host program. You should type the pathname of the file that contains your assembled external routines. You saved your assembly language workfiles on NEWPASCAL3, so type /NEWPASCAL3/ASMSUBS . The Linker will open this library file, then ask you for another one. Only one library file is used in this sample, so just type a carriage return and the Linker will continue.

Next the Linker will ask you for the name of a file to use for the library map. You will find out all about maps in the chapter THE LIBRARY; for now, just type a carriage return when the Linker asks for Map file. The Linker will continue until the display looks like this:

```
Lib file?/NEWPASCAL3/ASMSUBS
Opening ASMSUBS.CODE
Lib file?
Map file?
Reading CALLASM
Reading TIMES2
Output file?
```

Now you should type in the pathname for the file that will contain the complete program produced by the Linker. In this case, type /NEWPASCAL3/SAMP2. The Linker will continue with:

```
Linking CALLASM  # 1
    Copying func TIMES2
    Copying proc INCARRAY
```

When the Command prompt line is displayed, linking is complete: the codefile for the sample program is ready to execute. Type X for Execute, and when the computer responds with

```
Execute what file?
```

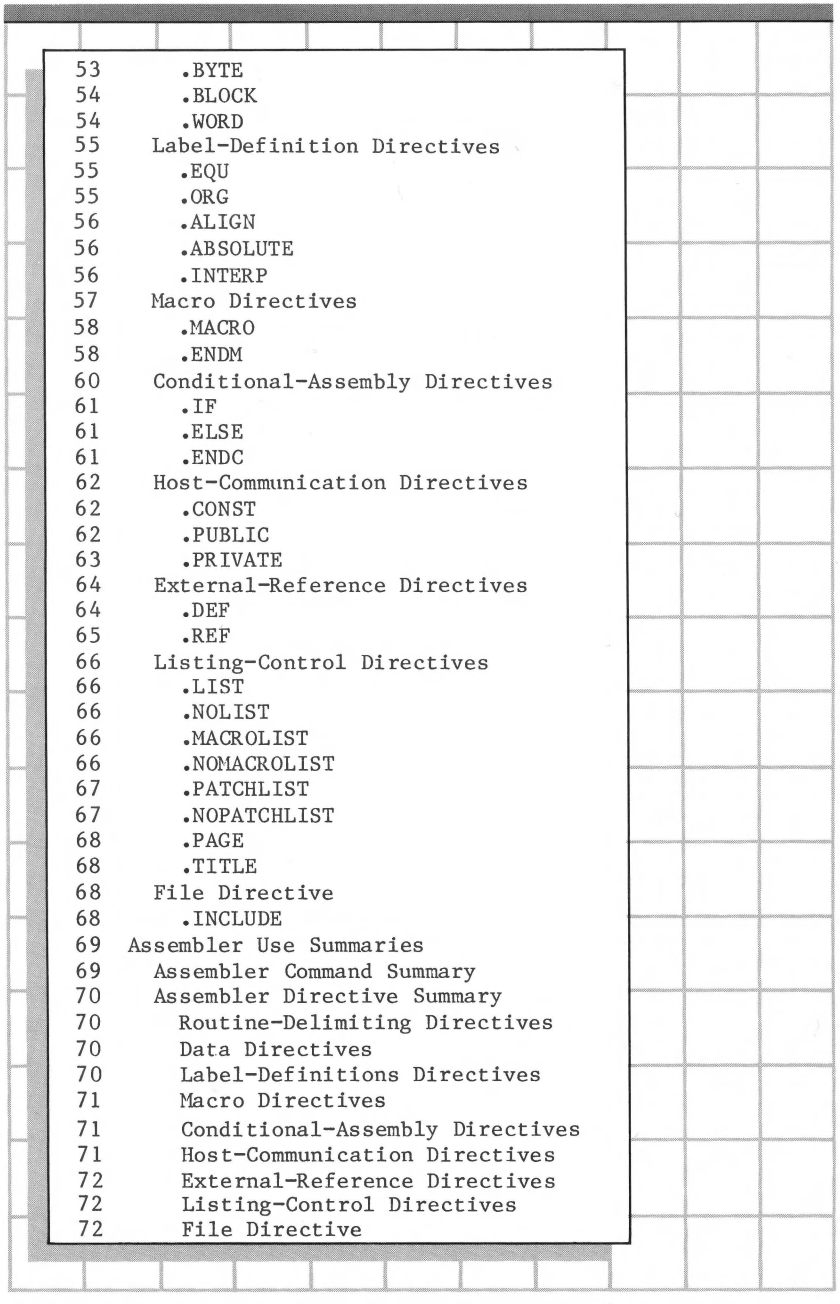
type the pathname /NEWPASCAL3/SAMP2 . The sample program will be loaded and executed. If it has been compiled and linked correctly, the sample program will display:

```
initial array:
Ø 1 2 3 4 5 6 7 8 9
array, incremented:
1 2 3 4 5 6 7 8 9 1Ø
incremented array, times 2:
2 4 6 8 1Ø 12 14 16 18 2Ø
```

This sample program is intended only to introduce you to the use of the Compiler, the Assembler, and the Linker. When you start to write real programs, you will need to learn more about these program preparation tools. You should read the remaining chapters of this manual and the chapters PROCEDURES AND FUNCTIONS, LIBRARY UNITS, and PROGRAM SEGMENTATION and the appendix THE APPLE III PASCAL COMPILER in the Apple III Pascal Programmer's Manual.

2

The Assembler



Introduction

Even if you write most of your programs in Pascal, you may occasionally need to write an assembly language routine for a part of your program that requires critical timing or that directly interfaces with hardware. The Apple III Pascal Assembler converts your assembly language routine into a codefile that can be linked with a Pascal host program. The Apple III Pascal Assembler is a version of the UCSD Adaptable Assembler, implemented specifically for the 65Ø2 microprocessor used in the Apple III computer.

This chapter tells how to use the Apple III Pascal Assembler, but it is not a complete description of the 65Ø2 assembly language used on the Apple III. For that you will need a reference book on 65Ø2 programming. Several good ones are listed in the Bibliography.

Files Needed

The assembly language routine you wish to assemble should be stored in a textfile. This file, called the source file, may be the text part of the system workfile. If there is no workfile currently assigned, you can specify any other textfile. The result of assembling the source file is a codefile called the object file.

In addition to the source textfile and disk space for the object codefile, you will need the following files to be able to use the Assembler:

```
SYSTEM.ASSMBLER
OPCODES.65Ø2
ERRORS.65Ø2 (optional)
SYSTEM.EDITOR (optional)
```

These files are supplied on the Apple III Pascal System diskettes, labelled PASCAL1 , PASCAL2 , and PASCAL3 . To gain more room for program files on the system diskettes, you can rearrange the system files. A convenient way to set up your system diskettes for program preparation is given in the section The System Diskette Files in the TABLES appendix and

is described in the chapter FIRST STEPS IN PROGRAM PREPARATION. The rearranged diskettes are named NEWPASCAL1 , NEWPASCAL2 , and NEWPASCAL3 .

Diskette file SYSTEM.ASSMBLER contains the Assembler program. File OPCODES.65Ø2 contains the instruction mnemonics for 65Ø2 assembly language as used in the Apple III. These files are supplied on diskette PASCAL2 ; in the rearranged system diskettes, the assembler files are on NEWPASCAL3 . They must be available on some diskette in one of the system's disk drives when you type A from the command level to invoke the Assembler.

Diskette file ERRORS.65Ø2, on diskette NEWPASCAL3 in the rearranged system diskettes, contains the Assembler error messages. This file is optional; if it is not available, the Assembler will report errors by number and you will have to look up the error descriptions in the ERROR MESSAGES appendix.

When the Assembler detects an error, it gives you the option of immediately invoking the Editor to correct the problem. If you type E for Edit from the Assembler, the file SYSTEM.EDITOR , which is normally found on your Pascal system diskette NEWPASCAL2 , must be on a diskette in one of the disk drives.

With NEWPASCAL2 in the built-in drive and NEWPASCAL3 in the first external drive, your Apple III has all the files needed to use the Pascal Editor, Compiler, Assembler, and Linker.

If your system has only two disk drives and you wish to assemble a textfile that is not on diskette NEWPASCAL2 or NEWPASCAL3 , you'll have to use the Filer's Transfer command to transfer that textfile onto either NEWPASCAL2 or NEWPASCAL3 before assembling.

Using the Assembler

You invoke the Assembler by typing A for Assembler from the Command level of the Apple III Pascal System. The screen immediately shows the message

```
Assembling...
```

If no workfile is available as a result of the Editor's Update command or the Filer's Get command, the system prompts you with the message:

Assemble what text?

You should respond by typing the pathname of the source file, that is, the textfile that contains the routines you wish to assemble. It is not necessary to type the suffix .TEXT ; the suffix is automatically supplied by the Assembler if you don't type it. If you wish to defeat this feature in order to assemble a textfile whose pathname does not end in .TEXT , type a period (.) after the last character of your pathname.

Next you will be asked for the name of the codefile where you wish to save the assembled version of your routine:

To what codefile?

If you simply press the RETURN key the command will not be terminated, as you might expect. Instead, the assembled version of your routine will be saved on the Pascal system diskette's workfile SYSTEM.WRK.CODE .

Press the ESCAPE key and then press RETURN in response to this prompt or the previous one to abandon the assembly and return to the Command level.

If you want your object codefile to have the same pathname as the source textfile (with the suffix .CODE instead of .TEXT), you should respond to this prompt by typing a dollar sign ($) and pressing the RETURN key. This feature makes it easy to use the same name for both versions of your routine. The dollar sign repeats your entire source-file pathname, including the volume identifier, so do not specify the volume before you type the dollar sign.

If you want your codefile to be stored under some other pathname, type that pathname in response to the prompt. It is not necessary to type the suffix .CODE ; the suffix is automatically supplied by the Assembler. If you wish to defeat this feature in order to specify a pathname that does not have a .CODE suffix, type a period (.) after the last character of your pathname.

After the source and object files for the assembly have been specified, the next prompt line is:

```
65Ø2 Assembler [A3/1.Ø]
Output file for Assembler listing (<CR> for none):
```

Now you must specify where you want the Assembler to send the assembly listing. The assembly listing gives the address and the assembled object code for each statement in the source routine. It also includes reference symbol tables, described below. This listing is independent of the object codefile that is saved as the final output of the assembly. See the example later in this chapter for a sample assembly listing.

If you wish, you can have the assembly listing sent to a diskette file, to the console, or to the printer. As usual for a console or printer output, the word CONSOLE or PRINTER must be preceded by a period, i.e., .CONSOLE .

If you specify a disk file for the assembly listing, you do not need to type the .TEXT suffix; .TEXT will be added automatically if it is needed. Unlike many parts of the system, ending the specified pathname with a period does not suppress the addition of the .TEXT suffix. However, if the pathname you type includes the string .TEXT anywhere in it, the pathname is used exactly as typed.

If you store an assembly listing as a diskette file, you can use the Editor to look at it, but the control characters in the file make it very difficult to edit. Form feeds (CONTROL-L) in the file are interpreted as clear screen characters by the Editor. Data-link escape commands (CONTROL-P), followed by a number, appear as tabs, but cause the Editor to lose track of the cursor position. You can use the Editor's Replace command to remove the CONTROL-Ls and CONTROL-Ps when you start editing a listing file.

Press the RETURN key if you do not want this listing. If you wish to abandon the assembly at this point, press the ESCAPE key and then press RETURN.

There is an additional diskette file requirement that is not obvious. The Assembler uses a small area on the Pascal system diskette for a temporary file containing information that will be needed if the Linker is used. This diskette file does not normally appear in the

diskette's directory, but space for it (usually less than four blocks) must be available on the Pascal system diskette after the Assembler has opened both the output codefile and the output textfile for the listing (if any). An attempt to assemble without space on the main system diskette for this file causes the message IO ERROR: NO ROOM ON VOL and a message prompting you to press the spacebar to reinitialize the system. After you have done this, your Pascal system diskette may contain a new file named LINKER.INFO, of zero length and type Datafile. You can remove this file if you wish.

If you have specified a destination for the assembly listing, the system reports whether or not the output device is on line, that is, connected and operating.

After you tell the Assembler what to do with the listing file, it starts assembling the source file. If you told the Assembler to send the assembly listing to .CONSOLE , the listing appears on the display screen, one line at a time. If you did not direct the assembly listing to .CONSOLE , a simple display showing the assembly's progress appears on the screen instead. In this mode, the Assembler displays a dot for each line of code assembled and a line counter every 5Ø lines. Upon completing each procedure or function, it displays the number of words of space available for the reference symbol table (described below) in brackets, followed by the message

```
Current minimum space is XX words
```

If you used the INCLUDE directive in your routine, the Assembler will display the message:

```
.INCLUDE <pathname>
```

each time it encounters the directive, to inform you that the named file has been included in the assembly.

If the Assembler encounters an error, it displays a message that shows the offending text and indicates the nature of the error. For example, you might see

```
    $Ø4  .EQU *
Identifier previously declared
```

The error message is taken from the file ERRORS.65Ø2 . If this file is not on a disk drive, or if there is not enough

available memory to load it, only the error message number is given. In that case, you might see

```
      $Ø4  .EQU *
Error #   9
"ERRORS.65Ø2" file not found
```

A complete list of Assembler syntax-error messages that correspond to these error numbers appears in this manual's appendix ERROR MESSAGES. Note that the descriptive error message is given only at the time the error is detected, and is not given by the Editor as it is when you use the Compiler. After each error is found, the Assembler prompts you with the following choice:

```
E(dit,<space>,<esc>
```

This is similar to the choice that you are given when the Compiler encounters an error. If you wish to proceed with the assembly, looking for more errors, press the spacebar. If you decide to terminate the assembly and return to the outermost Command level, press the ESCAPE key. If you wish to correct the error, type E. The Editor will be loaded into the computer and the workfile will be read into the Editor, ready for editing. If the file you are assembling is not the workfile, this prompt appears:

```
>Edit:
No workfile is present. File? (<ret> for no file <esc> to exit)
```

You should type the pathname of the source file being assembled. If the error occurred in an include file, you should type the name of that file, which is given in the last include message that was displayed. The file you specified will then be read into the Editor, the Editor will display a general error message, and the cursor will be placed at the point in the text where the error was detected.

The Editor does not display specific messages for errors reported by the Assembler. Therefore you should be sure to note which error is being reported by the Assembler before you type E to invoke the Editor.

When the assembly is finished, the Assembler displays a message telling you that it is finished and the number of errors that it found.

If the Assembler found no errors, it stores the object code in the Pascal system diskette's workfile SYSTEM.WRK.CODE or in the codefile with the pathname that you specified earlier. The assembled codefile cannot be executed by itself; it can only be used by Linking it with a host program's codefile. For information about linking, see the example later in this chapter, and also see this manual's chapter THE LINKER. For information about putting assembled codefiles into a library file so that the Run command will automatically link them to a host program, see this manual's chapter USING THE LIBRARY.

The code part of the Pascal system diskette's workfile, SYSTEM.WRK.CODE, is automatically erased whenever you use the Editor's Update command to update the text part of the workfile.

Reference Symbol Table

To help you locate the symbols in the listing of your assembly language routines, the Assembler generates Reference Symbol Tables. A Reference Symbol Table, entitled SYMBOLTABLE DUMP, appears in the assembly listing following the listing of each procedure or function. Each entry in the Reference Symbol Table contains three items. The first item is the symbol itself--the entries are listed alphabetically by symbol. The second item is the symbol type, using the abbreviations listed at the top of each table. If the symbol represents a label or an absolute, the third item is the definition address. A label's definition address is the four-digit hexadecimal number shown in the left-most column of the assembly listing for the statement that defines the label. If the symbol represents neither an absolute nor a label, the third item is filled in with dashes. A vertical bar (|) ends each entry. Here is one of the Reference Symbol Tables from the assembly listing example given later in this chapter:

```
PAGE -   5  INCARRAY  FILE:  SYMBOLTABLE DUMP


AB - Absolute      LB - Label      UD - Undefined      MC - Macro
RF - Ref           DF - Def        PR - Proc           FC - Func
PB - Public        PV - Private    CS - Consts


ALOOP     LB ØØ12|  INCARRAY PR ----|  POP      MC ----|
PSEUDO    AB ØØEØ|  PUSH     MC ----|  RETURN   AB ØØØØ|
SIZE      AB ØØØ2
```

The first entry is for a label named ALOOP, defined at address ØØ12. The second entry shows that INCARRAY is the name of the procedure. The third entry shows that POP is the name of a macro. The fourth entry shows that PSEUDO is an absolute that has been assigned the value ØØEØ .

A Sample Program

The sample program that follows includes the following items:

1) The assembly language source text of an external function, TIMES2, and an external procedure, INCARRAY.

2) The assembly listings for the function and the procedure.

3) A Pascal host program that calls the function and the procedure.

4) Sample commands for compiling the Pascal host program.

5) Sample commands for linking the assembly language routines to the Pascal host program.

Assembly-Language Routines

You can create your program textfile in the system workfile and assemble it, as described in the previous chapter. The alternative approach is to use named files for your program text and code. That approach is demonstrated in the sample that follows.

First, use the Filer's New command to remove any existing workfile. Then, using the Editor's Insert command, type the

following assembly language routine into the computer just as it appears here. Be careful with punctuation and special characters.

Note: The text following the semicolon (;) on each line is a comment. You can omit the semicolon and the comment if you wish.

```
;
;       sample macro POPs word from eval. stack
;
        .MACRO  POP
        PLA
        STA     %1
        PLA
        STA     %1+1
        .ENDM

;
;       sample macro PUSHes word to eval. stack
;
        .MACRO  PUSH
        LDA     %1+1
        PHA
        LDA     %1
        PHA
        .ENDM

;
; sample function for Pascal, declared:
; function TIMES2(data:integer):integer;
;
       .FUNC    TIMES2,1        ;one word of params
RETURN .EQU     Ø               ;temp store rtn addr

        POP     RETURN          ;save Pascal rtn ad
        PLA                     ;discard 4 bytes
        PLA                     ;of stack bias
        PLA                     ;(only need to do
        PLA                     ;for .func)

        PLA                     ;lsb of data
        ASL     A               ;times 2
        TAX                     ;save in x
        PLA                     ;msb of data
        ROL     A               ;times 2, with carry
        PHA                     ;move msb to
```

```
                                    ;evaluation stack
        TXA                         ;restore lsb to acc
        PHA                         ;move lsb to
                                    ;evaluation stack

        PUSH    RETURN              ;restore Pascal
                                    ;return address
        RTS                         ;RETURN to Pascal

;
; sample procedure for Pascal, declared:
; procedure INCARRAY(size:integer; var data: list);
;
        .PROC   INCARRAY,2          ;2 words of params
RETURN  .EQU    Ø                   ;temp store rtn ad
SIZE    .EQU    2                   ;temp store SIZE
PSEUDO  .EQU    ØEØ                 ;Pascal pseudo-reg.

        POP     RETURN              ;save Pascal rtn ad
        PLA                         ;lsb of array addr
        STA     PSEUDO
        PLA                         ;msb of array addr
        STA     PSEUDO+1
        PLA                         ;lsb of SIZE param.
        STA     SIZE
        PLA                         ;msb of SIZE discard

        LDY     #Ø                  ;init'ize array indx
ALOOP   CLC                         ;clear for add
        LDA     @PSEUDO,Y           ;load array byte
        ADC     #1                  ;increment
        STA     @PSEUDO,Y           ;store incd ar'y byt
        INY                         ;incrm't array index
        CPY     SIZE                ;test vs array SIZE
        BCC     ALOOP               ;repeat if lt or eq

        PUSH    RETURN              ;restore Pascal
                                    ;return address
        RTS                         ;RETURN to Pascal

        .END                        ;end of assembly
```

The Assembly Listing

After you have typed the assembly language sample with the Editor, type Q for Quit, select the Write option, and save the program in a diskette file named ASMSUBS . If your Apple III has two or more external disk drives, you can keep your user diskette in the second one (.D3). For this example, the user diskette is named MYDISK and the assembly language text file is named /MYDISK/ASMSUBS.TEXT . If you only have one external drive, you'll have to keep your developing program on one of the system diskettes. If you save your text and code files on NEWPASCAL2 , you can put your listing files on NEWPASCAL3 .

After you have saved your textfile, and with the system at the Command level, type A to invoke the Assembler. The system loads the Assembler, which displays a prompt asking for the source textfile:

```
Assembling...
Assemble what text?
```

If your user diskette is MYDISK , you should respond with the pathname /MYDISK/ASMSUBS . If your text file is on the system diskette, respond with /NEWPASCAL2/ASMSUBS . Next, the Assembler asks you for the name to use for the assembled object codefile:

```
To what code file?
```

To save the codefile with the same name as the textfile, except for the suffix, you may either type the pathname /MYDISK/ASMSUBS again, or simply type a dollar sign ($). It is usually convenient to use the same pathname for both versions of your program, and most commands can use the suffix .TEXT or .CODE to choose the appropriate version.

Note: if the source textfile had been available in the main system diskette's workfile SYSTEM.WRK.TEXT or some other workfile that you designated using the Filer's Get command, the prompting questions shown above would not have appeared. Instead, the Assembler would have assembled the text workfile and would have stored the object codefile as SYSTEM.WRK.CODE .

The next messages from the Assembler identify the version of the Assembler and ask you where to send the assembly listing:

```
65Ø2 Assembler [A3/1.Ø]
Output file for assembler listing (<CR> for none):
```

If you have a printer connected to your Apple III, type .PRINTER to send the assembly listing to the printer. If you respond to the last prompting message by typing .CONSOLE , the assembly listing will be sent to the monitor screen in place of the Assembler's usual screen display. If you want the listing file to be saved, specify the pathname where you want it sent: for example, /MYDISK/ASUBS.LST .

After you specify the disposition of the listing, the assembly process will begin. The console screen will show the usual Assembler display: a dot for each line of the source program, and messages that tell you how much memory space, in 16-bit words, is available at each stage of the assembly. The screen display will look something like this:

```
[1Ø768]<   Ø>.............
2 blocks for procedure code  9642 words left
[ 9635]<  26>.....................................
[ 9528]<  5Ø>........
Current minimum space is 9595 words
[ 9616]<  57>.............................
Current minimum space is 9571 words
[ 9592]<  92>
Assembly complete:       94 lines
     Ø   Errors flagged on this Assembly
```

Meanwhile, the printer has been printing the assembly listing, which looks like this:

```
PAGE -   Ø
Current memory available:  1Ø192
ØØØØ|
ØØØØ|                      ;
ØØØØ|                      ;       sample macro pops word from eval. stack
ØØØØ|                      ;
ØØØØ|                              .MACRO  POP
ØØØØ|                              PLA
ØØØØ|                              STA     %1
ØØØØ|                              PLA
ØØØØ|                              STA     %1+1
ØØØØ|                              .ENDM
```

```
0000|
0000|                          ;
0000|                          ;       sample macro pushes word to eval. stack
0000|                          ;
0000|                                  .MACRO  PUSH
0000|                                  LDA     %1+1
0000|                                  PHA
0000|                                  LDA     %1
0000|                                  PHA
0000|                                  .ENDM
0000|
0000|                          ;
0000|                          ; sample function for Pascal, declared:
0000|                          ; function TIMES2(data:integer):integer;
0000|                          ;
2 blocks for procedure code  9642 words left
PAGE -   1  TIMES2    FILE:ASMSUBS


0000|                                  .FUNC   TIMES2,1        ;one word of params
Current memory available:   96Ø3
0000| 0000                     RETURN  .EQU    Ø               ;temp store rtn addr
0000|
0000|                                  POP     RETURN          ;save Pascal rtn ad
0000| 68                    #         PLA
0001| 85 ØØ                 #         STA     RETURN
0003| 68                    #         PLA
0004| 85 Ø1                 #         STA     RETURN+1
0006| 68                              PLA                     ;discard 4 bytes
0007| 68                              PLA                     ;of stack bias
0008| 68                              PLA                     ;(only need to do
0009| 68                              PLA                     ;for .func)
000A|
000A| 68                              PLA                     ;lsb of data
000B| ØA                              ASL     A               ;times 2
000C| AA                              TAX                     ;save in x
000D| 68                              PLA                     ;msb of data
000E| 2A                              ROL     A               ;times 2, with carry
000F| 48                              PHA                     ;move msb to
000F|                                                           ;evaluation stack
0010| 8A                              TXA                     ;restore lsb to acc
0011| 48                              PHA                     ;move lsb to
0011|                                                           ;evaluation stack
0012|
```

```
ØØ12|                                   PUSH    RETURN          ;restore Pascal
ØØ12|                                                           ;return address
ØØ12| A5 Ø1                 #           LDA     RETURN+1
ØØ14| 48                    #           PHA
ØØ15| A5 ØØ                 #           LDA     RETURN
ØØ17| 48                    #           PHA
ØØ18| 6Ø                              RTS                     ;RETURN to Pascal
ØØ19|
ØØ19|
ØØ19|                       ;
ØØ19|                       ; sample procedure for Pascal, declared:
ØØ19|                       ; procedure INCARRAY(size:integer; var data: list);
ØØ19|                       ;
PAGE -   2  TIMES2    FILE:ASMSUBS  SYMBOLTABLE DUMP


AB - Absolute     LB - Label     UD - Undefined     MC - Macro
RF - Ref          DF - Def       PR - Proc          FC - Func
PB - Public       PV - Private   CS - Consts

POP      MC ----|  PUSH     MC ----|  RETURN   AB ØØØØ|  TIMES2   FC ----|
PAGE -   3  TIMES2    FILE:ASMSUBS


Current minimum space is 9595 words
PAGE -   4  INCARRAY  FILE:ASMSUBS


ØØØØ|                                   .PROC   INCARRAY,2      ;2 words of params
Current memory available:   96Ø3
ØØØØ| ØØØØ                  RETURN    .EQU    Ø                 ;temp store rtn ad
ØØØØ| ØØØ2                  SIZE      .EQU    2                 ;temp store SIZE
ØØØØ| ØØEØ                  PSEUDO    .EQU    ØEØ               ;Pascal pseudo reg
ØØØØ|
ØØØØ|                                   POP     RETURN          ;save Pascal RTN ad
ØØØØ| 68                    #           PLA
ØØØ1| 85 ØØ                 #           STA     RETURN
ØØØ3| 68                    #           PLA
ØØØ4| 85 Ø1                 #           STA     RETURN+1
ØØØ6| 68                                PLA                     ;lsb of array addr
ØØØ7| 85 EØ                            STA     PSEUDO
ØØØ9| 68                                PLA                     ;msb of array addr
ØØØA| 85 E1                            STA     PSEUDO+1
ØØØC| 68                                PLA                     ;lsb of SIZE param.
ØØØD| 85 Ø2                            STA     SIZE
ØØØF| 68                                PLA                     ;msb of SIZE discard
```

```
ØØ1Ø|
ØØ1Ø| AØ ØØ                        LDY     #Ø              ;init'ize array indx
ØØ12| 18                   ALOOP   CLC                     ;clear for add
ØØ13| B1 EØ                        LDA     @PSEUDO,Y       ;larray byte
ØØ15| 69 Ø1                        ADC     #1              ;increment
ØØ17| 91 EØ                        STA     @PSEUDO,Y       ;store incd ar'y byt
ØØ19| C8                          INY                     ;incrm't array index
ØØ1A| C4 Ø2                        CPY     SIZE            ;test vs array SIZE
ØØ1C| 3ØF4                        BMI     ALOOP           ;do while less than
ØØ1E|
ØØ1E|                             PUSH    RETURN          ;restore Pascal
ØØ1E|                                                      ;return address
ØØ1E| A5 Ø1                 #       LDA     RETURN+1
ØØ2Ø| 48                   #       PHA
ØØ21| A5 ØØ                 #       LDA     RETURN
ØØ23| 48                   #       PHA
ØØ24| 6Ø                          RTS                     ;RETURN to Pascal
ØØ25|
ØØ25|                             .END                    ;end of assembly
PAGE -   5  INCARRAY  FILE:ASMSUBS  SYMBOLTABLE DUMP


AB - Absolute     LB - Label      UD - Undefined     MC - Macro
RF - Ref          DF - Def        PR - Proc          FC - Func
PB - Public       PV - Private    CS - Consts


ALOOP    LB ØØ12|  INCARRAY PR ----|  POP      MC ----|  PSEUDO   AB ØØEØ|
PUSH     MC ----|  RETURN   AB ØØØØ|  SIZE     AB ØØØ2

PAGE -   6  INCARRAY  FILE:ASMSUBS


Current minimum space is 9571 words

Assembly complete:       94 lines
     Ø   Errors flagged on this Assembly
```

Only the assembled object code shown on the left side of the assembly listing is saved in the file /MYDISK/ASMSUBS.CODE .

Notes about the sample assembly listing:

1) The addresses given in the Symboltable dump correspond to the addresses shown in the left-most column of the listing.

2) Addresses in the object code appear in reverse byte order; that is, low byte first.

3) A number-sign (#) is printed at the left of all source statements that are expanded from macros.

4) The notation used for indirect addressing with the Apple III Pascal Assembler is not the same as the standard notation defined by the manufacturer of the 65Ø2 microprocessor. See the section on Addressing Modes under Syntax of Assembly Language Statements, later in this chapter.

A Pascal Host Program

The following sample Pascal host program calls the external function and procedure assembled earlier. You should use the Editor to type the program as shown, then save it under the pathname /MYDISK/CALLASM.TEXT . Note: the first line of the program directs the compiler to save the listing file as /MYDISK/CALL.LST.TEXT .

```
{$L /MYDISK/CALL.LST.TEXT}

program CALLASM;

{       sample Pascal host program with calls       }
{       to an external function and procedure       }


type list = packed array[Ø..9] of Ø..255;

var i,k: integer; aa: list;

procedure INCARRAY(size:integer; var data: list);
external;

function TIMES2(data:integer):integer;
external;
```

```
begin
   writeln('initial array:');
   for i := Ø to 9 do
      begin
      aa[i] := i;
      write(aa[i],' ');
      end;
   writeln;
   writeln('array, incremented:');
   INCARRAY(1Ø,aa);
   for i := Ø to 9 do write(aa[i],' ');
   writeln;
   writeln('incremented array, times 2:');
   for i := Ø to 9 do write(TIMES2(aa[i]),' ');
end.
```

Using the Host Program

Before the Pascal host program you have just typed can be used, the text version must be compiled to make an executable P-code version. This is done from the Command level by typing C for Compile. If you saved the program text in a file named /MYDISK/CALLASM.TEXT , the first prompting messages and your responses will look like this:

```
Compiling...
Compile what text?  /MYDISK/CALLASM
To what codefile?  $
```

The first response tells the Compiler to compile the program in the textfile /MYDISK/CALLASM.TEXT . The dollar sign ($) response to the second prompt tells the Compiler to save the resulting codefile with the same pathname as the textfile, except for the suffix: /MYDISK/CALLASM.CODE . Just as in the Assembler example, it is convenient to use the same name for the textfile and codefile versions of the program.

Now the actual compilation begins. The Compiler displays a dot for each line of the source program, and messages that tell you how much memory space, in 16-bit words, is available at each stage of the compilation. For a description of the Compiler messages, refer to the Apple III Pascal Programmer's Manual.

If there are no errors in the program, the Command prompt line will reappear. When you reach this point, compilation of

CALLASM is complete and the compiled codefile is stored as /MYDISK/CALLASM.CODE .

However, CALLASM is still not ready to execute: the external assembly language function and procedure in /MYDISK/ASMSUBS.CODE still have to be linked to the Pascal program. To do this, type an L for Link. The system's messages and your responses will make a dialog like the one shown below. For each prompting message from the system, the response you should make is given below. An explanation of each response is shown in parentheses.

```
Linking...

Apple /// Pascal Linker [A3/1.Ø]
Host file?  /MYDISK/CALLASM     (Host program codefile)
Opening /MYDISK/CALLASM.CODE
Lib file?  /MYDISK/ASMSUBS           (Routines to link)
Opening /MYDISK/ASMSUBS.CODE
Lib file?           (Press RETURN key--no more to link)
Map file?               (Press RETURN key--no map file)
Reading CALLASM
Reading TIMES2
Output file?  /MYDISK/SAMPL.CODE  (Executable codefile;
                                      type suffix .CODE)
Linking CALLASM # 1
     Copying func TIMES2
     Copying proc INCARRAY
```

The file SAMPL.CODE now contains your compiled Pascal host program CALLASM linked with the assembly language routines TIMES2 and INCARRAY. The completed program is now an executable codefile. If you type X for Execute, the Apple III will display:

```
Execute what file?
```

If you type /MYDISK/SAMPL, the program you just linked will be loaded into the Apple and executed, to produce this display:

```
initial array:
Ø 1 2 3 4 5 6 7 8 9
array, incremented:
1 2 3 4 5 6 7 8 9 1Ø
incremented array, times 2:
2 4 6 8 1Ø 12 14 16 18 2Ø
```

Assembler Information

This section defines the syntax of assembly language source files and of the statements they contain. It does not describe programming techniques, but only the way the program must be written for the Apple III Pascal Assembler to assemble it.

Syntax of Assembler Source Files

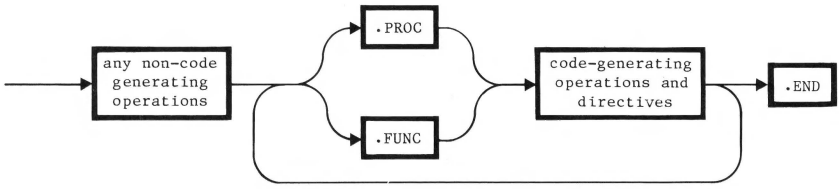
The routines you assemble with the Pascal Assembler will be external procedures and functions used with Pascal programs. Statements that do not generate actual instruction code can also occur outside the body of a procedure or function. These statements can be of two kinds: (1) statements in macro definitions, and (2) statements using any of the following assembler directives:

.EQU	.MACRO	.IF	.DEF	.LIST	.MACROLIST
.ABSOLUTE	.ENDM	.ELSE	.REF	.NOLIST	.NOMACROLIST
.INTERP		.ENDC		.PAGE	.PATCHLIST
				.TITLE	.NOPATCHLIST

All symbols defined before the first procedure or function stay in the symbol table throughout the assembly, so they can be referred to from any of the routines in the source file. When each new procedure or function begins, all symbols are removed from the symbol table except the ones that were defined before the first procedure or function.

The statements making up the body of a procedure or a function are preceded by a .PROC or a .FUNC statement. Each procedure or function ends at the occurrence of the next .PROC or .FUNC statement, except the last one. The last procedure or function in the source file must end with an .END statement, which terminates the assembly. All text beyond the .END statement is ignored by the Assembler.

Here is a general syntax diagram for assembly files:



ASSEMBLY-FILE SYNTAX

Syntax of Assembly Language Statements

Each assembly language statement occupies one line and contains four fields, arranged like this:

Label Operation Operand Comment

The fields are separated by one or more spaces or tabs. The normal practice is to use tabs so that the fields line up as columns on the listings, making them easier to read.

The label field can be occupied by an identifier or a local label, or it can be blank. The operation field is occupied either by an instruction mnemonic (op-code), by an assembler directive (pseudo-op) or by a macro identifier. The operand field is occupied by the arguments of the instruction or directive in the operation field. These can be expressions, identifiers, character strings, or other kinds of arguments, depending on the operation. In some cases, the operand field is blank.

The comment field contains text that is ignored by the Assembler. The comment field starts with a semicolon (;). A comment normally follows the other fields in a statement, but if a semicolon is the first non-blank character in a statement, the entire statement is treated as a comment.

Identifiers

An identifier is a character string starting with a letter. The subsequent characters can be letters, numbers, or the ASCII underline (_). Only the first eight characters (not counting any underlines) are actually used by the Assembler, although more can be typed in the identifier.

The underline character is ignored by the Pascal Compiler and Assembler. If you declare a procedure as EXT_PROC , it is just as if you had declared it EXTPROC or E_X_T_PROC .

The Assembler makes only one pass through the source. On encountering an undefined identifier in an expression, the assembler treats the identifier as an undefined label that will eventually be defined. Any identifier other than a label must be defined before it is used.

Labels

A label must begin in the first column, with no preceding spaces. A label can be followed by a colon; the colon will be ignored.

Using the Equate directive (.EQU), a label can be defined by an expression containing other labels or absolutes. A label that appears as the argument of an Equate directive can be undefined, but the undefined label cannot then be defined by a later Equate directive.

Local Labels

A local label has a dollar sign ($) as its first character, followed by as many as eight digits. A local label cannot be used on the left-hand side of an equate.

Local labels are mainly used to jump around within a small segment of code without using up space in the symbol table that will be needed for regular labels. The Assembler's local-label

table can hold up to 21 labels. The local-label table is emptied each time a regular label is encountered, thus making all local labels previously defined invalid beyond that point in the assembly. An example of the use of local labels is shown below, where the branch to label $Ø4 is made invalid by the intervening regular label REALLAB .

```
$Ø3         STA  4          ;LEGAL USE OF LOCAL LABEL
            .
            .
            BNE  $Ø3
            .
            .
            BNE  $Ø4        ;ILLEGAL USE OF LOCAL LABEL
REALLAB     LDA  #1
$Ø4         TAX
```

Constants

A constant must start with an integer from Ø through 9. For example, the hexadecimal constant FF must be written ØFF .

The default number base is hexadecimal. A decimal constant consists of a number followed by a period (decimal point).

```
EXAMPLE:          Hexadecimal:   13
                      Decimal:   19.
```

Remember that the default number base is hexadecimal.

Location Counter

The Assembler recognizes the asterisk (*) in the operand field as a reference to the current value of the location counter. For example, the statement

```
LOOP     JMP  *
```

would be assembled as a jump to the location of the jump itself; in other words, an infinite loop or virtual halt.

Addressing Modes

The value of an expression appearing in the operand field of an instruction is used either as immediate data, as an address, or as an indirect address. The default case is to use it as an address.

As usual with the 65Ø2, immediate data is indicated by a number sign (#) before an expression. For example,

```
LDA #5
```

is assembled as a load-accumulator immediate with data Ø5.

The notation used in the sample program for the indirect and indexed addressing modes is not the same as the standard notation defined by the manufacturer of the 65Ø2 microprocessor. (The Assembler will accept the standard notation, but note the warning below.)

An at-sign (@) before an expression appearing in the operand field causes the value of the expression to be used as an indirect address reference. For example, the instruction

```
LDA   DATADDR
```

would be assembled as a load-accumulator absolute that would load the byte of data at address DATADDR, but the instruction

```
LDA   @DATADDR,Y
```

would be assembled as a load-accumulator indirect, indexed by index register Y, that would use the word of data at address DATADDR as an address. The 65Ø2 would add the contents of the Y index register to the contents of the word found at DATADDR and load the accumulator with the byte of data found at the resulting address.

Notations for indirect and indexed addressing modes are shown in the following table:

Addressing Mode	Apple III Pascal Assembler Format	Standard 65Ø2 Assembler Format
Indirect	JMP @GOVECT	JMP (GOVECT)
X-indirect	LDA @LOC2,X	LDA (LOC2,X)
Y-indirect	LDA @LOC1,Y	LDA (LOC1),Y

The Apple III Pascal Assembler also accepts the standard notation for the indirect and indexed addressing modes, but it does not check the placement of the parentheses. It distinguishes between indirect indexed and indexed indirect modes by the presence of the X or the Y.

Expressions

The following operators can appear in expressions:

Unary operators:

+	positive
-	negative
~	ones complement

Binary operators:

+	addition
-	subtraction
*	multiplication
/	truncating division (DIV)
%	remainder division (MOD)
|	bit-by-bit OR
^	bit-by-bit exclusive OR
&	bit-by-bit AND
=	equal (valid only with .IF)
<>	not equal (valid only with .IF)

Expressions are evaluated from left to right, and all operators have the same precedence. To override the normal left-to-right precedence, use angle brackets <like this> around the part of the expression to be evaluated first. It is possible to create bracketed expressions too complex for the Assembler to evaluate, but most such expressions are too long to fit onto one line anyway.

Normally, a label can be used in an address expression such as

```
LDA  LABEL+5      ; Legal expression with label
```

only if the expression adds or subtracts a constant value from the address of the label. An expression such as

```
LDA  LABEL*2      ; Illegal expression with label
```

will not be accepted by the Assembler unless you are assembling using the .ABSOLUTE directive (discussed later in this chapter) and LABEL is previously defined. Likewise, a label must not appear in an expression used to make an absolute constant unless the label is absolute. A statement such as

```
LDA #LABEL+5      ; Illegal absolute constant with label
```

will not be accepted by the Assembler unless the .ABSOLUTE directive is in use and LABEL is previously defined.

The following portion of an assembled listing illustrates expression syntax as used in the Assembler. The examples are not an actual, useful program.

```
 PAGE - 1 TEMP1 FILE:EXPRSYNTAX

ØØØØ|                                .PROC TEMP1
Current memory available:   1ØØ88
ØØØØ|                                        ; CONSTANTS
ØØØØ|
ØØØØ| ØØØA                 CON1Ø    .EQU  1Ø.
ØØØØ| ØØBF                 OTHØ     .EQU  ØBF
ØØØØ| ØØF7                 ONEØ     .EQU  ØF7
ØØØØ|
ØØØØ|                                      ; example EXPRESSIONS
ØØØØ|
ØØØØ| A5 Ø5                          LDA  5
ØØØ2| A5 4D                          LDA  5+6*7
ØØØ4| A5 4D                          LDA  <5+6>*7
ØØØ6| A5 ØA                          LDA  7*6/4
ØØØ8| A5 Ø7                          LDA  7*<6/4>
ØØØA| A5 Ø1                          LDA  6%5
ØØØC| A5 Ø2                          LDA  5+11%5
ØØØE| A5 Ø7                 LABEL    LDA  5+<11%5>
ØØ1Ø| A5 48                          LDA  OTHØ^ONEØ
ØØ12| A5 B7                          LDA  OTHØ&ONEØ
ØØ14| AD ØEØØ                        LDA  LABEL
```

```
ØØ17| AD Ø9ØØ                    LDA  LABEL-5
ØØ1A| AD 4ØØØ                    LDA  LABEL+<5*CON1Ø>

        LDA  LABEL*2
ill-formed expression
E(dit,<space>,<esc>                [ Spacebar pressed here,
                                    to continue assembly. ]

ØØ1D|                            LDA  LABEL*2
ØØ1D| A9 Ø5                      LDA  #5
ØØ1F| A9 1C                      LDA  #5*<CON1Ø / 2> +3

        LDA  #<LABEL+5>
operand not absolute
E(dit,<space>,<esc>                [ Spacebar pressed here,
                                    to continue assembly. ]

ØØ21| A9                        LDA  #<LABEL+5>

        LDA  #LABEL
operand not absolute
E(dit,<space>,<esc>                [ Spacebar pressed here,
                                    to continue assembly. ]

ØØ22| A9                         LDA  #LABEL
ØØ23|
ØØ23|                           .END
```

Linkage to Assembly Language Routines

A routine is declared EXTERNAL in a Pascal host program in much the same way that a Pascal routine is declared FORWARD. The routine is declared by a standard PROCEDURE or FUNCTION heading followed by the keyword EXTERNAL. Calls to the external routine use standard Pascal syntax, and the Compiler checks that each call agrees in type and number of parameters with the declaration for that routine. It is the programmer's responsibility to ensure that the assembly language routine agrees with the host program's EXTERNAL declaration. The Linker checks only for the same number of words of parameters in the host program's EXTERNAL declaration and in the external routine's .PROC or .FUNC declaration. For more information on the Linker's functions, see this manual's chapter THE LINKER.

When the host program executes a call to an external procedure or function, parameters to be passed are pushed on the evaluation stack in the order they are encountered in the host program's calling statement: the first parameter is pushed on the stack, high byte first, then the second parameter, and so on. Long integers and sets are passed as the number of words used in the host program. After a long integer or set, a word indicating the number of words passed is pushed onto the stack. Again, each word is pushed on the stack high byte first. Strings, records, arrays, and VAR parameters are passed by address, high byte first. The host program's EXTERNAL declaration may declare a VAR parameter without a type. This allows a parameter of indeterminate size to be passed by address. When all the parameters have been passed, the host program's return address is pushed on the stack, high byte first, then low byte.

The assembly language routine being called must save the return address, and then push it back on the stack just before returning to the calling program. The passed parameters are available on the stack in reverse order: the last one passed is at the top of the stack.

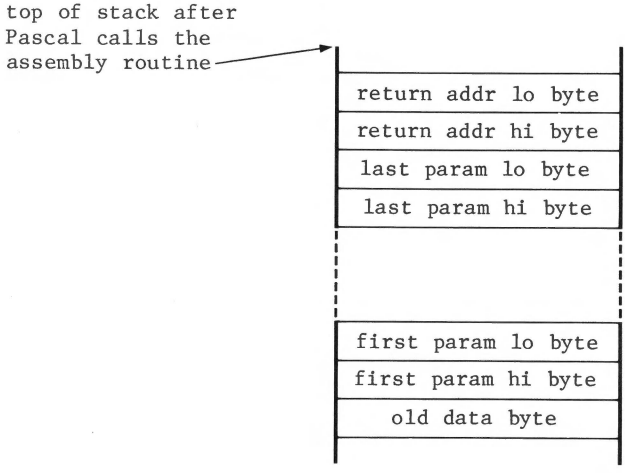


Figure 2-1. The Order of Parameters on the Stack

The TIMES2 function in the assembly language example earlier in this chapter uses parameter-passing by value. The function first removes the return address from the stack and saves it in location RETURN. After discarding the four extra bytes added to the stack because the host program was calling a function, the function then picks up the data word, one byte at a time. When it is finished, the function pushes the result back onto the stack, followed by the return address.

Conventions

When you write assembly language routines for the Apple III, you must respect the SOS conventions concerning register use and calling sequences. All the 65Ø2 registers are available, and zero-page hexadecimal locations Ø through 35 are available for storing temporary variables. However, the Apple III Pascal System also uses these locations as temporaries, so you should not expect data to remain there from one execution of a routine to the next. You can save variables in non-zero-page memory by using the .BYTE or .WORD directives to reserve space in your assembly language routine.

There are two Pascal conventions that apply only to functions:

1) When a function is called, the host program pushes two words (four bytes) of zeros on the evaluation stack after any parameters are pushed and before the return address is pushed.

2) When a function is finished, it must push the result (a scalar, real, or pointer, maximum two words) on the stack, high byte first, just before it pushes the return address.

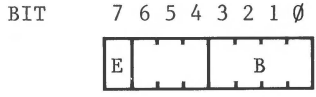
Enhanced Indirect Addressing

Enhanced indirect addressing is the method used in the Apple III to extend its memory addressing beyond 64K bytes. It involves the use of 65Ø2 indirect-X and indirect-Y addressing modes and depends on hardware interaction between the zero page and its corresponding extension page (X-page).

SOS permanently assigns locations $1AØØ through $1AFF as the user zero page, and the hardware automatically associates locations $16ØØ through $16FF as the X-page. Although the

zero-page data actually resides at $1AØØ through $1AFF, instructions that refer to the zero page still use address values $ØØ through $FF. Most Apple III instructions behave exactly like their 65Ø2 counterparts, except for indirect-X and indirect-Y instructions. Depending on the values in the X-page, these instructions can invoke enhanced indirect addressing.

Consider an indirect-X or indirect-Y reference through zero-page location n. As usual for the 65Ø2, zero-page locations n and n+1 are expected to contain the operand address (disregarding indexing by X or Y, for the moment). Since the zero page is mapped, this operand address is actually stored at locations $1AØØ+n and $1AØ1+n, with the least-significant byte at the lower address. Location $16Ø1+n contains the X-byte for this addressing operation. The X-byte is interpreted as follows:



Bit 7 is the enhanced-addressing bit, or E-bit. If it is zero, normal 65Ø2 addressing takes place and the rest of the X-byte is ignored. Normal 65Ø2 addressing means addressing in the 64K address space consisting of a lower 8K portion followed by the currently switched-in 32K bank followed by an upper 24K portion. Of course, the normal user should not access the lower 8K or the upper 24K, because these are occupied by the operating system.

If bit 7 of the X-byte is a one, enhanced indirect addressing takes place. The four-bit B field specifies a bank pair consisting of banks B and B+1. These two banks together make up a contiguous 64K address space. The address word stored in zero page is taken as the address of a location in this 64K bank pair, regardless of which bank is currently switched in.

The user should observe two warnings regarding enhanced indirect addressing. First, locations $ØØØØ to $ØØFF (the "zeroth" page) in each bank pair are actually mapped into the current user zero page (locations $1AØØ to $1AFF, in the portion of memory below the bank space). These locations should be addressed using ordinary zero-page addressing.

The second warning concerns SOS calls. Parameter addresses in SOS calls should not refer to locations $Ø1ØØ through $Ø1FF (page 1) of bank pair Ø-1.

Pascal Memory Usage

The Pascal system uses Apple III memory as follows:

(1) The areas above and below bank space are completely used up by SOS and the Pascal interpreter.

(2) The high end of the highest-numbered bank available is used by the interpreter, the SOS device drivers, and an interpreter buffer.

(3) If graphics are in use, Pascal reserves either 8K, 16K, or 32K in Bank Ø for graphics buffers. This space is accessible via the graphics driver or assembly language subroutines but not directly from Pascal programs.

(4) Of the space that remains, the Pascal system allocates the largest available contiguous block of memory up to 64K bytes long for use as the Pascal data space (stack/heap space). This space must reside in a single bank pair. All Pascal data resides in this space, and Pascal code segments are loaded here if there is no other space for them.

(5) If there is any memory space left over after all of the above allocations are made, the Pascal system uses it to hold code segments as they are dynamically loaded. If there is no left over space or if it has already been filled with code segments when space is needed to load another code segment, the new segment will be loaded in Pascal data space (see (4), above).

Accessing Pascal Data Storage

To access Pascal data space, an assembly language routine must perform indirect-X or indirect-Y addressing using an appropriate X-byte value. For example, if Pascal data space is in bank 1-2, the appropriate X-byte is $81. The high-order bit set to one enables enhanced indirect addressing and the low-order four bits specify the bank pair 1-2.

Since the Pascal subroutine linkage mechanism only passes two-byte addresses (the X-byte is excluded), the assembly

language writer must make sure the X-byte is set properly. The Pascal system presets locations $16E1, $16E3, $16E5, and so on through $16EF to the X-byte value for Pascal data space at boot time. Thus, assembly language routines can copy parameter addresses into locations $EØ-$E1, $E2-$E3, and so on through $EE-$EF and perform indirect-X or indirect-Y addressing with these zero-page addresses to access the parameters in Pascal data space.

The following example shows how to access .PUBLIC data by using this approach:

```
        .PROC    TEST
DATR    .EQU     ØEØ      ;first Pascal pseudo-register
        .PUBLIC  DATA     ;data belongs to the host
         LDA     ADATA    ;move address...
         STA     DATR
         LDA     ADATA+1
         STA     DATR+1   ;...into pseudo-register

         LDY     #Ø
         LDA     @DATR,Y  ;load the DATA into the accumulator

         LDY     #1Ø      ;if DATA = PACKED ARRAY[Ø...2Ø] OF CHAR,
         LDA     @DATR,Y  ;this loads DATA[1Ø]

ADATA   .WORD    DATA     ;the host's address of DATA
```

Enhanced indirect addressing also occurs in the assembly language example given earlier. The INCARRAY procedure pulls the return address from the stack and saves it at location RETURN. It then pulls the address of the array from the stack and stores it in the pseudo-register at hex location ØØEØ. After getting the remaining parameter from the stack, the procedure uses enhanced indirect addressing (indirect-Y addressing) to modify the array data where it is stored in memory.

All parameters that are passed by address must be accessed by means of enhanced indirect addressing.

The example given earlier in this chapter includes an external assembly language procedure, an external assembly language function, and a Pascal host program that calls them. The example demonstrates the way return addresses, passed parameters, and a function's returned value are handled in assembly language routines. In the example, the external routines are linked into the Pascal host program by direct user commands. For information about installing a routine into the system library so that it can be linked automatically, see this manual's chapter USING THE LIBRARY.

The Assembler Directives

Assembler directives are statements you put in your program to cause certain operations to be performed during program assembly. Assembler directives resemble machine instructions in appearance but, unlike instructions, they do not get assembled into corresponding op-codes, so they are sometimes called pseudo-ops. To make them easier to distinguish from instructions in program listings, all assembler directives begin with a period.

The Apple III Pascal Assembler directives described below are those of the UCSD Adaptable Assembler. They are different from the directives used by the various 65Ø2 microprocessor manufacturers.

In the descriptions that follow, punctuation marks and items that appear in upper case are to be typed in just as they appear. Items that appear in lower case are the names of element types, which you replace with the appropriate elements when you use the directives. Items that are enclosed in angle brackets <like this> are required elements, which you must supply. Items enclosed in square brackets [like this] are optional elements, which you may supply. If an element type is not shown with a particular directive, the element type is not used with that directive.

EXAMPLE:

```
[label]   .ASCII   "<character string>"
```

This notation indicates that you may supply a label, though you

don't need to, and that between the required double quotation marks you must supply the character string. You should not type the brackets.

Some of the common types of elements are defined in the following table.

Type:	Definition:
value	Any numerical value, label, constant, or expression.
valuelist	A list of one or more values separated by commas.
identifierlist	A list of one or more identifiers separated by commas.
expression	Any legal expression as defined under Syntax of Assembly Language Statements.
identifier[:integer]list	A list of one or more identifier:integer pairs separated by commas. The colon-integer is optional in each pair; the default value of integer is 1.

Examples are included after each directive definition to show you the specific syntax and form of the directive. Also, the example assembly language routine earlier in this chapter includes some assembler directives in operation.

Routine-Delimiting Directives

Every assembler source textfile must include at least one .PROC statement or .FUNC statement and one .END statement. The .PROC and .FUNC statements declare and delimit the procedures and functions that will be called by a Pascal host program. The .END statement appears at the end of the last routine and serves as the final delimiter.

The Pascal host program refers to an assembly language routine by means of an EXTERNAL declaration. At the time a routine is declared, the actual parameter names are given. For example, for the assembly language procedure that begins with the statement

```
.PROC FARKLE,4
```

the declaration in the Pascal host program might be

```
PROCEDURE FARKLE(X,Y:REAL);
EXTERNAL;
```

The use of these directives is demonstrated in the example given earlier in this chapter.

.PROC

Identifies a procedure, which returns no value. A procedure is terminated by the occurrence of a new .PROC or .FUNC statement, or by an .END statement.

FORM: .PROC <identifier>[,expression]

[expression] indicates the number of words of parameters expected in calls to this routine. The default value is Ø.

EXAMPLE: .PROC DLDRIVE,2

.FUNC

Identifies a function, which returns a value. The host program pushes two words onto the stack before it pushes the return address. A function is terminated by the occurrence of a new .PROC or .FUNC statement, or by an .END statement.

FORM: .FUNC <identifier>[,expression]

[expression] indicates the number of words of parameters expected in calls to this routine. The default value is Ø.

EXAMPLE: .FUNC RANDOM,4

.END

Indicates the end of the last routine in an assembly language source file.

FORM: .END

EXAMPLE: .END

Data Directives

The next four assembler directives are for inserting data into the stream of code being generated by the assembler. The .BYTE , .WORD , and .BLOCK directives can also be used to allocate space for storing variables.

.ASCII

Converts character values to ASCII-equivalent byte constants and places them into the code stream. If a label is present, its value is the address of the first byte stored.

FORM: [label] .ASCII "<character string>"

where the character string is any string of printable ASCII characters, including spaces. The length of the string must be less than 8Ø characters. The double quotation marks are used as delimiters for the characters to be converted. If you want to put a double quotation mark into the string, you must insert it separately, using the .BYTE directive, as shown below.

EXAMPLE: .ASCII "HELLO"

For the insertion of a string containing a double quotation mark, such as AB"CD, use the following technique:

```
.ASCII    "AB"
.BYTE      22      ; An ASCII  "
.ASCII    "CD"
```

Note: The 22 is the hexadecimal ASCII code for a double quotation mark.

.BYTE

Generates a byte of data in the code stream for each value in the list. Only the first 7 items in the list appear in the assembly listing, even though all of the items are assembled into the code stream. Each value must be between -128 and +255. If the value is outside this range an error will be flagged. If a label is present, its value is the address of the first byte stored.

FORM: [label] .BYTE [valuelist]

the default for no stated value is Ø.

EXAMPLE: TEMP .BYTE 4

the associated output would be: Ø4

.BLOCK

Generates a block of repeated data in the code stream. The length of the block is in bytes, and each byte in the block has the same value. If no value is specified, the default is Ø. If a label is present, its value is the address of the first byte in the block.

FORM: [label] .BLOCK <length>[,value]

EXAMPLE: TEMP .BLOCK 4,6

the associated output would be:

```
Ø6    (four bytes, each with value Ø6)
Ø6
Ø6
Ø6
```

.WORD

Generates a two-byte word of data in the code stream for each value in the valuelist. Assigns the label the value of the address of the first word.

FORM: [label] .WORD [valuelist]

EXAMPLE: TEMP .WORD Ø,2,4,...

the associated output would be:

```
ØØØØ
ØØØ2
ØØØ4
 .
 .
 .
```

```
EXAMPLE:    A1    .WORD  A2
                    .
                    .
                    .
            A2    .EQU   *     ; * denotes L.C. value
                  .WORD  5.
                    .
```

The statement A2 .EQU * assigns the current value of the assembler's location counter (L.C.) to label A2 . If the value of the location counter is 5Ø at the .EQU , the associated output would be:

```
ØØ5Ø  (assignment of the value of A2)
ØØØ5  (assignment due to .WORD 5)
```

Label-Definition Directives

The next four directives control the definitions of the labels used in your program.

.EQU

Assigns a value to a label. A label can be equated to an expression containing labels or absolutes, but the labels must already be defined. A local label can neither appear in nor be defined by an .EQU .

```
FORM:     <label>  .EQU     <value>

EXAMPLE:    BASE   .EQU     LABEL6
```

.ORG

Sets the location counter to the value of the operand. Words or bytes of zeros are generated in the code stream to fill the space between the old and new positions of the location counter. If the new value is less than the current location counter, an error will be generated.

```
FORM:               .ORG     <value>

EXAMPLE:            .ORG     ØDØØØ
```

.ALIGN Sets the location counter to the next higher address that is an even multiple of the value of the operand.

FORM: <label> .ALIGN <value>

EXAMPLE: PAGE .ALIGN Ø1ØØ

.ABSOLUTE This directive forces the assembler to interpret the arguments of all .ORG directives as absolute memory locations. Since the use of .ABSOLUTE has the effect of cancelling the generation of relocation information, the resulting object code cannot be linked to a Pascal host. Such an object file must be loaded by the user. It also makes it possible to treat any defined (i.e., non-forward-referenced) labels as absolute numbers. Thus such labels may be multiplied and divided, etc. The .ABSOLUTE directive must occur before the first .PROC or .FUNC directive and is in effect for the entire assembly.

FORM: .ABSOLUTE

EXAMPLE: .ABSOLUTE

.INTERP Used in expressions to specify locations relative to the beginning of a special table in the interpreter. Interpreter-relative labels may be defined as shown in the example. The rules regarding the use of such labels are the same as for any other specially defined labels (e.g., .PUBLIC and .PRIVATE).

EXAMPLE: STUFF .EQU .INTERP+25

Certain interpreter entry points may be accessed by means of an instruction such as

JMP @.INTERP+n

For more information on interpreter entry points, refer to the appendix SPECIAL MEMORY LOCATIONS.

Macro Directives

A macro is a named block of statements. After it is defined, it can be inserted into the text wherever it is needed simply by using its name as an operator. The text of the macro can include parameters so that each insertion results in a different version of the macro statements. A macro whose definition precedes the first .PROC or .FUNC statement in an assembly language source file can be used in any of the procedures or functions in the file.

A macro is invoked by using its name as an operator. The Assembler inserts the text of the macro definition into the program immediately after the statement that invokes it. A statement that invokes a macro can have a list of up to nine arguments, separated by commas, in its operand field. Each time it is invoked, the macro text is modified by substituting the arguments for the macro parameters. If n is a single decimal digit greater than zero, the n-th invocation argument is substituted wherever the parameter %n occurs in the macro definition. If a particular invocation provides fewer arguments than there are parameters in the macro's definition, a null string is substituted for each missing argument.

A macro definition cannot contain another macro definition. However, a definition can include other macro invocations. The nesting of macro invocations can be up to five levels deep.

You can put a macro definition in either the main file or in an INCLUDE file, but the macro definition must be completely contained within one text file. It is illegal to start a macro definition in the main source file and continue it into an INCLUDE file, or to start the definition in one INCLUDE file and continue it in another INCLUDE file.

Each time a macro is invoked, the macro text will appear in the listing file unless .NOMACROLIST was in effect when the macro was defined. Macro expansion text is flagged in the listing by a number sign (#) at the left of each macro statement. Comments occurring in the macro definition are not repeated in the expansion.

.MACRO Indicates the start of a macro definition and gives it an identifier.

.ENDM Indicates the end of a macro definition.

```
FORM:           .MACRO     <identifier>
                  .
                  .                    ; (macro body)
                  .
                .ENDM

EXAMPLE:        .MACRO HELP
                STA    %1            ; ( comment )
                LDA    %2            ; ( comment )
                .ENDM
```

The assembly listing beginning at the point where this macro is invoked might look like this:

```
                HELP   ALPHA,BETA
     #          STA    ALPHA
     #          LDA    BETA
```

The statement HELP invokes the defined macro using two arguments, ALPHA and BETA. These arguments are used in forming the macro expansion (flagged in the listing by number signs) that follows the invoking statement. In the expansion, the first macro-invocation argument (variable ALPHA) is substituted for the definition's parameter %1 , and the second argument (variable BETA) is substituted for parameter %2 .

The following portion of an assembly listing illustrates the syntax used when defining and invoking macros. The procedure itself is not meant to be an actual, useful program. Other examples of macros occur in the program example given in the first part of this chapter.

```
 PAGE -   1  TEMP2      FILE:MACROCALL

ØØØØ|                              .PROC TEMP2
Current memory available:   1ØØ88
ØØØØ|                                      ; CONSTANTS
ØØØØ|
ØØØØ| ØØØA              CON1Ø     .EQU  1Ø.
ØØØØ| ØØBF              OTHØ      .EQU  ØBF
ØØØØ| ØØF7              ONEØ      .EQU  ØF7
ØØØØ|
ØØØØ|                                      ; MACRO DEFINITIONS
ØØØØ|
ØØØØ|                              .MACRO M2
ØØØØ|                                CLC
ØØØØ|                                LDA   PREDEFL+%1

ØØØØ|                              .ENDM
ØØØØ|
ØØØØ|                              .MACRO TESTM
ØØØØ|                                JMP  %1
ØØØØ|                                LDA  #5+%2
ØØØØ|                                M2  %2 ; MACRO CALL INSIDE A MACRO DEF'N
ØØØØ|                                LDA  %3
ØØØØ|                                LDA  %4
ØØØØ|                                LDA  %5
ØØØØ|                                JMP  %6
ØØØØ|                              .ENDM
ØØØØ|
ØØØØ| A5 Ø5           PREDEFL   LDA  5  ; A PRE-DEFINED LABEL
ØØØ2|
ØØØ2|                                      ; MACRO CALL WITH ALL PARAMETERS
ØØØ2|                                      ; & NO LEADING OR TRAILING SPACES
ØØØ2|
ØØØ2|                             TESTM PREDEFL,<5*CON1Ø+6>,#55,#6,1,LABEL2
ØØØ2| 4C ØØØØ            #        JMP  PREDEFL
ØØØ5| A9 3D              #        LDA  #5+<5*CON1Ø+6>
ØØØ7|                    #        M2  <5*CON1Ø+6>
ØØØ7| 18                 #        CLC
ØØØ8| AD 38ØØ            #        LDA   PREDEFL+<5*CON1Ø+6>
ØØØB| A9 55              #        LDA  #55
ØØØD| A9 Ø6              #        LDA  #6
ØØØF| A5 Ø1              #        LDA  1
ØØ11| 4C ****            #        JMP  LABEL2
ØØ14|
ØØ14|                             M2   5  ; SIMPLE MACRO CALL
ØØ14| 18                 #        CLC
ØØ15| AD Ø5ØØ            #        LDA   PREDEFL+5
```

```
ØØ18|
ØØ18|                                   ; MACRO CALL WITH NULL PARAMS
ØØ18|                                   ; AND LEADING & TRAILING SPACES
ØØ18|
ØØ18|                          TESTM  ,CON1Ø,,   XX  ,ØFØ, PREDEFL

       JMP
not enough operands
E(dit,<space>,<esc>       [ Spacebar pressed here, to continue assembly. ]

ØØ18|                      #       JMP
ØØ18| A9 ØF               #       LDA  #5+CON1Ø
ØØ1A|                      #       M2   CON1Ø
ØØ1A| 18                   #       CLC
ØØ1B| AD ØAØØ             #       LDA   PREDEFL+CON1Ø

       LDA
ill formed operand
E(dit,<space>,<esc>       [ Spacebar pressed here, to continue assembly. ]

ØØ1E|                      #       LDA
ØØ1E| AD ****              #       LDA  XX
ØØ21| A5 FØ                #       LDA  ØFØ
ØØ23| 4C ØØØØ              #       JMP  PREDEFL
ØØ26|
ØØ26|                           .END
```

Conditional-Assembly Directives

Conditional-Assembly Directives are used to exclude or include selected sections of a source file at the time it is assembled. When the Assembler encounters an .IF directive, it evaluates the expression in its argument. If the expression is false, the Assembler simply discards the text until an .ENDC is reached, unless an .ELSE is encountered. If there is an .ELSE directive between the .IF and .ENDC directives, the text before the .ELSE is assembled if the expression is true. If the expression is false, the text after the .ELSE is assembled. The unassembled section of code will not be included in any listing. Conditional-assembly directives may be nested.

The conditional expression takes one of two forms. The first is the normal arithmetic/logical expression used elsewhere in the Assembler. This type of expression is considered false if it evaluates to zero; otherwise, it is true. The second form of conditional expression is comparison for equality, indicated

by an equal sign (=), or inequality, indicated by less-than greater-than (<>). The objects compared may be strings, characters, or arithmetic/logical expressions.

.IF Identifies the beginning of the conditional text and defines the conditional expression.

.ELSE Identifies the alternate section of text, which is used if the conditional expression is false.

.ENDC Identifies the end of the conditional text.

```
FORM:   [label]  .IF  <expression>
                   .
                   .
                   .
               [ .ELSE ]
                   .
                   .
                   .
                 .ENDC
```

```
EXAMPLE:
            .IF LABEL1-LABEL2  ;Arithmetic expression.
              .                 ; This text assembled
              .                 ; only if subtraction
              .                 ; result is non-zero

            .IF  "%1" ="STUFF"  ;Comparison expression.
              .                 ; This text assembled
              .                 ; if subtraction above
              .                 ; was true and if text
              .                 ; of first parameter
              .                 ; (assuming in macro)
              .                 ; is equal to "STUFF".
              .
            .ENDC               ; End of nested cond.
              .
              .
```

```
.ELSE
  .                    ; This text assembled
  .                    ;if subtraction result
  .                    ; was zero.
  .
.ENDC                  ; Terminates outer
                       ; level of conditional.
```

Host-Communication Directives

The directives .CONST , .PUBLIC , and .PRIVATE enable an assembly language routine to share addresses and data space with the Pascal program that calls it. Data values and locations are referred to by name in both the program and the routine. The Linker picks up and transfers the address values necessary to resolve these external references. Refer to this manual's chapter THE LINKER for further information.

Note that the locations defined using these directives are in the Apple III memory bank allocated to the Pascal program's data. The assembly language routine must use indirect addressing though one of the pseudo-registers to access this data. Refer to the discussion of Extended Addressing in the earlier section, Linkage to Assembly Language Routines.

.CONST

Allows constants that are declared global in the Pascal host program to be accessed by the assembly language routine. Only 16-bit objects can be accessed by means of the .CONST directive.

FORM: .CONST <identifierlist>

EXAMPLE: (see example after .PRIVATE)

.PUBLIC

Allows variables declared global by the Pascal host program to be used by the assembly language routine as well as the host program.

FORM: .PUBLIC <identifierlist>

EXAMPLE: (see example after .PRIVATE)

.PRIVATE Allows variable data used by the assembly language routine to be stored in the Pascal host program's global data segment and yet be inaccessible to the host program. These variables retain their values for the entire execution of the program.

FORM: .PRIVATE <identifier[:integer] list>

Each identifier will be allocated the number of words given by integer. The default is one word.

EXAMPLE: (for .CONST, .PUBLIC, and .PRIVATE)

Assume that the host program is the following Pascal program:

```
PROGRAM EXAMPLE;
CONST SETSIZE=5Ø; LENGTH=8Ø;

VAR I,J,F,HOLD,COUNTER,LDC:INTEGER;
    LST1:ARRAY[Ø..9] OF CHAR;

BEGIN
  .
  .
  .
END.
```

The following statement

```
.CONST     LENGTH
```

occuring in an assembly language routine called by the Pascal host program will allow the constant LENGTH to be used in the assembly language routine almost as if the line

```
LENGTH .EQU  8Ø.
```

had been written. Remember the limitation mentioned above: .CONST identifiers can be used only for 16-bit objects.

If the statements

```
.PRIVATE   PRT,LST2:9
.PUBLIC    LDC,I,J
```

appear in the assembly language routine, the variables LDC, I and J can be used by both the host program and the assembly language routine, while the variables PRT and LST2 can be used only by the assembly language routine. Also, the argument LST2:9 causes the variable LST2 to correspond to the beginning of a nine-word block of space in the Pascal host's global data segment.

External-Reference Directives

Separate assembly language routines can share data structures and subroutines by means of the .DEF and .REF directives. These directives cause the Assembler to generate information that the Linker uses to resolve external references between separate routines in the same assembly or between routines assembled separately. For example, by using these directives, one assembly language routine can call subroutines defined in another assembly language routine.

Note that procedures and functions can refer to identifiers defined before the first procedure or function in the same source file without using .DEF and .REF .

The use of the .DEF and .REF directives is similar to the use of the .PUBLIC directive. The .DEF and .REF directives enable you to associate labels between two assembly language routines rather than between an assembly language routine and a Pascal host program. Just as with .PRIVATE and .PUBLIC , these external references must eventually be resolved by the Linker.

The .PROC and .FUNC directives implicitly generate a .DEF with the same name. This means that an assembly language routine can call external procedures and functions if they are declared with a .REF directive in the calling assembly language routine.

.DEF

Declares that a label defined in the current routine is available for use (by means of .REF) from procedures or functions in other assembly language routines.

FORM: .DEF <identifierlist>

EXAMPLE: The following outline routine declares the labels DOIT and THINK in a .DEF statement. The subroutines labelled DOIT and THINK may then be used by other assembly language routines (see example for .REF).

```
        .PROC FARKLE,3
        .DEF DOIT,THINK
             .
             .
            BNE THINK
             .
DOIT        LDA
             .
            RTS
             .
THINK       LDY
             .
            RTS
             .
            .END
```

.REF

Identifies a label, used in the current routine, that is defined and declared as available (by means of a .DEF directive) in another routine. During the linking process, corresponding labels declared in .DEFs and .REFs are matched.

FORM: .REF <identifierlist>

EXAMPLE: The following outline assembly language routine defines the external label DOIT in a .REF statement. (DOIT was declared available for such reference by the .DEF in the previous example). It then uses the label DOIT just as if it referred to a labelled subroutine within the routine itself.

```
        .PROC SAMPLE
        .REF DOIT
          .
          .
```

```
        JSR DOIT
          .
          .
        .END
```

Listing-Control Directives

The listing-control directives determine what is sent to the assembly listing file. This is the file that is specified in response to the Assembler prompt

```
Output file for assembled listing (<CR> for none):
```

If the assembly listing file is omitted, all listing-control directives are ignored.

.LIST and .NOLIST

These two directives allow selective listing of assembly language routines. Statements assembled after a .LIST directive go to the specified listing file. Statements assembled after a .NOLIST directive are not listed. Listing may be turned on and off repeatedly within an assembly. .LIST is the default option.

```
FORM:       .LIST
            .NOLIST
```

.MACROLIST and .NOMACROLIST

Allow selective listing of macro expansions. The textual expansion of a macro will appear in the assembly listing if the .MACROLIST option was in effect when the macro was defined. The expansion text will not appear in the listing if the .NOMACROLIST option was in effect when the macro was defined. These options may be used repeatedly throughout a assembly language source file, to select those macros whose expansion text will appear in the assembly listing. The Assembler defaults to the .MACROLIST option.

Macro expansion text is flagged in the listing by a number sign (#) at the left of each expanded line. Comments in a macro's definition do not appear in the expansion. In the example assembly listing earlier in this chapter, the definition of macro POP appears

on PAGE-Ø ; the macro's expansion text appears on PAGE-1 and PAGE-4 .

When assembling nested macro invocations, listing of expansion text continues until the Assembler encounters the first macro defined with .NOMACROLIST in effect. Listing does not resume until that macro's invocation is complete, regardless of the listing state of the macros invoked by the non-listing macro.

FORM: .MACROLIST
.NOMACROLIST

EXAMPLE: .NOMACROLIST

The .NOLIST option takes precedence over the .MACROLIST option.

.PATCHLIST and .NOPATCHLIST

Allow control over listing of back-patches made to the code file. These options may be used repeatedly throughout an assembly.

When an undefined label is encountered, the assembled listing shows an asterisk (*) for each hexadecimal digit to be filled in later. For example:

```
ØØ19| 1Ø**            BPL DONE
```

When the forward reference is resolved, the back-patch is listed in the form

```
ØØ19* Ø5
ØØ1F| A9 ØØ     DONE  LDA #Ø
```

where the number to the left of the asterisk is the address of the patched location and the number to the right of the asterisk is that location's new value.

.PATCHLIST is the default state.

FORM: .PATCHLIST
.NOPATCHLIST

EXAMPLE: .NOPATCHLIST

.PAGE

Inserts a top-of-form page break in the assembly listing.

FORM: .PAGE

EXAMPLE: .PAGE

.TITLE

Specifies the title to appear at the top of each page of the assembly listing. At the beginning of each routine the title is set to blanks and must be reset if a title is desired for that routine. The title is cleared at the start of the file.

In the example assembly listing earlier in this chapter, the title SYMBOLTABLE DUMP was not set by a .TITLE directive. That title is always used on pages containing reference symbol tables. After each symbol table is listed, the title printed reverts to its previous setting.

FORM: .TITLE "<title>"

where <title> is any string of printable ASCII characters, including spaces. The string must be less than 8Ø characters. The double quotes are used to delimit the string, so a title may not include the double quote character.

EXAMPLE: .TITLE "QRC12 INTERPRETER"

File Directive

.INCLUDE

Causes the specified source file to be included in the assembly immediately after the .INCLUDE .

FORM: .INCLUDE <pathname>

where the pathname specifies an assembly language textfile to be included.

If you don't add the suffix .TEXT , the system will add it for you. The last character of the pathname must be the last non-space character on that line; no comment may follow on the same line.

```
CORRECT EXAMPLE:
     .INCLUDE  /VOL1/SHORTSTART.TEXT

CORRECT EXAMPLE:
     .INCLUDE  /VOL1/SHORTSTART.TEXT
               ; CALLS STARTER

INCORRECT EXAMPLE:
     .INCLUDE  /VOL1/SHORTSTART.TEXT  ; CALLS STARTER
```

The text of any included file is treated by the assembler just as if you had typed that text into the original file at the position of the .INCLUDE directive. For example, if the included file contains an .END directive, the assembly ends there.

A file that is included in an assembly via an .INCLUDE directive cannot itself contain .INCLUDE directives. In other words, you can't nest .INCLUDEs.

Assembler Use Summaries

This section contains summaries of the Assembler commands and the Assembler directives.

Assembler Command Summary

1. From Command level, select Assemble.

2. If a text workfile exists, the Assembler assembles that file automatically. Otherwise, the assembler prompts you to specify a source textfile and then to specify a destination codefile.

3. Finally, the Assembler prompts you to specify an output textfile for the assembly listing, if you want one.

4. If the Assembler finds an error, select the Editor to correct the source file, then assemble again.

Assembler Directive Summary

Square brackets [like this] surround optional elements, which you may supply. Angle brackets <like this> surround required elements, which you must supply. The brackets and the brief definitions at the right side of the table are not to be typed.

Routine-Delimiting Directives

.PROC	<identifier>[,expression]	Begins a procedure.
.FUNC	<identifier>[,expression]	Begins a function.
.END		Ends the entire assembly.

Data Directives

[label]	.ASCII	"<character string>"	Inserts ASCII values of characters.
[label]	.BYTE	[valuelist]	Inserts bytes of listed values.
[label]	.BLOCK	<length>[,value]	Inserts block of given value and length.
[label]	.WORD	[valuelist]	Inserts words of listed values.

Label-Definitions Directives

<label>	.EQU	<value>	Assigns value to label.
	.ORG	<value>	Location of next byte will be (start of assembly file) + value.
	.ABSOLUTE		Causes all .ORGs to put next byte at absolute location = value.

.ALIGN <value>	Increases the location counter to the next whole multiple of value.
.INTERP	First location of interpreter relative location table; used in relative-location expressions.

Macro Directives

.MACRO <identifier>	Begins a macro definition.
.ENDM	Ends a macro definition.

Conditional-Assembly Directives

[label] .IF <expression>	Begins conditional assembly. If true, assembles next text
[.ELSE]	[up to .ELSE]; if false, only assembles text after .ELSE .
.ENDC	Ends conditional assembly.

Host-Communication Directives

.CONST <identifierlist>	Takes value from global constant in Pascal host.
.PUBLIC <identifierlist>	Uses a global variable from the Pascal host.

.PRIVATE <identifier[:integer] list>	Creates a global variable not accessible to the Pascal host. Default: 1 word per identifier.

External-Reference Directives

.DEF <identifierlist>	Makes label available to other routines.
.REF <identifierlist>	Refers to label .DEF'd in some other routine.

Listing-Control Directives

.LIST and .NOLIST	Turn assembly listing on and off.
.MACROLIST and .NOMACROLIST	Turn listing of macro expansions on and off.
.PATCHLIST and .NOPATCHLIST	Turn listing of back-patches on and off.
.PAGE	Puts page-break in listing.
.TITLE "<title>"	Titles each page of current .PROC or .FUNC .

File Directive

.INCLUDE <pathname>	Includes named text file in the assembly.

3

The Linker

Introduction

The Linker provides a way to incorporate separately compiled or assembled routines into your program without having to re-compile or re-assemble them. For example, you might have a real-time application that requires an assembly language routine to obtain the necessary speed. This routine could be assembled separately and then added to your program by using the Linker.

All compiled or assembled codefiles include data that describe external references and entry points. The Linker uses this data to resolve references between separate codefiles. For details about the way Linker data is stored in the codefiles, see the appendix FILE FORMATS in the Apple III Pascal: Introduction, Filer, and Editor manual.

A Pascal program or Unit that calls linked subroutines is called a host program. In order for your host program to use linked routines, the program must declare them as external. This notifies the Compiler that the routines may be called, but have not yet been provided. The Compiler sets a flag in the linker data in the codefile to indicate that linking is required before the program can be executed. The example in this manual's chapter THE ASSEMBLER shows a Pascal host program, a procedure and a function, both in assembly language, and the linking process that combines them into an executable codefile.

You also use the Linker to link in a regular Unit, which is a group of related routines that will be used together. You don't need the Linker to use the Intrinsic Units that are provided with the Apple III Pascal language, such as TRANSCEND and APPLESTUFF ; your Pascal program picks them up directly from SYSTEM.LIBRARY or from the program library. On the other hand, you use the Linker to build Intrinsic Units of your own. For information about Units and Intrinsic Units, refer to the Apple III Pascal Programmer's Manual.

Linking Using the Link Command

You invoke the Linker explicitly by typing L for Link from the Command level. There are two situations in which this is the only way you can use the Linker:

- The host file into which Units or external routines are to be linked is not the codefile resulting from a successful compilation initiated by the Run command.

- Any of the Units or external routines to be linked reside in files other than the Pascal system diskette's SYSTEM.LIBRARY .

Files Needed

The following files must be present for you to use the Linker explicitly:

- SYSTEM.LINKER
- the host codefile needing external routines
- the library codefiles holding the external routines

Any time the Linker is invoked, SYSTEM.LINKER must be available in some drive. This file is normally found on diskette NEWPASCAL3 . After the Linker prompt line appears, SYSTEM.LINKER is no longer needed, so the diskette it is on may be removed from the system to make room for other diskettes.

On a system with three or four disk drives, diskette NEWPASCAL2 is normally your Pascal system diskette during program preparation. If NEWPASCAL2 is in the built-in drive and NEWPASCAL3 is in the second drive, you can put the diskette that contains your program into the third or fourth drive and have available all the diskette files you need to use the Linker.

If you only have two drives, and you want to Link when the host program file and the library file are not already on NEWPASCAL2 or NEWPASCAL3 , you can use the Filer to transfer the needed files onto NEWPASCAL3 before linking. Alternatively, if the Command prompt line is showing, if Linking is your only task, and if all your host and library files are on another diskette such as MYDISK , you can put MYDISK in the built-in drive and NEWPASCAL3 in the external drive. When the linking process is complete, the system will attempt to return to the Command

level. When it does not find the Pascal system diskette in the built-in drive, the system will prompt you to put it in.

The Host File

When you type L at the Command level to invoke the Linker explicitly, the system displays the messages:

```
Linking...

Apple /// Linker [A3/1.Ø]
Host file?
```

The host file is the Pascal program codefile into which the external routines or Units are to be linked. Note that the Linker will not accept an assembly routine codefile as the host file; the host must be a Pascal program codefile.

If you respond to the prompt by pressing the RETURN key, the Linker uses the Pascal system diskette's workfile SYSTEM.WRK.CODE as the host file. If either the Run command or the Compile command has just caused the Compiler to save a compiled codefile, the Linker uses that file as the host file even if it is not SYSTEM.WRK.CODE .

You can also respond by typing the pathname of any other Pascal codefile. If the Linker cannot find a file with the exact pathname you typed and that pathname does not end in .CODE or in .LIBRARY, it adds the suffix .CODE to the pathname and tries again. The Linker always displays the pathname of the last file it tried to find.

The Library Files

After the Linker finds the host file, it asks for the name of a library file that contains needed Pascal Units or external routines. The prompt is:

```
Lib file?
```

You should respond by typing the pathname of any codefile containing a Pascal Unit or external routine that you want linked into the host program. This file can be either a codefile produced by the Compiler or by the Assembler, or a library file created by the Librarian.

The Linker looks first for the exact pathname that you type, then, if the search is unsuccessful, it adds the suffix .CODE and looks again. In any case, it always displays the local filename of the file actually found. When the Linker finds the specified file, it displays the same prompt again and waits for you to type the pathname of another file containing a needed Unit or routine. You can include up to eight library files in one linking operation. If you type an asterisk (*) and then press the RETURN key, the Linker will look for Units or external routines in the file SYSTEM.LIBRARY on the Pascal system diskette.

EXAMPLE:

```
Lib file? *
Opening SYSTEM.LIBRARY
```

If a file you specify as a host file or a library file is not a codefile, the system will display an error message. These files must contain either compiled Pascal P-code or assembled 65Ø2 assembly code. For information on library files and the Librarian see this manual's chapter USING THE LIBRARY.

When you have supplied the names of all the library files needed, respond to the next "Lib file?" prompt by pressing the RETURN key.

The Map File

When you have finished specifying library files, the Linker will prompt with:

```
Map file?
```

The map file is a textfile produced by the Linker. It contains a map or directory of the labels involved in the linking process. If you respond by typing a pathname, the Linker writes the map file with that pathname. You need not type the suffix .TEXT ; if the pathname you type does not end with .TEXT or a period (.), the Linker will add the suffix.

If you respond to this prompt by simply pressing the RETURN key, no map file will be written. The map file is primarily a diagnostic and system programming tool, and is not required for most uses of the Linker. Note: You can get a more useful map of a library or codefile using the Library Mapper described in the chapter THE LIBRARY.

The Output File

After you have specified the disposition of the map file, the Linker reads the files required to start the linking process. Then it asks you:

```
Output file?
```

Type the pathname for the linked output codefile. This pathname will often be the same as that of the host file, but the Linker will not accept the dollar-sign ($) same-name option used with the Compiler and the Assembler. You need not type the suffix .CODE ; the Linker will supply it for you, unless you end the pathname with a period (.).

If you respond with no pathname, by pressing the RETURN key only, the linked output will be saved in the Pascal system diskette's workfile, SYSTEM.WRK.CODE .

After you type the output pathname and press the RETURN key, the actual linking will begin.

If a specified library file is not available in any drive, this message appears:

```
No file <filename>
Type <sp>(continue), <esc>(terminate)
```

Again, if you elect to continue linking without some needed Units or routines, the resulting codefile cannot be executed until you explicitly link them in.

Linking Using the Run Command

If the linking needed in your program is simple enough, you can let the Run command invoke the Linker at the same time you compile and execute your program. Linking is needed if your program contains external declarations or uses Units other than Intrinsic Units. Intrinsic Units needed at execute time are not linked; they can be in either SYSTEM.LIBRARY or the program library, on the same diskette as the program codefile.

If all of the regular Units and external routines to be linked reside in the Pascal system diskette's SYSTEM.LIBRARY , you can

use the Run command to compile, link, and execute your program. Otherwise, you'll have to use the Link command explicitly, as described in the previous section.

Files Needed

The following diskette files must be present if the Linker is invoked by the Run command:

- SYSTEM.PASCAL ;
- The host program needing external routines;
- SYSTEM.LINKER ;
- SYSTEM.LIBRARY containing external routines to link and Intrinsic Units needed at execution time;
- The program library containing Intrinsic Units needed at execution time.

The following files must also be present when you use the Run command, if the condition shown applies:

- SYSTEM.COMPILER, if host program is a textfile;
- SYSTEM.SYNTAX, for Compiler error messages;
- SYSTEM.EDITOR, to fix errors found by Compiler.

These files are supplied on the Apple III Pascal System diskettes labelled PASCAL1 , PASCAL2 , and PASCAL3 . The examples in this section use rearranged system diskettes named NEWPASCAL1 , NEWPASCAL2 , and NEWPASCAL3 . The arrangement of the files on these diskettes is given in the TABLES appendix, and the steps in setting them up are described in the chapter FIRST STEPS IN PROGRAM PREPARATION. Remember that you can rearrange the system files any way you wish; this arrangement is only a suggestion.

The system returns to the Command level for an instant between any two of the system programs invoked by the Run command. If the Pascal system diskette is not in the built-in drive when this happens, the system will prompt you to put it in.

If you use the Run command with a text workfile, the Compiler will be invoked first, so the file SYSTEM.COMPILER must be available. It is normally found on NEWPASCAL3 , but it may be on any diskette in any drive. If the workfile has already been compiled into its code version, the Run command will not invoke the Compiler, so SYSTEM.COMPILER is not needed.

If linking is needed after the program has compiled successfully, the Linker is invoked automatically, so the file SYSTEM.LINKER must be available in some drive. This file is normally found on diskette NEWPASCAL3 .

When the Linker is invoked by the Run command, it automatically uses the codefile that resulted from the latest successful compilation as the host file, even if this file is not the code workfile.

When invoked by the Run command, the Linker automatically looks for needed Units and external routines in the file SYSTEM.LIBRARY . File SYSTEM.LIBRARY must be on the Pascal system diskette (NEWPASCAL2), but the diskette may be in any disk drive. Finally, following successful compilation and linking, the program is executed. At that time, if SYSTEM.LIBRARY is required for execution, it must be on the Pascal system diskette in the built-in drive.

If the file SYSTEM.LIBRARY is not available on the Pascal system diskette, this message appears:

```
No file *SYSTEM.LIBRARY
Type <sp>(continue), <esc>(terminate)
```

When the Linker is used with the Run command, it automatically looks for the file SYSTEM.LIBRARY on the Pascal system diskette. When it is invoked by the Run command, the Linker will not allow you to specify a library file other than SYSTEM.LIBRARY; if you want to use any other library files, you will have to invoke the Linker explicitly. If you get this error message, press the ESC key to go back to Command level.

Linker Command Summary

1. From Command level, select Run or Link.

2. Run automatically links the compiled workfile to Units and routines found in SYSTEM.LIBRARY . Link prompts you to specify a host codefile and then to specify as many library codefiles as needed. Press the RETURN key when you want to stop specifying library files.

3. Next, the Linker prompts you to specify a map textfile for storing Linker information. Normally, you press the RETURN key to go on.

4. Finally, the Linker prompts you to specify an output codefile for the linked program.

4

The Library

What is a Library?

A library is just a codefile containing routines that are used by your program. Intrinsic Units in a library will be picked up automatically when your program is executed. Regular Units and assembly language routines must be linked into your program by using the Linker. For information on the Linker, see this manual's chapter THE LINKER.

Whenever you execute a program that uses Intrinsic Units, the system looks for those units in two files: the program library and the system library. The program library file must be on the same volume as the program codefile; the system library must be on the Pascal system diskette. The system first looks for the needed Units in the program library, then, is they aren't found there, the system looks for them in .D1/SYSTEM.LIBRARY. Thus, if there is a Unit with the same name in both library files, the one in the program library is used.

The program library's name is derived from the name of the program codefile. This means that the program library is used only with its corresponding program. The rules for deriving a program library name from a program codefile name are:

(1) Drop the suffix .CODE, if present;
(2) Truncate the name to 11 characters, if necessary;
(3) Add the suffix .LIB at the end.

For example, if your program is named MYTHING.CODE , the program library file will be named MYTHING.LIB .

If your program uses regular Units and assembly language external routines, the system will not pick up the needed items at execution time. Instead, you must use the Linker to link the required Units and routines into your program codefile. For information about using the Linker, see the chapter THE LINKER in this manual.

The System Librarian

The Librarian is the system program that you use to combine separately compiled or assembled codefiles into a single library file. One way you can use it is to put all of your Pascal Units and assembly language routines into a single convenient library codefile for linking into your programs.

When you use the Run command, the system will automatically find and link needed Units and assembly language routines if they are in the file named SYSTEM.LIBRARY on the Pascal system diskette. The system will also find needed Intrinsic Units that are in SYSTEM.LIBRARY and use them without linking. You use the Librarian to add, change, and delete routines in SYSTEM.LIBRARY .

When you use the Link and Execute commands explicitly, you can use a library file other than SYSTEM.LIBRARY. Using the Librarian, you can set up a library file for the exclusive use of a particular program. This file is called a program library, and it will be used automatically if its name is the same as that of the program except that it ends in .LIB . For more information about Units and Intrinsic Units, see the chapter LIBRARY UNITS in the Apple III Pascal Programmer's Manual.

The Librarian cannot be invoked just by typing a letter from the Command level; instead, you must use the Execute command and specify the Librarian program codefile by name. System programs such as this that are invoked by the Execute command are sometimes referred to as Utility programs.

When you wish to add a new Unit or routine to a library codefile, or to delete one, you must first use the Librarian to create a new, empty library file. Next, you specify each item in the original library file that you want to keep so that the Librarian can copy it into the new library file. You can then add new items by having the Librarian transfer other codefiles into the new library file. After you have created a library you want the system to use automatically, you must either move it to the Pascal system diskette and change its name to SYSTEM.LIBRARY, or move it to the same diskette as your program and change its name to the appropriate program library name.

Files Needed

The following files must be in some disk drives for you to use the Librarian:

- LIBRARY.CODE
- Codefiles containing Units and routines to be put into the new library
- Output file (after the Librarian has started)

The file LIBRARY.CODE is normally found on diskette NEWPASCAL1 , but it may be in any drive. Once the Librarian has started, this file is no longer needed.

A file containing a Unit or routine to be put into the new library is called an Input file. Each Input file only needs to be available on some diskette in some drive during the time it is being loaded. Once the Librarian prompts you for the next Input file, you can remove the diskette containing the previous one.

The Librarian builds the new library in a file called the Output file. Once the Librarian has started, you must leave the diskette containing the Output file in its drive until the new library is complete.

If you only have two disk drives, you will still need to have NEWPASCAL1 in the external drive when you execute the Librarian codefile. After the first Librarian prompt line appears on the screen, you can remove NEWPASCAL1 and NEWPASCAL2 and put in other diskettes as needed. Remember to leave the diskette containing the Output file in its drive until you finish using the Librarian.

When you finish using the Librarian, your Pascal system diskette should be in the built-in drive. If it is not there, the system prompts you to replace it.

Using the Librarian

To invoke the Librarian, the system must be at the Command level and the Librarian program codefile must be in some disk drive. Type X for Execute. The system will prompt you with:

```
Execute what file?
```

You should respond by typing the pathname of the Librarian program codefile:

```
/NEWPASCAL1/LIBRARY
```

You do not need to type the suffix .CODE; the system will append the .CODE suffix automatically. The system executes LIBRARY.CODE, which displays the program identification message and the first prompt:

```
Apple /// Pascal Librarian [A3/1.Ø]


Output file ->
```

At this point, you can remove diskette NEWPASCAL1 from its drive if you need to.

The Output File

You should respond to the Output file prompt with the pathname of your new library file. This pathname is used exactly as you type it; no suffix is added by the system. If you respond to this prompt by typing SYSTEM.LIBRARY, the system will remove the original SYSTEM.LIBRARY file when the Librarian is finished, and replace it with your new library file. Typing an asterisk (*) in response to this prompt is the same as typing SYSTEM.LIBRARY.

The Input Files

The Librarian now displays the prompt:

```
Input File ->
```

You should respond to this prompt by typing the pathname of a library or codefile that contains Units or routines you wish to include in your new library file. If you want to copy Units

or routines from the system library, you should type SYSTEM.LIBRARY in response to this prompt. Typing an asterisk (*) here is the same as typing SYSTEM.LIBRARY.

The Librarian first looks on the specified diskette for a file whose pathname is exactly as you typed it. If there is no file with that exact pathname and that pathname does not end in .CODE, the suffix .CODE is added to the pathname and the search is repeated. If the search is still unsuccessful, the Librarian will display the message

I/O ERROR # 1Ø Type <space> to continue

if your file was not found, or the message

I/O ERROR # 9 Type <space> to continue

if your diskette was not found. After you type the space, the Librarian will prompt you to try again. The only way to escape from the program at this point is by typing a correct file specification or by pressing the RETURN key and then typing A for Abort.

Moving Segments into a Library

There can be up to 16 code or data segments in any Apple III program codefile or library codefile, and the Librarian assigns each one a numbered slot. After you have specified the name of a codefile, the Librarian displays a table that gives the slot number, the segment number (in parentheses), the name, and the length in bytes of each Unit or routine in the file.

An Intrinsic Unit can occupy two slots, one for the code segment and one for the data segment. The segment numbers will already have been assigned: see the chapter PROGRAM SEGMENTATION in the Apple III Pascal Programmer's Manual. The number of bytes given for a segment is its length as stored in the library. For Regular Units and external-procedure segments, this length includes linker data that are not placed in your program, so it is a little larger than the number of bytes the segment will occupy when used in your program.

You should not put more than one Intrinsic Unit with the same segment number into a library.

The slot table for the file you specify will be displayed immediately after the Input codefile prompt. It will look something like this:

```
Input File -> /NEWPASCAL2/SYSTEM.LIBRARY
        Ø-(3Ø) LONGINTI  2452          8-              Ø
        1-(31) PASCALIO  1238          9-              Ø
        2-(29) TRANSCEN  1168         1Ø-              Ø
        3-(22) APPLESTU   662         11-              Ø
        4-                Ø           12-              Ø
        5-                Ø           13-              Ø
        6-                Ø           14-              Ø
        7-                Ø           15-              Ø
```

When the Librarian displays a slot table, it also displays this command line at the top of the screen:

```
Slot to copy and <space>, = for all, ? for Select, N(ew file, Q(uit, A(bort
```

This command line shows the ways you can specify the slot or slots containing segments that you wish to include in the new library you are creating. To specify a particular segment, you refer to the table and type the appropriate slot number followed by a space.

For each slot number you select from the table, the Librarian will display the prompt:

```
Slot to copy into?
```

You should respond by typing the slot number that you want the previously-specified segment to occupy when it is placed in the new library file. After you type the slot number, press the spacebar to terminate your entry. The Librarian will then transfer the specified segment into the Output codefile.

Segments may be placed in any available library slot, in any order. After each segment is transferred, the Librarian displays a new slot table for the Output codefile, which is your new library file.

To copy the first and third of the four segments in the Input file whose slot table is shown above, you would type the slot numbers as shown below. The repeated message "Slot to copy into?" is displayed by the Librarian.

```
Ø <space>
Slot to copy into?  Ø <space>
2 <space>
Slot to copy into?  1 <space>
```

If you want to include all of the segments in the Input file in your new library, or even most of them, you can use one of the other options given in the prompt line. If you type an equals sign (=), the Librarian will copy every segment from its slot in the Input file into the same slot in the Output file. If you type a question mark (?), the Librarian will step through the table and give you the option to select each input segment in turn. If you type a question mark with a table similar to the one shown above, the first prompt will be:

```
Copy slot Ø?
```

You should type Y if you wish the segment in slot Ø of the Input file to be copied into slot Ø of your new library file. Type N if you do not wish to copy that segment. The Librarian will repeat the prompting message for each occupied slot in the Input file. When you use either of the multiple-slot options, each segment copied from a slot in the Input file will automatically be placed in the slot with the same number in the Output file, which contains your new library.

If you attempt to put an input segment into an output slot that is already occupied, this message will appear:

```
WARNING - Slot xx already copied.  Please reconfirm (Y/N)
```

To abandon the current move, type N . If you type Y , the segment you previously placed in the specified slot will be replaced by the segment you are currently moving.

Note that the actual code for the replaced segment is not removed from the library. The best way to avoid this unwanted increase in the size of your library file is to start a new library by copying only the old library segments that you want, then adding the new ones, rather than replacing the unwanted items.

When all of the segments that you want from this Input file have been copied into the Output file, you can request a new input file by typing N for New file. The Librarian will prompt you again:

```
Input File ->
```

Type the name of the next Input file. The Librarian will prompt you to copy the desired segments, as above.

Each time the Librarian puts a segment into the Output file, it displays a new output slot table. For example, the Output file prompt line and the display of the output library table might look like this:

```
Output File -> /MYDISK/NEW.LIBRARY
File length - 29
        Ø-(3Ø) LONGINTI  2452          8-              Ø
        1-(31) PASCALIO  1238          9-              Ø
        2-(29) TRANSCEN  1168         1Ø-              Ø
        3-(22) APPLESTU   662         11-              Ø
        4-                  Ø         12-              Ø
        5-                  Ø         13-              Ø
        6-                  Ø         14-              Ø
        7-(25) PILFER     362         15-              Ø
```

The File length displayed with the table is the new library's length in blocks--in this example, it is 29.

For more information about segments and Units, see the chapters PROGRAM SEGMENTATION and LIBRARY UNITS in the Apple III Pascal Programmer's Manual.

Remember that a library file has the same internal format as a codefile. This means that a codefile generated by the Compiler can be used as a library.

Inserting a Copyright Notice

Once the needed segments from all Input codefiles have been put into your new library's Output codefile, you tell the Librarian you are finished by typing Q for Quit. The Librarian then displays this prompt at the bottom of the screen:

```
Notice?
```

This prompt enables you to put a copyright notice in your library file. The notice will be displayed each time a library map is produced for your file, as described in the next section of this manual. For example, you might type:

Copyright (c) 1981 Apple Computer Inc.

or any other message up to one line long. If you do not want a copyright line in your library file, simply press the RETURN key. When the Command prompt line reappears, indicating that the Librarian is finished, your new library is complete.

The Librarian program does not copy the copyright notice from a Input file into the Output file. If you make a new library file with the same name as the old one, any previous copyright notice is lost.

Library Mapping

The Library Mapper program creates a Map textfile for a library file, or for any codefile. The Map textfile lists information about multi-part programs that you are creating, including:

- Linker information for each segment
- The interface section of each Pascal Unit
- Procedures and Functions in each segment
- The parameters for each Procedure and Function

See the chapters PROCEDURES and FUNCTIONS, LIBRARY UNITS, and PROGRAM SEGMENTATION in the Apple III Pascal Programmer's Manual for information about Procedures, Functions, Units, and Segments.

Files Needed

The following files must be present in drives for you to use the Library Mapper program:

- LIBMAP.CODE
- Library Codefiles to be mapped
- Map output textfile (after the Library Mapper has started)

The file LIBMAP.CODE is normally found on diskette NEWPASCAL1 , but it can be in any drive. Once the Library Mapper program has started, you can remove the diskette with LIBMAP.CODE on it from its drive.

The library codefiles can be in any drives. Each library codefile only has to be present while it is being mapped. As soon as the program prompts you for the next codefile, you can remove the diskette with the previous file on it.

The output file for the map itself must be present throughout the mapping process. If you don't specify a file for the map, it will be sent to .CONSOLE .

When you terminate the library mapping utility program, your Pascal system diskette (NEWPASCAL2) should be in the built-in drive. If it is not there, the system will prompt you to put it in.

Using the Library Mapper

With the Command prompt line showing, and with diskette NEWPASCAL1 in any available disk drive, type X for Execute. When the prompt

```
Execute what file?
```

appears, respond by typing

```
/NEWPASCAL1/LIBMAP
```

Note that, as usual, the suffix .CODE is supplied automatically if you don't type it. Soon this message appears

```
Apple /// Library Map Utility [A3/1.Ø]
```

and the program prompts you to

```
Enter library name:
```

You should respond by typing the pathname of a library file or codefile. The program first tries to find the file exactly as specified. If this search fails, the program adds the suffix .CODE to the pathname and tries again. If the specified file is not found, the following message appears:

```
Bad file
Enter library name:
```

If the file you specify is not a codefile, this message appears:

```
Not a code file
Enter library name:
```

Typing an asterisk (*) in response to the library name prompt is the same as typing */SYSTEM.LIBRARY . This specifies the system library on the main system diskette as the input library file.

The Library Mapper is normally used for listing the information in the interfaces of the Units in a library, but the option is also available to show unresolved symbol references. The program will offer you the option by displaying this prompt:

```
List linker info table (Y/N)?
```

If you do not want this information, type an N or just press RETURN. If you respond to this prompt by typing a Y , the program will prompt further:

```
List referenced items (Y/N)?
```

Pressing the space bar or the RETURN key is considered an N.

The program now prompts you for:

```
Map output file name:
```

You should respond by typing the pathname where you want the program to send the map information. Note that if you don't add the suffix .TEXT to the pathname, the system automatically adds it for you. To override this feature, just type a period after the pathname. If you respond by pressing only the RETURN key, the program sends the map output to .CONSOLE .

Several codefiles can be mapped in succession. When the program has finished mapping the current codefile, it prompts you again:

```
Enter library name:
```

The Library Mapper will create a map for each library or program codefile you specify. These maps will all be sent to the same Map Output textfile.

To quit the Library Mapper, press the RETURN key the next time the program prompts:

```
Enter library name:
```

When you finish using the Library Mapper, your main system diskette should be in the built-in drive. If it is not there, the system will prompt you to put it in.

Library Map Example

Here is an example showing the map information for the sample program presented in the introduction to this manual and in the chapter THE ASSEMBLER. The prompt messages are described above; they are shown here just as they appear on the display. Note: the lines of dashes separating map information for different files are output by the program.

```
X
Execute what file? .d2/libmap
Apple /// Library Map Utility [A3/1.Ø]
enter library name: .d2/callasm
list linker info table (Y/N)? y
list referenced items (Y/N)? y
map output file name:              { RETURN pressed to send }
                                   { map output to .CONSOLE }


  LIBRARY MAP FOR .d2/callasm.CODE


Segment # 1:
System version = A3/1.Ø, code type is P-Code (least sig. 1st)
CALLASM   Pascal host outer block
    AA        public var  base = 5
    I         public var  base = 4
    K         public var  base = 3
    INCARRAY external proc  P #2
    TIMES2   external proc  P #3

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

```
enter library name: .d2/asmsubs
list linker info table (Y/N)? y
list referenced items (Y/N)? y


  LIBRARY MAP FOR .d2/asmsubs.CODE


Segment # 1:
System version = A3/1.Ø, code type is 65Ø2
TIMES2    separate procedure segment
    TIMES2    separate proc  P #1
    TIMES2    global addr    P #1,  I #Ø
    INCARRAY separate proc  P #2
    INCARRAY global addr    P #2,  I #Ø

     ------------------------------------------------------

enter library name: .d2/sample
list linker info table (Y/N)? y
list references items (Y/N)? y


  LIBRARY MAP FOR .d2/sample.CODE


Segment # 1:
System version = A3/1.Ø, code type is P-Code (least sig. lst)
SAMPLE    completely linked segment

     ------------------------------------------------------

enter library name:     { RETURN pressed to stop mapping }
```

Library Use Summaries

This section contains summaries of the operation of the system Librarian and the Library Mapper.

The System Librarian

1. Type X from the Command level. When prompted Execute what file? , type /NEWPASCAL1/LIBRARY .

2. When prompted for an Output file, type a filename for the new library file, e.g., /MYDISK/NEW.LIBRARY .

3. When prompted for an Input file, type the name of the file containing the first items to put in the new library, e.g., /NEWPASCAL2/SYSTEM.LIBRARY .

4. To transfer an item from the Input file to the new library Output file, type the item's Input file slot number (Ø to 15) and press the spacebar. When prompted for Slot to copy into? , type the number of the slot you want the item to occupy in the Output file and press the spacebar.

5. Type N to begin taking items from a new Input file.

6. When all desired items have been transferred to the new library, type Q for Quit. When prompted for Notice, type a copyright notice or other message or press RETURN.

7. To use Intrinsic Units from the new library automatically, you must either move it to your main system diskette and name it SYSTEM.LIBRARY or move it to the same volume as your program codefile and make it the program library by giving it the program name with the suffix .LIB .

Library Mapping

1. Type X from the Command level. When prompted Execute what file? , type /NEWPASCAL1/LIBMAP .

2. When prompted Enter Library name: , type the pathname of the library or other code file whose contents you wish to see mapped, e.g., /NEWPASCAL2/SYSTEM.LIBRARY .

3. When prompted for Linker info table? , type Y if you want that information, otherwise press the spacebar or RETURN key.

4. When prompted for Map output file name: , type the pathname of the diskette file or other device to which you wish the map sent. Just pressing the RETURN key sends the map to .CONSOLE .

5. When prompted again to Enter Library name: , type the pathname of the next library file whose contents you wish mapped, or press the RETURN key to quit the program.

A

A Complex Sample Program

Introduction

This appendix shows how a complex program is created using the Apple III Pascal system. It shows the procedures for compiling, assembling, linking, and using the Librarian to put together a sample program with a regular Unit, Intrinsic Units, and external procedures. The sample program includes the use of .Public, .Private and .Const directives to access data structures from assembly language routines.

In creating the sample program, you will use features of the Pascal system described in this manual and in Apple III Pascal Programmer's Manual. These features are not described here. You should read the appropriate sections of the manuals and study the other sample programs before you try to create the complex sample program.

Figure A-1 illustrates the different program sections and the sequence of operations required to put together the sample program.

The text of each program section is given below. You should use the Editor and type each one, then save it with the file name given. The procedures to use in creating the program are given in the last section of this appendix.

For this example, the system files are on two diskettes, in a different arrangement from the one used in the rest of the manual. Here, all program textfiles and codefiles are on a diskette called BOOTWRK in the built-in drive. Besides the program files mentioned below, BOOTWRK has copies of SYSTEM.MISCINFO, SYSTEM.FILER and SYSTEM.PASCAL on it. The Assembler, Compiler, Linker and Librarian programs are all on another diskette in drive .D2 . The section on PROGRAM PREPARATION DISKETTES in the chapter FIRST STEPS IN PROGRAM PREPARATION describes the procedure for rearranging the system files.

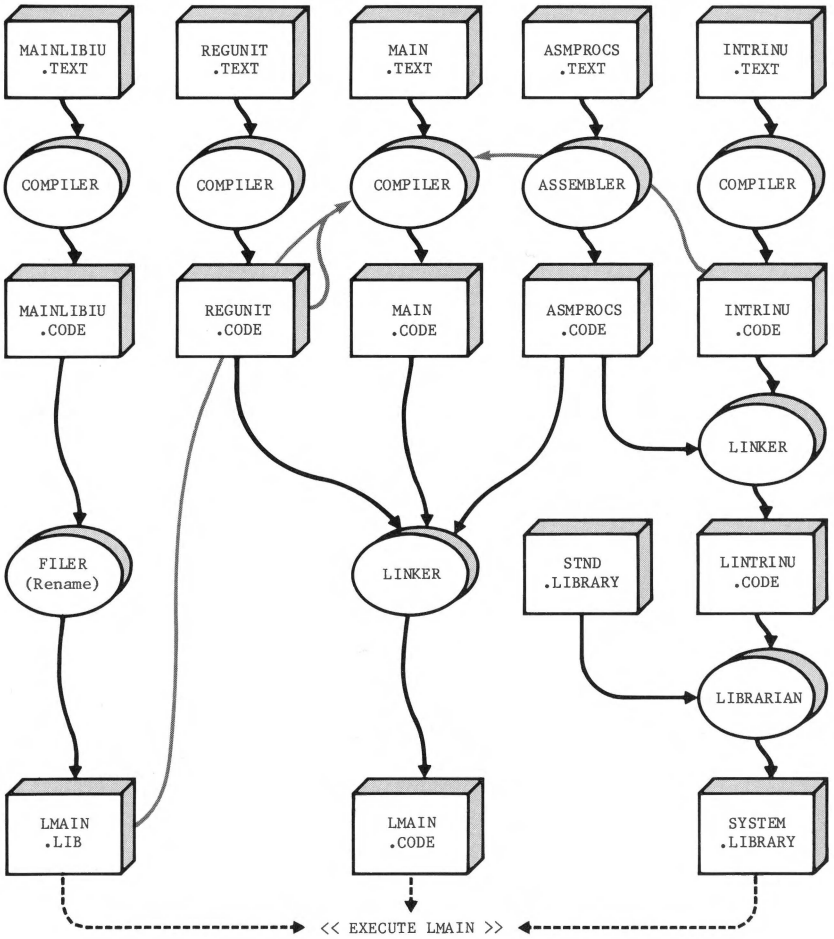


Figure A-1. Creating a Complex Program

The Host Program

Use the Editor to create a textfile named MAIN containing the following text:

```
PROGRAM MAIN;

  USES APPLESTUFF,
         { MAINLIBIU is first since REGUNIT uses it also }
       {$U LMAIN.LIB} MAINLIBIU, {$U REGUNIT.CODE } REGUNIT,
       {$U INTRINU.CODE} INTRINU;

  CONST LENGTH = 8Ø;
  VAR   I,RESULT,INVAL,IOR:INTEGER;
        MSTR:STRING;

    { Return INT multiplied by 2 }
  FUNCTION MULT2(INT:INTEGER):INTEGER; EXTERNAL;

    { Store INT in I if INT >= LENGTH }
  PROCEDURE STOREI(INT:INTEGER); EXTERNAL;

BEGIN
  WRITE('Starting main, enter MSTR:');
  READLN(MSTR);
  REGUPROC(MSTR);
  MAINLIBP;
  I:=Ø;
    {$I-}  { Turn off system input checking; do it ourselves }
  REPEAT
    WRITE('Value to multiply by 2:');
    READ(INVAL);
    IOR:=IORESULT;
    READLN
  UNTIL IOR=Ø;
    {$I+}
  RESULT:=MULT2(INVAL);
  STOREI(RESULT);
  WRITELN('I=',I,', RESULT=',RESULT);
  INTRINUPROC;
  WRITE('Press any key:');
    { Wait for user to press a key }
  WHILE NOT KEYPRESS DO;
  WRITELN('Done!')
END.
```

The Regular Unit

Use the Editor to create a textfile named REGUNIT containing the following text:

```
UNIT REGUNIT;

  INTERFACE

      { Regular units can use intrinsics
        but intrinsics cannot use regulars }
    USES {$U LMAIN.LIB} MAINLIBIU;

    PROCEDURE REGUPROC(ST:STRING);

  IMPLEMENTATION

      { External procedures are only allowed
        in the implementation part of units. }
      { Return in SUM the added ASCII values of chars in STR }
    PROCEDURE CHKSUM(VAR SUM:INTEGER; STR:STRING); EXTERNAL;

    PROCEDURE REGUPROC;
      VAR CSUM:INTEGER;
    BEGIN
      CSUM:=Ø;
      CHKSUM(CSUM,ST);
      WRITELN('ST=',ST,', CHECKSUM=',CSUM)
    END;

BEGIN
END.
```

The Intrinsic Units

Use the Editor to create a textfile named MAINLIBIU containing the following text:

```
UNIT MAINLIBIU; INTRINSIC CODE 1Ø;
  INTERFACE

    PROCEDURE MAINLIBP;

  IMPLEMENTATION

    PROCEDURE MAINLIBP;
    BEGIN
      WRITELN('In mainlibp')
    END;

BEGIN
END.
```

Use the Editor to create a textfile named INTRINU containing the following text:

```
UNIT INTRINU; INTRINSIC CODE 4Ø DATA 41;

  INTERFACE

      { Nested intrinsic units}
      { (Regular units can be nested too.) }
    USES {$U LMAIN.LIB} MAINLIBIU;

      { This variable requires a data segment }
    VAR IUSTRING:STRING;

    PROCEDURE INTRINUPROC;

  IMPLEMENTATION

      { Just to show that intrinsic units can
        have external procedures and functions. }
    PROCEDURE DONOTHING; EXTERNAL;

    PROCEDURE INTRINUPROC;
    BEGIN
      WRITELN(IUSTRING);
      WRITELN('Calling DONOTHING');
      DONOTHING;
      WRITELN('Called DONOTHING');
      MAINLIBP
    END;
```

```
BEGIN
    { This code is only executed once }
  IUSTRING:='Hi, I am an intrinsic unit!  What are you?'
END.
```

The Assembly Language Routines

Use the Editor to create a textfile named ASMPROCS containing the following text:

```
                .PROC   CHKSUM,2

                ;   as defined in Host Program:
                ; PROCEDURE CHKSUM(VAR SUM:INTEGER; STR:STRING);
                ;       EXTERNAL;

                ; EØ..EE are 8 byte-pairs with
                ; pre-defined extend bytes.
STRPTR          .EQU    ØEØ             ;Byte pair EØ-E1
SUMPTR          .EQU    ØE2             ;Byte pair E2-E3

                PLA                     ;save return address
                STA     RET
                PLA
                STA     RET+1

                ;These parameters are pointers.
                ;The ADDRESS of a string is always passed to
                ;an external procedure even if the string is
                ;not a VAR parameter.
                ;The ADDRESS of SUM is passed since it is a
                ;VAR parameter
                PLA
                STA     STRPTR          ;Pointer to STR
                PLA
                STA     STRPTR+1
                PLA
                STA     SUMPTR          ;Pointer to SUM
                PLA
                STA     SUMPTR+1
```

```
                LDY     #Ø              ;Zero sum and set Y to Ø.
                STY     SUM
                STY     SUM+1
                LDA     @STRPTR,Y       ;Get length of string.
                                        ;Y is still Ø, in case
                                        ;we take the branch on
                                        ;zero length string.
                BEQ     FINISH
                TAY

NXTCHAR         LDA     @STRPTR,Y       ;Start at last char and
                                        ;add sum of all chars.
                CLC
                ADC     SUM
                STA     SUM
                LDA     #Ø
                ADC     SUM+1
                STA     SUM+1
                DEY
                BNE     NXTCHAR         ;Add another char if
                                        ;more are left.

                ;We assume Y is zero on entry to FINISH
FINISH          LDA     SUM             ;Store the results
                STA     @SUMPTR,Y       ;low byte
                INY
                LDA     SUM+1
                STA     @SUMPTR,Y       ;high byte

                LDA     RET+1           ;Go back to caller
                PHA
                LDA     RET
                PHA
                RTS

SUM             .WORD   Ø
RET             .WORD   Ø

                ;.END    Do this so all the assembly procedures
                         ;can be assembled in one assembly.


                .PROC STOREI,1

                ;   Pascal declaration is:
                ; PROCEDURE STOREI(INT:INTEGER); EXTERNAL;
```

```
                .MACRO  MOVADR
                LDA     %1
                STA     %2
                LDA     %1+1
                STA     %2+1
                .ENDM

                ;As you will see,
                ;using these requires the utmost care!
                .CONST  LENGTH          ;All 3 of these data
                .PUBLIC I               ;are in the global data
                .PRIVATE PRET           ;segment of MAIN, only
                                        ;PRET is not accessable
                                        ;by MAIN.

ZI              .EQU    ØEØ             ;See comment about EØ..
ZRET            .EQU    ØE2             ;..EE in CHKSUM above


                MOVADR  II,ZI           ;Use macro MOVADR to
                MOVADR  RET,ZRET        ;move global addresses
                                        ;into zero page

                LDY     #Ø
                PLA                     ;Store return address
                                        ;in private global area.
                STA     @ZRET,Y         ;Low byte.
                INY
                PLA
                STA     @ZRET,Y         ;High byte.
                DEY

                ;If INT is >= LENGTH then store it in I
                SEC
                PLA
                TAX                     ;Low byte of INT
                SBC     LEN             ;INT - LENGTH
                INY
                PLA
                STA     TMP             ;High byte of INT
                SBC     LEN+1
                BCC     RETURN          ;Branch if INT < LENGTH
```

```
                ;Store INT in I
                DEY
                TXA
                STA     @ZI,Y           ;Low byte
                LDA     TMP
                INY
                STA     @ZI,Y           ;High byte

RETURN          LDA     @ZRET,Y         ;High byte.
                PHA
                DEY
                LDA     @ZRET,Y         ;Low byte.
                PHA
                RTS

TMP             .BYTE   Ø
LEN             .WORD   LENGTH
II              .WORD   I
RET             .WORD   PRET

                ;.END    Do this so all the assembly procedures
                        ;can be assembled in one assembly.


                .FUNC   MULT2,1

                ;FUNCTION MULT2(INT:INTEGER):INTEGER; EXTERNAL;

                PLA                     ;Store return address
                STA     RET
                PLA
                STA     RET+1

                PLA                     ;Pull 4 bytes Function
                PLA                     ;filler off stack
                PLA
                PLA

                PLA                     ;Low byte of INT
                ASL     A               ;Low byte times 2 and
                TAX                     ;high bit into carry.
                PLA
                ROL     A               ;High byte times 2 with
                PHA                     ;low bit from carry.
                TXA                     ;Push result. No check
                PHA                     ;for overflow.
```

```
                LDA     RET+1           ;Push return and go
                PHA                     ;back to caller.
                LDA     RET
                PHA
                RTS

RET             .WORD   Ø

                ;.END


                .PROC   DONOTHING

                RTS                     ;This is a simple one
                                        ;just to show that
                                        ;intrinsic units can
                                        ;have EXTERNAL
                .END                    ;procs too!
```

Putting the Pieces Together

This section shows the command sequences for generating an executable codefile named LMAIN.CODE . The messages exchanged between you and the system are shown between lines of dashes (----), with the responses you type shown in upper case and the messages from the system shown in lower case. All sequences start from the main command line of the system.

The next command sequence, with a different textfile name, is used several times during the development of the sample program. When the procedures below call for the compile sequence, this is what they mean:

```
-----------------------------------
C
Compile what text? MAINLIBIU<RETURN>
To what codefile? $<RETURN>
-----------------------------------
```

Since MAINLIBIU is an Intrinsic Unit, you might as well put it into a library. For this example, use the program library. The executable codefile is named LMAIN.CODE , so the program library is named LMAIN.LIB . When you have a library with only one code file in it, you don't need to use the Librarian. Instead, you can use the following shortcut:

```
-------------------------------------------------
F
T
Transfer what file? MAINLIBIU.CODE,LMAIN.LIB<RETURN>
Q
-------------------------------------------------
```

Next, you should perform the compile sequence shown above on the files REGUNIT and INTRINU.

After that you should assemble the assembly procedures. For this example, all of the assembly procedures are in one file called ASMPROCS.

```
-----------------------------------------------------------
A
Assemble what text? ASMPROCS<RETURN>
To what codefile? $<RETURN>
Output file for Assembler listing (<CR> for none): <RETURN>
-----------------------------------------------------------
```

Now it is time to do the linking. First, link the intrinsic unit INTRINU, as shown below. Note: the lines starting with "Opening" and "Reading" are output by the Linker.

```
-----------------------------
L
Host file? INTRINU<RETURN>
Opening INTRINU.CODE
Lib file? ASMPROCS<RETURN>
Opening ASMPROCS.CODE
Lib file? <RETURN>
Map file? <RETURN>
Reading INTRINU
Reading CHKSUM
Output file? LINTRINU<RETURN>
-----------------------------
```

The next thing to do is to put this unit into a SYSTEM.LIBRARY along with the standard intrinsic unit APPLESTUFF. Before doing this, you should rename the standard SYSTEM.LIBRARY as STND.LIBRARY . It can be on any volume as it will not be required to run the program; here, it is on BOOTWRK. Note: the slot tables displayed by the Librarian program are not shown here.

```
-------------------------------------
X
Execute what file? .D2/LIBRARY<RETURN>
Output file -> SYSTEM.LIBRARY<RETURN>
Input file -> STND.LIBRARY<RETURN>
Ø<SPACE>                { number of slot APPLESTUFF is in }
Ø<RETURN>          { APPLESTUFF can go into any empty slot }
N
Input file -> LINTRINU<RETURN>
=
Q
Notice? <RETURN>
-------------------------------------
```

Next you should perform the Compile sequence on MAIN. Once you have done that, you can do the final link:

```
-------------------------------
L
Host file? MAIN<RETURN>
Opening MAIN.CODE
Lib file? ASMPROCS<RETURN>
Opening ASMPROCS.CODE
Lib file? REGUNIT
Opening REGUNIT.CODE
Lib file? <RETURN>
Map file? <RETURN>
Reading MAIN
Reading REGUNIT
Reading CHKSUM
Output code file? LMAIN<RETURN>
Linking REGUNIT #7
  Copying proc CHKSUM
Linking MAIN #1
  Copying proc STOREI
  Copying proc MULT2
-------------------------------
```

The file produced by the Linker is the executable codefile LMAIN. Type X LMAIN to execute it.

Figure A-2 is a composite picture of the operations involved in creating any complex program. It shows the different ways you can put a program together a piece at a time. Compare it with Figure A-1, which shows the program files making up the complex sample. Note that the compiler automatically picks up the interface text for imbedded Units. This means that those Units

must be compiled first and the resulting codefiles must be in some drive when the main program is compiled.

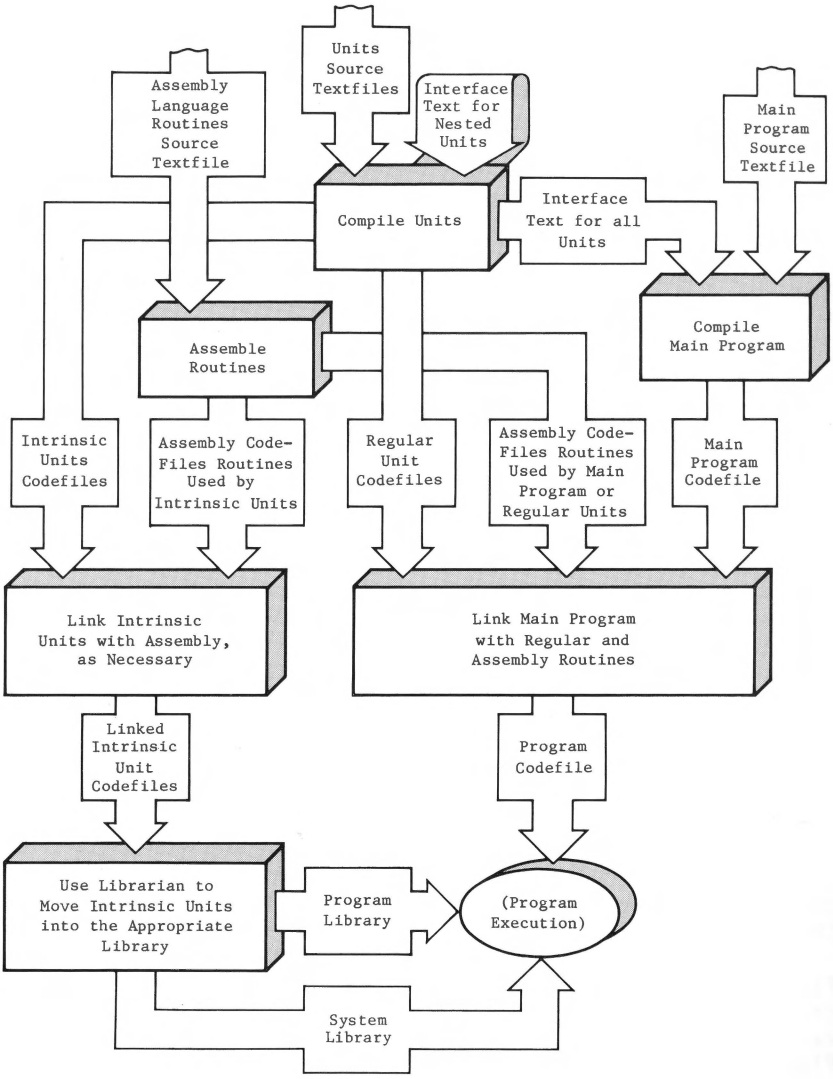


Figure A-2. Overview of Program Preparation

B

Special Memory Locations

Introduction

There are certain memory locations that will be useful if you are writing assembly language subroutines for your Pascal programs. The functions of some of these memory locations are defined by the Apple III hardware and the functions of the others are defined by the Apple III Pascal system.

Note that all addresses in this appendix are given in hexadecimal, as indicated by the dollar-sign prefix ($).

Apple III Hardware Control

$FFDØ Zero Page Register. Indicates which page is the current zero page. During execution of a Pascal program, this location will contain the page number defined by SOS for the zero page of a user program. SOS itself runs using a different zero page. For Pascal, the value in this location is $1A, indicating that the zero page resides at physical locations $1AØØ-$1AFF. The user should never change the contents of this register.

$FFEF Bank Register. The low-order four bits indicate the currently switched-in memory bank. The upper four bits contain additional machine state data. Setting the lower four bits to value b switches in bank b. User programs should not change the value of this register.

Special Pages

$1AØØ... $1AFF User program zero page.

$1BØØ... $1BFF User program 65Ø2 machine stack. Used as the Pascal "evaluation stack."

$16ØØ... $16FF User program address extension page corresponding to zero page $1AØØ-$1AFF. Used by the Apple III enhanced indirect addressing mechanism.

Locations Defined by Pascal

$EØ,$E1 $E2,$E3 ... $EE,$EF	Space reserved for user to set up pointers into Pascal data space. The user can set any of these two-byte "pseudo registers" to point to a location in Pascal data space and take advantage of the fact that the Pascal system sets the corresponding address extension page locations to the bank pair number of the Pascal data space.
$16E1 $16E3 ... $16EF	Address extension page locations corresponding to the zero page locations above. When the Pascal system is booted, it stores the bank pair number for the Pascal data space into these locations. The user must not change the contents of these locations.
$16FE	Bank number of highest memory bank in the machine. Total machine memory is 64K + 32K x <value in 16FE>.

Table Pointed to by .INTERP

When used in an expression, .INTERP is the address of a nine-word table in the interpreter. Each word in the table contains the address of a potentially useful part of the interpreter. The normal user will probably not use this feature of the assembler.

.INTERP+Ø	Address of the interpreter's run time execution error posting routine. The user can load the A register with the error number and execute JSR @.INTERP to invoke the system error message routine.
.INTERP+2	Address of BIOS (Input/Output handling routine) dispatching table.
.INTERP+4	Address of the location that contains the address of SYSCOM (the area used to communicate between the interpreter and Pascal Operating System).

.INTERP+6 Address of the location that contains the extension byte of the address of SYSCOM.

.INTERP+8 Address of the location that contains the address of the current global data area.

.INTERP+A Address of the location that contains the extension byte of the address of the current global data area.

.INTERP+C Address of the vector of words used by the interpreter for intermediate results of interpreter computations.

.INTERP+E Address of the table of pointers to code segments.

.INTERP+1Ø Address of the table of extension bytes corresponding to the segment pointers of the previous table.

C

Tables

The Pascal System Diskettes

The diskettes that contain the programs making up the Apple III Pascal System are called the system diskettes. This section describes these program files and defines the operations performed when starting up the Pascal system.

Definitions

When you first turn on the Apple III, there is nothing stored in its program memory. Special circuitry inside the computer starts loading the system software from the diskette in the built-in drive. This initial loading of the system software is called bootstrap loading, or just booting, because it seems as though the system is trying to pull itself up by its own bootstraps.

Bootstrapping the Pascal system involves loading the Apple III's operating system (files SOS.KERNEL and SOS.DRIVER), the P-code interpreter (SOS.INTERP), the Pascal command processor (SYSTEM.PASCAL), and the miscellaneous-information file (SYSTEM.MISCINFO). Booting with a diskette that contains all five of these files in the built-in drive is called a one-stage boot.

If space constraints make it inconvenient to have all five of these files on a single diskette, you can have a two-stage boot. In this case, you start the bootstrap operation by turning on the power or pressing CONTROL-RESET with a diskette that contains the first three files in the built-in drive. When its part of the bootstrap operation is complete, you remove it and insert a diskette that contains the Pascal command processor, SYSTEM.PASCAL , and SYSTEM.MISCINFO .

A diskette that is used in a bootstrap process is called a boot diskette. The diskette containing the file SYSTEM.PASCAL is called the Pascal system diskette and is normally kept in the built-in drive during operation of the Pascal system.

Starting up the system as if you had just turned on the power is sometimes called a cold boot. This is what happens when you press CONTROL-RESET. The second-stage boot, sometimes called a warm boot, is the same as what happens when you invoke the Halt command. The effects of these different kinds of re-start are summarized in the following table.

Operation:	Files Loaded:	Software Initialized:	Comment:
CONTROL-RESET	SOS.KERNEL SOS.INTERP SOS.DRIVER SYSTEM.PASCAL SYSTEM.MISCINFO	(all)	This is called a cold boot.
HALT	SYSTEM.PASCAL SYSTEM.MISCINFO	SOS.INTERP SYSTEM.PASCAL	This is called a warm boot.
INITIALIZE	(none)	SYSTEM.PASCAL	Initialization of the Pascal system

The System Files

The following tables show the contents of each of the Pascal system diskettes. The first table shows the contents of the diskettes as supplied.

It is sometimes more convenient to have the system diskettes configured differently. For program development, you can use a two-stage boot configuration, as shown in the second table.

When one of these files is needed by the system, it usually doesn't matter which diskette the file is on or which drive the diskette is in. The cases when a file must be on a particular diskette or in a particular drive are pointed out in the table The System Diskette Files: By Command, below.

The System Diskettes, as Supplied

The files making up the Apple III Pascal System are supplied on three diskettes. This table shows the system files that are found on each diskette. The order of the files on any diskette is unimportant.

PASCAL1	PASCAL2	PASCAL3
SOS.KERNEL	SYSTEM.EDITOR	LIBMAP.CODE
SOS.DRIVER	SYSTEM.SYNTAX	LIBRARY.CODE
SOS.INTERP	SYSTEM.COMPILER	SETUP.CODE
SYSTEM.PASCAL	SYSTEM.ASSMBLER	AIIFORMAT.CODE
SYSTEM.MISCINFO	OPCODES.65Ø2	
SYSTEM.LIBRARY	ERRORS.65Ø2	
SYSTEM.FILER	SYSTEM.LINKER	

PASCAL1 is both the boot diskette and the Pascal system diskette for running programs created with the Apple III Pascal system. This diskette contains all of the system files necessary for bootstrapping the system. It also contains the Filer, so that you can use it to move files around as soon as you boot the system.

PASCAL2 contains the programs used in program development, including the Editor, the Pascal Compiler, the Assembler, and the Linker.

PASCAL3 contains utility programs used for setting up the Pascal system, for building program libraries, and for formatting Apple II Pascal format diskettes.

System Diskettes for Program Development

The Pascal system diskettes as supplied are configured for a one-stage boot. You can use the Filer to make a set of system diskettes that will be more convenient for program development. There is a detailed description of the way this is done in the first chapter of this manual.

By moving the Apple III system files (SOS.KERNEL, SOS.DRIVER, and SOS.INTERP) onto a separate diskette, you can make room for most of the Pascal system programs you need for program development. This means you have to use a two-stage boot: the diskette with the SOS files on it is used only for the first stage of a cold boot.

Here is the way your program development diskettes should look, except for the order of the files on each diskette, which doesn't matter.

NEWPASCAL1	NEWPASCAL2	NEWPASCAL3
SOS.KERNEL	SYSTEM.PASCAL	OPCODES.65Ø2
SOS.DRIVER	SYSTEM.MISCINFO	ERRORS.65Ø2
SOS.INTERP	SYSTEM.LIBRARY	SYSTEM.LINKER
SETUP.CODE	SYSTEM.EDITOR	SYSTEM.ASSMBLER
AIIFORMATTER.CODE	SYSTEM.SYNTAX	SYSTEM.FILER
LIBRARY.CODE	SYSTEM.COMPILER	
LIBMAP.CODE		

NEWPASCAL1 contains the three SOS files needed for starting a cold boot. When the first stage of the boot is finished, the system will prompt you to remove this diskette and insert a Pascal system diskette into the built-in drive. NEWPASCAL1 also contains the utility programs, which can be in any drive when they are executed.

NEWPASCAL2 is your Pascal system diskette for program development. It contains SYSTEM.PASCAL , so it must be in the built-in drive whenever the system returns to the command level. If you are using the workfile, it will also be on this diskette.

NEWPASCAL3 contains the system programs there is no room for on the Pascal system diskette. It also contains the Filer. When you need to Get or Save a file on a user diskette, invoke the Filer with NEWPASCAL3 in a drive. Then you can remove it if necessary to make a drive available for your user diskette, and move your file to or from the Pascal system diskette, which is still in the built-in drive.

The System Files: By Command

The following table lists the files needed by each of the command options. With a few exceptions, the required files can be on any diskette and in any drive. Some of the commands listed require that certain files be on the Pascal system diskette and in the built-in drive. If you are using the system diskettes as supplied, the Pascal system diskette is PASCAL1 . With the recommended configuration for program development, the Pascal system diskette is NEWPASCAL2 .

Command	Files Needed	Where Files Must Be Found
File	SYSTEM.FILER	any disk, any drive; needed only at start
	Files to be moved	any disks, any drives; Transfer requires presence of source file; can prompt for destination file
Edit	SYSTEM.EDITOR	any disk, any drive
	Textfile to be Edited	any disk, any drive; optional; default is system workfile, on Pascal system diskette
Compile	SYSTEM.COMPILER	any disk, any drive
	Textfile to be Compiled	any disk, any drive; default is Pascal system diskette's system workfile, built-in drive
	SYSTEM.LIBRARY	Pascal system diskette, built-in drive; required only if program USES Intrinsic Units
	SYSTEM.EDITOR	any disk, any drive; optional; to fix errors found by Compiler
	SYSTEM.SYNTAX	Pascal system diskette, built-in drive; optional; provides error messages on entering Editor
Assemble	SYSTEM.ASSMBLER	any disk, any drive
	OPCODES.65Ø2	any disk, any drive; required
	ERRORS.65Ø2	any disk, any drive; optional; provides error messages in Assembler
	Textfile to be Assembled	any disk, any drive; default is Pascal system diskette's system workfile
	SYSTEM.EDITOR	any disk, any drive; optional; to fix errors found by Assembler
Link	SYSTEM.LINKER	any disk, any drive; needed only to start
	Host codefile	any disk, any drive; default is Pascal system diskette's system workfile, built-in drive
	Library codefile	any disk, any drive

Execute	Codefile to be Executed	any disk, any drive; required only when loading, if no overlays
	Program library	same disk as program codefile, any drive; required if program needs Intrinsic Units it contains
	SYSTEM.LIBRARY	Pascal system diskette, built-in drive; required if the program uses long integers, does file I/O using reals or SEEK, or USES Intrinsic Units that are not in program library
Run	Text or Codefile to be Run	any disk, any drive; default is Pascal system diskette's system workfile
	SYSTEM.COMPILER	any disk, any drive; required only if file being Run is a textfile
	SYSTEM.EDITOR	any disk, any drive; optional; to fix errors found by Compiler
	SYSTEM.SYNTAX	Pascal system diskette, any drive; optional; provides error messages for Editor
	SYSTEM.LINKER	any disk, any drive; required only if routines need to be Linked; no Link needed to USE Intrinsic Units
	SYSTEM.LIBRARY	Pascal system diskette, built-in drive; required if program uses long integers, does file I/O using reals or SEEK, or USES Intrinsic Units, or if it holds needed routines if Linking
	Program library	same disk as program codefile, any drive; required if program needs Intrinsic Units it contains
	SYSTEM.PASCAL	Pascal system diskette, built-in drive; required between Compiling, Linking, and eXecuting.
User restart	All files needed by previous program	same setup and files required by previous program

The System Files: By Filename

The next table gives more information about the files making up the Apple III Pascal System.

Filename	Contents of File	Use of File	When Needed
SOS.KERNEL	Operating system, written in 65Ø2 machine language	Interfaces all other programs to Apple III	Power-on, CONTROL-RESET
SOS.DRIVER	SOS I/O Drivers, written in 65Ø2 machine language	Interfaces all other programs to I/O devices	Power-on, CONTROL-RESET
SOS.INTERP	Interpreter, written in 65Ø2 machine language	Executes P-code on Apple III's processor	Power-on, CONTROL-RESET
SYSTEM.PASCAL	Command level portion of software-development system	Lets you pick Edit, File, Run, etc.	Power-on, CONTROL-RESET, Halt, return to command level
SYSTEM.MISCINFO	Information about system configuration	Tells system about system hardware	Power-on, CONTROL-RESET, Halt
SYSTEM.EDITOR	Text Editor	Lets you make & change text files	Edit, Compile Run, Assemble
SYSTEM.FILER	Filer	Lets you store, delete & move files	File
SYSTEM.LIBRARY	Routines for long integers, trigonometric functions, graphics, I/O, and optional user-defined Intrinsic Units	Many programs use these library routines	Run, Execute, Link, Compile, if program uses library routines
SYSTEM.SYNTAX	Compiler error messages	Provides message in Editor after Compiler finds an error	Run, Compile followed by Edit after an error

SYSTEM.COMPILER	Pascal Compiler	Converts Pascal program text to P-code	Compile, Run
SYSTEM.LINKER	Linker	Puts library routines into your program	Link, Run
SYSTEM.ASSMBLER	65Ø2 Assembler	Converts 65Ø2 assembly text into machine code	Assemble
OPCODES.65Ø2	Instruction set for Assembler	Used by the Assembler	Assemble
ERRORS.65Ø2	Assembler error messages	Optional; Provides message after Assembler finds an error	Assemble
LIBRARY.CODE	Utility program	Puts routines into library	Execute LIBRARY
LIBMAP.CODE	Utility program	Displays contents of library file	Execute LIBMAP
SETUP.CODE	Utility program	Makes new file SYSTEM.MISCINFO describing the system configuration	Execute SETUP
AIIFORMAT.CODE	Utility program	Converts a SOS-formatted diskette to an Apple II Pascal formatted diskette	Execute AIIFORMAT

Pascal I/O Device Volumes

The Apple III Pascal System assigns volume numbers and volume names to the various input/output devices as shown in the following table. The SOS device names shown here are typical; you can use the System Configuration Program to change them.

Pascal Device Number	Pascal Device Name	SOS Device Name	SOS Volume Name	Description of Input/Output Device
#Ø:	--	--	--	(not used)
#1:	CONSOLE:	.CONSOLE	--	Screen & keyboard (echo on input)
#2:	SYSTERM:	.CONSOLE	--	Screen & keyboard (no echo on input)
#3:	GRAPHIC:	.GRAFIX	--	Graphics
#4:	--	.D1	diskette name	Built-in disk drive
#5:	--	.D2	diskette name	1st external disk drive
#6:	PRINTER:	.PRINTER	--	Printer
#7:	REMIN:	.RS232	--	Remote input
#8:	REMOUT:	.RS232	--	Remote output
#9:	--	.D3	diskette name	2nd external disk drive
#1Ø:	--	.D4	diskette name	3rd external disk drive

ASCII Character Codes

Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char
Ø	ØØ	NUL	32	2Ø	SP	64	4Ø	@	96	6Ø	`
1	Ø1	SOH	33	21	!	65	41	A	97	61	a
2	Ø2	STX	34	22	"	66	42	B	98	62	b
3	Ø3	ETX	35	23	#	67	43	C	99	63	c
4	Ø4	EOT	36	24	$	68	44	D	1ØØ	64	d
5	Ø5	ENQ	37	25	%	69	45	E	1Ø1	65	e
6	Ø6	ACK	38	26	&	7Ø	46	F	1Ø2	66	f
7	Ø7	BEL	39	27	'	71	47	G	1Ø3	67	g
8	Ø8	BS	4Ø	28	(	72	48	H	1Ø4	68	h
9	Ø9	HT	41	29	)	73	49	I	1Ø5	69	i
1Ø	ØA	LF	42	2A	*	74	4A	J	1Ø6	6A	j
11	ØB	VT	43	2B	+	75	4B	K	1Ø7	6B	k
12	ØC	FF	44	2C	,	76	4C	L	1Ø8	6C	l
13	ØD	CR	45	2D	-	77	4D	M	1Ø9	6D	m
14	ØE	SO	46	2E	.	78	4E	N	11Ø	6E	n
15	ØF	SI	47	2F	/	79	4F	O	111	6F	o
16	1Ø	DLE	48	3Ø	Ø	8Ø	5Ø	P	112	7Ø	p
17	11	DC1	49	31	1	81	51	Q	113	71	q
18	12	DC2	5Ø	32	2	82	52	R	114	72	r
19	13	DC3	51	33	3	83	53	S	115	73	s
2Ø	14	DC4	52	34	4	84	54	T	116	74	t
21	15	NAK	53	35	5	85	55	U	117	75	u
22	16	SYN	54	36	6	86	56	V	118	76	v
23	17	ETB	55	37	7	87	57	W	119	77	w
24	18	CAN	56	38	8	88	58	X	12Ø	78	x
25	19	EM	57	39	9	89	59	Y	121	79	y
26	1A	SUB	58	3A	:	9Ø	5A	Z	122	7A	z
27	1B	ESC	59	3B	;	91	5B	[	123	7B	{
28	1C	FS	6Ø	3C	<	92	5C	\	124	7C	|
29	1D	GS	61	3D	=	93	5D	]	125	7D	}
3Ø	1E	RS	62	3E	>	94	5E	^	126	7E	~
31	1F	US	63	3F	?	95	5F	_	127	7F	DEL

D

Command Summaries

Command Summaries

This appendix contains summaries of the commands used with the system programs described in this manual: the Assembler, the Linker, the Librarian, and the Library Mapper. For descriptions of the commands used with the Editor and the Filer, refer to Apple III Pascal: Introduction, Filer, and Editor. For the commands used with the Pascal Compiler, refer to the Apple III Pascal Programmer's Manual.

Assembler Commands

1. From Command level, select Assemble.

2. If a text workfile exists, that file is assembled automatically. Otherwise, you are prompted to specify a source textfile and then to specify a destination codefile.

3. Finally, you are prompted to specify an output textfile for the assembly listing, if you want one.

4. If the Assembler finds an error, select the Editor to fix it.

Linker Commands

1. From Command level, select Run or Link.

2. Run automatically links the compiled workfile to Units and routines found in SYSTEM.LIBRARY . Link prompts you to specify a host codefile and then to specify as many library codefiles as needed. Press the RETURN key when you want to stop specifying library files.

3. Next, the Linker prompts you to specify a map textfile for storing Linker information. Normally, you press the RETURN key to go on.

4. Finally, the Linker prompts you to specify an output codefile for the linked program.

Librarian Commands

1. From Command level, select the Execute option. When you are prompted EXECUTE WHAT FILE?, type the name of the desired librarian program.

There are two programs that enable you to edit library codefiles:

LIBRARY	Puts Units and routines into a library file. The Library program prompts you for the name of the new library file and the name of the file containing the items to put into the library. It also enables you to install a copyright notice.
LIBMAP	Shows the contents of a library file or other codefile.

E

User Notes

Making a Turnkey Diskette

It is sometimes convenient to have the Apple III start running a user program as soon as the machine is turned on. A system that works this way is called a turnkey system, because all the user has to do is turn power on--"turn the key"--and the system comes up running the desired program. Using the Apple III Pascal System, you can set up a diskette so that the Apple III will automatically begin running your program when you insert the turnkey diskette and turn the Apple on.

To set up a turnkey system with a program you have created on the Pascal System, you start with a formatted blank diskette with a name you will recognize: for example, TURNKEY . Using the Filer's Transfer command, transfer all of the system files on diskette PASCAL1 onto TURNKEY, by typing

```
/PASCAL1/=, /TURNKEY/=
```

when the Transfer command prompts you for file names. Next, you remove the file SYSTEM.FILER and transfer a copy of your program codefile onto the turnkey diskette, giving this new copy of your program the filename SYSTEM.STARTUP . If your program uses a program library, re-name the program library file SYSTEM.STAR.LIB . Make sure your turnkey diskette contains the following files:

```
SOS.KERNEL
SOS.DRIVER  (use SCP to edit as needed)
SOS.INTERP
SYSTEM.PASCAL
SYSTEM.MISCINFO
SYSTEM.STARTUP  (your program code file)
SYSTEM.LIBRARY  (if needed by your program)
SYSTEM.STAR.LIB  (if needed by your program)
```

To run your turnkey program, insert the turnkey diskette into the built-in drive and start a cold boot by pressing CONTROL-RESET. Soon, with no further action on your part, SYSTEM.STARTUP is executed. Thereafter, the program file you named SYSTEM.STARTUP will be executed each time the system is re-booted, as long as the diskette containing your SYSTEM.STARTUP is in the built-in drive.

Exec Files

An exec file consists of commands stored in an ASCII file. When an exec file is executed, each command included in the file is executed, just as if you were typing the commands from the keyboard. Exec files are used to store sequences of commands that must be entered into the system over and over again.

This section contains an explanation of exec files and an example demonstrating how to create and execute one.

Using Exec Files

To create an exec file, Type M for Make from the main Command level. The system will prompt you with the message

```
New exec name:
```

Type the pathname you want to give to your exec file, following the same rules that govern the naming of other Pascal files. Now you will see the prompt

```
Terminator=%, change it?
```

The terminator is a character that is used to signify the beginning and end of an exec file. The terminator character that marks the beginning of the file is automatically supplied by the system. Two terminator characters indicate the end of the file; these must be typed by the user. If you answer the above prompt by typing N for No, the system will use a percent sign as a terminator. If you type Y, meaning that you want to change the terminator character, the system will ask for

```
New terminator:
```

The character you type becomes the terminator character for that exec file. The system will accept any character as the terminator character. Note that some keystrokes do not produce characters, and thus cannot be used as the terminator character.

The terminator character that signals the beginning of an exec file is supplied by the system. Do not begin an exec file by typing the terminator character. If you do, the system-supplied terminator immediately followed by your typed terminator will be interpreted as the end-of-file signal and the system will close the exec file.

Once the system knows what your terminator character is, you can begin typing the series of commands that will make up your exec file. Commands will be executed as you type them. To end the exec file, type the terminator character twice.

When you are ready to execute your exec file, type X for Execute from the main Command level. When the system prompts with

```
Execute what file?
```

you should respond by typing

```
EXEC//<pathname>
```

The system will start performing the series of commands listed in your exec file, flashing the prompts and your previously entered responses as it goes.

When using an exec file, you must make sure that the system will be able to go through EXACTLY the same sequence of events that it went through when you created the exec file. For example, suppose you create an exec file that enters the Filer, transfers the file MYFILE.TEXT from diskette OLDSTUF to diskette NEWSTUF and then returns to the main Command level. If you later run your exec file without removing the original diskette NEWSTUF from its disk drive, the system would find MYFILE.TEXT already present on that diskette. Consequently, the system will ask

```
Remove old /NEWSTUF/MYFILE ?
```

This is a question that was not asked when the exec file was created. The system will use as its response the next character in the exec file which, in this case, happens to be Q for Quit. In order for the system to remove a file under these conditions, it must receive an N for No or a Y for Yes as the

response to the above question. Thus, the old version of MYFILE will not be removed and the new version of MYFILE will not be transferred. Because the Q was used to respond to this question, the exec file never uses the Q to Quit the Filer and the exec file closes with the system still at the Filer level. Thus, when creating an exec file, you must make sure that the steps the system goes through will not change from one execution to another.

There is no way to stop the execution of an exec file part way through except by pressing RESET. You can use the control command CONTROL-7 to stop or freeze the output on the screen temporarilty. The control command CONTROL-9 flushes the output: the program continues to run but its output is not sent to the screen.

The keyboard remains open during the execution of an exec file. Thus characters that are entered while an exec file is running are saved and then used as console input once the exec file is closed.

If you use the Make command to create an exec file on a diskette that already has a file with the name you have given your exec file, the system will eliminate the original file when you close the new exec file.

It is not permissible to create an exec file from within another exec file. If you try, the system will warn you that

```
Nested exec commands illegal
```

After you see this message, you can continue entering commands into the exec file.

If you run a Pascal program while you are making an exec file, typed responses to any UNITREAD procedures specifying unit 1 or unit 2 (.CONSOLE) are used by the Pascal program, but are NOT stored in the exec file. The KEYPRESS function also will take its input from the console and not from the exec file. READ, READLN, and GET procedures will take their input from the exec file, if reading from the standard FILE input.

The system error routine will close an open exec file if an error occurs while the system is getting console input from the exec file.

A Sample Exec File

Suppose you want to create an exec file that removes all the files on whatever diskette is in the drive .D2 and then copies all the files from the diskette in drive .D1 (the built-in drive) onto the diskette in drive .D2 . To start making your exec file, type M for Make from the Command Level. The system will prompt you with the message

```
New exec name:
```

In response, type

```
/NEWSTUF/UPDATE
```

You have just started an exec file named UPDATE that will be saved on diskette NEWSTUF. Next the system will ask:

```
Terminator=%, change it?
```

Respond by typing

```
N
```

Entering this response tells the system that the terminator character used to signal the beginning and end of the exec file will remain the percent sign.

Now you are ready to enter the list of commands that will make up your exec file.

Keystrokes	Explanation
F	Enter Filer
R	Execute Remove command
.D2/=	Response to prompt: Remove what file? telling the system to remove all files on the diskette in drive .D2
RETURN	RETURN ending the previous response
<space>	To handle "Press space to continue" prompts
<space>	
<space>	
Y	Response to prompt: Update directory?
T	Execute Transfer command
.D1/=	Response to prompt: Transfer what file?
RETURN	RETURN ending the previous response

.D2/=	Response to prompt: To where?
RETURN	RETURN ending the previous response
Q	Exit from the Filer
%%	Terminator characters indicating the end of the exec file

Note that each of these commands is executed in the normal fashion when it is typed into the exec file. Thus, after carrying out the above list of commands, the files on diskette in drive .D2 will be replaced by copies of the files on the diskette in drive .D1.

When you are ready to execute the exec file, type X for Execute from the Command Level, then type

```
EXEC///NEWSTUF/UPDATE
```

You then will see the system enter the Filer, Remove the files from diskette in drive .D2, transfer all the files from the diskette in drive .D1 to the diskette in drive .D2, and return to the Command level.

Notice that the Exec command is followed by three slashes: the first two delimit the command and the third one starts the pathname. You must type all three.

F

Bibliography

Bibliography

There are several books on 65Ø2 assembly language programming. A few of them are listed here, along with the programming reference manual published by the manufacturers of the 65Ø2 microprocessor.

Barden, William, Jr.: How to Program Microcomputers. Howard W. Sams, Indianapolis, Indiana, 1977.

de Jong, Marvin L.: Programming and Interfacing the 65Ø2, with Experiments. Howard W. Sams, Indianapolis, Indiana, 198Ø.

Foster, Caxton C.: Programming a Micro-computer: 65Ø2. Addison-Wesley, Reading, Massachusetts, 1978.

Inman, Don, and Inman, Kurt: Apple Machine Language. Reston Publishing Company, Inc., Reston, Virginia, 1981.

Leventhal, Lance A.: 65Ø2 Assembly Language Programming. Osborne/McGraw-Hill, Berkeley, California, 1979.

MOS Technology, Inc.: MCS65ØØ Microcomputer Family Programming Manual. MOS Technology, Inc., Norristown, Pennsylvania, 1975. (also published by: Synertek, Santa Clara, California, 1976.)

Scanlon, Leo J.: 65Ø2 Software Design. Howard W. Sams, Indianapolis, Indiana, 198Ø.

Weller, W. J.: Practical Microcomputer Programming: the 65Ø2. Northern Technology Books, Evanston, Illinois, 198Ø.

Zaks, Rodnay: Programming the 65Ø2. Sybex, Berkeley, California, 198Ø.

--- Programming Excercises for the 65Ø2. Sybex, 198Ø.

--- 65Ø2 Applications Book. Sybex, 198Ø.

G

Error Messages

Execution Error Messages

When an error is detected during execution of a user program, it is reported in one of the following forms:

By Number:	By Name:
Exec err # 1Ø	I/O error: Vol not found
S# 1, P# 7, I# 56	S# 1, P# 7, I# 56
Type <space> to continue	Type <space> to continue

The first line of the message reports the error by number or by a short description. The second line of the message gives the location in the program where the error was detected. The number after S# is the segment number, the number after P# is the procedure number within that segment, and the number after I# is the byte offset in that procedure. The Compiler will list segment, procedure, and byte-offset information when you compile a program. See the discussion of the LIST+ (compiled listing) compiler option in the Apple III Pascal Programmer's Manual.

In the table that follows, most of the error numbers correspond to error-message numbers in the UCSD Pascal system on which the Apple III Pascal System is based. Missing numbers correspond to messages that are not used in the Apple III Pascal System.

Those errors listed as fatal either cause the system to warm boot itself or, if the error was totally lethal to the system, force you to cold boot the system. All other errors cause the system to re-initialize itself by invoking the system Initialize command. This usually happens after you respond to a message that tells you to press the spacebar to continue.

Error Number	Error Message and Description
1	Invalid index, value out of range for string or subrange. Does not occur if RANGECHECK- compiler option used.
2	No segment: bad code file. File reads correctly from disk, but is not a valid segment.
3	Procedure not present at exit time: attempt to EXIT from a procedure that was not previously called or active.
4	Stack overflow: the program data space has exceeded available user memory.
5	Integer overflow. Long integer arithmetic gave an intermediate result greater than 36 digits or final result was assigned to variable of insufficient size.
6	Divide by zero.
8	User break: "break" (CONTROL-\) key pressed or HALT instruction executed.
9	System I/O error: error in attempting to read an operating system segment from disk.
1Ø	User I/O error: error when user's program attempted a blockread, blockwrite, get, or put. If file SYSTEM.PASCAL available, this error is further reported as an I/O ERROR (See table of I/O Error Messages, below).
11	Unimplemented instruction: codefile has probably been damaged.
12	Floating point math error: error in real number format, overflow, underflow, etc.
13	String too long: attempt to store a source string into a destination string of insufficient size.

I/O Error Messages

This table lists all the error numbers that can be returned by the IORESULT function described in the Apple III Pascal Programmer's Manual chapter INTRODUCTION TO FILES AND I/O. The notation (SOS) in the table indicates an error reported by SOS; most of the SOS errors are extremely unlikely under the Pascal system, and are included here for completeness only.

Ø	No error; normal I/O completion
2	Bad unit number
3	Illegal operation, e.g., read from .PRINTER
5	Lost unit -- no longer on line
6	Lost file -- file is no longer in directory
7	Illegal pathname
8	No room -- insufficient space on volume
9	No unit -- unit is not on line
1Ø	No such file in specified directory
11	Duplicate pathname
12	Attempt to open an already open file
13	Attempt to access a closed file
14	Bad input format -- error in reading number
15	Ring buffer overflow -- input arriving too fast
16	Write-protect error -- volume is protected
19	Too many files open for system to handle
32	(SOS) Invalid request code
34	(SOS) Invalid control-parameter list
35	(SØS) Character device not open
36	(SØS) Device not available
37	(SØS) Resource not available
44	(SØS) Invalid byte count
45	(SØS) Invalid block number
46	(SØS) Disk switched
48...63	(SØS) Device-specific error
64	(SØS) Device error -- bad address or data on volume
65	(SØS) Too many character files open
66	(SØS) Too many block files open
67	(SØS) Invalid file reference number
73	(SØS) Directory full
74	(SØS) Incompatible file format
75	(SØS) Unsupported storage type
76	(SØS) Attempted read past end of file
77	(SØS) File position out of range
78	(SØS) Illegal access attempted

79 (SØS) User's buffer too small
8Ø (SØS) File busy
82 (SØS) Volume format is not SOS
83 (SØS) Invalid value in list parameter
84 (SØS) Out of memory for SOS system buffer
85 (SØS) Buffer table full
86 (SØS) Invalid system buffer parameter
87 (SØS) Duplicate volume error
123...127 (SØS) System call error number [error - 122]

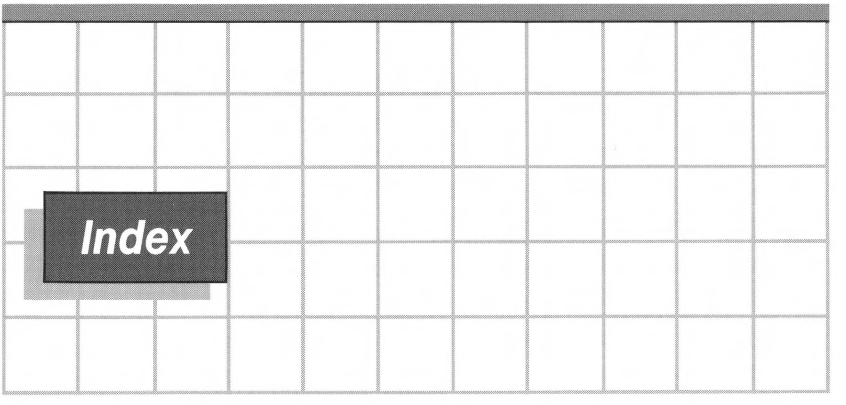
Assembler Error Messages

When the Apple III Pascal Assembler discovers an error in your assembly language routine, it displays an error message taken from the file ERRORS.65Ø2. If the file ERRORS.65Ø2 is not available in any drive, the Assembler will report errors by number only.

The error message for each Assembler error number is given in the table below. If you wish, you can gain some additional diskette space by removing the file ERRORS.65Ø2 and looking up error numbers in this table.

1: Undefined label
2: Operand out of range
3: Must have procedure name
4: Number of parameters expected
5: Extra garbage on line
6: Input line over 8Ø characters
7: Not enough .IF's
8: Must be declared in .ASECT before used
9: Identifier previously declared
1Ø: Improper format
11: .EQU expected
12: Must .EQU before use if not to a label
13: Macro identifier expected
14: Word addressed machine
15: Backward .ORG currently not allowed
16: Identifier expected
17: Constant expected
18: Invalid structure
19: Extra special symbol
2Ø: Branch too far
21: Variable not PC relative
22: Illegal macro parameter index
23: Not enough macro parameters
24: Operand not absolute
25: Illegal use of special symbols
26: Ill-formed expression
27: Not enough operands
28: Cannot handle this relative expression
29: Constant overflow
3Ø: Illegal decimal constant

31: Illegal octal constant
32: Illegal binary constant
33: Invalid key word
34: Macro stack overflow: 5 nested limit
35: Include files may not be nested
36: Unexpected end of input
37: This is a bad place for an .INCLUDE file
38: Only labels & comments may occupy column 1
39: Expected local label
4Ø: Local label stack overflow
41: String constant must be on one line
42: String constant exceeds 8Ø characters
43: Illegal use of macro parameter
44: No local labels in .ASECT
45: Expected key word
46: String expected
47: Bad block, parity error (CRC)
48: Bad unit number
49: Bad mode, illegal operation
5Ø: Undefined hardware error
51: Lost unit, unit is no longer on-line
52: Lost file, file is no longer in directory
53: Bad title, illegal file name
54: No room, insufficient space on disk
55: No unit, no such volume on-line
56: No file, no such file on volume
57: Duplicate file
58: Not closed, attempt to open an open file
59: Not open, attempt to access a closed file
6Ø: Bad format, error in reading real or integer
61: Nested macro definitions illegal
62: '=' or '<>' expected
63: May not .EQU to undefined labels
64: Must declare .ABSOLUTE before 1st .PROC
76: Index register required
77: 'X' or 'Y' expected
78: Zero-page address required
79: Illegal use of register
8Ø: Index register expected
81: Ill-formed operand
82: 'X' expected for indexed addressing
83: Must use 'X' index register



A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

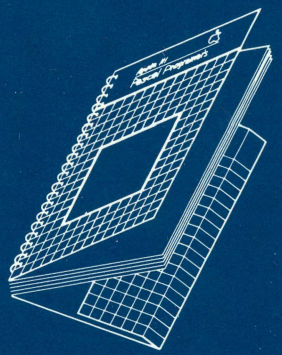
Y

Z

$

1,2,3,4,5,6,7,8,9,0

Tuck end flap
inside back cover
when using manual.



Apple III Pascal: *Program Preparation Tools*

apple computer
10260 Bandley Drive
Cupertino, California 95014
(408) 996-1010

030-0239-A