Apple III

COBOL

Introduction and Operating System Manual

Customer Satisfaction

If you discover physical defects in the manuals distributed with an Apple product or in the media on which a software product is distributed, Apple will replace the documentation or media at no charge to you during the 90-day period after you purchased the product.

In addition, if Apple releases a corrective update to a software product during the 90-day period after you purchased the software, Apple will replace the applicable diskettes and documentation with the revised version at no charge to you during the six months after the date of purchase.

In some countries the replacement period may be different; check with your authorized Apple dealer. Return any item to be replaced with proof of purchase to Apple or an authorized Apple dealer.

Limitation on Warranties and Liability

Even though Apple has tested the software described in this manual and reviewed its contents, neither Apple nor its software suppliers make any warranty or representation, either express or implied, with respect to this manual or to the software described in this manual, their quality, performance, merchantability, or fitness for any particular purpose. As a result, this software and manual are sold "as is", and you the purchaser are assuming the entire risk as to their quality and performance. In no event will Apple or its software suppliers be liable for direct, indirect, incidental, or consequential damages resulting from any defect in the software or manual, even if they have been advised of the possibility of such damages. In particular, they shall have no liability for any programs or data stored in or used with Apple products, including the costs of recovering or reproducing these programs or data. 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.

Copyright

Product Revisions

Apple cannot guarantee that you will receive notice of a revision to the software described in this manual, even if you have returned a registration card received with the product. You should periodically check with your authorized Apple Dealer.

Simultaneously published in the U.S.A and Canada.

Reorder Apple Product #A3D0021

Apple III COBOL

Introduction and Operating System Manual

Acknowledgements

COBOL is an industry language and is not the property of any company or group of companies, or of any organization or group of organizations.

No warranty, expressed or implied, is made by any contributor or by the CODASYL Programming Language Committee as to the accuracy and functioning of the programming system and language. Moreover, no responsibility is assumed by any contributor, or by the committee, in connection herewith.

The authors and copyright holders of the copyrighted material used herein:

> FLOW-MATIC (Trademark for Sperry Rand Corporation) Programming for the Univac® I and II, Data Automation Systems copyrighted 1958, 1959, by Sperry Rand Corporation; IBM Commercial Translator Form No. F28-8013, copyrighted 1959 by IBM; FACT, DSI27A5260-2760, copyrighted 1960 by Minneapolis-Honeywell.

have specifically authorized the use of this material in whole or in part, in the COBOL specifications. Such authorization extends to the reproduction and use of COBOL specifications in programming manuals or similar publications.

Contents

3 The COBOL System and Main Command Line 39

4 Compiler Directives 57

5 Application Design and Development 69

6 Apple III Device Control 95

Appendices

Preface

This manual, together with its companion manual, the *Apple III COBOL Language Reference Manual,* contains the information you need to write and run COBOL programs on the Apple *III*. The material in these two manuals is presented in a condensed format in the *Apple III COBOL Quick Reference Guide.*

This manual is not a tutorial for the COBOL language; if you are unfamiliar with COBOL, you should read a good COBOL textbook. The manual does contain some discussion of COBOL statements, primarily where the information is specific to the Apple *III* implementation.

If you are unfamiliar with the Apple *III* computer, we encourage you to read the *Apple III Owner's Guide.* However, you can start reading this manual and just refer to the *Owner's Guide* as you need it.

The parts of the Apple *III* COBOL system covered here include:

The Run-Time System, used to handle files and to control the compilation, execution, and debugging of COBOL programs.

The COBOL Compiler, used to translate files of COBOL source code into intermediate code for execution by the Run-Time System.

The system interface to the console keyboard and display, and to other peripherals available on the Apple *III*.

The FORMS2 utility, used to generate COBOL source code for screen-oriented interaction between a COBOL application program and its operators or users.

The Animator, a screen-oriented interactive debugger used to study and debug COBOL programs during execution.

Content of this Manual

Chapter 1 is a basic orientation to the COBOL system and to the Apple /// computer; it does not assume any knowledge of COBOL.

Chapter 2 uses the sample programs on the /DEMO disk to examine the most basic features of the COBOL system; it is a tutorial intended to familiarize you with the features of the system. The first example illustrates how a non-programmer can use the system to generate simple but useful filing programs.

Chapter 3 describes the COBOL system's Command Line options, and gives general instructions for running programs and setting run-time switches. It also explains how the operator can interact with a running program.

Chapter 4 describes Compiler directives to control listings and object code files, and discusses other operational aspects of the Compiler.

Chapter 5 describes the program design and development process, with special reference to program editing, the subprogram CALL facility, and segmentation (overlays).

Chapter 6 describes Apple /// facilities for interaction with the console and other devices attached to the computer.

Chapter 7 decribes the FORMS2 utility program, which automates the most common programming tasks in console interaction.

Chapter 8 describes the Animator, a screen-oriented tool for run-time debugging of COBOL programs.

Appendix A is an alphabetical list of Compiler directives available in Apple /// COBOL.

Appendix B lists all error messages produced by the Apple /// COBOL Compiler.

Appendix C lists all error messages that can be signaled by the COBOL Run-Time System during program execution.

Appendix D summarizes the use of the FORMS2 utility.

Appendix E summarizes the use of the Animator.

Appendix F describes file naming conventions and formats used by Apple /// COBOL under SOS (the Apple /// Sophisticated Operating System).

Appendix G discusses the implementation limits of the Apple /// COBOL system and considerations for transferring existing COBOL programs onto the Apple /// system.

Appendix H gives specific information for such transfers using the ACCESS /// utility.

Symbols Used in this Manual

Throughout this manual, you will see illustrations of the Apple /// display as it appears during use of the COBOL system. There are also numerous gray boxes, showing just a few lines from the full display.

The tutorials in Chapters Two, Seven, and Eight, as well as a number of examples elsewhere, expect you to interact with the system by typing

commands or responding to prompts. To distinguish what you type from the text that the system puts on the display, the examples show what you type in boldface.

At any point in the manual, you may see one of the following symbols:

The helping hand indicates an especially useful or noteworthy piece of information.

The eye means "watch out." It warns of a potential hazard.

1

Overview—Getting Started with Apple /// COBOL

This chapter presents an overview of the Apple /// COBOL Language System. It tells you what you need to use the system, and points to other places in this manual and other Apple /// manuals where you can find supplementary information. Even if you are familiar with the Apple ///, you should at least skim this chapter before proceeding. Then, you can go ahead to the tutorial and demonstration programs in Chapter 2 or to the detailed reference material in later chapters.

General Introduction

COBOL (COmmon Business Oriented Language) is the most widely used programming language for commercial and administrative data processing applications. Apple /// COBOL, ANSI standard and certified by the Federal Compiler Testing Center at High Intermediate level, is a full-featured implementation including the most frequently used modules.

Most COBOL programmers currently work in a large computer ("mainframe") environment with a batch-oriented operating system. While Apple /// COBOL is compatible with traditional mainframe applications, it also extends the language to the interactive personal computer environment, providing COBOL programmers with immediacy not possible on a mainframe. The FORMS2 source-code generator feature, for example, lets you begin with a blank screen, create and edit data entry screens, and complete your programming with a fully-operational program. Using the Animator Option you can conveniently debug your program at

the source level, maintaining complete control while you actually watch execution on your monitor screen.

Most COBOL programs developed on other systems can be run on the Apple ///. Appendices G and H include complete information about transporting COBOL programs to the Apple ///.

Apple /// COBOL is based on the ANSI COBOL as specified in "American National Standard Programming Language COBOL" (ANSI X3.23-1974), and fully implements the following modules at Levels 1 and 2:

Nucleus

Table Handling

Sequential Input and Output

Relative Input and Output

Inter-Program Communication

Sort Merge

In addition, the following modules are fully implemented at Level 1:

Segmentation

Library

Debug

Using Apple /// COBOL, a COBOL source file may be created with any Apple /// text editor that will produce ASCII or Pascal TEXT files. A program may also be entered directly from the keyboard. The COBOL Compiler generates an intermediate code file (more than one if the program is segmented) and a listing file that includes any error messages. The intermediate code is then interpreted by the Run-Time System (RTS).

The COBOL Run-Time System uses SOS, the Apple /// Sophisticated Operating System, to handle files and the devices (disk drives, printers, and the keyboard and display) attached to the computer. SOS provides an integrated and consistent way for your programs to interact with each other and with the outside world. It also provides a common framework for all Apple /// language systems, such as Pascal and Business BASIC as well as COBOL.

The COBOL Run-Time System does not use the same interpreter as the Apple /// Pascal Language System. TEXT or ASCII files created by the Pascal system can be read by the COBOL system, but programs generated by the Pascal system will not run on the COBOL system. You will need to load ("boot") the Pascal system to run Pascal programs or any of the utilities provided with the Pascal system. In particular, you will need to boot another system to edit programs.

The Apple /// COBOL System includes utilities for copying files, listing directories, typing files to the console, deleting files, etc. For more complex tasks (for example, formatting disks or reconfiguring the system) you will need to use the SOS Utilities, described in Chapter 4 of the *Apple /// Owner's Guide.*

Hardware Requirements

COBOL programs can be executed on any Apple /// computer; however, for development work and for large programs, it is useful to have 256K of random access memory and one or more external disk drives. The Apple /// has one built-in disk drive and it allows multiple external drives of several types: floppy disk drives such as the Disk II or Disk /// and larger (hard disk) mass storage devices such as the ProFile. The SOS operating system can be configured to drive other devices as well. See Chapter 4 of the *Apple /// Owner's Guide* for information about the System Configuration Program utility; for details on individual devices see the *Apple /// Standard Device Drivers Manual* or the manuals that come with the device, for example, the *Apple /// ProFile Owner's Manual.*

Reconfiguring the System

The COBOL Language System is initially configured to drive the console (keyboard input and display output), the built-in disk drive, and two external floppy disk drives. The configuration doesn't include a driver for a printer or for a large disk, such as a ProFile. If you have such devices, you will want to reconfigure your SOS.DRIVER files to include them. (Information about reconfiguring your system is included in Chapter 4 of the *Owner's Guide* and in the *Standard Device Drivers Manual.*)

Note to ProFile Users:
If you have a ProFile or other large disk, you should move all the COBOL system files onto it from the release disks; this will speed up operations and save your having to shift floppies in and out of the smaller drives. Files that should be transferred are those in Table 1-1 that are indented under the COBOL/ sub-directories. To copy these files, first make a sub-directory COBOL/ on the large disk; this sub-directory should be at the first level on the disk. Then copy the COBOL/ files from the system disks to this new sub-directory on the larger disk. Note that the files ADIS, ISAM and UTIL that appear on several of the disks are identical; they are repeated as a convenience to simplify development using smaller disks. You only need one copy of them on a large disk. For instructions on making a sub-directory and copying files, see the *Apple III Owner's Guide,* Chapter 5: "Operations on Files".

Although the SOS.DRIVER file provided is configured for two external disk drives, the examples in this manual will assume only one external drive. In general, if you have the two external drives, you will not have to follow the directions to swap diskettes. The Apple *III* COBOL system is designed to find the system files it needs in any drive that has been configured.

COBOL Disk Contents

The COBOL system comes with four write-protected disks, /COBOLBOOT, /COMPILER, /ANIMATOR, and /FORMS2. It also includes a disk named /DEMO with several sample programs. Before you start Apple /// COBOL for the first time, make a copy of each of these disks. In addition, you should always keep two copies of important files: one for everyday use and one for backup. If something happens to the working copy, you can go to the backup, make a new working copy and proceed. For instructions on copying a disk, see the *Apple /// Owner's Guide,* Chapter 4. Table 1-1 is a list of the files on these disks. The indentation below COBOL/ indicates that this name stands for a sub-directory containing the files indented below it.

The Compiler, Animator, and FORMS2 are segmented programs; that is, they all have a "root" segment that is loaded when you invoke them by giving a command to the Run-Time System, plus "overlay" segments that are loaded at various times during their execution. The Run-Time System looks for these overlays in the same directory in which it found the root; do not move these disks during a run of the Compiler, the Animator, or FORMS2.

The SOS files (SOS.KERNEL, SOS.DRIVER, and SOS.INTERP) don't need to be on-line after the system has been loaded and the COBOL Command Line is displayed.

The /FORMS2 disk contains numerous files. Several of these are "screens" displayed during a run of FORMS2 (the ones ending ".I01", ..., ".W02"); these are all ASCII files and can be listed on the console or a printer. Several others (".CH1", ".CH2", ".GN1", and ".GN2") are COBOL source code; they are used with FORMS2 to create complete COBOL programs to check FORMS2 screen output or for maintaining simple indexed files. These files are also ASCII files; COBOL programmers may want to list these and study them as examples of Apple /// COBOL. They are also duplicated on the /DEMO disk, for convenience in demonstrations in this manual.

COBOLBOOT	COMPILER	ANIMATOR	DEMO	FORMS2
SOS.KERNEL	COBOL/	COBOL/	PI.CBL	COBOL/
SOS.DRIVER	COBOL	ANIM	STOCK1.CBL	FORMS2
SOS.INTERP	COBOL.D00	ANIM.D00	STOCK2.CBL	FORMS2.ISR
COBOL/	COBOL.ERR	ANIM.ISR	DEMO.CBL	FORMS2.I51
ADIS	COBOL.ISR	ANIM.I01	COBOL/	FORMS2.I52
ISAM	COBOL.I01	ANIM.I02	FORMS2.CH1	FORMS2.I53
UTIL	COBOL.I02	ANIM.I03	FORMS2.CH2	FORMS2.CH1
	COBOL.I03	ANIM.I08	FORMS2.GN1	FORMS2.CH2
	ADIS	ANIM.I09	FORMS2.GN2	FORMS2.GN1
	ISAM	ADIS		FORMS2.GN2
	UTIL	ISAM		FORMS2.I01
				FORMS2.I02
				FORMS2.H01
				FORMS2.H02
				FORMS2.H03
				FORMS2.H04
				FORMS2.W01
				FORMS2.W02
				ADIS
				ISAM

Note: the files ADIS, ISAM and UTIL are included on several disks. These files control certain aspects of the COBOL Run-Time System (COBOL ACCEPT/DISPLAY extensions, Indexed Sequential File handling, and the COBOL System Utilities). These files must be on-line (i.e., accessible in some disk drive) when the features they control are used.

Table 1-1. COBOL System Disk Contents

SOS Files

Table 1-1 shows that files on the Apple /// have a hierarchical structure; each disk has a name (its "volume name") and contains a set of files. The volume name in fact refers to a "directory" on the disk that tells the system what files the disk contains and where to find them. Some of these files in turn can be sub-directories to other files, and so on to arbitrary levels.

In addition, SOS thinks of each device on the system as a file and refers to it by the name of its driver (.CONSOLE, .PRINTER, .PROFILE, etc.) You can call a disk by its volume name (for example, /COBOLBOOT) or by the name of the drive it is in at any given time (for example, .D1 for the built-in drive).

In general, SOS files have names composed of letters, digits and the special characters "/" (slash) and "." (period). As an illustration of the file hierarchy, Figure 1-1 shows the structure SOS sees when the /COBOLBOOT disk is in the built-in drive.

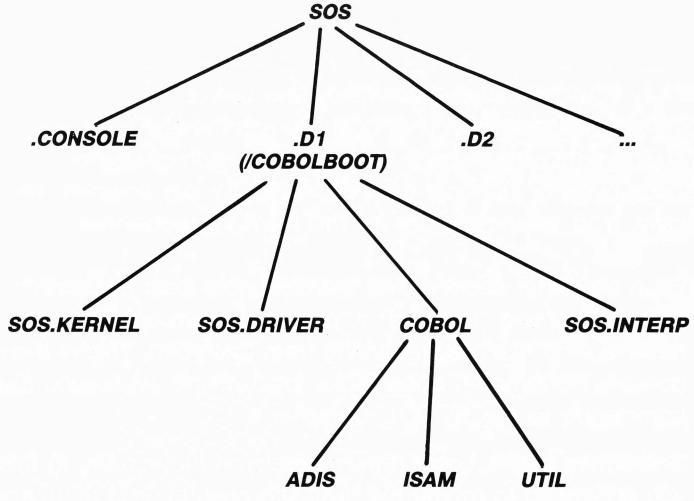


Figure 1-1. SOS File Hierarchy

Using the Apple III Console

Data and program entry on the Apple *III* is based on habits you probably acquired from typing, keypunching, or terminal use on other systems. The main body of the keyboard is similar to a typewriter keyboard, with a few extra keys. To the right of the main keyboard is a "numeric keypad" like that of a calculator. Four keys with arrows pointing up, down, left and right form the lower right corner of the main keyboard. These are "cursor control keys": they control the position on the display of the solid rectangle that marks the next expected input. The ALPHA LOCK key is in the lower left corner. This isn't quite the same thing as a typewriter's shift lock—it does shift the alphabetic keys to upper-case, but numbers and punctuation keys still work normally. With the ALPHA LOCK key down, you must still shift to get an "!" instead of a "1", for example.

To enhance readability, Apple *III* COBOL recognizes lower case letters; however, standard ANSI COBOL does not. If you want to write standard programs (for example, for portability reasons) on the Apple *III*, you may want to push the ALPHA LOCK key down whenever you write a COBOL program.

SOS is responsible for handling the characters you type. It does make some allowances for human fallibility. That is, you can correct your typing mistakes before a program starts doing dreadful things with them. Most programs ask the console for a line of input at a time; until you press the RETURN key you can usually backspace and correct your errors. To backspace over characters you've typed, press the LEFT-ARROW key (the key to the right of the space bar) once for each character you want to backspace over, or just hold the LEFT-ARROW key down—it will repeat automatically. You will notice one of the following behaviors:

- Each character backed over will disappear from the screen. In effect, you have "untyped" it; once you have untyped back to the mistake, you can resume typing from that point. This is the normal console input mode, observable, for example, when you give commands to the COBOL Compiler.
- Characters backed over remain visible on the screen. In this case, you only need to retype the mistaken characters. When you press RETURN, the whole line is sent, just as you see it, to the program. Of course, if you typed an extra character or omitted a

character, you may need to retype more than one character to correct the line. This mode is typical of the COBOL Utilities options; you can observe it by experimenting with listing directories or typing files. See the examples in Chapter Two or refer to Chapter Three for details.

If you completely change your mind about what you want to type, or if there are numerous mistakes throughout a line, you can conveniently erase the whole line and start over at any time before you press RETURN. To cancel a line of input, type CONTROL-X by holding down the CONTROL key on the left of the keyboard and pressing the X key.

SOS never forces you to use upper-case letters. When choosing commands from the COBOL command line, answering prompts at the Utilities level, or giving directives to the Compiler, you may use either lower-case or upper-case letters.

For more information on SOS files, specifically on console input, see Chapter 3 in this manual, or refer to the *Apple /// Owner's Guide.*

Running the COBOL System

This section explains how to start up Apple /// COBOL. Note that the actual keystrokes you will need to type are shown in boldface type.

To start ("boot up") Apple /// COBOL, insert a copy of the disk labelled /COBOLBOOT into the built-in drive and turn on your monitor and the Apple ///; or if the power is already on press the **CONTROL** key and the **RESET** button (inset at the top of the keyboard, near the built-in drive). The built-in drive's red IN USE light will come on, and the drive will whir for a bit, and give a buzz for any external drive that is empty or has its door open. Then the screen illustrated in Figure 1-2 is displayed.

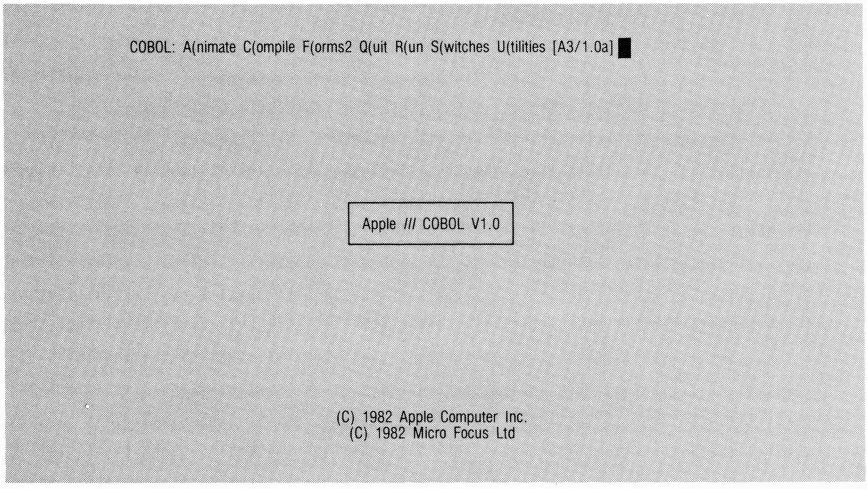


Figure 1-2. System Startup Screen

The top line of this display lists the options available to you; this manual refers to it as the "COBOL Command Line". We will study the commands in detail in Chapter Three; for now a simple example will suffice to show how the options are invoked. The cursor is placed at the end of the command line, waiting for you to type one of the initial letters shown: A, C, F, Q, R, S, or U.

Type **U** (for Utilities). The start-up screen disappears and the top line is replaced by the Utilities menu:

```
Util: C(opy,D(ate,L(dir,E(xt-dir,P(refix,R(emove,T(ype,Q(uit
```

This line lists the options available at the Utilities level; to start, type **L** (for List directory). Underneath the menu for the Utilities, you will now see:

```
List what directory ? /COBOLBOOT
```

with the cursor positioned over the "/". Just press **RETURN**; you should see information about the files on the /COBOLBOOT disk. Figure 1-3 shows what the display should look like. Note that only the first level of the file hierarchy on the disk is shown; none of the files contained in the subdirectory COBOL appear.

```
Util: C(opy,D(ate,L(dir,E(xt-dir,P(refix,R(emove,T(ype,Q(uit
/COBOLBOOT                          Size      Modified    Time    File type  EOF(bytes)
  COBOL                               4    30-Jul-82    16:22    Directory        2048
  SOS.KERNEL                         44    01-Feb-82    00:00    Sosfile         22016
  SOS.INTERP                         65    02-Aug-82    14:06    Datafile        32768
  SOS.DRIVER                         14    02-Aug-82    13:50    Sosfile          6656
4 files / 127 blocks listed; 69 blocks available out of 280
```

Figure 1-3. /COBOLBOOT Directory Listing (First Level)

To conclude our initial demonstration, type **Q** (for Quit) to exit the Utilities; the COBOL Command Line will reappear. Type **Q** once more, and the system will halt and wait for you to reinitialize it. At this stage you can try out the demonstrations in Chapter Two, or go ahead to the detailed discussions in the rest of the manual.

2

Demonstration Programs

This chapter is a tutorial that uses many of the commands from the COBOL Command Line to help you become familiar with the operation of the system. Most of the examples assume some knowledge of COBOL. The first example does not; its purpose is to illustrate how you can use the system even without mastering COBOL. Subsequent examples explore the use of the COBOL Compiler and the Run-Time System utilities.

This demonstration assumes you are using one external disk drive.

Phone Book (FORMS2 Demonstration)

This example demonstrates how the many pieces of the Apple /// COBOL System work together. The example is given as a step-by-step "recipe", with explanations postponed until subsequent examples. Because we are anticipating information that is discussed later, some of the steps may seem rather mysterious. Please follow the steps exactly! If things don't seem to be going as you expect, don't worry; you can start over by resetting the Apple III—simply press the **CONTROL** key while holding the **RESET** button.

This example uses the FORMS2 utility to create an on-line phone book containing names, address, and telephone numbers. Similar files can be created for other purposes; in fact you will see that an operator with no knowledge of COBOL can follow this "recipe" to set up and maintain files,

possibly translating existing paper files into computer format. Note that only simple files can be handled this way; more complex files and integration of files into a full application system will, of course, require serious programming effort.

Initialization

1. *Set Up.* Locate your backup copies of the /DEMO, /COBOLBOOT, /FORMS2 and /COMPILER disks. Place the /COBOLBOOT disk into the built-in drive, and turn on the power or simultaneously press the **CONTROL** key and the **RESET** button if the power is already on.

   After you see the COBOL start-up screen, type **U** to get the Utilities menu and then **P** to choose the Prefix option. You will see the following message:

```
Current prefix is /COBOLBOOT
New Prefix :
```

   Type **/DEMO** and **RETURN**; then type **Q** to Quit the Utilities. The Prefix now tells SOS to look for files on the /DEMO disk unless you specifically tell it otherwise. Now remove /COBOLBOOT from the built-in drive and insert the /DEMO disk. Place the /FORMS2 disk in your external drive.

2. *Running FORMS2.* Type **F**; this loads and runs the FORMS2 utility program. Note that FORMS2 is a program built on top of the Apple III COBOL System; you can create similar tools for your own programs. FORMS2 then displays the screen (I01), shown in Figure 2-1.

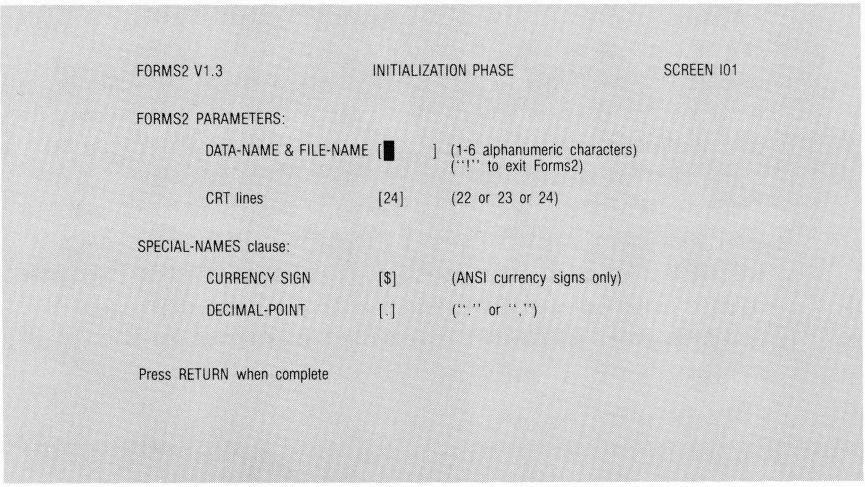


Figure 2-1. FORMS2 Initialization; First Screen

3. *Initialization: Step 1.* The screen shows an area in which you can type up to six characters. Type **PHONE** and then press **RETURN**; this name will be used as the basename for the files generated during this run. If you mistype, you can back up and retype the line any time before you press RETURN. If you have already pressed RETURN, don't worry—whatever you typed is probably an acceptable name anyway. Just use this name instead of PHONE for the rest of the example.

4. *Initialization: Step 2.* The next screen (I02, shown in Figure 2-2) is a "menu" of options, A through G; type **G** (for Generate) followed by **RETURN** to tell FORMS2 you want to generate a program that creates and retrieves records. If you are redoing the demonstration and you reached this point earlier, there may be some old files around with names like PHONE.DDS and PHONE.GEN; if so, you will be asked at this stage if you want to overwrite them. Answer "yes" by typing **Y** and then pressing **RETURN** after each such question. (The same situation may arise at later stages of this demonstration, too; in each case, type **Y** and then press **RETURN** to overwrite the previous file and continue the program.)

```
FORMS2 V1.3                    INITIALIZATION PHASE                    SCREEN I02

FILES TO BE CREATED:

        FILE COMBINATIONS      [ ]     (A = DDS)
                                       (B = DDS & CHK)
                                       (C = DDS & CHK & Snn)
                                       (D = DDS & Snn)
                                       (E = Snn)
                                       (F = No files output)
                                       (G = DDS & Snn & GEN)

DEVICE/DIRECTORY PREFIX (0-40 Chars) [                                      ]


Press RETURN when complete
```

Figure 2-2. FORMS2 Initialization; Second Screen

The First Work Phase

5. *Initializing the Work Screen.* Now a "work screen" (W01; see Figure 2-3) appears. This screen, like the last one, is a "menu" from which you can select any one of the options; in this case, however, just press **RETURN** to select the default option (A), the option the system automatically chooses unless you tell it to do otherwise. Then you can begin creating the screen image you want to appear on the display when the PHONE program is running.

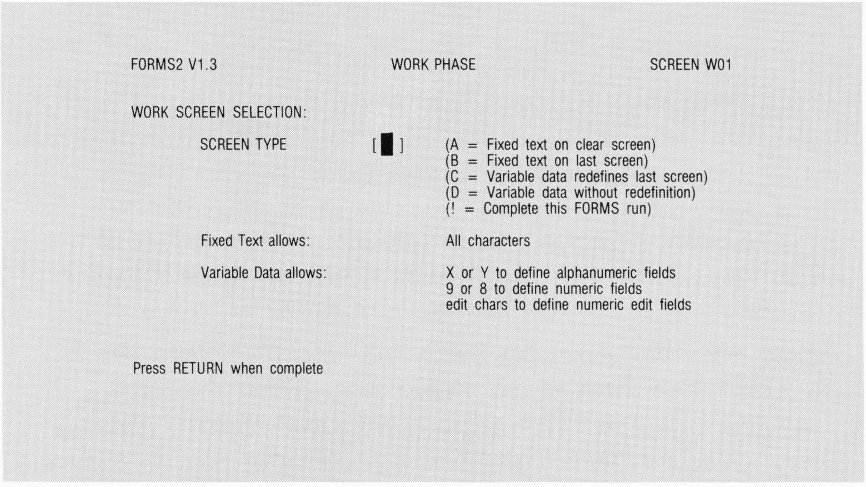


Figure 2-3. FORMS2 Initialization; Third Screen

6. *Labelling the Form.* Now the display clears and the cursor appears in the upper left corner. Note that the bottom line of the screen has a line of hyphens ("-"); this area is used by the program for messages, so you shouldn't use it for the form.

   To create the form, you'll start on the first or second line of the screen and type a "form" suitable for an address book. Your completed form will look like this:

```
Name    [Last:                    ,First:                    ]

Address [                                                    ]
        [                                                    ]
        [                                                    ]
        [State:                                ZIP:          ]

Phone   [(nnn) nnn-nnnn  Ext:                                ]
```

Figure 2-4. "Address Book" Form

To create this screen, simply type the information in Figure 2-4 when we tell you to. But first, here are a few things to keep in mind:

- It isn't critical how many spaces you leave after each of the entries.
- DON'T press RETURN at the end of a line. FORMS2 uses the DOWN-ARROW key (the lower right corner of the main keyboard) for moving between lines, saving the RETURN key for a different function. Press the DOWN-ARROW key twice to leave a blank line between the name line and the address and before the telephone number line. (The other three ARROW keys can be used to move around the screen, too.) If you accidentally press RETURN, at the end of a line, press RETURN again to get back to where you were.
- If you don't like the way a line looks, you can move back and type over it until you are satisfied. You will learn about more complicated editing commands in Chapter 7, which discusses FORMS2 in detail.

Now, type the information shown in Figure 2-4.

7. *Releasing the Screen.* When you have finished typing the form, press **RETURN**, then the **space bar**, and then **RETURN** again. Note that although after the first RETURN the cursor will cover any text in the upper lefthand corner (if you started typing on the first line), or write something there (if you started typing on the second line), no harm has been done. This completes the first phase of work. You will see on the display the COBOL source code created for your screen, a redisplay of the screen, and a message telling you that FORMS2 has created a file named PHONE.S00. When it is done, FORMS2 will ask you to press **RETURN** to go on to the next step.

The Second Work Phase

8. *Reinitializing.* The work screen W01 (Figure 2-3) should now be displayed, with a new option selected: option C. Don't change the option, just press **RETURN**.

   FORMS2 now re-displays the fixed text screen you drew in the first phase. The first phase created the headings that appear on the screen when the program is used; the second phase defines how the user fills in the form with data.

   Variable data is defined using the standard COBOL data symbols X and 9 to stand respectively for alphanumeric data (that is, any key on the keyboard that produces a visible character on the screen) and numeric data (digits 0 to 9).

   What you must do is go through the form once more and mark each position that the operator should be allowed to type into. Type **X**'s over the name and address lines and **9**'s over the phone number and ZIP code areas. (Remember that the keys may be held down to create a whole row of characters very quickly.) When you are done, you should have a screen that looks like this:

```
Name     [ Last: XXXXXXXXXXXXXXXXXX,First: XXXXXXXXXXXXXXXXXX ]

Address  [ XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX ]
         [ XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX ]
         [ XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX ]
         [ State: XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX  ZIP:99999 ]

Phone    [ (999) 999-9999  Ext: XXXXXXXXXXXXXXXXXXXXXXXXXXXXXX ]
```

9. *Separating Key and Data Areas.* FORMS2 needs to know which part of this screen will be the "key" that will be used to keep the records in order, that is, what index to use for the file you are creating.

   The key must be one or more fields beginning with the first field on the screen (in this case, last name). In this example, we'll designate the first and last name as the key. To do this, place the cursor on the first X in the address field and press **RETURN**, then *****, then **RETURN** again. This three-key sequence tells FORMS2 where you want to start the (non-key) data area.

10. *Completing the FORMS2 Run.* To have this screen accepted, the command is the same as in step 7: **RETURN**, **space bar** and **RETURN**. Next you see a warning

```
WORK SCREEN VALIDATION in progress
DO NOT press RETURN
```

while FORMS2 checks that it understands your X's and 9's. Then it shows you the COBOL statements it has created and tells you that it has written the three files PHONE.SO1, PHONE.DDS and PHONE.GEN to the built-in drive. (Press **RETURN** as the system asks you to, as this information is transmitted.) You are now through with the FORMS2 run, and the original COBOL Command Line reappears on the top of the screen.

Running the PHONE Program

11. *Compiling the FORMS2 Output.* Remove the /FORMS2 disk from the external drive and replace it with /COMPILER. To compile the program, type **C**. When the Compiler has been loaded, the cursor will reappear on the screen; type

PHONE.GEN LIST COPYLIST

(and press **RETURN**). Actually, you only need the first part of this line. Typing PHONE.GEN tells the Compiler what file to compile; LIST directs the Compiler to list the COBOL source code on the display as it is being compiled. Notice COPY statements in this code for the standard pieces FORMS2.GN1 and FORMS2.GN2 as well as for PHONE.DDS, which has code specific to this

particular example. Listing the code on the screen just gives you something to watch for a few minutes while the Compiler works. When the compilation finishes, a message appears at the end of the listing:

```
*ERRORS=00000 DATA=01792 CODE=02048 DICT=02909:59165/62074 GSA FLAGS=OFF
```

Some parts of this message may be different, if your form isn't the same as the example. In any case, there should be no errors, and you can go on to the next stage.

12. *Loading the PHONE program.* The cursor should now be on the top line of the screen; type **R** to start the program running. The Run-Time System asks you what program you want to run, prompting you with the message

```
File to run :
```

Type **PHONE** (and press **RETURN**). The program now loads and displays your original screen, placing spaces over all the X's and zeros over the 9's you typed in during the second phase of the FORMS2 run. The program is now ready to accept input. You may find it helpful to move around with the cursor keys for a bit before you try typing a record. The cursor keys will only move in the areas you filled in during the second work phase. Note that the UP- and DOWN-ARROW keys move from field to field, backwards or forwards. Don't press RETURN; that is reserved for when you are done with a record. Finally, note that the ESCAPE key will move you to the start of the key area on the screen. Feel free to experiment; even if you press RETURN, the worst that can happen is for you to accidentally terminate the run. You can always start over at the beginning of this step.

13. *Creating New Records.* Press **ESCAPE** to get to the first character position in the Name field. Type in any last name you want; after typing the last name, you can press the DOWN-ARROW key to move to the First Name field. Or just type spaces or RIGHT-ARROWs to get there. Fill in name, address and phone number. Try typing something other than a number in the ZIP Code or Phone Number field; you'll hear a "beep" and nothing will appear on the screen. Only a digit (0 to 9) will be accepted in a field you defined as numeric (with 9's) during the FORMS2 run.

When you have completed filling in the form, press **RETURN**. You will see, underneath the form, the message

```
NEW RECORD WRITTEN
```

Insert a few more records in this way. Assuming you have changed both the key area (the name) and some of the data (address or phone number), a new record will be written each time.

Note that any data left on the screen from one record is still present to be picked up by the next. So, for example, you may be entering several records for people at the same address; in that case it is convenient not to have to retype the same data. But you have to be sure to blank out (space over) anything that doesn't belong in the new record.

If you just change the key, you can also make a new record with the same address and phone number; for example, as we mentioned, you may want several records for people who all live or work at the same place. To do this, change the last name. Then press **ESCAPE** to put the cursor on the first character of the last name. Then press **RETURN**.

If you just change the key but fail to press ESCAPE or use the ARROW keys to move back to the first characters before you press RETURN, you will see instead the message

```
RECORD NOT FOUND
```

and the data areas will be blanked out. The program thinks you are looking for a record already filed under the name you just typed; pressing ESCAPE first tells it there is no such record—go ahead and make one. The next section explores the options for looking up existing records and for changing or deleting them; for now we are just interested in creating the file in the first place.

14. *Performing Other Operations.* For the address book to be useful, you need to be able to look up a name, to "thumb through" the book when you're not sure of a name, to make corrections or changes to the entries, and to delete entries when they become obsolete. All of this can be done by the program we have generated. In principle, all you do to look up a record is type in the name and RETURN, or just keep pressing RETURNs to get to

any record in the file that is alphabetically later than the one you are at. Table 2-1 shows all of the possible operations; this section will give some brief examples.

To start with, type **AAAA** and press **RETURN**. Probably you will get the RECORD NOT FOUND message, unless you happen to have a record for something like the AAAA-1 Car Repair Shop. In any case, you will now be positioned at the beginning of the file; press **RETURN** a couple of times. The display will show the next couple of records in alphabetical order after AAAA, regardless of the order in which you created them.

Similarly, if you type **N** (and blanks to write over other letters in the key area) the program will, in effect, flip to the second half of the alphabet. The next **RETURN** will then bring up the first record after N in alphabetical order. If you are looking up Rumpelstiltskin, just type the key Rum; then a **RETURN** or two will probably be enough to get to the right entry without your having to type the whole name. For any name you type, the program will either tell you that the file contains no such name or it will display the data that it has stored for that name.

Key Changed	Data Changed	Specified Key Already Exists*	Cursor Position	Result
yes	yes	no	anywhere	NEW RECORD WRITTEN
yes	yes	yes	anywhere	RECORD ALREADY EXISTS
yes	no	no	start of key	NEW RECORD WRITTEN
yes	no	no	elsewhere	RECORD NOT FOUND
yes	no	yes	start of key	RECORD ALREADY EXISTS
yes	no	yes	elsewhere	displays the record
no	yes	no	anywhere	NEW RECORD WRITTEN
no	yes	yes	anywhere	RECORD AMENDED
no	no	no	anywhere	displays next record
no	no	yes	start of key	RECORD DELETED
no	no	yes	elsewhere	displays the record

* Note that if the key has been changed, this column refers to the new key.

Table 2-1. Indexed File (.GEN) Program Operations

Press **RETURN** enough times, and you will "run off the end" of the file. The program will warn you

```
END OF FILE - RETURN WILL TERMINATE
```

Go ahead and press **RETURN**; you will be back at the COBOL Command Line. You can rerun the PHONE program at any time; as in step 12, just type **R** and **PHONE**. Whenever the program is loaded, you will be positioned at the start of the file, and pressing **RETURN** will show you the first record in alphabetical order.

The only way to terminate the program is to press **RETURN** after the final record. However, you can get to the end quickly (without entering a lot of RETURNs) by looking up a key like ZZZZZ; this won't be found unless you have created it as a "sentinel" record at the end, but in any case another couple of RETURNs will then be enough to finish the run.

Now try creating a new record for a name you already have in the file; that is, type the name and some new data for it. The program will not accept this new record; instead it will tell you:

```
RECORD ALREADY EXISTS WITH THIS KEY
```

This behavior prevents you from accidentally writing over an old record. But if you want to change the old record, you can do it, in the following way. Type the key (and **RETURN**) to retrieve the old record; now with the old data showing on the display, type over it to make any changes you want. Now when you press **RETURN** you will see the message

```
RECORD AMENDED
```

and the new version of the record will replace the old one in the file.

Finally, to remove an obsolete record from the file, first retrieve it by typing its key or by stepping to it with **RETURN**s, then press **ESCAPE** to move the cursor to the start of the key and press **RETURN**. This will delete the record from the file and confirm by giving you the message

```
RECORD DELETED
```

This example has shown one way to create a COBOL program for simple file creation and maintenance. FORMS2 actually has much more general use, as a tool to assist a programmer in writing code for console interaction. For a detailed look at FORMS2, see Chapter Seven.

Entering a Program from the Console

Our next example illustrates how the COBOL Compiler accepts input directly from the console and processes it line by line as you type it.

If you haven't done the first example, you can set up for the rest of this chapter as described in step 1 of the previous section; that is, boot with the /COBOLBOOT disk in the built-in drive, change the SOS prefix to /DEMO: type **U**, **P**, **/DEMO**, **RETURN**, and **Q** to load the Utilities package, change the Prefix to **/DEMO** and **Q**uit the Utilities. Remove /COBOLBOOT from the built-in drive and replace it with /DEMO; place the /COMPILER disk in the external drive.

After you type **C** to load the Compiler, you will see this message at the top of the display under the COBOL Command Line:

```
* Apple /// COBOL V1.0 (C) 1982 Apple Computer Inc.
```

The cursor is below this, waiting for your commands to the Compiler; type

.CONSOLE LIST"SAMPLE.CBL"

and press **RETURN**. As in the FORMS2 example, the first word on the line is the name of the source file; .CONSOLE is the SOS name of the console driver file. In this example, you want the Compiler listing to be written to a file rather than sent to the display. That's why you typed LIST"SAMPLE.CBL" and not simply LIST.

The Compiler will signify acceptance of this directive and then prompt you for input:

```
* Accepted - LIST"SAMPLE.CBL"
* Compiling console input
```

Type in the program displayed in Figure 2-5. Remember to type over to column 8 for Area A and to column 12 for Area B. If you mistype something, you can correct it before the end of the line by backing over it with the LEFT-ARROW key. Then press **RETURN** at the end of each line. Once you have pressed RETURN, it is too late to make a correction; the Compiler has accepted the line and is processing it.

```
COBOL: A(nimate C(ompile F(orms2 Q(uit R(un S(witches U(tilities [A3/1.0a]
* Apple /// COBOL V1.0 (C) 1982 Apple Computer Inc.
.CONSOLE LIST"SAMPLE.CBL"
* Accepted - LIST"SAMPLE.CBL"
* Compiling console input

      WORKING-STORAGE SECTION.
      01  GREETING PIC X(8)  VALUE "HELLO!!!".

      PROCEDURE DIVISION.
          DISPLAY GREETING AT 0735.
          STOP RUN.

* ERRORS=00000 DATA=00768 CODE=00256 DICT=00030:10328/10358 GSA FLAGS = OFF
```

Figure 2-5. Console Example Listing

Whether you've made a mistake or not, press **RETURN** after the last line and finish the input by typing **CONTROL-C** (that is, type C while holding down the CONTROL key) and **RETURN**. The Compiler finishes its work and displays the final line shown in Figure 2-5, with the total ERRORS, DATA, and so on. The cursor then reappears at the top of the display, to the right of the COBOL command line.

List the directory on /DEMO to see what the Compiler has done: type **U** to get the Utilities, **L** to list, and press **RETURN** to accept the default /DEMO. You should see some new files—SAMPLE.CBL and some others. The other files all have names with the same base (CONSOL) and different extensions; CONSOL.INT is the intermediate code for running the

program. The other files are used by Animator, the Apple /// COBOL debugger. To examine the listing, type **T**; the Utilities respond by asking you

```
Type what file ?
```

Respond by typing **SAMPLE.CBL** (and pressing **RETURN**); Figure 2-5 should appear on your display. Now type **Q** to quit the Utilities level and get back to the main command line. If there were any error messages in your listing, start over again, entering the program exactly as in Figure 2-5.

Once you have compiled the program without error, it is time to try running it. Type **R**, and answer the prompt

```
File to run :
```

by typing **CONSOL** and pressing **RETURN**, and see the result!

This example has been contrived, but it illustrates a number of points besides the mechanics of invoking the Run-Time System and the Compiler.

- Apple /// COBOL extends the DISPLAY verb of COBOL to allow full use of the display; our message was placed at column 35 of line 7 on the screen.
- Apple /// COBOL, like many implementations, relaxes the rules of standard COBOL on what must be present, and in what order, to constitute a correct program; here, for example, there is no IDENTIFICATION or ENVIRONMENT DIVISION and an abbreviated DATA DIVISION is allowed.
- It is possible, if awkward, to enter a program directly from the keyboard. However, you can't modify a program this way. Console input is very useful for testing one feature of a program, or checking your understanding of some aspect of COBOL or of the Apple /// implementation. You can also save such a program for later recompilation; the Compiler will read SAMPLE.CBL as a source file, ignoring the asterisked lines added during the previous compilation.

In any case, you will want to use an editor for practical program development. See Chapter Four of this manual for suggestions on using Apple /// editors to write COBOL programs.

- Apple /// COBOL allows lower-case letters in COBOL reserved words and data-names. Use of lower-case can be extremely beneficial for legibility; it certainly emphasizes COBOL's similarity to English sentences. Upper-case letters can be reserved for standard usage or for emphasis, as in

  Move FICA-payment to TAX-form-line-13.

Caution: While lower-case letters are much more legible than upper-case, you may have portability problems if you have to move an Apple /// program to an all-upper-case system. See Chapter Five for techniques to convert lower-case program files to upper-case.

Another Example (Calculation of PI)

The /DEMO disk contains several files of COBOL source code. We will use three of these (PI.CBL, STOCK1.CBL and STOCK2.CBL) to demonstrate some of the options available to you in using the Compiler. The cursor should be at the end of the COBOL command line. Type **C** again; after the Compiler's initial message, type

PI LIST FLAG(LOW)

and press **RETURN**. The first part of this line indicates the source file is PI.CBL; the file name extension .CBL can be typed, or it can be left to the Compiler to supply by default, as here. The directives specify that the Compiler will create a listing on the console and that it will "flag" (by underlining in the listing) any line in the source code that does not conform to the low level GSA certification of COBOL implementation. The Compiler will signal its acceptance of these directives, and tell you it has started working, and then list the code on your screen. The last part of this listing is shown in Figure 2-6.

The flags appear in the left margin of the listing, with dashes continuing to the right up to the item flagged. Thus, "H-I" in Figure 2-6 stands for the "High-Intermediate" level of COBOL and indicates items (like the COMPUTE verb or the "<" symbol in comparisons) which are not

implemented in the low level of the COBOL Nucleus. The flags with the "A///" label indicate Apple /// extensions to COBOL (like the optional THEN in the IF sentence or the extended use of DISPLAY in our example). The total number of flags is reported among the counters at the end of the listing.

If you now run this program (as before, at the COBOL Command Line level type **R** and then **PI** in response to the prompt), you you will see the screen updated each time around the LOOP of Figure 2-6 with a new term to be added to the accumulating value of pi.

```
COBOL: A(nimate C(ompile F(orms2 Q(uit R(un S(witches U(tilities [A3/1.0a]
           COMPUTE TERM = (TERM * (N - 2) ** 2) / (4 * N * (N - 1)).
**H-I------------                                                    --
           IF TERM < 0.0000000000001 THEN GO TO HALT.
**H-I----------------                                               --
**A///-------------------------------                              --
           ADD TERM TO PI.
           COMPUTE ED = PI * 6.
**H-I------------                                                    --
           MOVE ED TO DI-PI2.
           MOVE TERM TO ED.
           MOVE ED TO DI-TERM2.
           DISPLAY DI-2.
**A///-----------------                                             --
           ADD 2 TO N.
           IF N < 100 GO TO LOOP.
**H-I------------                                                    --
      HALT.
           STOP RUN.
* Apple /// COBOL V1.0 REVISION 0                        URN AA/0000/HA
* Compiler (C) 1982 Apple Computer Inc.
*
* ERRORS=00000 DATA=02560 CODE=00768 DICT=00512:09846/10358 GSA FLAGS =00010
```

Figure 2-6. Partial Listing of PI Program

Notice that the listing doesn't have sequence numbers in the left margin. Try compiling PI.CBL once again, typing in the following line to the Compiler after you type **C**:

PI LIST RESEQ NOFORM

The RESEQ directive instructs the Compiler to supply sequence numbers in the first six columns of each line. The other new directive, NOFORM, suppresses the page headers that otherwise appear in the listing. You can create a sequenced copy of the source file by directing the listing to disk, for example

```
PI LIST(PI2.CBL) RESEQ NOFORM
```

With this command line, the Compiler will create a sequenced version of the source file. Note that the list file must have a different name from the source file—you can't write the new listing file on top of the existing source code file.

File Handling Example

Our final examples, STOCK1.CBL and STOCK2.CBL, illustrate some of the display-handling capabilities of Apple /// COBOL. This topic is dealt with in detail in Chapter 6; our exploration here will be brief. Return to the top level COBOL command line if you are still on the Utilities level. To compile STOCK1, type **C**, then

```
STOCK1 NOLIST
```

NOLIST is the negative form of the LIST directive; it tells the Compiler not to output any listing file. Wait until the compilation is finished and the Compiler status line appears showing zero errors. Then run the stock file program (type **R** and then **STOCK1**, followed by **RETURN**); you should see Figure 2-7.

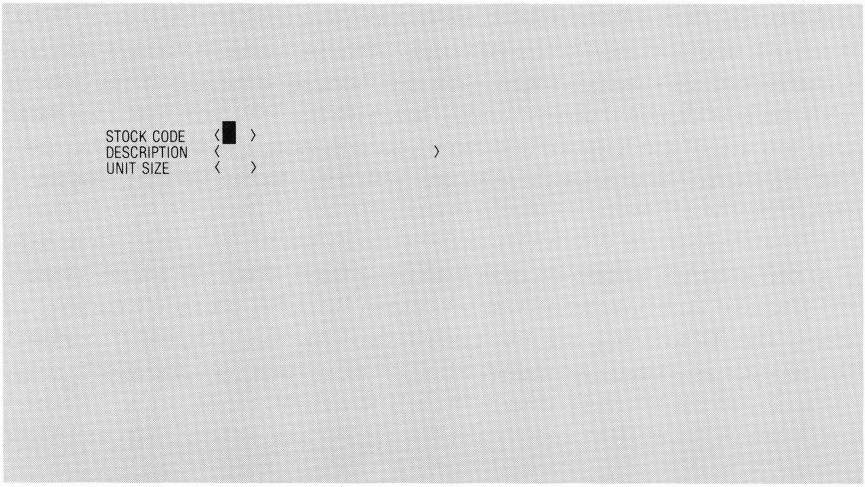


Figure 2-7. Screen from STOCK1 Program

The cursor is positioned at the start of the Stock Code field (which is the key for the index sequential file generated by this program). Type in **A-01**, and you will then see the cursor move to the start of the Description field. Type in something here, for example

Left-hand widget cleaner

After 24 characters of description, the cursor will move down to the Unit Size field. If you have typed more than 24 characters, you can use the LEFT-ARROW key to back up to the start of this field. In any case, the Run-Time System will not let you type in anything but digits into Unit Size; it will beep at you if you try. It will also prevent you from moving into the protected areas outside the angle brackets.

Experiment with the cursor keys, the ESCAPE key, and the TAB key to see what you can do. Getting back to the Unit Size field, type in four digits, say **0012**; then press **RETURN** to finish the first record and to clear the entry fields for a new record. This time try stock code **B-13**, and a short description like

Blivet: 3-pronged

and then press the **DOWN-ARROW** key to go to the Unit Size field. Type 4 for unit size and then press **RETURN**. What happened? You will see the same record still displayed; Unit Size is supposed to be four digits but contains one digit and three blanks. Return to the Unit Size field and step past the 4 (or type it again) and then type a **period**; the field will now be "left zero-filled" and you can write the record out to the file by pressing **RETURN**.

You can continue adding records with other keys; the program will close the files and terminate whenever you press RETURN with the Key field blank. The /DEMO disk will now contain two new files. STOCK.IT contains the records you have generated. You can examine it by using the T(ype command in the Utilities package: type **U** to get the Utilities, **T** to select Type, and **STOCK.IT** to specify the file. Remember to type **Q** to exit the Utilities.

Type can be used to examine any ordinary character file; that is, any file of ASCII characters or any file in the special TEXT format used by the Pascal System Editor. When you list a directory (L option in the Utilities), you will see these files classified as type Asciifile and Textfile respectively. For more information about file types, see Appendix F.

A large file will move down the display screen too fast for study. To pause during the Type operation, hold down the **CONTROL** key and type the **7** key on the numeric keypad; type **7** a second time to resume the display output.

For the final example in this chapter, we will simulate a more realistic COBOL application system by using a second program to interrogate the file produced by STOCK1. Once again, from the COBOL Command Line (which should be visible at the top of the screen after you terminate the STOCK1 program), type **C** to load the Compiler. Now type only the name of the source code file:

STOCK2

This illustrates a final permutation on the LIST directive: a listing file will be output to the /DEMO disk, with the file name STOCK2.LST. That is, unless you tell it otherwise, the Compiler will take the source file name STOCK2.CBL, strip off the extension .CBL, and add .LST to create a name for a file which it uses for listing output. As the compilation proceeds, you will see an error message on the screen. The error has no impact on the program; it was introduced simply to show the format of error messages. The error message will appear in the listing file directly underneath the source code line it refers to; it will also appear on the screen, as in Figure 2-8.

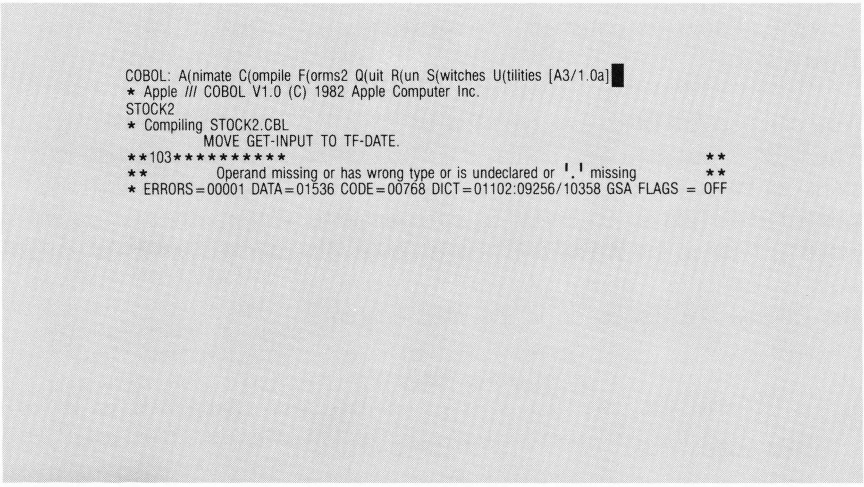


Figure 2-8. STOCK2 Error Message

The STOCK2 program generates a SEQUENTIAL file, STOCK.TRS, of orders against stock items in the file STOCK.IT which you just created with STOCK1. If you run this program, you will see another screen with

fields for entering information. Type one of the stock codes you used in the last example; then type an arbitrary order number, delivery date and number of units. When you press **RETURN**, the description from the original stock record will appear next to the stock code, and the unit size will be multiplied by the number of units to show you the total quantity ordered. You can then accept the order (by typing **Y**, **RETURN**) or reject it (by typing **N**, **RETURN**) and start over. Remember that to terminate the program, you can just press RETURN when the fields are blank.

These two stock file programs illustrate a good part of the screen and file handling capabilities of Apple *///* COBOL. You should find it instructive to play with the programs a bit and to read through the source code to see how the effects are obtained. The full listing of STOCK2.LST is given in Figure 2-9.

You can use the Utilities to type this or STOCK1.CBL onto the display. The Utilities package also contains a Copy utility which you can use to print out these files, if you reconfigure your SOS.DRIVER to contain the driver (.PRINTER or .SILENTYPE) appropriate to your hardware. You will need to refer to the *Apple /// Owner's Guide*, Chapter 4, for details on how to configure a printer into your system. Copy and other functions of the Utilities are discussed in the next chapter.

Listing of the STOCK2 Program

```
* Apple /// COBOL V1.0                STOCK2.CBL                PAGE: 0001
*
    IDENTIFICATION DIVISION.
    PROGRAM-ID. GOODS-IN.
    AUTHOR. MICRO FOCUS LTD.
    ENVIRONMENT DIVISION.
    CONFIGURATION SECTION.
    SOURCE-COMPUTER. APPLE-III.
    OBJECT-COMPUTER. APPLE-III.
    SPECIAL-NAMES. CONSOLE IS CRT.
    INPUT-OUTPUT SECTION.
    FILE-CONTROL.
        SELECT STOCK-FILE ASSIGN "STOCK.IT"
        ORGANIZATION INDEXED
        ACCESS DYNAMIC
        RECORD KEY STOCK-CODE.
        SELECT TRANS-FILE
        ASSIGN "STOCK.TRS"
        ORGANIZATION SEQUENTIAL.
```

```
* Apple /// COBOL V1.0                STOCK2.CBL                PAGE: 0002
*
  /
    DATA DIVISION.
    FILE SECTION.
    FD  STOCK-FILE; RECORD 32.
    01  STOCK-ITEM.
        02  STOCK-CODE PIC X(4).
        02  STOCK-DESCRIPT PIC X(24).
        02  UNIT-SIZE PIC 9(4).
    FD  TRANS-FILE; RECORD 30.
    01  TRANS-RECORD.
        02  TRAN-NO PIC 9(4).
        02  TF-STOCK-CODE PIC X(4).
        02  TF-QUANTITY PIC 9(8).
        02  TF-ORDER-NO PIC X(6).
        02  TF-DATE PIC X(8).
    WORKING-STORAGE SECTION.
    01  STOCK-INWARD-FORM.
        02  PRG-TITLE PIC X(20) VALUE "        GOODS INWARD".
        02  FILLER PIC X(140).
        02  CODE-HDNG PIC X(23) VALUE "STOCK CODE      〈    〉".
        02  FILLER PIC X(57).
        02  ORDER-NO-HDNG PIC X(23) VALUE "ORDER NO      〈      〉".
        02  FILLER PIC X(57).
        02  DATE-HDNG PIC X(24) VALUE "DELIVERY DATE  MM/DD/YY".
        02  FILLER PIC X(56).
        02  UNITS-HDNG PIC X(23) VALUE "NO OF UNITS      〈    〉".
    01  STOCK-RECEIPT REDEFINES STOCK-INWARD-FORM.
        02  FILLER PIC X(178).
        02  SR-STOCK-CODE PIC X(4).
        02  FILLER PIC X(74).
        02  SR-ORDER-NO PIC X(6).
        02  FILLER PIC X(73).
        02  SR-DATE.
            04  SR-MM PIC 99.
            04  FILLER PIC X.
            04  SR-DD PIC 99.
            04  FILLER PIC X.
            04  SR-YY PIC 99.
        02  FILLER PIC X(75).
        02  SR-NO-OF-UNITS PIC 9(4).
    01  CONFIRM-MSG REDEFINES STOCK-INWARD-FORM.
        02  FILLER PIC X(184).
        02  CM-STOCK-DESCRIPT PIC X(24).
        02  FILLER PIC X(352).
        02  UNIT-SIZE-HDNG PIC X(18).
        02  CM-UNIT-SIZE PIC 9(4).
        02  FILLER PIC X(58).
```

```
* Apple /// COBOL V1.0                STOCK2.CBL                PAGE: 0003
*
 /
      02  QUANTITY-HDNG PIC X(14).
      02  CM-QUANTITY PIC 9(8).
      02  FILLER PIC X(58).
      02  OK-HDNG PIC X(3).
      02  CM-Y-OR-N PIC X.
  PROCEDURE DIVISION.
  START-PROC.
      OPEN I-O STOCK-FILE.
      OPEN OUTPUT TRANS-FILE.
      DISPLAY SPACE.
      MOVE 0 TO TRAN-NO.
      DISPLAY STOCK-INWARD-FORM.
  GET-INPUT.
      ACCEPT STOCK-RECEIPT.
      IF SR-STOCK-CODE = SPACE GO TO END-IT.
      IF SR-NO-OF-UNITS NOT NUMERIC GO TO INVALID-ENTRY.
      MOVE SR-STOCK-CODE TO STOCK-CODE.
      READ STOCK-FILE; INVALID GO TO INVALID-CODE.
* VALID ENTRY, CALCULATE AND DISPLAY TOTAL QUANTITY IN TO CONFIRM
      MOVE STOCK-DESCRIPT TO CM-STOCK-DESCRIPT.
      MOVE "UNIT SIZE" TO UNIT-SIZE-HDNG.
      MOVE UNIT-SIZE TO CM-UNIT-SIZE.
      MOVE "QUANTITY IN" TO QUANTITY-HDNG.
      MOVE UNIT-SIZE TO TF-QUANTITY.
      MULTIPLY SR-NO-OF-UNITS BY TF-QUANTITY.
      MOVE TF-QUANTITY TO CM-QUANTITY.
      MOVE "OK?" TO OK-HDNG.
      DISPLAY CONFIRM-MSG.
      ACCEPT CM-Y-OR-N AT 1004.
      IF CM-Y-OR-N = "Y" PERFORM WRITE-TRANS.
* CLEAR INPUT DATA ON SCREEN
      MOVE SPACE TO CONFIRM-MSG.
      MOVE "MM/DD/YY" TO SR-DATE.
      DISPLAY STOCK-RECEIPT.
      DISPLAY CONFIRM-MSG.
      GO TO GET-INPUT.
  WRITE-TRANS.
      ADD 1 TO TRAN-NO.
      MOVE STOCK-CODE TO TF-STOCK-CODE.
      MOVE SR-ORDER-NO TO TF-ORDER-NO.
      MOVE GET-INPUT TO TF-DATE.
**103**********                                                    **
**  Operand has wrong data-type or is not declared                **
      WRITE TRANS-RECORD.
  INVALID-ENTRY.
      DISPLAY "NON-NUMERIC NO OF UNITS" AT 0325.
      GO TO GET-INPUT.
```

```
* Apple /// COBOL V1.0                    STOCK2.CBL                    PAGE: 0004
*
  /
    INVALID-CODE.
         DISPLAY "INVALID CODE          " AT 0325.
         GO TO GET-INPUT.
    END-IT.
         CLOSE STOCK-FILE.
         CLOSE TRANS-FILE.
         DISPLAY SPACE.
         DISPLAY "END OF PROGRAM".
         STOP RUN.
* Apple /// COBOL V1.0                                          URN AA/0000/HA
* Compiler ©1982 Apple Computer Inc.
*
* ERRORS=00001 DATA=01536 CODE=00768 DICT=01102:60972/62074 GSA
FLAGS = OFF
```

Figure 2-9. Listing of the STOCK2 Program

3

The COBOL System and Main Command Line

Running under SOS

As you type commands to the Apple /// COBOL System, your input is handled by SOS, the Apple /// Sophisticated Operating System. SOS controls the Apple /// file system and all its on-line peripheral devices. These devices are integrated into the file system by means of "driver" files, so that a programmer or operator can refer to them and use them much like other files. Refer to the *Apple III Standard Device Drivers Manual* for details about .CONSOLE, .PRINTER, .RS232 and other drivers.

The first chapters of this manual have already looked briefly at some of the aspects of SOS, namely console input and output and the file hierarchy. This chapter fills in some of the gaps left by that discussion and refers you to appropriate places for more information.

Type-ahead

Chapter One discussed how to correct typing mistakes made at the keyboard. Here, we take up another SOS service for console input. Whenever you type a key while the system or a program is not expecting input, SOS stores the key code in a type-ahead buffer. When the program does request input from the console, the driver will first send along any characters waiting in this buffer. At this point, the character will be "echoed" on the screen. The character doesn't show up while it is in the type-ahead buffer, but it isn't lost—just pending. Type-ahead is limited to

128 characters, enough so that even a fast typist will not usually get too far ahead of program responses. If you do fill the buffer, you will hear a "beep" from the console when you type the next character. After the program consumes some of the pending input, SOS will accept further keystrokes.

The situation in which you will most likely use type-ahead is in selecting COBOL system commands and Utility options. A single keystroke will select the command and begin loading the appropriate code to handle it; you need not wait for the initial message or intermediate menu to continue your typing. If your request is simple or your typing secure, you can complete the request without waiting for each input character to be displayed.

If you make a typing mistake while typing ahead, there is no harm done—just wait for the program to catch up with you and correct the mistake as usual. You can also erase everything in the type-ahead buffer by pressing the CONTROL key and the 6 on the numeric keypad at the same time.

CONTROL Operations

Recall from Chapter One that CONTROL-X erases a whole line of input (at any point before you press RETURN); this and the buffer erase (CONTROL-6) operation are part of a general scheme of operations invoked by holding down the CONTROL key and typing another key at the same time. (We refer to this operation as CONTROL-k, where "k" stands for the key that is typed while CONTROL is held down.) Some of these control operations are defined as part of the ASCII character-set used by the Apple *III*. For example, the LEFT-ARROW key is simply a convenient substitute for the ASCII backspace character which you also get when you type CONTROL-H.

Another ASCII definition is the CONTROL-C we used in Chapter Two for terminating console source code input to the Compiler; the CONTROL-C is just the ASCII End-of-Text (ETX) character. Also, the CONTROL-X used to cancel a line of input is the ASCII CANCEL code, CAN.

The Apple *III* extends this scheme in a number of ways. One use of CONTROL special to the Apple *III* is to activate the RESET key inset in the top of the keyboard. If you push these two keys at the same time, the

Apple /// will reset (cold start) the system as if you had turned the power switch off and then on again.

Another special operation is CONTROL-\ (that is, CONTROL and the back-slash key in the top right corner of the main keyboard). CONTROL-\ aborts any program running and returns control to the COBOL Command Line.

Note that the COBOL Run-Time System will complete file operations and close any open files before terminating after a CONTROL-\. This means, for example, that when a program reads input from the console, you will have to press RETURN after the CONTROL-\ (in order to end the input operation).

Also note that the CONTROL-\ operation will cause the Run-Time System to issue a post-mortem message; for example, if you abort the PI program of Chapter Two, you will see something like:

```
PI.INT Segment:Root
COBOL PC 00DCH : RTS Error 150
```

The first line of this message names the file running at the time of the CONTROL-\ and the segment (program overlay) that was interrupted. Programs without overlays will always specify the root segment; programs like the Compiler which use overlays may name some segment other than the root. COBOL PC refers to the Run-Time System program counter; in this case it was interrupted at a count of DC (hexadecimal). Error number 150 signifies "Program interrupted by user" (see the list of Run-Time Errors in Appendix C).

Finally, there are some special console controls using the numeric keypad to the right of the main keyboard. Several of these are "toggle" switches—you push them once to get one effect and a second time to turn off the effect. The numeric controls are:

CONTROL-5 Video Output Switch. This toggle switch turns console video output off (when first pushed) and back on again (the next time), and so on. The Apple /// runs about 25 percent faster with the video turned off; so you can speed up programs (notably, long compiles) by using this switch. Note that a console input request from a program will automatically turn the video back on.

CONTROL-6 Flush Type-Ahead Buffer. Typing this removes any characters in the type-ahead buffer (anything you've typed that hasn't yet been sent to a program as input).

CONTROL-7 Output Pause. This toggle switch "freezes" the console screen in its current state; if a program tries to put something new on the screen, it will be suspended until you toggle the switch again to unfreeze the screen.

This control is very useful. Use it to pause at some point in a compilation, with the listing going out to the screen; or use it while using the Type command in Utilities to examine some part of a file in detail and then let the rest go past at top speed. Since most programs interact with the user at the console, this switch is effectively a general purpose PAUSE switch.

CONTROL-8 Control Character Display. This toggle switch causes subsequent control codes to appear on the screen with visible representations (two-character abbreviations squeezed into one character position on the screen).

These control operations do not work with the numeric keys on the main keyboard; you must use the keys on the separate numeric keypad. For more information on console control operations, refer to the Apple *III Standard Device Drivers Manual.*

File Names

SOS Pathnames

SOS files have names composed of letters, digits and the special characters "/" (slash) and "." (period). The letters may be typed in upper- or lower-case, since SOS capitalizes lower-case letters internally. The full SOS name of a file always begins either with a "/", indicating the root directory of a volume currently on-line, or with a ".", indicating a device driver. Devices may be block structured, as in the case of disk drives, or they may be character-oriented, for example .CONSOLE or .PRINTER. A block device may contain many files, organized in the hierarchical fashion illustrated above in Chapter One, Figure 1-2. In the case of a chararacter device, the device name is the full name of the file. For any file on a block device, the full name traces out a "path" in the file hierarchy from the root

to the position of the file. Thus, the full "pathname" of the file ISAM on /COBOLBOOT is:

/COBOLBOOT/COBOL/ISAM or .D1/COBOL/ISAM

Thus COBOL is a "subdirectory" file that contains a list of other files. You can use the device name (.D1 for the built-in drive) instead of the volume name; but the volume name is more general. If you begin with /COBOLBOOT, SOS looks for the volume in each on-line drive until it finds it; you don't need to remember where it is. If SOS can't find the file (if you mistyped the name, or if the disk is not in the device you specified, or not in any device) you will see an error message. For example, if you are trying to run a program, the error message is similar to that from a CONTROL-\ abort. The Compiler and the Utilities use a message like the following to report being unable to find a file:

```
Can't open <file name>
```

It can be inconvenient to type out the full pathname of a file, especially if it is far down in the hierarchy and if you have to keep typing it in many times. SOS maintains a "prefix" which it will place in front of any file name you give it that doesn't begin with a period or a slash. That is, if you type .CONSOLE, or .D2/PI.CBL or /DEMO/STOCK.IT, SOS assumes you have typed a complete pathname. However, if the prefix is /VOL1/DIRA (for example) and you type FILEX, SOS will interpret this to mean:

/VOL1/DIRA/FILEX

Whenever you start or reset the system, the prefix is initialized to the volume name of the disk in the built-in drive. To change this value, use the Prefix Utility as in the example in Chapter One.

COBOL File Name Extensions

The period has another use in filenames besides beginning the name of a device. You may want to group together several files in the same directory with related names. By using a base file name (for example, STOCK1) you can create a cluster of related files with extensions of the name—STOCK1.CBL for the source code, STOCK1.INT for the intermediate code object file, etc.

SOS doesn't care about periods in the middle of file names, but the COBOL Run-Time System commands use this method to differentiate the functions of related files. The system also allows certain default values for these file extensions. Thus, the Compiler expects source file input to have the extension .CBL. It will accept any file name specified in full (for example, XYZ.A.B.PQR), but if no extension is given, it will supply the .CBL that it expects. The most important of the file name extensions used in the COBOL system are:

.ACP	Created by the Compiler for use by the Animator (for handling COPY statements).
.ANM	Animator file (created by Compiler directive ANIM).
.CBL	COBOL source code file; default extension for the Compiler input.
.CHK	FORMS2 screen checkout program source code.
.DDS	Data Description Statements; COBOL source code defining the screens created in a FORMS2 run.
.Dnn	Dictionary files used by both the Compiler and the Animator; one per segment (number nn) in the program.
.GEN	FORMS2 index file generation program source code.
.Inn	Program segment intermediate code (overlays).
.IDX	Index file for Index Sequential file organization (see Appendix F for more details).
.INT	Intermediate code file (root segment) output by the Compiler; default extension for Run.
.ISR	Inter-segment reference file for segmented programs.
.SXn	Work-files for the COBOL Sort-Merge Module (n = 1 or 2).

.Snn FORMS2 screen image files for each screen created in a FORMS2 run (n = 00, 01, 02, ...).

.TMn Temporary Compiler files (n = 1 or 2).

Run-Time System ? Wildcard

If you abort the Compiler (by a CONTROL-\), the RTS Error 150 message indicates that the file name of the intermediate code file for the Compiler was

?/COBOL/COBOL

The Animator, FORMS2, and the Utilities all are invoked with similar names (?/COBOL/ANIM, ?/COBOL/FORMS2, and ?/COBOL/UTIL respectively). The "?" here is a special COBOL (not SOS) feature, telling the Run-Time System to search for the file in all drives on-line. You may use this feature elsewhere, for example in telling the Run command what file to execute, or in a COBOL COPY statement like the one in PHONE.GEN from Chapter Two:

COPY "?/COBOL/FORMS2.GN1".

Turnkey Systems

For an application that runs a single major program, it is useful to be able to run the program in a "turnkey" mode. That is, the operator can place a specially tailored disk in the built-in drive, turn on (or reset) the computer and begin work immediately, without having to interact with the COBOL Command Line. As far as such a user is concerned, the Apple /// has become a dedicated machine for his application.

To create a turnkey system, rename the program you want to run COBOL.START; for a relatively small application it is convenient to place this file on a boot disk (for example, you could copy STOCK1.INT from /DEMO to a duplicate of your /COBOLBOOT disk, renaming it COBOL.START by means of the copy operation). But it isn't necessary for the COBOL.START program to be on the boot disk. As long as it is a first-level directory entry in any volume on-line at the time the Run-Time System starts up (at a reset or when the power is turned on), it will be loaded and executed. Notice that if you switch disks, the COBOL system

may start running a different COBOL.START from another disk. Only if the file COBOL.START is not found will the Run-Time System call its normal instruction analyzer and display the COBOL Command Line.

When the COBOL.START program terminates, either normally (by executing a COBOL STOP statement) or as a result of a CONTROL-\, the Run-Time System will start it again from the beginning as if you had just powered up. This is convenient for a dedicated application, but it also means that you will not be able to execute any of the normal COBOL Command Line options from a turnkey disk. Note, however, that you can use the cold-start assembler call to exit your application (see Chapter Four).

COBOL Command Line Options

Whenever you boot up the Apple /// COBOL System, and whenever a program finishes, either normally or by a CONTROL-\ abort from the keyboard, you will find the cursor positioned at the end of the COBOL Command Line, waiting for you to select one of the command options listed there:

```
COBOL: A(nimate C(ompile F(orms2 Q(uit R(un S(witches U(tilities [A3 1.5a]
```

The remainder of this chapter discusses these commands in detail, except for the first three, which are given detailed treatment in later chapters and so only briefly sketched here. Each of these commands is invoked by typing the key labelled with the command's initial letter (A, C, F, Q, R, S or U); do not press RETURN after the key. If you type any other key, the Run-Time System will "beep" at you and again wait for one of these choices.

Animate

The **A** command invokes the Animator, the COBOL debugger that brings your programs to life. After the Animator has been loaded, you will see its initial message:

```
* A/// Animator V1.0 (C) 1982 Apple Computer Inc.
```

The cursor will reappear underneath this message, awaiting input of the name of the program to be animated. Type the name of the intermediate

code file, usually a name like program.INT; however, any file name is acceptable, for example COBOL.START if you are debugging a turnkey system. If Animator cannot find the file using the name you typed, and if the name didn't end with .INT, the Animator will supply the extension .INT and look again. So you can use the "basename" of the program and let Animator add the .INT by default. If you change your mind and wish to exit the Animator, type **!** when asked for input.

Once Animator has found the intermediate code file, it looks for a source code file and several other files that were output by the Compiler. The Animator expects to find a dictionary (.D00) file corresponding to the intermediate code file; this file contains the name of the original source code file given to the Compiler, which Animator then searches for. Other files may also be needed, with extensions .ACP (for handling COPY statements), or .Inn and .Dnn for handling segment number nn of a segmented program. These files should all be on-line; usually they will be on the same disk, and Animator will look for them on the same drive as it found the intermediate code file on.

However, if Animator doesn't find a file on this drive, it will let you specify a different one. At the bottom of the display, you will see the message

```
FILE NOT FOUND - S(top run) C(ontinue) A(lter drive)
```

with the name of the file in question below this. You can then type **A**, followed by a directory name, and Animator will prefix that name to the file name and look again.

Compile

The **C** command invokes the COBOL Compiler, the subject of Chapter Four. After the Compiler has been loaded, you will see its initial message:

```
* Apple /// COBOL V1.0 (C) 1982 Apple Computer Inc.
```

The cursor will reappear underneath this message, awaiting input of the name of the source code file to be compiled plus optional Compiler directives. If the Compiler can't find the name of the file as you typed it, and if the name you typed doesn't already end with the extension .CBL, the Compiler will append .CBL to the name and try to find it with this extension. Note that if you entered the Compiler by accident, you may type **!** followed by **RETURN** to exit immediately.

Forms2

The **F** command invokes the FORMS2 screen manipulation utility, the subject of Chapter Seven. After FORMS2 has been loaded, you will see the first of a series of initialization screens. See Chapter Two for an example and Chapter Seven for detailed discussion. If you type F by mistake, you can exit immediately from FORMS2 by typing an exclamation point (**!**) followed by **RETURN**.

Quit

The **Q** command exits the Run-Time System. You will see the following message in large letters:

```
INSERT SYSTEM DISK AND REBOOT
```

No further action is possible until you reset the machine. As you need a different run time system (for example, the Apple *///* Pascal System) for editing files, you will in fact switch from one system to another rather frequently. This command is the cleanest way to end a session with the COBOL Run-Time System.

Run

The **R** command loads and executes a program. You will see the prompt line

```
File to run :
```

Type the pathname of the intermediate code file; after you press **RETURN**, the file will be loaded and execution will begin. The command assumes that the file name ends with the extension .INT; if the name you type doesn't end in .INT, the Run-Time System will supply this extension automatically. To suppress this, you must type an extra period (.) character at the end of the name. For example, to run a program named EXAMPLE (not EXAMPLE.INT) you answer the prompt as follows:

```
File to run : EXAMPLE.
```

Switches

The S command displays or sets run-time switches. There are nine switches available for use in tailoring different runs of the same program. One of these switches is the ANSI Standard Debug switch. If this switch is turned off, then any statements in the program with the USE FOR DEBUGGING clause are inhibited; if it is on, they come into effect. The other eight switches, labeled with the digits 0 through 7, may be used for arbitrary purposes in a program. These are tested in a program by means of ON STATUS condition-names and OFF STATUS condition- names defined in the SPECIAL-NAMES paragraph of the ENVIRONMENT DIVISION; see the *Apple III COBOL Language Reference Manual,* Chapter Three, for the definition of the syntax needed in this paragraph. After you type **S** on the COBOL Command Line, you will see below the COBOL Command Line the display

```
0-7, A(nsi C(lear) <ret> Switches ---------
```

with digits 0 to 7 or a letter A showing in place of the corresponding dash when the switch is set. Type **A** or a digit to toggle the switch (set it if it was off, clear it if it was set). You can clear all the switches at once by typing **C**. Press **RETURN** to finish your changes and return to the COBOL Command Line. Once set, the switches remain set until they are changed by a following S command. This facility is provided primarily for compatibility with batch systems.

Utilities

The **U** command invokes the COBOL Utilities package. Once this is loaded, the Utilities menu will appear, and you may select one of the options. Of these, some do file handling (listing directories, deleting files, etc.), and some deal with SOS system parameters. All the options use the LEFT-ARROW cursor control somewhat differently than the rest of the COBOL system. Under the control of the Utilities program the backspace is "non-destructive"; when you back up with the LEFT-ARROW, the characters backed over remain on the screen and they will be sent to the utilities program when you press the RETURN key. Thus you can correct a single letter typing error without having to retype everything past it on the line.

Note also that the Utilities program will throw away any input on the line after its first blank. If you change a long line into a short one, it isn't necessary to overtype the tail end of the line with blanks—one blank at the end of the corrected line is enough to terminate it properly.

The Utilities menu is

```
Util: C(opy,D(ate,L(ist-dir,E(xt-dir,P(refix,R(emove,T(ype,Q(uit
```

The individual options are

- Copy. This option makes a new copy (the target) of any file (the source). After you type **C** to invoke the option, you will be asked:

```
Copy what file ?
```

  and after typing in the SOS pathname of the source file, you will be asked for the target:

```
to what file ?
```

  The target file does not need to be a disk file. To get a hard-copy listing of any ASCII or TEXT file, copy it to a printer (.SILENTYPE or .PRINTER device file). A copy to the device .CONSOLE is essentially the same command as the Type utility. In fact, it is considerably faster.

- Date. This option sets Apple *///* date and time; it displays the current date and time and prompts for your changes with the following line below the Utilities menu:

```
Enter date and time 00- -00 00:00:00
```

  or with the zeros and blanks filled in if you have a clock in your system or have already set the date and time since the last time the Apple *///* was powered up. Type over any field to change it. The blanks above can be replaced by any of the standard three-letter month abbreviations. You may type any combination of lower- and upper-case: internally the field will be translated to an initial capital and two lower-case letters. The fields for day, year, and time will accept any digit values. Press **RETURN** to enter any changes you have made, or to get back to the command level.

Note that if your Apple /// has an internal clock, the time fields will be incremented automatically while the power is on; otherwise, the time may be used as a simple "time-stamp" to make files of an individual work session.

- Ext-dir
- List-dir. Both of these options list a directory on a block device. Ext-dir gives an "extended directory listing", by listing all levels of the file hierarchy under the specified directory; List-dir only lists the files that are directly below the specified directory. The format of an Ext-dir listing is illustrated in Figure 3-1; the corresponding List-dir listing appears in Figure 1-3.

```
Util: C(opy,D(ate,L(dir,E(xt-dir,P(refix,R(emove,T(ype,Q(uit
/COBOLBOOT                    Size      Modified    Time    File type  EOF(bytes)
 COBOL                          4    30-Jul-82   16:22    Directory       2048
  ISAM                         33    20-Jul-82   18:20    Datafile       16384
  ADIS                         11    13-Jul-82   15:00    Datafile        4864
  UTIL                         33    30-Jul-82   13:15    Datafile       16384
 SOS.KERNEL                    44    01-Feb-82   00:00    Sosfile        22016
 SOS.INTERP                    65    02-Aug-82   14:06    Datafile       32768
 SOS.DRIVER                    14    02-Aug-82   13:50    Sosfile         6656
7 files / 204 blocks listed; 69 blocks available out of 280
```

Figure 3-1. Extended Directory Listing

The current prefix appears as the default value in the prompt line

```
List what directory ? /COBOLBOOT
```

(This example shows the prompt as it would be if you booted from the original COBOL System boot disk and haven't changed the prefix since. Whenever you type E or L, you will see the current prefix in this prompt.) If you want to list this directory, just press RETURN. Otherwise, type over the prefix with a disk device name or a directory and then press **RETURN**. As you start typing, the default value will vanish from the prompt line.

- Prefix. This option displays or changes the prefix that SOS attaches to file names not beginning with the "/" or "." characters. The display resulting from this option is similar to

```
Current Prefix is /COBOLBOOT
New Prefix:
```

with a display of the current value of the prefix and the cursor positioned to receive a new value. If you just press **RETURN**, the current prefix will remain in effect. If you type a device name (for example, .d2), SOS will read the volume name of the disk currently on that device as the new value of the prefix string, and you will see this volume name replace the string you typed on the second line of the message. Also if you type a volume name in lower-case, the Run-Time System will display the name in capitalized form on this line. At the initial loading of the Run-Time System, the prefix is set to the volume name of the boot disk in the built-in drive.

- Remove. This option deletes a file (only one at a time; no "wild-card" conventions are recognized). The prompt is

```
File to remove :
```

Type in the pathname of the file to be deleted. You will then be asked to confirm this:

```
Remove <file name> ?
```

Type **Y** to go ahead with the deletion; any other response returns you to the Utilities menu.

- Type. This option lists the contents of a file onto the console. Answer the prompt

```
Type what file ?
```

with the pathname of the file to be listed. With this option, it is possible to use the "?" character as shorthand for "any drive" in place of a device name. To pause at some point during the listing, use the CONTROL-7 operation.

- Quit. This option exits from the Utilities to the main COBOL command line. It is the only way to get out of the Utilities except for a CONTROL-\ abort.

Summary

Command Entry

- Upper- and lower-case letters are equally acceptable.
- Single keystrokes invoke the options at the main command level and the Utilities level; elsewhere, an entry must be terminated by a RETURN.
- Commands may be typed ahead without waiting for the display of intermediate prompts.
- Backing up with the LEFT-ARROW key allows the correction of typing mistakes on the current line.

Control Operations

- CONTROL-C (ASCII ETX) End of file for console input.
- CONTROL-X (ASCII CAN) Erase the current line of input.
- CONTROL-\ Abort a program and return control to the COBOL command line. May require a following RETURN if the system is processing an ACCEPT statement.
- Numeric keypad controls:

CONTROL-5	Turn CRT screen on/off; any output to the console between CONTROL-5's will be lost.
CONTROL-6	Erase any characters typed ahead (and not yet processed as input by a program).
CONTROL-7	Toggle console acceptance of output; the program halts until the next CONTROL-7 allows output to resume from where it was stopped.
CONTROL-8	Toggle the visible representation on the display of control characters in console output.

COBOL Command Line Summary

A Animate command. Invokes the Apple /// COBOL Animator. Type the pathname of the intermediate code file to be animated. Type ! to exit.

C Compile command. Invokes the Apple /// COBOL Compiler. Type the pathname of the source code file (extension .CBL assumed by default) and Compiler directives. Type ! to exit.

F Forms2 command. Invokes FORMS2 utility. Type ! to exit.

Q Quit command. Exits the Run-Time System. No further action is possible until you reset the machine.

R Run command. Loads and executes a program. After the prompt, type the pathname of the intermediate code file (use an extra period to terminate the name if the file name doesn't end with the extension .INT; if it does so end, you don't need to type the extension).

S Switches command. Display or set run-time switches. Displays the current settings, dashes when the switch is clear, a digit (0 through 7) or A when the switch is set. Type A or a digit to toggle the switch (set it if it was off, clear it if it was set). C clears all switches. Press RETURN to exit back to the COBOL command line.

U Utilities command. Invokes COBOL utilities. At this level, backing over characters with the LEFT-ARROW key doesn't erase them; pressing RETURN sends the entire visible line to the utility. Options are:

- Copy - copies any disk file to another SOS file; the target can be a character file (such as .PRINTER) or a disk file. The local name may be changed in the course of the copy; it must be changed if a second copy of the file is made to the same directory as the original.
- Date - sets Apple /// date and time. Displays the current date and time. Type over any field to change its value.
- List-dir - lists an Apple /// disk directory.

- Ext-dir - lists a disk directory and sub-directories. The prompt line gives the current prefix directory; type over it to list a different directory.
- Prefix - sets Apple *///* prefix. Displays the current prefix string; type a new value to change the prefix.
- Remove - removes a file. Type the pathname of the file to be deleted.
- Type - lists a file on the console. Answer the prompt with the pathname of the file to be listed. To pause at some point of the listing, use the CONTROL-7 control.
- Quit - exits from the utilities to the Run-Time System.

4

Compiler Directives

Compiler Command Format

After you have invoked the Compiler by selecting the Compile option from the COBOL Command Line, and the Compiler has issued its initial message, it expects you to type in a command line in the following format:

file name [directive directive ...]

where

file name is the pathname of a file containing Apple *III* COBOL source statements. If this name doesn't begin with a period or a slash, it is appended to the system prefix. The Compiler searches for this (possibly prefixed) name on-line; if it fails to find the name, it will try again with a new name formed by appending ".CBL" to the original name. If the name is still not found, the Compiler will issue the error message

```
Open fail : [prefix/]file-name[.CBL]
```

and return control to the main COBOL command line.

directive is one of the optional Apple *III* COBOL directives described in this chapter. Each directive must be separated by one or more spaces from any preceding directive or file name and from any succeeding directive. If the list of directives is too

long to fit on one line of the screen, it may be continued on a subsequent line by typing **&** (ampersand) followed by **RETURN**. Here is how an example might look:

/PROFILE/COBOL/DEVELOP/LEDGER RESEQ NOFORM & ⟨R⟩

(the line ends with an ampersand and a RETURN). The system will respond to this with

```
* Accepted - RESEQ
* Accepted - NOFORM
* Accepted - &
```

and then the next line may be typed in:

DATE ''JULY 28, 1982 17:05'' NOINT

whereupon the system will respond with

```
* Accepted - date '' July 28, 1982 17:05''
* Accepted - noint
```

and begin compilation.

Directives may appear in any of the following forms:

keyword
keyword[ ](argument)
keyword[ ]"argument"
NO[ ]keyword

The bracketed space means that you may optionally use one or more spaces before an argument or between the NO and the directive it negates. In any case where an argument contains a space, you must use the quotation mark format; otherwise you may use either parentheses or quotation marks.

If you press RETURN without entering a file name (and possibly some directives), the Compiler will assume that the source code will be entered from the console and that default settings are to be used for the directives. If you type in only a file name, the default directives will also be used. Table 4-1 specifies all the directives available in the Apple *III* COBOL Compiler; default directive settings are marked with a "*" and have their arguments filled in with the default values.

Directive	Argument	Negative Directive
* ANIM	—	NO ANIM
BRIEF	—	* NO BRIEF
COMP	—	* NO COMP
COPYLIST	—	* NO COPYLIST
* CRTWIDTH	"128"	NO CRTWIDTH
* DATE**	"literal-1"	NO DATE
* ECHO	—	NO ECHO
ERRLIST	—	* NO ERRLIST
FLAG	"level"	* NO FLAG
* FORM	"60"	NO FORM
IBM	—	* NO IBM
* INT	"basename.INT"	NO INT
* LIST	"basename.LST"	NO LIST
* PRINT	"basename.LST"	NO PRINT
REF	—	* NO REF
RESEQ	—	* NO RESEQ
FORMFEED	"function-name"	—
SYSIN	"function-name"	—
SYSOUT	"function-name"	—
TAB	"function-name"	—

* marks the default value of the directive

— signifies that there is no argument (or, as in the case of DATE, no negative)

basename is the name given for the source file (minus .CBL extension if any)

** If literal-1 is not specified, the system date and time will be used.

Table 4-1. Compiler Directives

Some directives are mutually exclusive. When you type in a command line, the Compiler will scan it to see whether it can accept all the directives; if it rejects any directive (either because it can't recognize it or because of

mutual exclusion) it will print out a message to that effect on the console and return control to the COBOL Command Line. Table 4-2 shows the exclusions in effect; most of these concern various listing directives.

Directive	Excludes
NO LIST (In cases where [NO] appears, both the positive and the negative form of the directive are excluded.)	COPYLIST ERRLIST [NO] FORM LIST PRINT [NO] REF RESEQ
ERRLIST	COPYLIST [NO] REF RESEQ
ANIM	NOCRTWIDTH

Table 4-2. Excluded Combinations of Directives

Description of Compiler Directives

ANIM

Format: ANIM
NO ANIM

The ANIM directive causes the Compiler to generate files that can be used by the Animator for program testing and debugging. If the source code file is basename.CBL, this option produces the files basename.ANM and basename.ACP. The Animator also uses the intermediate code files and dictionary files that are always produced for the root, and any overlay segments. NO ANIM will suppress the output of those files needed only by the Animator.

Note that ANIM is mutually exclusive with the NO CRTWIDTH directive, since the Animator makes use of the internal buffers allocated by the use of CRTWIDTH.

The default setting of this directive is ANIM.

BRIEF

Format: BRIEF
NO BRIEF

BRIEF causes the Compiler to omit the explanatory text in error messages, that is to output only the error number to the listing and to the console. The Compiler automatically goes into BRIEF mode if it cannot find the file COBOL.ERR. NO BRIEF causes the explanatory messages to be output (if the Compiler can find the file COBOL.ERR on-line).

The default setting is NO BRIEF.

COMP

Format: COMP
NO COMP

This directive causes the Compiler to generate smaller and faster code for arithmetic on PIC 99 and PIC 9(4) operands where no ON SIZE ERROR clause is specified. In addition to the speed and smaller code size, another advantage is that it becomes possible to do computations on characters, for example converting lower- to upper-case. The disadvantage of COMP is that when it is in effect, some MOVEs may give non-ANSI standard results. PIC 99 fields will overflow at 256 rather than 100.

The default setting is NO COMP.

COPYLIST

Format: COPYLIST
NO COPYLIST

COPYLIST causes the contents of a file named in a COPY statement in the COBOL source code to be output in the Compiler listing.

At page breaks in the listing, the page header names any COPY file open at the break. NO COPYLIST suppresses the listing of COPY file contents.

The default setting is NO COPYLIST.

CRTWIDTH

Format: CRTWIDTH "line-size"
NO CRTWIDTH

CRTWIDTH specifies the logical line-size of the CRT screen for Format 1 (ANSI standard) DISPLAY statements. NO CRTWIDTH tells the Compiler that no ANSI standard DISPLAY statement will be compiled. In this case, the Compiler can use memory otherwise allocated to control tables to increase dictionary size.

The default setting is CRTWIDTH "128" (a value chosen to match a common limit of main-frame COBOL implementations).

DATE

Format: DATE "literal-1"

DATE causes the Compiler to replace the comment entry in the DATE-COMPILED paragraph of the IDENTIFICATION DIVISION with the value of literal-1. If literal-1 is not specified, then the system date and time will be used in its place. If the NO DATE directive is issued, or there is no system clock and the date has not been set manually, then date-time insertion into the DATE-COMPILED paragraph is suppressed.

The default is DATE.

ECHO

Format: ECHO
NO ECHO

ECHO causes the Compiler to list error lines on the console (that is, the source code line in error, a line with the error number underneath this, and—unless the BRIEF directive is in effect—a final line with explanatory text). NO ECHO turns off console listing of errors.

The default setting is ECHO.

ERRLIST

Format: ERRLIST
NO ERRLIST

ERRLIST causes the Compiler to limit the listing file (LIST or PRINT output) to error lines (the same listing, in other words, that appears on the console when ECHO is set). Only erroneous source code lines, lines with the error numbers and (unless a BRIEF listing is selected) lines giving a short explanation of the errors are written out to the listing file.

The default setting is NO ERRLIST, that is, a full listing file.

FLAG

Format: FLAG "level"
NO FLAG

FLAG causes the Compiler to produce General Services Administration (GSA) Compiler validation flags as part of the compilation listing. The flags are also listed on the console if the ECHO directive is in effect.

The parameter "level" must be one of the following:

LOW	Produces validation flags for all features higher than the Low Level of GSA Compiler certification.
L-I	Produces validation flags for all features higher than the Low-Intermediate Level of GSA Compiler certification.
H-I	Produces validation flags for all features higher than the High-Intermediate Level of GSA Compiler certification.
HIGH	Produces validation flags for all features higher than the High Level of GSA Compiler certification.
A*III*	Produces validation flags for the Apple *III* (that is, Micro-Focus Compact Interactive Standard) extensions to standard COBOL as it is specified in ANSI Standard X3.23 – 1974. For a list of these extensions, see the *Apple III COBOL Language Reference Manual.*

The FLAG "A*III*" directive is the best way to find any part of an Apple *III* COBOL program that is not in accordance with the ANSI standard. If the program compiles with no line flagged A*III*, you may be sure that the program is a standard COBOL program.

IBM — Flags several optional extensions to Standard COBOL compatible with IBM 8100 DPPX COBOL (these features are enabled by specifying the IBM directive below). For details of this feature, refer to Appendix J of the *Apple III COBOL Language Reference Manual.*

The default setting is NOFLAG.

FORM

Format: FORM "page-length"
NO FORM

FORM sets the number of lines to be placed on each page of the compilation listing file; the minimum number of lines per page is five. Each output page will have this specified number of lines, except where the source code has a "/" in the Indicator Area (column seven), which forces a page eject. NO FORM inhibits the pagination of the listing file; this is particularly useful when using the Compiler to generate a sequenced version of the source code file (see RESEQ below).

The default setting is FORM "60".

FORMFEED

See SPECIAL-NAMES Directives.

IBM

Format: IBM
NO IBM

IBM enables the language extensions specified in Appendix J of the *Apple III COBOL Language Reference Manual.* These extensions are compatible with DPPX COBOL on the IBM 8100. NO IBM directs the Compiler to

regard these features as syntax errors.

The default setting is NO IBM.

INT

Format: INT "pathname"
NO INT

INT causes the Compiler to output an intermediate code object file with (root segment) name given by the quoted SOS pathname. Other intermediate code files will also be output in the case of a segmented program; see Chapter Five for details. The NO INT directive tells the Compiler not to generate any object files for the program.

It is not an error to type several INT directives in one command line, but it is pointless—the Compiler scans its directives from left to right and will simply take the last setting of any directive and ignore earlier ones.

The default setting is INT "basename.INT" where basename is the file name used at the beginning of the command line, minus the ".CBL" extension if that was present. If you wish to generate a code file whose name does not end in ".INT", then issue the command INT"basename.". Note that the "." at the end of basename prevents the Compiler from appending the ".INT" suffix.

LIST

Format: LIST
LIST "pathname"
NO LIST

LIST causes the Compiler to produce a listing file such as that illustrated in Figure 2-9, written to the file specified by the pathname argument. If the directive is given in the first format (just LIST with no following argument) the listing is written to the console; that is, the effect is the same as a LIST ".CONSOLE" directive.

The default setting is LIST "basename.LST" where basename is the file name used at the beginning of the command line, minus the .CBL file extension if that was present.

PRINT

Format: PRINT
PRINT "pathname"
NO PRINT

PRINT is identical in effect to LIST and may be used interchangeably with it.

Default setting: see LIST.

REF

Format: REF
NO REF

REF directs the Compiler to put hexadecimal location references to the right of each line of source code in the listing file. These locations are offsets of data items in the DATA DIVISION or offsets of the lines in the PROCEDURE DIVISION. NO REF omits these location references.

When a program terminates with a run-time error, the error message contains a reference to the COBOL program counter:

xxxxxxxxx Segment:yyyy
COBOL PC nnnnH : RTS Error mmm

Here, xxxxxxxxx refers to the intermediate code file name, yyyy is "Root' or an overlay segment number, mmm is the error number (see Appendix C), and nnnn is PROCEDURE DIVISION offset. This offset will be one of the offsets printed by the REF directive, or in between two if the error occurs somewhere in the middle of the statement.

The default setting is NO REF.

RESEQ

Format: RESEQ
NO RESEQ

RESEQ causes the Compiler to generate COBOL sequence numbers in the first six columns of the listing file. The first source line is numbered

000010, and subsequent lines increment this by 10. With NO RESEQ, the Compiler ignores the sequence number area; characters in this area will be printed in the listing, but they are used for documentation purposes only.

Using the directive combination

LIST ".Dn/basename.CBL" RESEQ NOFORM NOINT

is a useful way of creating a (re)numbered version of a source file. (The "n" in .Dn above is intended to specify a different disk drive from that containing the source file —the Compiler will object if you try to overwrite the original source.) The listing file will contain an extra line at the beginning (listing the directives in force) and three extra lines at the end, but these will be ignored by the Compiler if you use the new file as input. Of course, you can also delete these lines by using a text editor.

SPECIAL-NAMES Directives

The remaining four directives all have the same format and comparable significance. The format is:

FORMFEED "function-name"
SYSIN "function-name"
SYSOUT "function-name"
TAB "function-name"

The SPECIAL-NAMES paragraph of the ENVIRONMENT DIVISION allows the programmer to associate arbitrary mnemonic names with the functions SYSIN and SYSOUT (for the system logical input and output) and TAB and FORM (for certain WRITE ADVANCING clauses). See the *Apple III COBOL Language Reference Manual* for the form of such statements.

As a consequence of these functions, the four names FORM, SYSIN, SYSOUT, and TAB are usually reserved words for Apple *III* COBOL. However, you may need to use these names as data-names in a program; for example, you may be transporting a program from a different COBOL system.

These directives allow you to change the mnemonic-names in the ENVIRONMENT DIVISION to any arbitrary value, to avoid such conflicts. For example:

TAB "AIII-TAB"

directs the Compiler to change the implementor-name of the TAB function to AIII-TAB; thereby, the name TAB loses all special significance in compiling this source code.

There are no default settings for these directives, and the Compiler will reject a negative directive (such as NO SYSIN).

Compiler Sign-Off

At the conclusion of processing, the Compiler will place on the screen (and at the end of the listing file, as in Figure 2-9) a line of summary information specifying

ERRORS=nnnnn	Total number of errors found
DATA=nnnnn	Total number of bytes used for data items in the object file
CODE=nnnnn	Total number of bytes used for the PROCEDURE DIVISION in the object file
DICT=nnnnn:mmmmm/ppppp	Data dictionary space usage— nnnnn is the amount used, ppppp is the total available and mmmmm is the number of bytes left
GSA FLAGS=nnnnn	Total number of source items flagged—always 0 if NO FLAG directive is in effect

5

Application Design and Development

Apple /// COBOL provides standard COBOL facilities for loading programs dynamically, for overlaying segments in memory and for invoking programs (subroutines) written in COBOL. These subroutines must conform to the Level One specifications of the COBOL modules Segmentation and Inter-Program Communication. In addition, the COPY verb (of the COBOL Library module) allows the sharing of complex program pieces, notably record and file descriptions, among multiple programs with a single source text to update as changes are made.

With these facilities, large and complex Apple /// COBOL applications can be developed. In particular, the total size of the application is not constrained by intrinsic hardware limits. This chapter describes the use of these structural facilities, with emphasis on operational and design issues. Details of the Apple /// COBOL Language elements for handling Inter-Program Communication and Segmentation features are given in the *Apple /// COBOL Language Reference Manual.*

Program Editing

This chapter is principally devoted to issues of program design, which is a conceptual and analytic task, that is, deskwork rather than work at the computer console. Other chapters of the manual deal with the mechanics of using the Apple /// and operating the Compiler and the other system software. The interface between design and the actual compilation and

testing of programs lies at the level of program editing; therefore we begin by considering how to use Apple /// text editors in COBOL application development.

The Apple /// COBOL System does not include a text editor. The COBOL Run-Time System uses a different interpreter than that in other Apple /// Language systems, in order to handle the unique features of COBOL as efficiently as possible. We assume that you have one of the standard Apple editors at hand: Apple Writer /// or the Pascal Editor or any other editor that can output ASCII or Pascal TEXT files. For details on their use or the use of individual commands mentioned below, you should refer to the appropriate manual (the *Apple Writer /// Operating Manual* or the *Apple /// Pascal Introduction, Filer, and Editor* manual).

COBOL Formatting

The formatting rules for COBOL programs—Indicator Area in character position 7, Area A from positions 8 through 11, and Area B from 12 to the end of the line—originated in an era dominated by punched card input. Column numbers are less convenient in the context of screen editing, but most editors do have the ability to keep track of margins or set tab positions; some, like the Apple /// Pascal Editor, can automatically follow your pattern of indentation (such as in IF-ELSE sentences) to assist you in the layout of "structured" code patterns.

For Apple /// COBOL, the important character positions on a line are position numbers 7 (Indicator Area), 8 (start of Area A), 12 (start of Area B), and 72 (last character position read). The Compiler ignores anything after position 72 and anything in the first six character positions (the Sequence Number Area)— though it can place sequence numbers there if you direct it to do so by using the RESEQ directive (see Chapter Four).

Both of the editors discussed here count columns differently than COBOL does: they begin with column 0 at the left of the display and reach column 79 at the right (rather than columns 1 and 80). (Both allow lines longer than eighty characters, but that is not relevant in a COBOL setting.) You will need to learn to decrement the COBOL boundary values by one to correspond to the way the editors count; for example, Area A is editor columns seven to ten, and the Indicator Area is editor column six.

A COBOL Program Template

It is worthwhile to create a skeleton COBOL program, containing nothing but Division headers and your standard IDENTIFICATION and ENVIRONMENT DIVISION entries, such as AUTHOR, INSTALLATION, DATE-WRITTEN, DATE-COMPILED, SOURCE-COMPUTER, OBJECT-COMPUTER. Many of these entries are unnecessary for the functioning of a COBOL program, but they are all good documentation and should be included in any program. Note that Apple *III* COBOL will update the DATE-COMPILED comment to assist you in tracking different versions of a program (see the DATE directive in Chapter Four).

Apple *III* COBOL allows you to omit some of the entries required by the ANSI standard. For example, the console example in Chapter Two omitted all division headers except for the PROCEDURE DIVISION. Items which can be omitted in an Apple *III* COBOL program, but are required by the ANSI standard, are marked in the *Apple III COBOL Language Reference Manual* by special notation; they are surrounded by square brackets overstruck with hyphens. These program documentation entries can be omitted from "quick and dirty" programs, but should be present in any production program. In any case, the items required by the ANSI standard must be present for the program to be portable. It is good practice, therefore, to have a model or template file around containing well thought-out versions of these entries. If you make such a file and save it, say with the name MODEL.CBL, then you can always begin a new program by loading MODEL.CBL and simply changing its name, when you write it out to disk, to whatever you like for the new program. Then when you quit editing, the new file keeps whatever you have incorporated from the model, and the original MODEL.CBL is still there for the next time. Use system utilities to write-protect MODEL.CBL.

Such a file can also contain sample formats for FD entries or for your preferred indentation patterns for COBOL statements. While you are editing the new program on top of the model it is easy to delete items you don't need or to fill in the holes in a sentence template. It is counterproductive to be too elaborate with editing templates, but simple versions can be very useful in jogging your memory and maintaining a consistent style (as in data names or in the layout of structured statements) across several programs.

Program Editing with Apple Writer III

Apple Writer and the Pascal Editor are very different in approach to editing a text. If you have a choice, you should experiment with both to determine your own preference. Apple Writer is more like a typewriter or keypunch in its operation, and is heavily weighted in its design to word processing; the Pascal Editor is specifically designed to help in writing structured code.

For COBOL programming purposes, the main advantage of Apple Writer is that it constantly displays, at the top of the CRT screen, a line of data about the position of the cursor. In particular, the "Tab" field in this display tells you what column you are on. The value reported ranges from 0 at the left hand margin to 79 at the right of the screen, and past that for longer lines. As mentioned above, you need to adjust COBOL positions down one in this context, since COBOL numbering begins with 1 in the leftmost column. Once you have adjusted to this, however, it is easy to keep to the required COBOL format as you edit your program. For example, when you approach the end of a line you can type up through column 71 and then press RETURN and continue the line by inserting a hyphen in column 6.

Be sure to end every line with a RETURN. The Apple Writer documentation suggests otherwise, but it is discussing word processing and trying to anticipate text justification and filling operations. You DON'T want these operations applied to your source code!

Apple Writer also allows you to set "Tab stops" anywhere along a line. These are initially set to columns 8, 16, 24, etc., which is not a useful setting for COBOL. However, you can clear these default values, and reset your own tabs as you move across the screen. Normally you will find use for stops at each level of a record description and at convenient indentations within nested statements in the PROCEDURE DIVISION. One disadvantage of Apple Writer is that you can't move the cursor vertically up and down the screen; the DOWN-ARROW and UP-ARROW keys move to the start of the next line or the end of the previous one. To have column alignment of PICTURE or VALUE phrases, you will need to use tab stops liberally.

Capitalizing Lower-Case Files

Standard COBOL uses only upper-case letters, except in alphanumeric literals. If your programs are going to be moved from one system to another, you should probably follow the standard in this regard when writing them.

In order to enter a program in all upper-case letters, push down the ALPHA LOCK key below the shift key in the lower left corner of the keyboard. This key will shift all the alphabetical keys into upper-case, but it has no effect on any of the other keys. This is a hardware function of the Apple *///* keyboard; it applies to any editing software that will run on the Apple *///*.

Occasionally, you may find it necessary to convert a lower-case source code file; for example, if you want to transport an Apple *///* COBOL program to another system. Some editors have special commands to convert between cases. In Apple Writer, for example, you can use CONTROL-C followed by the RIGHT-ARROW or LEFT-ARROW keys to move across the screen converting every character you pass over to upper-case; this process is rather impractical for a large file, however.

To convert a large file to upper-case, you can issue 26 separate "find and replace" commands, replacing all a's with A's, all b's with B's, ..., and all z's with Z's. The last of these commands to Apple Writer, for example, would be:

CONTROL-F /z/Z/a

where the CONTROL-F invokes the find and replace function. This process is still a bit unwieldy, but note that the Apple Writer WPL feature allows you to make a file of these commands, which can then be applied to any file you like.

Editing with the Apple III Pascal Editor

The Pascal editor works in an editing "environment" context that determines (among other items) margin settings and the name and type of the file. Normally, when you begin editing a new file margins are set at the far left and right of the display, and there is a paragraph indentation of five spaces. For a new file (that is, if you haven't read in an old file to change it) the editor assumes the pathname /SYSTEM.WRK.TEXT and a special file type TEXT. All of this can be changed using the Set Environment command. Once you have entered the editor, just type **S** and **E**, and you will see a display like that in Figure 5-1. To change a parameter setting, type its initial letter and the new value. (If the value can be more than one character long, you will need to include a **RETURN** as well.) Figure 5-1 illustrates one useful setting of the Environment. There are several points worth noting about the example:

- Like Apple Writer, this editor numbers columns from 0 to 79.
- If you type in the new file's name .d2/model.cbl without a period at the end, the editor will automatically append the extension .TEXT to what you typed. The final period is not part of the name—it just tells the editor NOT to extend the name. It's easy to forget this final period, but mostly it's only a minor annoyance. The virtue of setting the file name by a Set Environment command is that the editor will tell you right away what it thought you meant.
- The A(scii option is left False; that is, the file will show up in a directory listing as type TEXT, not as type ASCII. Either type can be used, but with an ASCII type file, the margin settings won't be saved when you save the file—you have to reset the environment for each edit.

```
)Environment: <options>, <ctrlC> accepts, <esc> escapes [A3/1.0]
    I(ndent auto        True
    F(illing            False          original value:
    L(eft margin        7                    0
    R(ight margin       71                   79
    P(ara margin        5
    C(ommand char       ^
    T(oken search       True
    A(scii file         False
    N(ame of file       .d2/model.cbl.       *SYSTEM.WRK.TEXT
```

Figure 5-1. A COBOL Editing Environment

After confirming these environment changes (by typing **CONTROL-C**), type **I** to enter the editor's insert mode, and start typing in a program.

As you type in **IDENTIFICATION DIVISION**, you will notice that the words appear at the far left of the screen. Unlike Apple Writer, the Pascal Editor doesn't tell you your column position along the line. How do you know that you are typing into Area A? One method of course is just to type in seven blanks before the word IDENTIFICATION; this works, but is prone to error from typing fewer than seven blanks. The environment margin settings permit another method that lets the editor do the counting for you. Leave the insert mode at some point in the typing of the division header (type **CONTROL-C**) and select the A(djust command by typing **A**. The menu at the top of the display gives you the options

```
)Adjust: L(eft, R(ight, C(enter, <Moving keys>, <ctrlC> accepts, <esc> escapes
```

Type L and the line the cursor is on will be adjusted, if necessary, to be flush against the Left Margin set in the Environment. Any time you want to be sure that a line starts in Area A, just Adjust it Left; if you want it in Area B, Adjust Left and then press the RIGHT-ARROW key four times to move the line to Area B. You must then type CONTROL-C to finish the adjustment operation.

As you continue typing in a program, you will notice that each time you press RETURN to end a line, the cursor will be in place to start the next line directly under the last one. This "auto-indentation" is very useful for typing structured code; for example, a "nested IF" statement, or a complex record description.

Consider the following record:

```
01  Rec-A.
    05  Name.
        10  Last      pic X(16).
        10  First     pic X(16).
        10  Middle    pic X(1).
    05  Id.
        10  SS-No     pic 9(9).
        10  Emp-No    pic 9(6).
    ...
```

You will have to type in blanks to indent from the 01 level to the 05 level and again to indent to the 10 level, but the editor will keep you at level 10 for each succeeding line until you use the LEFT-ARROW key to back up to a previous level. (Note that, like Apple Writer, this editor assumes tab stops at every eighth column; unlike Apple Writer, it won't allow you to change this.)

If you are deeply indented and need to get back to the start of Area B or Area A, you may find it convenient to start the new line at the current deep indentation and then stop inserting and use the Adjust command as we did above: Adjust Left to get to Area A or follow the Adjust Left with four RIGHT-ARROWs to get to Area B. Conclude as usual with CONTROL-C.

What happens when you get to the right margin in a long value clause? In this case, the line has to be "continued" by typing a hyphen in the Indicator Area of the next line. Note that the editor will "beep" at you as you reach the right margin—in fact, the "beep" will occur in the column before the right margin, allowing you one more character to complete the line. You should complete the line with this last character, then press RETURN. On the new line, type the hyphen and then exit from the insert mode with a CONTROL-C. You now need to adjust the hyphen into the Indicator Area; type A again and then the LEFT-ARROW to move the

hyphen into the Indicator Area. After obtaining acceptance of this adjustment with another CONTROL-C, you are ready to continue with insert mode. If you need to continue the next line also, auto-indentation will start the new line below the earlier hyphen.

It's easy to miss the last character position in a value clause literal, especially by over-writing the editor's "beep" by more than one character. However, you can also use the Adjust command to be sure of this position. Exit from your insertion and try the Adjust Right option. This will place the last character you typed exactly at the column set in the Environment as the right margin for your file. If the line moved when you adjusted it, your guess was wrong, but you can now correct it by deleting or inserting characters as necessary.

Setting Markers

The Pascal editor allows you to have up to ten "markers" at any points of your choice in a file. This facility allows you to jump around from point to point without having to wear out your finger or your patience on the cursor keys. You can also use markers with the Copy command to copy just part of an existing file into the program you are editing, or to duplicate code from one section of your file in another. When you select the **S**(et command from the editor's command line and type **M** for M(arker, the editor will prompt you with the message

```
Set what marker?
```

for a name (one to eight characters long, digits and special characters are allowed as well as letters) and it will make an internal note associating that name with the current cursor position. After you have set ten markers, you will have to replace one of the earlier ones if you want to mark a new point in your file.

Useful points to mark in any program would be the start of the DATA DIVISION and the PROCEDURE DIVISION. In some programs you may want to mark distinctions lower than divisions, for example the FILE-CONTROL paragraph in the ENVIRONMENT DIVISION or the LINKAGE SECTION in the DATA DIVISION, or individual paragraphs or statements in the PROCEDURE DIVISION. For segmented programs (see below) you will probably find it helpful to mark each of the sections of the PROCEDURE DIVISION with its segment number.

Note that you can set markers as well as margins in an ASCII type file; but the settings are lost when you quit the editor and write the file to disk. A TEXT file has an extra two blocks (1024 bytes) at the beginning, used by the editor to maintain this environmental information.

Some of the programs on the /DEMO disk illustrate the placement of markers. If you list the /DEMO directory, you will see that several of the files have file type Textfile, while others are type Asciifile. Use the Pascal editor to examine one of those listed as a Textfile. When you have read in the file, issue the Set Environment command (type **S** and then **E**); the markers set in the file will be named at the bottom of the Environment display.

You can use the Jump command to move from one marker to another. One advantage of the Pascal marker system is that an existing program can be marked by one programmer, and another programmer can immediately locate significant points of the program just by noticing what the first one marked.

Program Structure—Segmentation

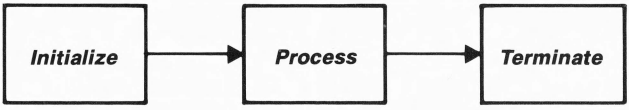
Recall that a COBOL PROCEDURE DIVISION consists of sentences, and the sentences may be grouped together into larger logical units, that is, paragraphs and sections. A paragraph is a collection of sentences given a name for purposes of documentation or as a target for transfer of control during program execution. A section may be one or more paragraphs, or it may be simply a number of sentences not grouped into paragraphs. For main-frame computers with large memories, program sections are primarily a notational convenience for PROCEDURE DIVISION parts intermediate between the paragraph and the program as a whole. Sometimes their only use is to allow abbreviations such as

```
PERFORM PARAGRAPH-A THRU PARAGRAPH-C.
```

to

```
PERFORM SECTION-1.
```

However, sections have a more active part in program development for the Apple *III*. A large procedure may be too large to fit into the computer's memory all at one time; it must be broken into several "segments", and each segment "overlaid" on top of the memory used by previous segments. Consider how frequently a program has the structure



There may be a substantial initialization phase; once it is done there is no need to have its code sitting in memory. Similarly, the termination code needn't be in memory until the end, and the central process can stay on disk until the initialization is done and be deleted when the termination code comes in. The central process might even have several independent parts which likewise can be split into segments.

The Compiler determines the overlay structure of your program on the basis of the sections you define in the PROCEDURE DIVISION of your source code. If any part of your PROCEDURE DIVISION is structured as a section, then all of it must be: every sentence and paragraph must be part of a section.

Segments and Overlays

The ENVIRONMENT and DATA DIVISION of a COBOL program plus some residual part of the PROCEDURE DIVISION must remain in memory at all times; even if you specify that each section of your program is a candidate for overlay, the Compiler will create a certain minimal amount of code needed by all the sections. This minimum segment that must be resident at all times is called the "root" or permanent segment of the program. The Run-Time System always loads the root of a program when you give it the Run command (Chapter Three). It will then bring in the other "independent" segments one at a time, taking no more than enough memory to hold the largest segment of the program. Such a situation is illustrated in Figure 5-2.

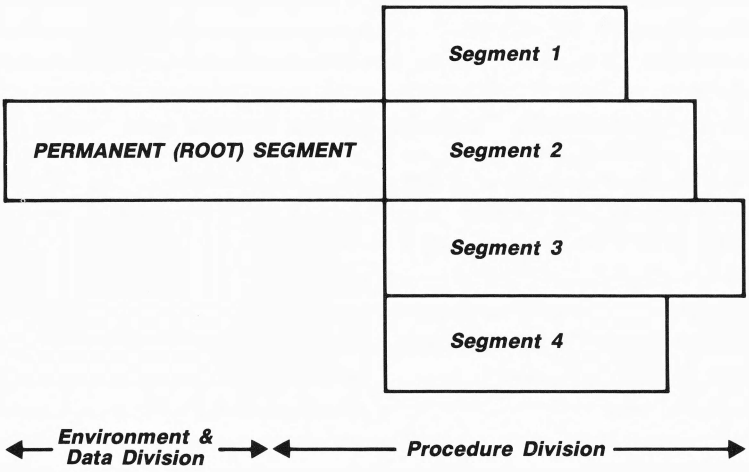


Figure 5-2. Segment Overlay

This figure shows a chunk of PROCEDURE DIVISION code in the root segment. In general, you will have some bookkeeping or general purpose routines that need to be performed by more than one segment; these should go into the root along with the program data.

As the program executes, and control transfers out of one segment into another, the Run-Time System checks whether the target is currently in memory. If it isn't, it is loaded into free memory or on top of existing overlayable segments.

Coding for Segmentation

In order to specify segmentation in a COBOL program, you must include a segment number in the section header of some of your sections in the PROCEDURE DIVISION. The segment number is a one or two digit number. Segment number values from 50 to 99 refer to independent segments, to be brought in (overlaid) as needed; values from 0 to 49 all refer to the root segment. You may omit the segment number value; in that case, the default value of 0 is assumed. (You might find it useful to have different segment values in the 0 to 49 range, in order to logically

differentiate sections that have different functions but must all be present in the root segment.) The following sample illustrates coding of a PROCEDURE DIVISION for segments:

```
PROCEDURE DIVISION.
ROOT-X-UPD                  SECTION.
X-UPD-A.
        MOVE A TO B.
        etc.
        .
        .
        STOP RUN.
OVLY-2-NEW-ACCOUNT      SECTION 60.
        MOVE X TO Y.
        etc.
```

Note that when segmentation is used, the entire procedure division must be segmented. Hence the first paragraph ("ROOT-X-UPD" here) must be a section.

In this example, the section ROOT-X-UPD will be included in the root segment—it will never need to be loaded when a PERFORM or other control transfer causes it to be executed. Since no segment number is given, the Compiler automatically places it in segment 00. The other section shown becomes segment 60. It is apparently intended to do special processing for a new account and will not normally be in memory at all; it will be loaded by the Run-Time System whenever control needs to be transferred there.

In general, root segment sections will precede overlay sections; that is, sections numbered 0 through 49 or not numbered at all must appear before sections numbered 50 through 99. Order within these two major groups is not important. However, good programming practice would suggest that the sections be numbered in increasing order and that the section numbers have some logic to them, in order to aid program documentation and maintenance.

Any control transfer (PERFORM or GO TO) to a segment that isn't in memory will cause the Run-Time System to load the segment on top of the current non-root segment. You can transfer to a paragraph in the middle of an overlay segment—it is not necessary to PERFORM or GO TO the section from its beginning. Other technical restrictions on segmentation are discussed in the *Apple III COBOL Language Reference Manual.*

Operational Considerations

The Compiler creates a dictionary file and an intermediate code file for each independent segment (that is, each section with a segment number of 50 or more) and for the root segment. The dictionary files have extensions .D00 (for the root) and .Dnn (for each segment number "nn" between 50 and 99). The intermediate code files have extensions .INT (for the root) and .Inn (for overlay segment nn). The dictionary files aren't needed for running the program; they are used by the Compiler and by the Animator. If you don't want to use the Animator to debug the program, you can delete these files after compilation.

Additionally, the Compiler generates an Intersegment Reference file (extension .ISR); the total size of the root, the largest overlay segment, and the .ISR file combined must be 64K bytes or less.

The Run-Time System thinks of any .INT file as a (potential) root of an overlay structure. Thus you may occasionally see messages (for example, when you abort a program with CONTROL-\) referring to a root even though you have no sections at all in the source code.

If you are trying to debug a large program, the Animator may be unable to operate because there isn't enough room for the program code in memory together with the Animator's own internal code. (This should not be a problem unless you are working in a configuration with less than 256K memory.) In such a case, you may find it useful to break the program into segments (or into smaller segments if it is already segmented). In this case, efficiency of operation is not a major consideration, so you can be fairly arbitrary in dividing the program into sections.

More generally, it requires considerable thought to use the segmentation facility efficiently. If you are developing a program on a 256K Apple *III*, you should always be able to use the Animator to trace program behavior. The basic idea is to consolidate into one section the sentences and paragraphs which tend to be needed at the same time during program execution. Most programs do have several clusters of code which act like that, a phenomenon known as "program locality" or "working set". The best rule of thumb is to follow the Initiate, Process and Terminate division illustrated above. Well-structured, modular programs usually divide into segments following internal logic in a natural way.

Program Structure—Inter-Program Communication

Segmentation is one means of modular decomposition of large or complex programming tasks. As such, it has two limitations on the modular breakup of a task: it does not provide for the modular decomposition of the Data Division, and it allows for only a finite set of procedural units, all of which must be present within one program source file.

Each module of a program will normally have its own "local" set of data items in working storage; it may well have a file or set of files that is relevant only for its task and not for the rest of the program. A programmer can use the REDEFINES clause in the DATA DIVISION to give different definitions for each module to a common area of memory, but over-use of this facility is dangerous and prone to inducing error. It is safer for each module to have its own local data items, with data and procedure code both loaded or overlaid together; the permanently resident root would contain only the procedures and data needed by all modules.

Furthermore, any on-going application will acquire a library of useful programs and standard subroutines for common tasks. It is possible to incorporate these in a new program by use of COPY statements or by use of editor copy capabilities; however, this approach requires a great deal of work and care to see that name conflicts are avoided, and to be sure that the library code works the same way in its new source code context.

COBOL addresses these problems by allowing separately compiled modules to cooperate, transferring control by CALL and EXIT statements and passing data by means of the LINKAGE SECTION. The Run-Time System loads a module when it is CALLed (unless it is already present in memory), bringing in its own local DATA DIVISION and PROCEDURE DIVISION. The module will then remain in memory until the program releases it with a CANCEL statement. By such means, the program can have a more detailed and complex control of the computer's memory usage than with segmentation.

The type of inter-module structuring possible using CALL is illustrated in Figure 5-3. Each box represents a separate source file, separately compiled and with its intermediate code available on disk. At each level, the modules shown CALL those underneath them that are attached by lines. None of the modules needs to know anything about any of the others except the ones directly under it.

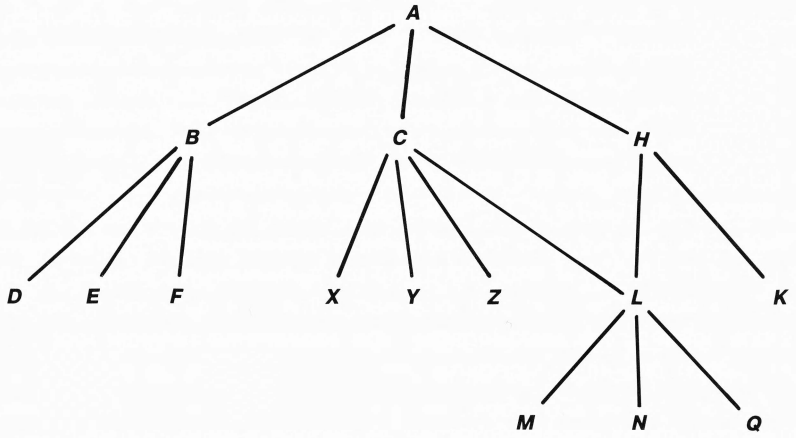


Figure 5-3. Sample CALL Structure

The main program A is permanently resident in memory. It calls B, C, and H, which are subsidiary stand-alone functions within the application. These programs in turn CALL other lower level functions. Since the functions B, C and H are independent, they don't need to be permanently resident in memory together. They can therefore be called as necessary, use the same memory, and then be CANCELed to allow room for the next function needed. The same applies to the lower functions at their level in the tree structure.

Memory Usage

Use of CALL and CANCEL should be planned to keep a common subroutine (like L in Figure 5-3) in memory to save load time. A CALLed program should also stay in memory if it has any open files; otherwise, it must re-open the files on every CALL.

If a file is opened during the execution of a program by means of a CALL statement, the file will be closed by any CANCEL statement that removes the program from memory.

Programs are loaded one after another, at the time they are called. Thus in Figure 5-3, assuming the CALL statements occur as in the left-to-right order of the diagram, program A will be loaded first, then B, and then D. If D is CANCELed, its memory then becomes available for E. However, suppose that D stays in memory until after the CALL to E. If D is subsequently CANCELed (but E isn't), there will be a "gap" opened with some free memory before E and some after. At the point that B CALLs F, F will be loaded into the larger of these two free areas. B must CANCEL each of D, E and F before its own EXIT, and A must CANCEL B in order to leave the maximum free space for loading C.

If you are not careful in CALL/CANCEL sequences, you can fragment the memory, leaving no room for a new CALL, even though the total amount of free memory might be enough. Figure 5-4 illustrates this problem:

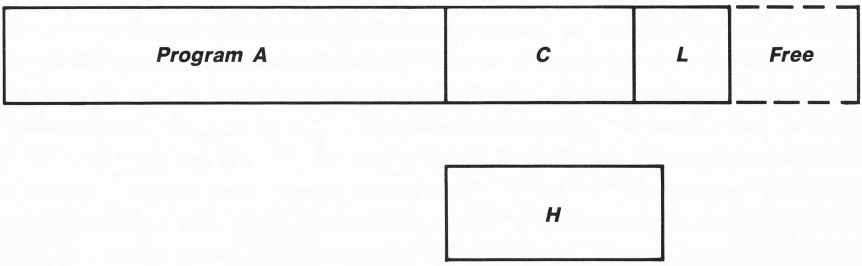


Figure 5-4. Memory Fragmentation

H is too big to fit in the memory obtained by CANCELing C and also too big to fit in the remaining free space after L. There are several possible solutions; the appropriate one will depend on the structure of the programming task. One possibility is for C to CALL L after CALLing (one or more of) X, Y or Z but before CANCELing them; this will place K further toward the high end of memory. Or possibly A can CALL H before C, so that C will fit in the space vacated by CANCELing H. Finally, A might be able to CALL L—not to do anything, but just to get it loaded.

There is one more complication to consider: any program can be segmented. When you CALL a segmented program, the Run-Time System allocates for it enough memory to hold the root segment and the largest overlay.

Coding for Program Calls

The most common use of the CALL statement in COBOL programming is the implementation of a modular hierarchy like that shown in Figure 5-3. Typically, the main program will have a number of statements like

```
CALL "B.INT".
```

or

```
CALL "C.INT" USING DATA-ITEM-1, DATA-ITEM-2.
```

where the items in the USING list of CALL to C must be 01 or 77 level items in the DATA DIVISION of A. These are the parameters of the CALL to C, and descriptions of them must appear in the LINKAGE SECTION of

C and in a USING list for the PROCEDURE DIVISION of C. The data-names used for these items in program C may (very likely will) be different from those used in A. The data descriptions may also be made different, but this is dangerous and should be done only with caution.

The object of the CALL verb may be the name of an alphanumeric data item as well as an alphanumeric literal. In either case, it must contain a valid SOS pathname for an intermediate code file. The name must be given in full; the Run-Time System will not supply the .INT extension when executing a CALL.

If the called program terminates with a STOP RUN statement or a simple EXIT statement, control will not return to the main program. Rather, the Run-Time System closes all files and returns to the main COBOL command line. To return control to the caller, the CALLed program must end with EXIT PROGRAM.

For details on CALL, CANCEL, EXIT PROGRAM and USING, refer to Chapter 11 of the *Apple III COBOL Language Reference Manual.*

Dynamic Program Hierarchies

There are situations in which a fixed CALL structure like Figure 5-3 would be inappropriate. For example, in a menu-based application environment, the user might execute any one of an indefinite number of programs from his library, or one program may need to be followed (but only in some circumstances) by another. Hard-coding all possibilities may be wasteful or impossible; you want instead to ACCEPT a name from the operator, or to build a name from pieces (like a list of standard file extensions and a basename) not necessarily known to the caller.

Fortunately, COBOL is sufficiently flexible to allow such dynamic determination of the program to call. The basic idea is to let program names be part of the data that is passed between programs and subprograms in the Linkage Section. With this method, a simple overall control loop for a main program can CALL an infinite number of other

programs, determined for it by some "menu selector". The example below illustrates this technique.

```
WORKING-STORAGE SECTION.

01  NEXT-PROG        PIC X(20)   VALUE SPACES.
01  CURRENT-
PROG                  PIC X(20)   VALUE "SELECTOR.INT".

PROCEDURE DIVISION.
LOOP.
        CALL    CURRENT-PROG USING NEXT-PROG.
        CANCEL CURRENT-PROG.
        IF NEXT-PROG = SPACES STOP RUN.
        MOVE NEXT-PROG TO CURRENT PROG.
        MOVE SPACES     TO NEXT PROG.
        GO TO LOOP.
```

Note that this is a complete program in Apple *///* COBOL. The actual programs to be run can then specify their successors as follows:

```
LINKAGE-SECTION.
01 NEXT-PROG PIC X(20).
.
.
.
PROCEDURE DIVISION USING NEXT-PROG.
.
.
.
MOVE "SUCCESSOR.INT" TO NEXT-PROG.
EXIT PROGRAM.
```

In this way, each independent subprogram CANCELs itself and changes the name in the CALL statement to CALL the next one. Note however that if the subprograms in turn CALL other programs, this simple method does not guarantee effective memory utilization.

There are also two other alternatives that could accomplish essentially the same end result as our "menu selector" program.

First, you could just name your master menu program COBOL.START. Then any program it called could be exited via the STOP RUN statement. This would cause the system to reinitialize, and COBOL.START would be run all over again. Of course, this method would not allow any information to be conveyed from the CALLed program back to the master menu program.

Secondly, you could use the chain facility as described in the section "Calls to the Operating System", later in this chapter. Chaining acts as if you had exited the calling program and typed R(un followed by the program name to chain to from the COBOL command line.

Operational Considerations

Each program in an Apple /// COBOL application must be written in COBOL; that is, you cannot CALL programs written in Pascal or any other language.

No intermediate code file may be larger than 64K bytes, includ- ing both DATA and PROCEDURE DIVISION, and the PROCEDURE DIVISION itself is limited to 60K bytes. If the program is segmented, the total of root plus largest segment plus .ISR file must meet both of these limits.

A main program is invoked via the COBOL Command Line option R (or A if the Animator is used for debugging it); a subprogram is loaded whenever a CALL statement referring to it is executed, and it is not already present in memory.

The CALLed intermediate code program file must be present on disk at the time of the first CALL to the file, or fatal error 164 will result (see Appendix C, Run-Time Errors). Disks can be shifted from drive to drive (for example, by the operator acting upon messages displayed on the console) while the program is running.

There must be room available in memory for the program to be loaded. The ON OVERFLOW clause can be used to specify program action if insufficient space is available. Otherwise the CALL statement is ignored and the next calling program instruction is performed.

Calls to the Operating System

While it is not possible to call an arbitrary non-COBOL program, you can call several SOS routines. These routines are known to the Compiler by a one-byte binary routine number, and you can write

CALL X"nn" USING ...

or

CALL data-name USING ..., where data-name is declared as PIC X VALUE X"nn".

The available routines and their routine numbers and parameters are as follows:

Destroy. Routine number X"A5". This routine deletes a disk file; that is, it functions like the Utility Remove. There are two parameters. The first parameter is the pathname of the file to destroy and the second is an indication of the success of the operation. They should be declared in WORKING-STORAGE as shown in the following example:

```
01  DESTROY                 PIC X       VALUE X"A5".
01  PATH-NAME-1.
    05  PN-LENGTH PIC 99 COMP VALUE 10.
    05  PN-CHARS PIC X(10) VALUE "/DEMO/TEST".
01  RESULT PIC 99 COMP.
```

Here, PN-LENGTH holds the (binary) count of the number of characters in the pathname, and PN-CHARS contains the characters. RESULT is a variable that will be set on return from the CALL and is a normal SOS error code. Its value is 0 if the file was successfully destroyed or non-zero in case of errors.

The actual call would appear as follows:

```
CALL DESTROY USING PATH-NAME-1, RESULT.
```

Set-Prefix. Routine number X"AA". This routine changes the SOS prefix value. There are two arguments, which should be declared as shown in the following example:

```
01  SET-PREFIX              PIC X       VALUE X"AA".
01  PATH-NAME-2.
    05  PN-LENGTH           PIC 99      COMP VALUE 14.
    05  PN-CHARS            PIC X(14)   VALUE "/PROFILE/COBOL".
01  RESULT                  PIC 99      COMP.
```

The actual call would appear as follows:

```
CALL SET-PREFIX USING PATH-NAME-2, RESULT.
```

Upon return from the call, the SOS pathname will be set to /PROFILE/COBOL, unless RESULT contains a non-zero value. If RESULT is non-zero, it will contain a SOS error code.

Get-Prefix. Routine number X"AB". This routine gets the value of the current SOS prefix. There are three parameters, which might be declared as follows:

```
01  PATH-NAME-2.
    05  PN-LENGTH           PIC 99 COMP.
    05  PN-CHARS            PIC X(99).
01  LENGTH-TO-RECEIVE-IN    PIC 99 COMP  VALUE 99.
01  RESULT                  PIC 99 COMP.
```

The actual call appears as follows:

```
CALL X"AB" USING PATH-NAME-2, LENGTH-TO-RECEIVE-IN, RESULT.
```

Note that the length of the pathname received is PN-LENGTH OF PATH-NAME-2. Meanwhile the length parameter given in the main call designates the size of buffer waiting to receive the pathname; it should be made amply large. RESULT is again a variable to receive a SOS error code and will be 0 if the call was successful.

Set-Time. Routine number X"AD". This routine sets the SOS date and time and will set the Apple /// clock if you have one. The argument can be declared as follows:

```
01  SET-TIME                PIC X        VALUE X"AD".
01  THE-TIME.
    02  CENTURY             PIC 99       VALUE 19.
    02  YEAR                PIC 99       VALUE 82.
    02  MONTH               PIC 99       VALUE 8.
    02  DAY-OF-MONTH        PIC 99       VALUE 5.
    02  FILLER              PIC 9.
    02  HOUR                PIC 99       VALUE 14.
    02  MINUTE              PIC 99       VALUE 29.
    02  SECOND              PIC 99       VALUE 16.
    02  FILLER              PIC 999.
```

In this case, the actual call could be

```
CALL SET-TIME USING THE-TIME.
```

Note that in this case, we have assigned a mnemonic name to the call we wish to make.

Chain. Routine number X"84". This routine chains to another program. It takes one parameter which can be declared as shown below:

```
01  CHAIN                   PIC X        VALUE X"84".
01  FILE-NAME-1             PIC X(30)    VALUE "/P/CHAINEDTO.INT".
```

The actual call will be

```
CALL CHAIN USING FILE-NAME-1.
```

Note that the entire file name must be given, including the ".INT" suffix, if present. The net effect of this command is as if you executed an immediate STOP RUN in the calling program and typed R(un from the COBOL command line. The query about what file to run is then answered with /P/CHAINEDTO.INT. Worth noting, however, is that unless you issue the chain command with another parameter, /P/CHAINEDTO.INT has become analogous to COBOL.START in that it will be run again if any program ends with a STOP RUN.

Cold-Start. Routine number X"A6". This routine brings the COBOL system to an immediate halt and issues the message

INSERT SYSTEM DISKETTE & REBOOT

It requires no parameters. Cold-Start is useful for providing the user with a clean exit from a turnkey application.

Get-Char. Routine number X"D8". This routine allows you to read a single character typed at the keyboard without a following carriage return. One obvious use is for menu selection. It requires one parameter:

```
01  GET-CHAR          PIC X      VALUE X"D8".
01  CHAR              PIC X.
```

The actual call would be given as follows:

```
CALL GET-CHAR USING CHAR.
```

On return from this call, CHAR will contain the ASCII character typed at the keyboard by the user.

Sysv. Routine number X"A0". This routine has two separate uses which will be discussed here. It takes two parameters declared as follows:

```
01  THE-FUNCTION          PIC 99 COMP.
01  PATH-NAME-3.
    05  PN-LENGTH         PIC 99 COMP.
    05  PN-CHARS          PIC X(99).
```

The following call

```
MOVE 4 TO THE-FUNCTION.
CALL X"A0" USING THE-FUNCTION, PATH-NAME-3.
```

will return the pathname of the directory of the main program in PATH-NAME-3.

The call

```
MOVE 5 TO THE-FUNCTION.

CALL X"A0" USING THE-FUNCTION, PATH-NAME-3.
```

will return the name of the currently executing program in PATH-NAME-3. Note once again that the length of the data received is stored in PN-LENGTH while the actual characters are stored in PN-CHARS.

Two major cautionary notes are in order here. First, please note that the use of these calls is dangerous because no validation is performed upon their parameters. Thus, improper use can destroy the COBOL execution environment. Secondly, the use of these calls will lead to programs which are non-standard and cannot be ported to other systems without conversion. In general, use of these calls should be avoided whenever possible.

6

Apple /// Device Control

COBOL is traditionally a batch processing language. However, a dedicated computer like the Apple /// naturally favors a conversational or interactive style of use. Even if a program has been moved to the Apple /// from a batch system, it may require some special information such as the names of files to be accessed. In the batch system version, this information and other run-time data tailoring would be done by some form of "Job Control Language" (JCL); on the Apple, it is obtained directly from the operator at the console. The COBOL Language makes special provision for console input and output with its ACCEPT and DISPLAY verbs. In most other regards, however, COBOL is oriented to input and output of large SEQUENTIAL or RANDOM files.

Apple /// disk drives are adaptable to traditional COBOL file processing. Implementation considerations for the standard FILE ORGANIZATION possibilities will be found in Appendix F. Here we simply note that RANDOM (index sequential) files in the Apple /// COBOL system are implemented as pairs of files, with an index file used to index a data file, which can be independently accessed as a SEQUENTIAL file. The same organization is used to implement work files for the SORT and MERGE verbs.

The Apple /// supports a large number of character devices; besides the console, there are asynchronous communications, printers, graphics functions on the console, audio, etc. For these devices, the standard file organizations are not entirely appropriate. Apple /// COBOL provides a number of extensions to permit support of these devices. There are special variants of the ACCEPT and DISPLAY verbs; there is a special file organization LINE SEQUENTIAL appropriate to printers and the console; and it is possible to deal with individual bytes as hexadecimal control codes.

LINE SEQUENTIAL file organization allows variable length records with an ASCII CR (carriage return; X"0D") to terminate a record. Such an organization is natural for text files. Note that both ASCII files and Pascal TEXT files can be read using this organization. Lines of characters are inherently of variable length; the other file organizations in Apple /// COBOL require fixed length records. Because of the structure of SOS, any Apple /// character device is, as far as a COBOL program is concerned, a file like any other file; hence the program can READ a record from the file or WRITE a record to it, using the LINE SEQUENTIAL organization.

File Status

The FILE CONTROL paragraph in the COBOL ENVIRONMENT DIVISION has an implementation-defined FILE STATUS clause; in Apple /// COBOL the FILE STATUS data item must be declared in the DATA DIVISION to be a PIC XX variable. The leading character of this variable is described in the *Apple /// COBOL Language Reference Manual,* in Chapters Five, Six, and Seven (which deal with SEQUENTIAL, RELATIVE, and INDEXED input and output respectively). The second character is in fact a binary value. It is significant only if the leading character is the digit 9; in this case its value is a binary number—one of the file error values listed in Appendix C of this

manual. In order to make use of the numeric value of this byte, the program must move it to a computational data item; for example

```
FILE-CONTROL
     ...STATUS is FILE-STATUS.
DATA DIVISION.
...
01   FILE-STATUS.
     02  STATUS-1          PIC9.
     02  STATUS-2          PIC X.
01   BINARY-FIELD.
     02  BINARY-ZERO       PIC X VALUE LOW-VALUE.
     02  BINARY-CHAR       PIC X.
01   BINARY-NO REDEFINES BINARY-FIELD PIC 9(4) COMP.

PROCEDURE DIVISION.
...
CHECK-STATUS.
     IF FILE-STATUS NOT = ZERO
          MOVE STATUS TO BINARY-CHAR
```

At this stage, reference to BINARY-NO will properly handle the arithmetic value of the STATUS-2 byte.

ANSI ACCEPT and DISPLAY

COBOL has two verbs especially designed for console handling: ACCEPT and DISPLAY. These verbs exist in Apple /// COBOL, both in their ANSI standard versions (which present a low-level system interface to the SOS driver), and in a higher level extension designed for ease of use and more legible and secure programs than are possible with the standard version.

When an ACCEPT statement is executed, or a READ from the file .CONSOLE, the cursor appears on the screen as an invitation for the operator to type in appropriate information. A DISPLAY or a WRITE to the console can prompt the operator for what is wanted. The prompting can be as elaborate as full-screen menus or as simple and unobtrusive as you like.

The ANSI forms of these statements are written

ACCEPT ... FROM CONSOLE. DISPLAY ... UPON CONSOLE.

with the CONSOLE phrases optional. The extended forms are

ACCEPT ... FROM CRT. DISPLAY .. UPON CRT.

with the CRT phrases required unless the program contains the entry CONSOLE IS CRT in the SPECIAL-NAMES paragraph of the ENVIRONMENT DIVISION. For example, in this chapter, the ANSI forms will all omit the CONSOLE phrases, and the Apple *III* extensions will all use the CRT phrases. For full specification of the statements and their options, refer to Chapter Three of the *Apple III COBOL Language Reference Manual.*

The standard ACCEPT and DISPLAY verbs allow the programmer to read and write strings of alphanumeric data using a "logical" line length defined by the CRTWIDTH directive to the Compiler. See Chapter Four for the use of this directive. The logical line length is that line length after which the COBOL system will send a carriage return. If the logical line length is greater than the number of characters on a line of the CRT screen (40 or 80 characters), the console driver splits the logical line whenever the physical line fills up. Logical line length can be set to any length from 0 to 255 characters; a length of 0 effectively turns off the use of the ANSI form of ACCEPT and DISPLAY. The default setting of logical line length is 128 characters, a common value in other COBOL implementations and thus relatively convenient for transporting existing programs to the Apple *III*.

In executing an ACCEPT statement, the console displays the cursor and gathers up all characters typed until a RETURN; then, the characters are moved to the data item specified in the statement, dropping any characters in excess of the logical line length.

Typing input to an ACCEPT statement is like typing commands to the COBOL Run-Time System: a LEFT-ARROW deletes the last character typed and a CONTROL-X will erase the whole line for you to start over. The other cursor motion keys (UP, DOWN and RIGHT ARROW keys) are read by the ACCEPT statement but not echoed to the display. Any other key can be used normally. The ENTER key on the numeric keypad works just like RETURN to terminate the input.

The DISPLAY statement can handle any number of alphanumeric literals and data items; it gathers the items, in the order named, character by character into logical lines. As each logical line is filled, or for any partial line left over, it sends the line to the console for display. No data item or literal is truncated in this process of line-filling; if the item is longer than the logical line length, it simply spills over onto several lines. Normally, you should compile your programs with the logical line length (CRTWIDTH) the same as the physical display line length (or possibly a multiple of it, such as 160) to avoid ragged displays from mismatches of the logical and physical lengths, for example

```
-----------------------------------
-----
-----------------------------------
---------
-----------------------------------
--
...
```

Console Control Codes

Apple *III* COBOL allows you to use hexadecimal values in alphanumeric literals and data items. Thus the full ASCII character set, control characters as well as printable ones, is available for use in ACCEPT and DISPLAY statements. This feature is valuable because the Apple *III* console driver uses the ASCII control characters for many of its "special effects" such as direct cursor positioning, inverse video, windows (viewports), color text, etc. For example, the statement

```
DISPLAY X"1C1A030712", " HELLO! ", X"110C" UPON
CONSOLE.
```

uses hexadecimal control characters to

- clear the screen (1C)
- move to column 3, row 7 (1A with parameters 03 and 07)
- change to inverse video (12)

and, after displaying the greeting, to

- return to normal video (11)
- home the cursor to the upper left corner (0C).

Note that hexadecimal values passed in ACCEPT/DISPLAY statements must be UPON CONSOLE, not UPON CRT. Therefore, if in SPECIAL-NAMES, you have defined CONSOLE IS CRT, you must explicitly use the UPON CONSOLE format when passing hexadecimal values to the console driver.

For full details on the console control characters, see the *Apple III Standard Device Drivers Manual,* Chapter 3. The discussion there uses decimal values for the control codes; however Appendix J to that manual also gives a conversion table for decimal to hexadecimal. Because hexadecimal codes are not very legible, it is useful to define data items for the codes you will use, for example:

```
77 CRLF PIC XX VALUE X"0D0A".
77 BS PIC X VALUE X"08".
77 CLR PIC X VALUE X"1C".
```

If you use console control codes in your DISPLAY statements, especially those which move the cursor, you will tend to conflict with the logical line defined by CRTWIDTH. Keep your display lists short and the line length long (say 240) in order to minimize conflicts.

The ACCEPT statement will also read these control codes, except for CONTROL-H (BS, X"08"), which is the same as the LEFT-ARROW, and CONTROL-X, which erases the line. CONTROL-\ (which should correspond to the "display clear" character X"1C") can't be input because it aborts the program after the RETURN ending the ACCEPT. Also note that CONTROL-M (ASCII CR, X"0D") is the same as RETURN, so it terminates the ACCEPT instead of being read into the data item. However, if you hold down the OPEN-APPLE key to the right of ALPHA LOCK, these keys will transmit the ASCII characters but with their high bit set; your program can recognize this and act accordingly. It is impossible to see these control characters as you type them—they aren't echoed on the screen as the printable characters are (unless you toggle CONTROL-8)—so it is hard to correct mistakes. In general, the codes are much more useful in DISPLAY than in ACCEPT.

Most of the screen control available to you through control codes and the ANSI standard DISPLAY statement can be done more simply and with less chance of error by the extensions that we will deal with in the next section. Ordinarily, you will use direct console control only when

- there is no room (on a turnkey disk for example) for the run-time ADIS module as well as the application. (The ADIS module is the part of the Run-Time System which handles Apple /// ACCEPT/DISPLAY extensions.)
- you need special effects, such as color changes. In this case, you can do most of the work through ADIS and keep the use of control codes to a minimum.
- or when higher-level (ADIS) features prove to be a performance bottle-neck. It is less efficient to go through ADIS than directly via control codes to the driver. But don't blindly assume that you need the extra efficiency; in most cases, efficiency is a result of structure and not of implementation.

Note that the use of console control codes in your DISPLAY or ACCEPT statements can cause difficulties if you later wish to Animate such programs. In general, the Animator will not handle such statements correctly. However, the following tips may be of some help: Do not use **X** or **G** to pass such statements but rather set a breakpoint after them, and **Z**(oom to the breakpoint. If by chance you do pass such statements using **X** or **G**, you may still be able to recover by issuing the **U** command followed by a space. However, there is no way to recover if you have issued a RESET VIEWPORT code to the console driver.

Apple III Graphics Control

Graphics is not supported in this release of Apple *III* COBOL. Due to limitations of the system, you will not be able to write to the .GRAFIX driver. You can create some of the effects that might be obtained by using .GRAFIX by creating alternate character sets and using them with the .CONSOLE driver.

Apple /// ADIS Features

The higher level screen control features of Apple /// COBOL are interpreted at run-time by the ADIS module in the subdirectory COBOL/ of the /COBOLBOOT disk. These features allow you to make the Apple /// console into an "intelligent" terminal; that is, programs can define screen "forms" with some fields available for user input (accepting only numeric values in numeric fields), while the rest of the screen is protected.

Most of the ACCEPT/DISPLAY extensions are record oriented and fit quite naturally into the structure of COBOL. The object of an ACCEPT or DISPLAY is a record; fields in the record may be numeric, alphanumeric, or FILLER items. Numeric fields will ACCEPT only numeric input; alphanumeric fields will ACCEPT any printable character (but not control codes—a user can't type these in unintentionally); and FILLER items are ignored for both input and output (except to determine spacing between fields).

During data entry, the operator has full cursor control by use of the ARROW keys. These can be moved anywhere on the screen without erasing any protected data or current input. Pressing RETURN finishes the ACCEPT and gives control to the next statement. Note that each character is moved into its field of the record as it is typed, so there is no delay at the RETURN while a whole screen is transmitted.

General Screen Control

The most useful of the special controls to the console driver are available in a legible and secure way using Apple *III* extensions of the ANSI statements.

1. To clear the screen, leaving the cursor in HOME position at the top left corner, use the statement

   DISPLAY SPACE UPON CRT.

   This is a special use of the COBOL reserved word SPACE; if you write instead DISPLAY " " UPON CRT, you will get a single blank character written out at the current location of the cursor.

2. To move the cursor to line yy and column xx, use

   DISPLAY ... AT yyxx UPON CRT.

   or

   ACCEPT ... AT yyxx FROM CRT.

   The cursor can be positioned this way for input (ACCEPT) as well as output (DISPLAY). The line number yy and the column number xx are decimal numbers; they start at 01 and range to 24 (for line numbers) and 80 (for columns). Note that each must be two digits, written without any separating spaces.

   ACCEPT statements allow another form of cursor control. If you use the

   CURSOR IS data-name

   entry in the SPECIAL-NAMES paragraph, where the data-name refers to a PIC 9(4) item, the item will be updated at the end of each ACCEPT and the next ACCEPT will automatically show the cursor at the next available (that is, non-FILLER) position. This facility is useful for recording the cursor position so that it can be restored after switching screens. For example, you might use this to restore a user ACCEPT screen after displaying a HELP screen.

3. To highlight a display in inverse video, use

```
DISPLAY ... UPON CRT-UNDER.
```

This has the effect of switching on the inverse video mode before the display and switching it off afterwards. This can be combined with the AT phrase to make a highlighted display anywhere on the screen. The name CRT-UNDER is used for historical reasons; it stands for underlining, which is an alternate form of highlighting on some types of terminals.

Thus we can perform the same example as in the last section with

```
DISPLAY SPACE UPON CRT.
DISPLAY "Hello!" AT 0804 UPON CRT-UNDER.
```

Screen-Record Definition

Any record (01 level data description) can be displayed upon or accepted from the CRT. Any field of the record which is named FILLER will be ignored except for its effect on spacing between fields. Assuming an 80 character monochrome screen, the record in Figure 6-1 defines four lines of the screen.

```
01  REC-1.
    05  FILLER  PIC X(200).
    05  NAME    PIC X(40)  VALUE "NAME:".
    05  FILLER  PIC X(40).
    05  SSN     PIC X(40)  VALUE "SS-N:nnn-nn-nnn".
```

Figure 6-1. Screen Record Description

You may define as much of the screen as you want or as little. No logical line length is assumed; the console driver will split lines as they exceed the line length of your screen. But you must provide enough FILLER to make a complete CRT line for each line desired in the display. (The last line can be left incomplete.)

DISPLAY REC-1 UPON CRT will place NAME: followed by 35 spaces (the rest of the X(40) of its definition) on the right half of the third line and SS-N:nnn-nn-nnn followed by 25 spaces on the line below it. Anything already on the screen in the FILLER areas (the top two lines and the first half of the next two) will still be there; DISPLAY only writes over the positions occupied by non-FILLER items. (If you want to clear the screen before displaying this record, use DISPLAY SPACE UPON CRT.)

ACCEPT REC-1 FROM CRT will place the cursor at the start of NAME: and will allow the operator to type in any printable characters in the right half of this line and the next one. In order to protect the legends on our fields, we need to redefine the record as in Figure 6-2.

```
01  ACC-REC REDEFINES REC-1.
    05  FILLER        PIC X(205).
    05  ACC-NAME      PIC X(35).
    05  FILLER        PIC X(45).
    05  SS.
        10  1ST       PIC 999.
        10  FILLER    PIC X.
        10  2ND       PIC 99.
        10  FILLER    PIC X.
        10  3RD       PIC 999.
```

Figure 6-2. Screen Record Redefinition

Now instead of REC-1 you can accept ACC-REC; the legends are in FILLER areas of the redefined record and hence untouchable. Furthermore, the console will "beep" at you if you try to put anything but numbers in the SS subfields.

A program must still do some validation of numeric input. The operator can't type in anything except numbers, but he might forget to type in anything. In our example, the fields will then contain "n"s.

The general screen control phrases (AT yyxx, UPON CRT-UNDER) may be used in connection with records. Note that when you display a record UPON CRT-UNDER only the non-FILLER fields will be displayed in inverse video. The control phrases and record redefinition provide a general means for constructing menu oriented applications.

Cursor Control in ACCEPT

During execution of ACCEPT statements, the cursor can be moved on the CRT screen by the cursor control keys on the console keyboard. Table 6-1 summarizes the relevant key functions. LEFT- and RIGHT-ARROW keys have their normal meaning, except that they skip over any protected fields. The TAB key moves to the next tab stop column (skipping protected fields).

The UP- and DOWN-ARROW are a bit different; UP-ARROW moves back to the start of the previous unprotected field (or is motionless if at the start of the first such field), and DOWN-ARROW moves forward to the start of the next unprotected field. ESCAPE acts as a "HOME" key; type it to return to the first input position.

Function	Key
Home	ESCAPE
Tab forward a field	⟨down-arrow⟩
Tab backward a field	⟨up-arrow ⟩
Forward Space	⟨right-arrow⟩
Backward Space	⟨left-arrow⟩
Column Tab	TAB
Left Zero Fill	"." (period)

Note that when the DECIMAL-POINT IS COMMA clause is specified in the user program, the Left Zero Fill function is performed instead by a comma.

Table 6-1. Apple /// Cursor Control Keys (ADIS)

The zero-fill operator (".") is included in the table because it does indeed move the cursor, as a side-effect of filling in a numeric field. When typing a small value (say 15) into a large numeric field, you can't just type the two digits you want—that is likely to leave garbage (possibly non-numeric) in most of the character positions of the field. If you type 15., the ADIS module will automatically right-adjust the numeric digits in the field and supply any leading zeros necessary to fill the field with a value of 15. Zero-fill works only on pure numeric fields; in a numeric edited field, the . is accepted as an editing character.

The console control facilities of Apple /// COBOL have been discussed here at length, since they are the foundation of interactive programming in COBOL. In particular, the FORMS2 package is built on these foundations; that is the subject of the next chapter of this manual.

7

The FORMS2 Utility

Record redefinition for ACCEPT and DISPLAY, with its special use of FILLER and the provision for validating input keystrokes, is the basis for a powerful forms-generation utility. FORMS2 is a source-code generator: the user creates an arbitrary image on the display screen and specifies what fields on the screen are to accept alphanumeric and numeric data; FORMS2 then automatically generates the COBOL record descriptions for these images. In addition, FORMS2 can create a complete COBOL program to check that DISPLAY and ACCEPT statements work as expected on these forms. And if the form describes the record of a simple index sequential file, FORMS2 can generate a complete COBOL program for creating or updating the file. Using this last feature, it is possible for a non-programmer to sit down at the Apple *III* console, type in a standard form (or create his own new one), redefine some parts of the form with X's and 9's, and get out a usable file program. (Chapter Two of this manual has a recipe for doing just that.) A non-programmer might need some advice and help from a programmer (especially for any fields that need COBOL numeric editing), but the process is understandable even without general knowledge of programming or specific training in COBOL.

FORMS2 works in a cyclical process of defining and redefining the appearance of the screen. Normally, you will lay out the fixed contents of the screen first, leaving spaces or prompting characters (as was done in Figure 6-1) in the fields to be typed in. Then you redefine the record with an X for each character of an alphanumeric field and a 9 for each digit in a numeric field. FORMS2 keeps the first record in the "Background" (but still displays it on the screen) while you are redefining it in the "Foreground". When you are done, you can start over with a clear screen

and a new record (not redefined on top of earlier ones) or you can stop there. More complex patterns than this are possible; for now the main point is the basic structure—the FORMS2 Background contains material you defined earlier, while the Foreground is the material you are currently working on.

FORMS2 presents the user with a series of menu screens to aid in the task of creating screen forms; these menus usually have some options selected by default. There are also four "Help" screens which you can consult while using FORMS2. They provide a brief reminder of the commands available to you. There are occasions for overriding the default options, but they work fine in normal use. While you are learning the use of the utility, it is best to leave them in the default settings.

FORMS2 Outputs

At the beginning of a FORMS2 run, you specify certain initial information that remains in effect for the rest of the run. In particular you tell FORMS2 the basename for the files it will produce; the initialization process will also tell FORMS2 which of its possible outputs you want in this run. The output files are named by adding extensions to the basename you typed in. The possible outputs are:

base.CHK COBOL source code for a checkout program to test the screens created by this FORMS2 run. This code uses the COPY statement to piece together the skeletons FORMS2.CH1 and FORMS2.CH2 and the record output file base.DDS into a complete program.

Using this program, you can demonstrate exactly how your screen records will work for DISPLAY and ACCEPT statements. The program displays each record in turn, and if it has been redefined with variable fields, you may then type in data under actual operational conditions.

This feature provides a check for your definitions. For example, did you forget a digit in the field SSN? It is also useful in application development; you can realistically demonstrate proposed screen formats without having a full system and thus can catch specification changes early.

base.DDS This file contains the COBOL source code for the records defined in this run. You can COPY this file into your COBOL program, or add to it by editing it to make a complete program. Records are similar to those shown in Figures 6-1 and 6-2 except that record and field names are formed on the same basename as the file; the first record is base-00, the second base-01, etc. and field names are also generated in sequence. Thus the record base-01 will have fields base-01-0000, base-01-0001, etc. along with FILLERs. (These names can be changed, not only by later editing, but also within the FORMS2 run, by the S9 programming command discussed below.)

base.GEN COBOL source code for index sequential file maintenance program. Like base.CHK, this uses base.DDS and skeleton files FORMS2.GN1 and FORMS2.GN2 as COPY files to make a complete program.

base.Snn Screen image files, one for each record in base.DDS. The numbering is the same as that of the records—base.S00 is the way the (foreground) screen looked when you finished the first record, etc.

These screen images can be read back in a later FORMS2 run in order to change (correct, update, etc.) the original outputs. You can also retrieve and display these later in the same run that creates them, for comparison and reference in defining other records.

These files will all be written to the same disk drive. One of the initialization parameters you can supply is a prefix to be used for this purpose. In the absence of this prefix (i.e., if you don't fill it in), the SOS prefix will determine the directory or subdirectory into which FORMS2 will write.

FORMS2 Tutorial

The best way to learn FORMS2 is to go through some examples, illustrating its main features and showing you how to operate it, as well as pointing out the features that can be ignored until you have acquired some experience with it. The next few pages explore FORMS2 in action. As in Chapter Two, we assume you have an Apple *III* with one external drive.

Running FORMS2

Begin by booting the COBOL Run-Time System; place the /FORMS2 disk in your external drive. Change the SOS prefix to /DEMO and place the /DEMO disk in the built-in drive. Run FORMS2 by typing F. You will go through the same initialization sequence as in Chapter Two, first seeing the screen in Figure 2-1. Now type in a name you want to use for this run. The example here uses "TEST":

```
DATA-NAME & FILE-NAME      [TEST  ] (1-6 alphanumeric characters)
```

You can ignore the rest of the options for now; we will study them in the next section. Press **RETURN**; FORMS2 will accept your name and present you with initialization screen I02 (Figure 2-2). Here also you can ignore the various options for now; press RETURN again. At this point "work screen" W01 (Figure 2-3) will appear. This also has a number of options for special purpose FORMS2 runs. In normal use (and for this test run) you should just press RETURN, to accept the default options. Now you should see a blank screen, and you can begin to type in a sample form.

Edit Mode and Command Mode

Take a few minutes first to move the cursor around the screen using the arrow keys on the lower right of the keyboard. These keys will take you to any position on the screen for the layout of forms. One thing to notice is that the UP- and DOWN-ARROW keys return the cursor to the start of the line, as well as moving up or down one line. They work this way because the RETURN key is used for a different purpose; it is a switch to go back and forth from "edit mode" (which allows you to move around the screen and type in your form) and "command mode".

Command mode allows you to give FORMS2 instructions about how to handle the form you create in edit mode. If you haven't already done so, press RETURN now; the cursor will jump to the upper left corner, and you won't be able to move it away from the two positions marked by underlines there. These two positions are there for you to give commands to FORMS2. Any time they aren't visible, you can get them by pressing RETURN; if you pressed RETURN accidentally, you can get back to edit mode by pressing it again.

For a quick tour of the commands available in command mode, type the question mark command. (That is, get into command mode by pressing **RETURN**, type **?**, and then press **RETURN** again to issue the command. Just typing within the command area doesn't cause FORMS2 to do anything; this prevents accidental mistypings from doing strange things to your forms!). The question-mark command shows you the first help screen. Keep typing **?** (and **RETURN**) to get the other help screens. To return to edit mode, simply press **RETURN** by itself instead of another question-mark. Now move the cursor to some suitable place, and type in the form in Figure 7-1.

```
Name:    [_____________________________________________]

Address: [                                             ]
         [                                             ]
         [                                             ]

Phone:   [(nnn) nnn-nnnn    Ext:                       ]
```

Figure 7-1. Fixed Text Screen (TEST.S00)

The spacing is not critical in this example, nor position on the screen. Modify it to suit yourself. When you are satisfied, get into command mode by pressing **RETURN**; then type a **space** and another **RETURN**. This tells FORMS2 that you are done with this screen. Several things will now happen: FORMS2 will generate COBOL statements describing your form, showing these on the screen as confirmation. Then you will see your form again as FORMS2 writes it to the output file TEST.S00:

```
File created = TEST.S00
```

You are then told to continue by pressing **RETURN**. Doing this takes you back to the W01 work screen. The option shown under the cursor is now C ("Variable data redefines last screen"); last time it was A. FORMS2 follows this sequence of choices automatically, since it is the most common and useful pattern.

You should now press **RETURN**; FORMS2 will once again show you the form you created in the first phase of work. But this time, the form has been placed in the Background; it is there for you to look at, but your editing won't change it. What you type in now will go into the Foreground; its purpose is to describe what an operator can type into the form.

The name and address fields should accept any letters or digits; the phone number should allow only digits. Possibly you will want to reserve the last five places in the address for ZIP code. The way you tell FORMS2 all this is to type an **X** to stand for any character or a **9** to stand for a digit at every place in the form that a user can write. The sample above then shows up on the screen as in Figure 7-2.

```
Name:    [XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX]

Address: [XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX]
         [XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX]
         [XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX 99999]

Phone:   [(999) 999-9999    Ext: XXXXXXXXXXXXXXXXXXXXXXXXXXX]
```

Figure 7-2. Variable Text Screen

When this form is used in a COBOL program, only the places you mark in this way can be typed into; the rest of the screen is protected. You are now done, but (in order to get some more practice with commands) press **RETURN** and then **F** and **RETURN** again. This command invokes another work screen (W02) and gives you some options for Foreground/Background manipulation. The option is initially set to H, which is Display Foreground. Press **RETURN**; you will see the X's and 9's you just typed, but the form (now in the Background) doesn't appear. RETURN will take you back to the W02 menu. To see the Background by itself, type **I** and press **RETURN** to select the Display Background option. Normally, you see displayed a merged image of both Foreground and Background; you can use these commands to see them separately. Press **RETURN** again and then option A, which is automatically selected after you return from H or I, will take you back to the merged image.

Once you are satisfied with the variable text entry screen, issue the command to process it. (Just as before, you get command mode by **RETURN**; press the **space bar** to select the "process screen" command, and **RETURN** to execute it.) This time you will get the message

```
WORK SCREEN VALIDATION in progress
```

This means that FORMS2 is checking to be sure it can write correct COBOL PICTURE descriptions of what you typed. When the validation is done, you will again see COBOL statements for your screen, a display of the (Foreground) screen and the name of the file it was written to (test.S01 in this case). Follow the instruction to press **RETURN** and once again, the W01 work screen will appear.

You can continue making as many Foreground/Background pairs as you want. To end this example, however, type an exclamation point (**!**) and press **RETURN**. At this stage, the COBOL Data Description Statements and the COBOL Checkout program using them will be written out to files on your /DEMO disk, and the FORMS2 run is complete.

Compiling and Running the Checkout Program

It remains to compile the Checkout program and run that as a test of your form. Place the /COMPILER disk in the external drive. Type **C** to invoke the Compiler, and when the cursor reappears, type

TEST.CHK LIST

and the Compiler will begin working on the Checkout program. The LIST directive lets you watch the compilation in progress. You should get a final message from the Compiler showing that there are no errors in the program. At that stage the COBOL Command Line will reappear.

Type **R** and answer the prompting message by typing **TEST** and pressing **RETURN**. Your form will appear on the screen, and you can try typing in a name, address and phone number. Notice that you can only type in digits for the phone number and in the ZIP code area; also notice that you can't type anything except in the fields you defined in screen S01. For example, if you used the same pattern as in the sample above and left a space between the the X's and the ZIP code 9's in the address, you will see that there is no way of typing over this space.

Assuming you used the same form as in our sample above, you will notice that there is no underlining in the name field. FORMS2 is set up to think of the underlines as just a convenient way of showing spacing; the actual underlining of a paper form isn't needed to guide entry on the screen. But if you want lines on the form, you can have them: there is a command (M, one of the Programming Commands discussed below) that can change the "visible space" character to anything else you prefer.

As soon as you press RETURN, the Checkout program will ask if you want it to repeat. Type **Y** (and press **RETURN**); the data you typed before will reappear. You can make corrections, try different entries or just experiment with your screen. This process can help you discover errors or problems of "human engineering" in your screens. Each time you press RETURN, you have the chance to repeat the cycle. If you made more than one form, the cycle will go through each of them in turn. When you are done, give a "no" answer to the repeat question (this is supplied by default if you just press RETURN).

Modifying Previous FORMS2 Output

If you are unsatisfied with your form after this checkout, you can go back to FORMS2 and change it. Place /FORMS2 in the external drive and again type **F** to run the utility. You will see the same sequence of initialization screens as before. Again you should give the basename for the files—**TEST** in our example—and otherwise just accept the default options by pressing **RETURN**. After the second initialization screen, FORMS2 will ask if you really want to replace the earlier files; you will see the message

```
File already exists: TEST.DDS

                    overwrite? [N] (Y=Yes)
```

with the cursor positioned on N. Since you do want to change the file, type in **Y** instead, and then press **RETURN**. FORMS2 will ask the same question about TEST.CHK, and again you should answer **Y**.

The work screen W01 will then appear; press **RETURN** to prepare for the first work phase. Now you could just type in a complete new form, but the point of this exercise is to make small changes in an existing form. You need to read in the old form and place it in the Foreground so that it can be edited. To do this, go into command mode (**RETURN**) and type

command **F** (Foreground/Background commands). After you press **RETURN** again, you will see the W02 menu. Type in option **F** (Merge screen into FOREGROUND), and the cursor will drop down to the field labeled FILE-NAME. This command will take any screen image (.Snn) file on disk and copy it into the FORMS2 Foreground. Type **TEST.S00** into the FILE-NAME field and then press **RETURN**. The menu will reappear with option H showing; you can then look at the Foreground that was read. If for some reason you read in the wrong screen, you can clear the Foreground (select option **B**) and try again. Once you have the right screen image, press **RETURN** to get back to screen W02 and press **RETURN** again to get into edit mode to make the changes you desire.

In our continuing example, let us add one line to the address. FORMS2 has some editing commands that will save you the work of retyping large portions of the screen. One of these commands will duplicate any line of your form up to nine times; we can use that to add a line to the address. Move the cursor down to the start of the last line of the address; press **RETURN** to get into command mode, and then type **A1** and press **RETURN**. (Be sure the cursor is at the beginning of the line; the command won't work if it isn't.) Notice that the command writes over the blank line between address and phone number. You can insert a new separator line by another FORMS2 command; move the cursor down to the phone number line and (in command mode) type **I1**. This inserts one line just before the line containing the cursor.

Make any other changes you like on the screen, then release it for processing (press the **space bar** in command mode, as before). FORMS2 will check with you before overwriting TEST.S00; then you can go on to process the variable screen. Here also you could use the command option **F** to read back in your original screen of X's and 9's. But for a small screen it is just as easy to fill it in again as to edit the original version. When you are done, terminate the FORMS2 run—press **RETURN** to get to command mode, press the **space bar** and **RETURN** to process the screen, type **Y** and press **RETURN** to overwrite TEST.S01, and type **!** and press **RETURN** to complete processing and exit FORMS2.

FORMS2 Commands

After the quick tour of FORMS2 above, you should be familiar with the main structure of the utility. FORMS2 works in phases, one screen at a time, usually with pairs of screens—a fixed text screen defining the form and a variable text screen redefining it with fields a user can write in on the form. To handle pairs of screens, FORMS2 has a Background (to show what has already been defined) and a Foreground (for the current phase of work). In any phase, you can switch from editing the form to a command mode for telling FORMS2 what to do with the form. This section studies the available commands in detail, following the sequence of screens you see during a run of FORMS2.

Initialization

The first screen that appears when you run FORMS2 (labeled I01) requests general parameters for this run. If you accidentally typed F at the COBOL command level, you can exit from FORMS2 by using the general command !; this can be issued instead of the initializing commands and at any time during the work phases to exit FORMS2 without further effects. In particular, you can use ! instead of giving a data-name parameter in the first field of the first initialization screen, or in place of output selection on the second initialization screen. This screen is illustrated in Figure 2-1. The entries on this screen are explained below:

```
DATA-NAME & FILE-NAME [      ] (1-6 alphanumeric characters)
```

Specifies the basename to be used by FORMS2 to construct file names and COBOL data-names. The name should be a valid COBOL data-name and a valid SOS file name; if it isn't valid, you won't be able to continue to the next field. Instead, the field will be filled with question-marks in inverse video to warn you that the name is unusable. As a special case, an "!" in the first character position will cause FORMS2 to abandon the run.

```
CRT lines                [24]      (22 or 23 or 24)
```

FORMS2 normally assumes you want to use a full 24 line screen. If you need to reserve the last line or last two lines for special purposes (for example, inverse video output of error messages), this parameter can be changed to 22 or 23.

```
CURRENCY SIGN            [$]
DECIMAL-POINT            [.]
```

Most COBOL programs follow the standard American conventions for currency sign and decimal point. If you use FORMS2 to create screens for a program with different conventions (that is, if the program has SPECIAL-NAMES clauses for DECIMAL-POINT or CURRENCY SIGN) you should tell FORMS2 the values to be used for these. The characters in these options can then be used in numeric edited variables on your screens.

The second initialization screen (I02, Figure 2-2) selects the output that will be generated by the FORMS2 run. Given a name "base" from the I01 screen, the options are:

A (DDS) Generate base.DDS only; with this option, the only file written by FORMS2 will be the file of COBOL Data Description Statements for the screen records.

B (DDS & CHK) Generate base.DDS and base.CHK, the Checkout program that cycles through DISPLAY and ACCEPT of the forms produced in this run, for demonstration or testing.

C (DDS & CHK & Snn) Generate base.DDS, base.CHK, and one file per screen generated. Screen files can be read in subsequent FORMS2 runs, to allow direct modification of their contents without editing the base.DDS file. This option is selected by default.

D (DDS & Snn) Generate base.DDS and one file per screen. Use this option to omit generation of the Checkout program.

E (Snn) Generate a screen image file for each work phase in the FORMS2 run (each screen created). This option is useful in initial design of an application; the screens can be shown on the CRT screen (by using the Type option of the Run-Time System Utilities) or transferred to a printer for hard copy samples in memos or design documentation.

F (No files output) Use for experimentation with FORMS2.

G (DDS & Snn & GEN) Generate base.DDS, screen image files and base.GEN, the indexed file generation program based on the form created in this run. When this option is selected, only commands that can be implemented in the indexed file program are allowed during the work phases.

```
DEVICE/DIRECTORY PREFIX (0-40 Chars) [                ...                ]
```

This entry specifies for FORMS2 where to place the output files; the prefix will also be used in COPY statements in the Index (base.GEN) or Checkout (base.CHK) programs to locate the file base.DDS; thus if you type .D2 in this field, base.CHK will contain the COPY statement

COPY ".D2/base.DDS".

If you make no entry in this field, FORMS2 will not prefix any file names; in that case, the SOS prefix will determine where the files are written, and the COPY statements will also omit prefixes.

After you press RETURN on this screen, FORMS2 will check whether you have asked for an output file (base.DDS, base.CHK or base.GEN) which already exists (in the directory given by the prefix). If so, you are asked whether you really mean to overwrite the current version. If you don't answer Y (for yes), the run will terminate.

The next screen to appear is W01, the initial screen for each work phase (that is, each screen definition). The main function of this screen is to tell FORMS2 what type of COBOL record to produce from the screen image you type. The options are

A (Fixed text on clear screen) This is the initial default as you begin work. With this option, FORMS2 creates a new record, not REDEFINED to overlay any previous record. It will also clear Foreground and Background to give you a clean slate to lay out your form. The COBOL record created by this option is a sequence of FILLER fields and PIC X(n) fields with VALUE phrases exactly corresponding to what you type on the screen (with the exception of a "visible spacing" character, initially set to be the underline "_"; see Programming Command M below).

B (Fixed text on last screen) With this option, FORMS2 again creates a record containing FILLERs and PIC X(n) fields that exactly reproduce the Foreground typed in by the user. The difference from option A is that here the record is REDEFINEd on top of the previous record. The use of this option is in multi-stage menu processing: the application program presents its operator with a screen and requests some input; it then adds some further details without erasing what is already on the screen. (The STOCK2 program on the DEMO diskette is an example of this type of processing.) A typical sequence of work phases in FORMS2 for this would be A, C, B, C, etc.

C (Variable data redefines last screen) This option is usually taken as the second phase after creating a fixed text form by option A. The record created at this stage is redefined on top of the previous one, with the Foreground screen now defining where on the form the user can type. Spaces in the Foreground will again generate FILLER fields; anything else creates alphanumeric, numeric, or numeric-edited fields in the new record. This type of record is validated when you release it for processing. Only the following characters are allowed:

9 A sequence of consecutive 9's generates a PIC 9(n) field

8 A sequence of consecutive 8's also generates a PIC 9(n) field. The point of this is that a contiguous field on the screen (ZIP code, for example) can be split into a number of sub-fields.

X Any sequence of consecutive X's (upper- or lower-case) will generate a PIC X(n) field.

Y Any sequence of consecutive Y's (upper- or lower-case) generates a PIC X(n) field; Strings of X's and Y's can split up contiguous user input into sub-fields as 8's and 9's do for numeric fields.

$ CURRENCY SIGN; redefinable at initialization screen I01
. DECIMAL-POINT; (also redefinable)

These and the other numeric editing characters

, + - * / 0 B Z CR and DB

may be used for numeric-edited fields; the operational characters

S, V and P

are not allowed. For a numeric-edited field, the user can type in a value with or without editing characters; the Apple /// ACCEPT statement turns any valid input into a normalized form.

Non-expert users should consult with COBOL programmers before using numeric-edited fields. FORMS2 can't fully validate these fields on output, and there may be errors in the resulting record description that only the Compiler will detect.

*D (Variable data without redefinition) This option works like C except that the record does not REDEFINE the previous record. The option is useful, for example, to display a "prompt" message at some point of the screen which will then receive variable data. By placing the variable data here, you can avoid wiping out the prompt at ACCEPT time. You could also use this option to generate "templates" to reset all user ACCEPT fields to default values in one MOVE statement.

Finally, three of the general commands (!, ?, and Q) described in the next section are available in place of one of these options.

General Commands

These are the commands listed on the first help screen. These commands, and those listed on the other three help screens, are always available in command mode, that is at any time after the W01 work phase initialization. The first three of them (!, ?, and Q) are also usable at some points during initialization. Commands are one or two characters long; letter commands can be typed in either upper- or lower-case.

The general commands are:

! (Terminate FORMS2 Run) This command causes FORMS2 to stop work and return control to the Run-Time System. If you are between work phases (that is, if screen W01 appears to ask for a record option for your next form) the command will cause output of DDS, CHK or GEN files that you requested at screen I02. At any other time, the run is abandoned without writing these files.

? (Display Help Screen) FORMS2 will display H01 as a reminder
?n of the general commands. If you press ? again, it will go on to H02, then to H03, then to H04 and finally back to H01 again. You can cycle around the help files as long as you like. To avoid looking at screens you don't want, you can also type ?2 or ?3 or ?4 to get a specific one of the later screens.

Q (Quit Screen Processing) Temporarily returns to the start of work phase; you may then change your mind about the type of record being generated. When you return to edit mode, the screen you typed will still be there, but will now be handled according to the new option. You may also type Q during the initialization phase (while selecting outputs at screen I02); in that case you will return to screen I01 and can modify any parameter settings you made there.

SPACE (Process Work Screen) Pressing the space bar releases the current screen for output. Processing will depend on which types of output you have selected and on the type of screen. Variable data screens are validated; DDS records and a screen image (Snn) file will be created if they have been requested. Processing can then resume with a new work phase for a new record, or it may terminate with the ! command.

_ (Re-enter Edit Mode) The underline character is the command to resume editing; since the command mode cursor is on an underline character when you enter edit mode, this becomes the default command. You may also use it explicitly.

X (Reposition Command Mode Cursor) If you use the upper left corner of the screen for your form, it may be confusing to have the command line appear there when you press RETURN. This doesn't destroy the actual contents of the Foreground screen, as you see when you return to edit mode; however, you may prefer to place the command line elsewhere on the screen. To do so, move the cursor (while in edit mode) to the position you want; then enter command mode and type X. The next time you enter command mode, the line will appear at the position you have selected.

* (Index Key Boundary) This command is used for index file program generation (output option G on screen I02). It is used to mark the separation of "key" from "data" areas in the file records for the index program. Place the edit mode cursor on the first data character of the variable screen and then issue * from command mode.

Screen Manipulation

The commands summarized on help screen H02 deal with general screen handling. Most of these commands assist in editing the Foreground screen; the F command invokes a subsidiary menu of commands (W02) for Foreground/Background handling.

An Repeats the current line (the line on which the edit mode cursor is found; the cursor must be at the start of the line) over the next n lines (n = 1 to 9).

Cn Inserts n blanks on the current line (n = 1 to 9), starting at the current edit mode cursor position. Everything to the right of the cursor moves n positions further right; if this shift takes any character off the screen, it is permanently lost.

Dn Deletes n characters on the current line (n = 1 to 9), starting with the character underneath the cursor. If the cursor is fewer than n positions from the end of the line, the rest of the line is deleted.

F
Fx Foreground/Background Manipulation. There are ten options, labeled A through J. If you type just F, FORMS2 will show the options menu (work screen W02); if you type F and an option letter (that is, FA, FB, ..., FJ), FORMS2 will act on the option selected without going through the menu. The options are

A Return to edit mode. No operation on either Foreground or Background.

B Clear Foreground. Erases anything already typed into the Foreground, without affecting the Background image.

C Clear Background. Erases anything now in the Background. Note that this doesn't affect any record created in an earlier work phase, but the record will no longer be visible as a template for the Foreground. If you need to retrieve an earlier record after using option C, use option G below.

D Overlay Background data onto Foreground. Whatever was in the Background is placed in the Foreground, just as if you had typed it in again. Useful for repeating all or part of a standard template form.

E Overlay Foreground data onto Background. Foreground contents become part of the Background template for editing, without having gone through the process of COBOL record generation. Useful to construct guidelines for future editing that you don't want or need in the COBOL record descriptions.

F Overlay Screen Image File onto Foreground. Allows you to edit an existing form, either to modify it or as a base for creating a new form. When you select option F, G, or J, FORMS2 will prompt for the name of the screen image file to be read in.

G Overlay Screen Image File onto Background. As for F except the file goes into the Background, where it remains visible, but does not affect the current edit.

H Display Foreground. Shows exactly what the current record contains, unconfused with any Background image.

I Display Background. Shows exactly what is in the Background. Occasionally you will type Foreground contents directly over Background data; in this case the original Background image will partly disappear. To see the full Background again, use option I.

J Display Screen Image File. Shows the contents of a screen file without placing it in either Foreground or Background. You can browse through a library of screens and choose the one you want to work with.

In Insert n blank lines (n = 1 to 9) before the current line. The current line and any beneath it move down on the screen, and any that move past the last line of the screen are permanently lost.

Kn Delete (kill) n lines (n = 1 to 9), starting at the current line. If there are fewer than n lines, all are deleted.

O
O1 These commands enable you to switch the automatic screen preparation of FORMS2 on and off. O (the letter "oh", not the digit "zero") turns on automatic screen preparation; when it is in effect, merging of the Foreground to the Background is done automatically when the screen is released at the end of a work phase. The O1 command ("oh one") turns off automatic screen handling.

Un
Vn These commands allow the cursor to move up or down without returning to the first column as it does under control of the UP- or DOWN-ARROW keys. Un moves it up n lines, Vn down n lines (n = 1 to 9). If there are fewer than n rows to move, the motion goes as far as it can.

Programming Commands

Help screen H03 summarizes some commands that are useful in tailoring FORMS2 output to special programming needs. Most of these commands are inconsistent with the index file generation program and are disallowed in that context.

G
G0 (Generate Screen Coordinates) Changes the names of the data-items in the screen records to have the form

base-rr-yyxx

for record number rr, with yy the row number and xx the column number for the position of the item on the screen. This may be helpful for a programmer using the Apple /// AT yyxx extension to the ACCEPT and DISPLAY statements. G0 may be used interchangeably with G.

G1 (Restore Default Naming) Resumes the standard naming of fields in screen records; instead of yyxx, the terminating four digits in the name are just the number of the field.

Jn (Multiple Space Reset) FORMS2 is initially set to create a FILLER whenever there are two or more spaces together on your screen. A large number of small FILLER items, however, can degrade the performance of your program. You can reset this parameter to prevent FORMS2 from creating FILLER items when there are n or fewer contiguous spaces (n = 0 to 9).

The initial setting is J1; with this any single space will be combined with named items before and after it. J2 will combine one or two spaces with the surrounding data-items, and so forth. J or J0 creates a FILLER even for a single isolated space.

Mx (Define Spacing Character) FORMS2 allows a character to be used to make visible areas that will be spaces in the final form; the initial character for that purpose is an underline "_". The Mx command resets the "visible space" character to x, where x can be any character on the keyboard. Note that you can press the space bar after typing M, but this is not a good idea; it causes all FILLER records to be suppressed.

P (Show Cursor Position) Causes FORMS2 to pause and display the current cursor position at the command line area.

Sn (Special File Control) A collection of special-purpose options related to the output of this current work phase:

S0 Cancels any other Sn (S3 or S9 below) command in effect and resumes normal processing.

S1 Suppresses COBOL statement generation for this screen; no data description statements will be written and no reference will be made to this screen in the Checkout program. The record number is still incremented for this screen; the next screen will have the next higher number. This option doesn't need to be cancelled by S0; generation of COBOL text resumes with the next screen unless the command is reissued then.

S2 Suppresses screen image generation (no file base.Snn is written for this screen). Commonly used to eliminate screen image files for the variable data (option C) work phase. Like S1, this lasts only for the current screen; no cancelling command is needed.

S3 Forces user to name screen image files (instead of forming the names base.Snn). In normal processing, when the file base.Snn already exists, you will be asked if you want to overwrite it. If not, you can then supply any file name you like for the new screen image. When S3 is in effect (i.e., until cancelled by a following S0) FORMS2 will always request that the file name be given explicitly.

S9 Causes FORMS2 to halt after generating each COBOL data description for you to examine it. You may then make minor editing changes before pressing RETURN to send the statement out to the .DDS file. This option is used primarily to change the data-names to more mnemonic values than base-rr-nnnn.

Windows

The final help screen (H04) displays commands for screen window creation. This facility allows a group of lines to be treated as an isolated screen. At the start of processing, FORMS2 will usually generate FILLER for lines skipped over at the top of the screen. This can be wasteful of memory if you want a form to appear only on the lower part of the CRT (for example, below some other screen). You can reset the top and bottom lines handled in a work phase to eliminate the excess FILLER. Note that a COBOL program must use the AT phrase in the DISPLAY statement to show the screen in the correct position. To assist in coding such DISPLAY statements, FORMS2 renames records created in windows: if the window starts at line ll, for example, the record name will be

base-rr-ll

instead of just base-rr. Windows in FORMS2 can be shown with or without a line of delimiters just outside their boundaries. These delimiters are simply a convenience for you while you are editing; they do not actually go into either Foreground or Background, and they aren't written into the record description.

Note that you can write on the screen outside the window. Such text doesn't become part of the record, but it will be written to the screen image file (and so may be used for documentation).

The window commands are

W (or W0) Position the cursor at the start of the window.

W1 Sets up a window starting at the current line, showing a line of delimiters on the previous line.

W2 Defines the end line of the window at the current line; a line of delimiters will show on the next line down.

W3 Sets up a window starting at the current line, without a line of delimiters before it.

W4 Sets end of window without delimiters showing.

W5 Shows delimiters before the start of the window.

W6 Shows delimiters after the end of the window.

W7 Erases starting delimiters (restoring work screen contents that the delimiters obscured).

W8 Erases end delimiters (restoring work screen contents).

W9 Positions the cursor at the end (lower right corner) of the current window.

Indexed File Program Generation

An indexed sequential file is a sequence of records that have a special field, called the key, to help in finding any record quickly just by naming its key. It is a standard mode of file organization throughout business data processing applications. FORMS2 can generate simple programs for creating and updating this type of file, without the operator having any knowledge of COBOL programming. The first example in Chapter Two gives a demonstration of this feature, and a practical guide to using it. This feature is not a panacea, but it can cope with many practical problems, leaving programming skills available for more complex work. Our discussion in this section is general in nature; for a tutorial on the feature, and for details on the use of the program once it has been generated, see Chapter Two.

The generator option is selected from the second initialization screen (I02). Instead of the default (option C), type G; FORMS2 will then set itself up and prevent you from giving any commands that are inconsistent with the job of generating the index file program. You will go through just two work phases: one defines the way the data entry screen will look to the user, and the other defines the data entered on the screen (and into the file).

The file is defined by fields in the variable text work screen; the file record will have exactly one byte for each character typed in this screen (X, Y, 8, 9 or numeric editing character). The key will be part of the record; it must be the first field or set of fields in the record and cannot be more than 32 characters long. Sometime during the editing of the variable screen, you

must give FORMS2 the * command, with the cursor resting on the first character past the key area. If you don't give a valid * command, FORMS2 will keep returning you to the edit mode to do so.

You will not be able to use most of the Programming or Window commands discussed above; they are incompatible with the code produced for the Index File Program. There will be a line of delimiters (hyphens) on the bottom of the screen, since the generated program uses a line for a number of its own messages.

When you finish this work screen (release it for processing by entering command mode and pressing the space bar and RETURN), FORMS2 will automatically exit: you won't get back to the W01 menu, since only one pair of screens can be handled in this mode.

8

Program Debugging and the Animator

Two independent types of debugging are available in Apple /// COBOL. The first involves optional "debugging lines" that are included if the "DEBUGGING MODE" switch is present in the "SOURCE-COMPUTER" sentence. The second is the Animator utility, a source-level, screen-oriented program testing and debugging tool.

Animator can in fact bring a program "alive" on the CRT screen, to allow study and debugging at a very high logical level. The Animator needs memory space to hold its own code and buffers as well as those of the program you are debugging. In addition, it must read in a "dictionary" file (name.D00 for a root segment or name.Dnn for an overlay segment) to handle the cross-reference between the COBOL source code and the intermediate object code. The dictionary space becomes a limiting factor on small systems; if your configuration has only 128K of memory, you will find it difficult to animate non-trivial programs. However, as memory increases to 256K, it should be possible to animate any program you can compile.

ANSI Debugging Mode

The ANSI standard for COBOL contains a number of elements to aid in program debugging. The principal idea is that certain parts of the source program are treated as comments unless a certain run-time switch is set; if it is set, those parts automatically swing into action. The programmer can place "debugging lines" of code into his program at any point, and the

Compiler takes care of activating these when the run-time switch is on. To prevent unnecessary code output, the Compiler requires a compile-time switch (the WITH DEBUGGING MODE clause of the SOURCE-COMPUTER paragraph) or it will ignore the ANSI debugging facilities.

The full list of COBOL language elements for debugging will be found in Chapter Ten of the *Apple /// COBOL Language Reference Manual.* Use of these elements is not special to the Apple /// COBOL system and will not be dealt with here. The Apple /// provides for level one ANSI debug with the run-time switch "A". Refer to Chapter Three of this manual for information on setting and clearing run-time switches. Because the Animator is far more convenient and powerful, it is envisioned that ANSI debug will be little used in Apple /// COBOL, however.

Animator — General Description

The Apple /// Animator is a COBOL-oriented debugging tool which applies to any COBOL program compiled by the Apple /// COBOL Compiler. No special language elements are needed, and any part of a program can be studied in normal operation using the Animator. It is thus a tool for studying or maintaining a program already in use and not simply a checkout for code under development. The Animator executes the same intermediate code file (.INT file) that would be run in normal operation under the Run-Time System R command. Several other files are needed by the Animator as well, to enable it to translate back and forth between the intermediate code and the original source code. These files are generated automatically by the default settings of the Apple /// Compiler.

Animator is a dynamic debugger; you can not only follow the execution of the program, you can also change the code or the flow of control as you interact with it. Thus you can actively check out your guesses about the behavior of the program and make experimental corrections during a program run.

The main aim of the Animator is to free the COBOL programmer from the need to be aware of the internal representations of either data or procedural code, so that his programs can effectively be debugged at the same logical level as they were written. This is done by using the screen as a "window" into the source program and "animating" execution by moving the cursor from statement to statement as execution proceeds.

You can speed up execution or slow it down; you can switch off the animation altogether or execute up to a "breakpoint" set at a point of interest. You can interrupt execution at any point, without previous preparation of breakpoints, simply by pressing the space bar on the keyboard. While execution is suspended, you can easily examine any part of the source code by means of simple commands acting on the screen display. This means that it is not even necessary to have a printed compilation listing in order to debug a program.

Various other debugging functions are available, all invoked by typing a key. Animator keeps you informed of what you can do at any point by using the bottom few lines of the screen for "menus" of the commands available at that point in execution. (Only the top 20 lines of the screen are used for the display of source code.)

Animator is "screen-oriented" in the same way that display editors are—you move the cursor around on the screen to some part of the code that interests you, and Animator determines from the cursor position what reference this makes to either data items or procedural statements. Or you can refer directly to data items or procedures by name. Where pointing with the cursor isn't enough, you instruct Animator using normal COBOL syntax. For instance, to change the value of a data item, you type the new value in COBOL literal format (so non-numeric literals are enclosed in quotes). It is even possible to type in a complete COBOL source statement for immediate execution.

Animator Tutorial

To gain some experience with the Animator and learn a few of its basic facilities, let us look again at the sample programs from Chapter Two. These should all have .ANM, .ACP and .D00 files on the DEMO disk, as well as the source and intermediate code files (extensions .CBL and .INT). Assuming that you have one external disk drive, place the /ANIMATOR in this and boot up the system; as usual for our examples, reset the prefix to **/DEMO** and place /DEMO in the built-in drive. Type **A** at the COBOL command line to load the Animator and when the initial message

```
* A/// Animator V1.0 (C) Apple Computer Inc.
```

appears, type the name of the program to be animated; for our first example, use the name **PI**, or equivalently, **PI.INT**—the Animator looks first for the intermediate code file and then for the other files related to it. When it has found the files, Animator will display on the screen the first 20 lines of the PROCEDURE DIVISION of PI.CBL; note that there are sequence numbers on these lines supplied by Animator, whether or not the program source code has sequence numbers. Below the delimiting line of hyphens, you see the basic Animator menu

```
B(rk-pnts) D(isp) E(xec) F(ind) L(evel) M(on) P(-c) Q(uery) S(creen) U(ser) ?
```

To get started, type **E** for Execute; this shows the subsidiary menu of options in executing the program:

```
EXECUTE - X(single step) sKip I(till next If) G(o) Z(oom) S(top run)
```

Type **X**; the cursor will move down to the second statement; type **X** several more times, and watch the cursor move.

Animator is actually tracing out the execution of the program. As you type X's, you will get to the bottom of the display. The Animator will then replace the display with another screenful of statements from the program. Anytime control transfers off the screen, Animator will bring up the source code transferred to.

To liven things up a bit, you can examine what is happening to the data in the program. Type **M** and you will see another subsidiary menu:

```
MONITOR - S(et) U(nset) N(ame)
```

To select a data item to monitor you can either move around with the cursor and use the Set option, or you can name the item. To watch the variable TERM as the calculation goes on, type **N** for Name and type in **TERM** (or **term**; lower-case is equivalent) and press **RETURN**. The current value of TERM will be displayed on a line below the main menu line. As you continue to type the **X** key, you will see at each statement that this display is erased and then replaced with whatever is the value of TERM at that point.

When you get tired of pushing X, type **G** instead; the menu line now reads

```
GO - 1-9(speed=4) Z(oom) space-bar(halt)
```

and the cursor will be traveling by itself over the statements of the LOOP paragraph. You can change the speed it is moving at by typing a digit from 1 to 9. Initially, the speed setting is 4; when you change it, the new value is reported on the menu line. Try typing in **1** and then **9** to see the extremes of the range. Then experiment to find a speed that you find comfortable to watch.

You can interrupt this process at any time by pressing the space bar. This will return you to the main menu. Try typing **U**; this command shows you the "user screen" buffer that Animator uses to keep track of DISPLAY statements made by the program. Since Animator uses all of the real display screen, it diverts a DISPLAY from the program to this buffer. It is always available for your examination; type **U** to get to it, then type any key to get the Animator screen again.

The user screen buffer is used only for the Apple *III* extensions to COBOL ACCEPT and DISPLAY statements. Animator refers ANSI standard ACCEPT and DISPLAY statements to the menu area below its line of hyphens.

This is probably a good time to notice that you can move freely around the screen using the ARROW keys, but you can't see the rest of the program that way. Type **S** to get the sub-menu

```
SCREEN - N(ext) P(revious) T(op) E(nd) V(iew) H(alf) F(ull) =/+/—
```

Typing **T**, for example, will replace the screen with the starting lines of the program. You can play around with these screen commands as you like. Note that F(ull) undoes the effect of H(alf). If you use a command incorrectly, Animator will flash a line of pound signs ("#") over the menu. By this stage you have probably lost your place in the program; type **P** for the program-counter menu

```
PROGRAM COUNTER - W(here) R(eset)
```

Type **W**, and the screen will be redrawn with the cursor at the next statement to execute. None of the screen manipulations change your

place in the program. Resume execution with **G9** and let the program run to its termination (after TERM becomes 0). Animator won't let the program exit automatically; it gives you an opportunity to study the program first, requiring you to ask it explicitly to stop:

```
WARNING - Next instruction is STOP RUN - S(top run) C(ontinue)
```

If you type **C**, you can examine the state of the program before it stops. For example, you can type **D** (for Display) and then type **N**; Animator will show you the value of N (it's 0037) after the computation loop. The cursor is underneath this displayed value. Animator gives you a chance to change the value if you wish to; otherwise just press **RETURN** to get back the main menu. Type **E** for Execute and **X** to single-step. You will get the warning again. Type **S** this time to stop run; you will see the final user screen display, and control will return to the COBOL Command Line.

Animation and ACCEPT

The PI program has no ACCEPT statements in it. To observe how the Animator handles such statements, let us use as the next example the stock file program of Chapter Two. Again type **A** to get the Animator, and STOCK1 to name the program to animate. Type **X** until you come to the ACCEPT statement; with one more X the source code screen will disappear and the user screen will take its place. For the ACCEPT statement to execute, it must receive some input from this screen, signaled by a **RETURN**.

Press **RETURN** leaving the stock code field blank; Animator will recreate the source code screen, with the cursor positioned after the ACCEPT statement. The next statement tests for a blank stock code field as a signal of termination; if you type **X** twice more, you will see the test succeed and program control transfer to the END-IT paragraph. This leaves only a little more of the program to go, but you can use Animator to reset the control flow. Use **UP-ARROW** to move the cursor back to the ACCEPT statement, and use the Reset option of the Program Counter command: type **P** and then **R** with the cursor on the ACCEPT. Now another **X** will show you the screen again and you can type something into the stock code field. Press **RETURN** to continue normal execution.

In a program like this, it is useful to have both PROCEDURE and DATA DIVISION code in view simultaneously. To do so, type **S** to get the screen command and take the **H** option. The lower half of the screen now shows the start of the program. If you type **S** and **N** a few times, you can advance this screen to show the record ENTER-IT, which is the object of the ACCEPT statement. Note that you can type either 〉 or . to advance the screen, also. You can also adjust the division on the screen to show more of the PROCEDURE DIVISION and less DATA; place the cursor on the mid-screen delimiter line, type **S** for screen and this time you have the options U (to move the delimiters up one line) and D (to move them down one line). You can adjust within either half of the screen by S+n or S—n commands (n being the number of lines up or down to scroll the screen).

With the various data-names visible on the screen, either in PROCEDURE or in DATA DIVISION, you can examine or change data values with the Query command. Place the cursor on the item you are interested in (use CRT-STOCK-CODE in our example) and then type **Q**. As with the Display command for a named item, the Q command will show you the current value at the bottom of the screen and leave the cursor below it for you to enter a new value. If you don't want to change the value, just press **RETURN**. Note that if you inquire about a record or an intermediate level data item, Animator will only show you the contents of the first elementary item contained in it.

When you are through playing with the code, type **E** to get the Execute menu and then **S** to stop the run. (This will be an abnormal termination of the stock program; if you prefer to exit normally through END-IT, go through the ACCEPT with a blank stock code field.)

This short survey demonstrates a few of the available commands. It should show you the approach taken by Animator to debug a program. You can refer to data or procedure by name or you can point to them with the cursor. You can execute statements, skip them or jump to arbitrary places to examine what will happen. If a program seems to be going off on a wrong track, you can look at the data, change it if necessary, and try the logic again. In short, Animator will do for you, using the actual executable code of your program, what you would ordinarily do laboriously with pencil on a printed program listing.

Operational Considerations

The Apple /// COBOL Compiler will produce all the files needed by the Animator; that is, the ANIM directive is in effect by default. If you know you are not going to use the Animator on a program, you can suppress the production of the Animator-related output files (.ANM and .ACP) by using the NOANIM directive. Unless you are compiling a large program that can't be animated within the constraints of your memory, it is a good idea to allow the default. Even established programs behave oddly at times, and you can begin tracking problems immediately, without having to recompile, if the Animator files are available. Compiling the program with ANIM in effect doesn't slow down the program in normal execution, nor does it produce a substantially larger code file.

Note that the Animator requires the source code to be available. Any library files referred to by COPY statements in the main source file must also be present when you run the Animator.

Program listings aren't necessary with the Animator, but they remain useful—20 lines is not a large piece of a program. If you make compilation listings, it is useful to give the Compiler the directives RESEQ and COPYLIST. This will ensure that your printed listing has sequence numbers that match the sequencing Animator supplies.

The name you supply Animator at the start of a run is the name of the executable code file (/directory/name.INT or just name if the directory concerned is the current prefix, since Animator will supply the default .INT extension). The other files do not need to be on the same disk as the .INT file; if Animator doesn't find them where it expects to, it will prompt you with the message

```
FILE BELOW NOT FOUND - S(top run) C(ontinue) A(lter drive)
<.dn/name.ext>
```

with the last line being what it expected. You should type **A** and the device name to tell Animator where to look for the file in question.

Note that using the Utility program to set the prefix to the volume and subdirectory containing the program to be animated will solve this problem. Generally, it is a good idea to use the same prefix when animating that you used to compile.

There is one "directive" that the Animator will accept on the command line, after the name of the program to be animated; this is the ZOOM directive. If you want to animate a called subprogram and don't want to animate all the levels before the subprogram is called, type

main-program-name ZOOM(subprogram-name)

Execution will proceed without interference by the Animator until the subprogram is reached.

Animator Commands

Commands to Animator are issued from menus appearing at the base of the screen. Most of the commands shown on the main menu have further sub-menus of command options. The heavily used execution commands (options of the E command on the main menu) may also be invoked when the main menu is showing. These options are

```
G (and GO speeds 1 to 9), I, K, V, W, X
screen commands: +n -n =n < , > .
```

In the descriptions below, any command that can be issued at the main command level will be marked with an asterisk ("*"). Note that some of these commands do not appear on the main prompt line. After execution of a sub-command, whether from a subsidiary menu or the main level, the main menu is redisplayed.

The main command menu is too long to fit completely on one CRT line. The rightmost listing on the first line is a question mark—if you type ? the remainder of the main menu will appear. Thus, on initial entry to the Animator, you see the menu

```
B(rk-pnts) D(isp) E(xec) F(ind) L(evel) M(on) P(-c) Q(uery) S(creen) U(ser) ?
```

Typing ? then displays the rest of the main level commands:

```
C(ompile) N(ame) unT(il) lOcA(te)?
```

A second ? will switch back to the first part of the main menu. All commands on both lines are available, whether or not the part of the menu listing them is on the screen at any given time.

You enter commands simply by typing the appropriate key. If an invalid entry is made at any time, Animator will signal the error by briefly replacing the prompt line with a line of pound signs ("#") and emitting a "beep".

Some commands require you move the cursor first to point to the appropriate place in the source code. The main menu shown above is displayed whenever execution is suspended. Anytime it is displayed, you can move the cursor using the normal cursor keys. UP- and DOWN-ARROW keys and the RETURN key have their normal meanings—RETURN moves the cursor to the start of the next line, and the cursor keys maintain vertical alignment.

The main Animator commands fall into three broad categories: screen manipulation (A, F, O, S and U); execution control commands (B, C, E, L, N, P and T); and display and modification of data (D, M and Q). Each of these will now be discussed in detail.

Screen Manipulation Commands

Animator uses the screen as a window into the source code text. The screen manipulation commands allow you to reposition the source code window to any point within the COBOL source program. Note that moving the cursor by itself doesn't affect the point at which execution will be resumed.

Animator automatically displays resequenced line numbers against the source text. These will be the same as those appearing on the compilation listing if the directives RESEQ (and COPYLIST if appropriate) were specified to the Compiler.

* *The S(creen) Command*

Type **S** and the following subordinate menu is displayed:

```
SCREEN - N(ext) P(revious) T(op) E(nd) V(iew) H(alf) F(ull) =/+/-
```

These commands reposition the window to display a different part of the source text. The options are

N
* \>
* .

Displays next screen of source text; actual amount shown depends on whether the screen is split (Half option). The new screen overlaps the current one by two lines. The "\>" key (think of it as an arrow pointing forward) also moves the display forward one page; *it may be typed when the main menu is displayed.* Note that you don't need to shift the key—both \> and . will display the next screen.

P
* \<
* ,

Displays the previous screen of source text. The \< key, usable from the main menu, also displays the previous screen. Like \>, this key doesn't need to be shifted; both \< and , display the previous screen.

T

Displays the top screen of source text (that is, the first lines of code).

E

Displays the screen at the end of the source text.

V

Repositions the window so that the source line indicated by the cursor is on the third line. (The cursor must be positioned first, before typing S.)

H

Splits the screen in half (two windows) with a dividing line of hyphens. The lower window is positioned to show the top of the source code. Subsequent screen commands operate on the window in which the cursor is positioned.

F

Restores the full screen display (single window).

* =n

Repositions the window so that the nth source line is aligned at the third line of the window. The Animator sequence numbers increment by ten; they are therefore ten times the line numbers. Drop the final zero of a sequence number to get an absolute line number for this command.

* +n

Scrolls the window forward n lines.

* −n

Scrolls the window back n lines.

Note: =, +, − all position the cursor for entry of a numeric quantity followed by RETURN.

There is one special case of the screen command. If the screen is split and you position the cursor on the dividing line of hyphens, then when you type S, the following subordinate menu is displayed:

```
SCREEN DIVIDER - U(p) D(own)
```

These two options allow the relative size of the two windows to be altered:

U	Moves the screen divider Up one line.
D	Moves the screen divider Down one line.

* *The F(ind) Command*

This command instructs the Animator to search forward from the current cursor position through the source text for a specified string of characters. If it is found, the screen window is positioned with the line containing this string as the third screen line and the cursor is positioned following the string. If it is not, the main screen display remains unchanged, but the Animator indicates the failure by beeping and restoring the main prompt line.

Type **F** and the cursor is positioned for entry of either

''string'' (followed by **RETURN**)

or

''string''M (followed by **RETURN**) (to search only in main file)

where a string is any sequence of characters (including spaces); it needn't be a complete word. Any character not forming part of the string can be used in place of the quotation marks to delimit the string.

Note that the search distinguishes upper- and lower-case, so you must type the string exactly as it appears in the source. The F operation only examines columns 7 through 72 of the source code; the displayed line numbers are ignored.

The optional M after the string instructs the Animator to search only the main source file and not any library (COPY) files.

* *The Locate Commands*

The lOcAte commands (O and A, suggested on the main menu by capitalizing these letters in lOcAte) allow you to find the declaration of any name in the program (either a data-name or procedure-name). They differ only in that O searches for the name on which the cursor is positioned, while A looks for a typed-out name. Either

position the cursor to rest on the start of any occurrence of the name, then type **O**,

or

type **A**, then type the name followed by **RETURN**.

The screen window will be repositioned in the source text, and the cursor will be placed on the declaration of the specified name.

* *The User Screen Command*

Type **U** and the user screen buffer (containing the screen image from the Apple *III* COBOL extensions of the DISPLAY statement) will appear on the screen, replacing the source code window(s). Note that this user screen is not used for standard ANSI DISPLAY statements; they are redirected by Animator to its menu area on the main screen.

Inverse video effects in the animated program, achieved by use of the DISPLAY ... UPON CRT-UNDER clause, will not be preserved in this user screen buffer. All characters will appear in the same positions on the display as during a normal program run, but all will be in normal video.

This display remains on the screen for examination until any other key is typed; at that point the source code screen will be redisplayed. Note that the user screen appears automatically during the execution of an ACCEPT FROM CRT statement and at the end of the Animator run (that is, as a result of the Stop Run command).

Execution Control Commands

These commands allow initiation and control of program execution. They also control the degree of animation used.

* The B(reakpoint) Command

Type **B** and the following subordinate menu is displayed:

```
BREAK-POINTS - S(et) U(nset) C(ancel) eX(amine) I(f)
```

The options allow breakpoints to be set at which execution will halt automatically. A breakpoint may have an associated condition. Some of these commands require the cursor to be positioned to point to the relevant COBOL statement. The cursor must be positioned on the first character of the COBOL verb before you type B. Up to four breakpoints may be set concurrently.

Type the appropriate key, where:

S Sets a breakpoint at the statement pointed to by the cursor.

U Clears (unsets) the breakpoint pointed to by the cursor.

C Cancels all breakpoints currently set.

X eXamines breakpoints by repositioning the window within the source code and positioning the cursor at the statement of the next breakpoint in the file. Successive calls on this function will move the cursor to point to each breakpoint in turn.

I Sets a breakpoint with an associated condition at the statement pointed to by the cursor. The Animator positions the cursor for entry of the required conditional expression which you write in COBOL format, followed by a RETURN. Subsequent examination of breakpoints (see X above) will both point to this breakpoint and display the condition. When execution reaches this breakpoint, the Animator will only halt if the condition is true.

Example: after typing

```
I
```

type

```
TRAN-NO IS EQUAL TO 0
```

Note that you do not need to type the word "IF".

* *The unT(il) Command*

This command allows specification of a conditional expression (without the IF) which will cause execution to halt if the condition becomes true at any point during execution. When execution halts, the condition is automatically "switched off". Note that this facility will significantly degrade the speed of execution, since the condition must be tested at the end of each statement in the source program.

Type **T** and the following subordinate menu is displayed:

```
UNTIL CONDITION - S(et) U(nset) eX(amine)
```

Type the appropriate key, where

S Positions the cursor for entry of a condition in COBOL format. Note that this only sets the condition. Note: program execution must be resumed by use of the GO or ZOOM commands.

U Cancels the previously set condition.

X Displays the previously set conditional expression.

* *The E(xecute) Command*

Type **E** and the following subordinate menu is displayed:

```
EXECUTE - X(single step) sK(ip) I(until next If) G(o) Z(oom) S(top run)
```

These options initiate execution, with or without animation, in a variety of ways.

Any of these commands, except for S(top run), may also be entered directly from the main menu line without preceding it with an E.

Type the appropriate key, where

* X eXecutes a single COBOL statement and moves the cursor to the next statement.

* K sKips a single COBOL statement, without executing it, and moves the cursor to the next statement.

  Note: If the final statement of a PERFORMed paragraph is skipped, control does not exit from the PERFORM but passes to the next statement in the source code.

* I Executes without animation up to the next IF statement; then execution halts and the cursor is repositioned at this IF statement.

* G Initiates animated execution. After each statement is executed, the cursor is moved to the next statement in the source code. The speed of execution can be varied by typing a digit from 1 to 9 (1 = slowest, 9 = fastest). The speed may also be entered before initiating execution. Execution proceeds until halted as described below.

* Z Initiates execution without animation (Zooms). Upon reaching the first DISPLAY UPON CRT or ACCEPT FROM CRT, the user screen is displayed, replacing the source code, and remains on the screen until execution is halted as described below.

  S Stops execution after displaying the current user screen.

After initiation by one of the above commands, execution proceeds as described above, unless it is halted in one of the following ways:

1. The space bar is pressed.
2. A previously set breakpoint is reached.
3. The condition specified by means of the unT(il) command becomes satisfied.

4. A STOP RUN statement is reached. In that case, Animator displays the following prompt:

```
WARNING - Next instruction is STOP RUN - S(top run) C(ontinue)
```

   Type **S** to stop, or **C** to regain control.

5. A run-time error occurs. In that case, Animator displays the following prompt:

```
RTS ERROR: nnn - S(top) C(ontinue)
```

   Type **S** to stop, or **C** to regain control. There is a listing of Run-Time Errors in Appendix C.

* *The L(evel) Command*

Animation normally traces execution into any level of nested PERFORMs. This command allows a "threshold" level to be set at any level of nesting, such that any PERFORMs subordinate to this level are treated as a single statement for animation purposes, that is, the cursor will not be moved into PERFORMed procedures below the threshold level.

Type **L** and the following subordinate menu is displayed:

```
PERFORM LEVEL = 01, THRESHOLD LEVEL = 03 S(et) U(nset) E(xit) Q(uit)
```

Note that the display indicates both the current PERFORM level and the threshold level currently set.

Type the appropriate key, where

S Sets the threshold level at the current level.

U Clears (unsets) the threshold level, restoring animation at all levels.

E Completes execution of the current PERFORM without animation, repositioning the cursor to the statement following the PERFORM, and then sets the threshold level at this point.

Q Causes immediate abandonment of the current PERFORM without executing further statements. This allows a tidy exit from a closed loop in a PERFORMed procedure.

* *The P(-C) Program Counter Command*

This command provides facilities to determine the point at which execution will start (or resume), or to change this point. To change the restart point, you must first place the cursor at the statement at which execution is to start.

Type **P** and the following subordinate menu is displayed:

```
PROGRAM COUNTER - W(here) R(eset)
```

Type the appropriate key, where

* W Repositions the screen window if necessary and places the cursor at the next statement to be executed. This is useful as a check after use of the source screen manipulation commands, but note that it is not necessary since the Animator will resume execution at the correct position and display the screen appropriately as it does. Note that this command may also be entered from the main menu line (without the preceding P).

If the program being animated contains a DISPLAY of any console control codes, animating the program may cause the display to become jumbled since Animator redirects ANSI DISPLAY statements to the menu area of its display. When animating such programs, it is useful to Zoom to a breakpoint placed after these DISPLAY statements. In any case, you can use the W command to clean up the display if it has become jumbled.

R Resets the execution start point (program counter) to the current cursor position. Before typing P, place the cursor on the first character of an executable statement (COBOL verb).

* *The C(ompile) Command*

This command enables immediate compilation and execution of a specified COBOL statement (or statements).

Type **C** and the cursor is positioned for entry of the required COBOL statements followed by **RETURN**. If you type in multiple statements, they should not be separated by periods. A final terminating period may be entered, but if omitted it is assumed. The statements typed aren't retained; they will be executed once only.

Note that any statement so compiled is not placed into the source and disappears after execution.

If a syntax error is found in the entered statements, the following prompt is displayed:

```
COMPILER ERROR NO. nnn - S(top) C(ontinue)
```

Type **S** to stop, or **C** to regain control. Compiler error numbers are detailed in Appendix B.

Certain complex statements cannot be handled (for example, INSPECT). These will return COMPILER ERROR NO. 001, and it will be necessary for you to type simpler statements to achieve the same effect.

* *The N(ame) Command*

Within a hierarchy of programs the Animator will by default animate all programs that were compiled with the ANIM directive in effect. Any other programs are executed normally. This command allows finer control over which programs are animated.

Type **N** and the following subordinate menu is displayed:

```
PROGRAM NAME - W(hich) A(ny) T(his) O(ther)
```

Type the appropriate key, where

W Displays the current program name.

A Causes any program compiled for animation to be animated. Note: this is the initial default; typing this key resumes default behavior after a previous T or O option in another N(ame) command.

T Tells Animator to animate only the program currently executing. Any others will be executed normally.

O Positions the cursor for entry of the name of the next program to be animated followed by RETURN. This must be entered as an alphanumeric literal (within quotes). Note: Animation of the current program continues until termination. All others except the named

program will be executed without animation. Animation resumes with the named program.

If you need to start the run by executing normally until a specified program is reached, use the following command directive after the main program name when running the Animator:

ZOOM (program name)

Display and Modification of Data

These commands provide facilities for examining and changing the contents of specified data items. You select the data items by either specifying the data-name or pointing with the cursor.

* *The D(isplay) Command*

This command allows you to examine or change any data item in the program, whether or not it currently appears on the screen, by typing the name.

Type **D** and the cursor is positioned for entry of the required data-name followed by RETURN.

The value of the item is displayed (with appropriate conversion in accordance with its PICTURE clause.) For alphanumeric or group items, non-ASCII characters are displayed as periods. Numeric data items display leading or trailing signs according to their pictures, and decimal points are inserted if used.

Following display of the data item, the cursor is positioned to the next line for entry of a replacement value if required. If you don't want to change the displayed value, simply press RETURN; otherwise type in the new value in COBOL literal format (alphanumeric literals must be in quotes). Replacement is performed in accordance with the standard rules for the COBOL MOVE statement.

No more than the first 80 characters will be displayed at one time. For longer data items, the remainder of the field can be displayed by specifying

data-name + n

where n is a numeric offset into the data item. Any changes will be applied only to the part of the item currently displayed.

* *The Q(uery) Command*

This command allows display or modification of a data item referenced by pointing with the cursor.

Position the cursor to the first character of any occurrence of the data-name within the source code, then type **Q**.

The data item is displayed and may be modified as described for the D(isplay) command, with the exception that only the first 80 characters of long data items can be referenced.

There are two special cases in the use of the Q)uery command. A Query on a condition-name is allowed, but no subsequent modifications are allowed. A Query on a file-identifier in an FD declaration or a SELECT statement gives the file status only and allows no modification.

* *The M(onitor) Command*

This command enables automatically repeated display of a single specified data item (without modification).

Type **M** and the following subordinate menu is displayed:

```
MONITOR - S(et) U(nset) N(ame)
```

Type the appropriate key, where

S Sets the data item pointed to by the cursor to be the object of monitoring. (The cursor must be moved to point to an occurrence of the data-name before you press M.)

U Clears (unsets) the monitor.

N Positions the cursor for entry of a data-name followed by RETURN. You can specify an offset into a long data item in the same manner as for the D(isplay) command.

The monitored item is redisplayed after each animation "step" (for example, after each statement in "GO" mode).

A

Summary of Compiler Directives

This appendix describes each of the available Compiler directives, in alphabetical order. Directives in effect by default are marked with an "*". Parameters to the directives may be specified as shown below (between quotation marks or in parentheses, with optional spaces between the directive and the parameter). The "NO" prefix to negate the effect of a directive may be joined to the directive (as below) or separated from it by one or more spaces.

***ANIM** Files are created which are necessary for later Animation of the program being compiled.

NOANIM Creation of the files need for Animation of a program is suppressed.

BRIEF The text of error messages is suppressed and only the error numbers are listed. This is assumed if the error message file COBOL.ERR cannot be found; otherwise the default is NOBRIEF.

***NOBRIEF** Explanatory error messages appear in the program listing underneath the line of asterisks flagging each syntax error.

COMP Enables the Compiler to generate code for binary treatment of PIC 99 and PIC 9(4) COMPUTATIONAL numeric variables; with this directive in effect, some statements involving such variables will have results different from the ANSI standard.

***NOCOMP** Forces standard ANSI treatment of PIC 99 and PIC 9(4) COMPUTATIONAL data items.

COPYLIST The contents of any file(s) named in COPY statements are listed; the listing file page headings will contain the name of any COPY file open at the time a page heading is output.

***NOCOPYLIST** The contents of files named in COPY statements in the program are not listed.

CRTWIDTH "integer" Specifies the width in characters of the user screen. This facility is used in Format 1 (ANSI standard) DISPLAY statements to enable the user to plan the separation points in display of data-items too long to fit on one physical CRT line.
The default is 128 character positions.

NOCRTWIDTH Specifies that Format 1 (standard ANSI) DISPLAY statements will not be compiled, with consequent space saving in memory due to control tables not being necessary. Any such statements will give an "UNRECOGNIZED VERB" error message.

DATE "string" The comment-entry in the DATE-COMPILED paragraph, if present in the program undergoing compilation, is replaced in its entirety by the character string as entered between parentheses in the DATE Compiler directive. This date is then printed at the top of every listing page under the file name. Note that in the absence of a directive, the system time and date will be used as if they had been entered as the string.

***ECHO** Error lines are echoed on the console CRT. For each error, the source line producing it, a flag line containing the error number and (unless the BRIEF directive is in effect) an explanatory message will appear on the screen.

NOECHO Suppresses echoing of error messages to the console.

ERRLIST The listing is limited to those COBOL lines containing any syntax errors or flags, together with the associated error messages.

***NOERRLIST** The listing contains all source code lines as well as any syntax error or flag messages.

FLAG "level" This directive specifies the output of GSA Compiler certification validation flags at compile time. The parameter "level" is one of the character strings:

LOW Produces validation flags for all features higher than the Low Level of Compiler certification of the General Services Administration (GSA).

L-I Produces validation flags for all features higher than the Low-Intermediate Level of Compiler certification of the GSA.

H-I Produces validation flags for all features higher than the High-Intermediate Level of Compiler certification of the GSA.

HIGH Produces validation flags for all features higher than the the High Level of Compiler certification of the GSA.

A*III* Produces validation flags for only the Apple *III* COBOL extensions to standard COBOL as it is specified in the ANSI COBOL Standard.

IBM Flags the IBM-compatible options listed in Appendix J of the *Apple III COBOL Language Reference Manual.*

***NOFLAG** Suppresses generation of Compiler validation flags.

FORM "integer" Specifies the number of COBOL lines per page of listing (minimum is five). The default is 60 lines per page.

NOFORM No form feed or page headings are to be output by the Compiler in the list file.

FORMFEED "function-name" See below under SYSIN.

IBM Enables the extensions listed in Appendix J of the *Apple III COBOL Language Reference Manual.*

***NOIBM** Treats these extensions as syntax errors.

***INT "external-file-name"** Specifies the file to which the intermediate code is to be directed. If the file exists already it will be overwritten. If INT

is specified without a file-name parameter, the Compiler appends the extension .INT to the name of the source file; this is the extension that the Run-Time System expects for an executable intermediate code file. INT, without file-name parameter, is the default.

NOINT No intermediate code file is output. The Compiler is in effect used for syntax checking only.

***LIST "external-file-name", *PRINT** Specifies the file to which the listing is to be directed; this may be a character device or a disk file. The default is "source-file.LST". If no file name is given, .CONSOLE is assumed. If output is to a file that already exists, that file will be overwritten.

NOLIST, NOPRINT No list file is produced; used for fast compilation of "clean" programs.

REF The four-digit location addresses are included on the right hand side of the listing file.

***NOREF** Suppresses output of the four-digit location addresses on the right hand side of the listing file.

RESEQ If this is specified, the Compiler generates COBOL sequence numbers, renumbering each line in increments of ten.

***NORESEQ** Suppresses generation of sequence numbers; columns 1 through 6 are printed on the listing just as they appear in the source file, with no effect on the compilation.

FORMFEED "function-name", SYSIN "function-name", SYSOUT "function-name", TAB "function-name" Changes the name recognized by the Compiler for certain function names in the SPECIAL-NAMES paragraph for this compilation.

B

Compile-Time Error Messages

This appendix lists all error messages produced by the Compiler, along with explanations of their meanings. You will notice that the numbers listed are not continuous; that is, there are gaps in the numbering. The Compiler should never have cause to generate an error message with an unlisted number. If you ever encounter such a number, consult your Apple *III* Product Technical Support office.

Error	Description
01	Compiler error; consult Technical Support
02	Illegal format : Data-name
03	Illegal format : Literal, or invalid use of ALL
04	Illegal format : Character
05	Data-name not unique
06	Too many data or procedure names declared or insufficient memory
07	Illegal character in column 7 or continuation error
08	Nested COPY statement or unknown COPY file specified
09	'.' missing
10	Statement starts in wrong area of source line
21	'.' missing
22	DIVISION missing
23	SECTION missing
24	IDENTIFICATION missing
25	PROGRAM-ID missing
26	AUTHOR missing

27 INSTALLATION missing
28 DATE-WRITTEN missing
29 SECURITY missing
30 ENVIRONMENT missing

31 CONFIGURATION missing
32 SOURCE-COMPUTER missing
33 OBJECT-COMPUTER/SPECIAL-NAMES clause error
34 OBJECT-COMPUTER missing
36 SPECIAL-NAMES missing
37 SWITCH clause error or system name/mnemonic name error
38 DECIMAL-POINT clause error
39 CONSOLE clause error
40 Illegal currency symbol

41 '.' missing
42 DIVISION missing
43 SECTION missing
44 INPUT-OUTPUT missing
45 FILE-CONTROL missing
46 ASSIGN missing
47 SEQUENTIAL or RELATIVE or INDEXED missing
48 ACCESS missing on indexed/relative file
49 SEQUENTIAL or DYNAMIC missing or >64 alternate keys
50 Illegal ORGANIZATION/ACCESS/KEY combination

51 Unrecognized phrase in SELECT clause
52 RERUN clause syntax error
53 SAME AREA clause syntax error
54 Missing or illegal file-name
55 DATA DIVISION missing
56 PROCEDURE DIVISION missing or unknown statement
57 Program collating sequence not defined

61 '.' missing
62 DIVISION missing
63 SECTION missing
64 File-name not specified in SELECT statement or invalid CD name
65 RECORD SIZE integer missing or line sequential record >1024 bytes

66 Illegal level no (01-49),01 level required, or level hierarchy wrong
67 FD, CD or SD qualification syntax error
68 WORKING-STORAGE missing
69 PROCEDURE DIVISION missing or unknown statement
70 Data description qualifier or '.' missing

71 Incompatible PICTURE clause and qualifiers
72 BLANK illegal with non-numeric data-item
73 PICTURE clause too long
74 VALUE with non-elementary item, wrong data-type or value truncated
75 VALUE in error or illegal for PICTURE type
76 Non-elementary item has FILLER/SYNC/JUST/BLANK clause
77 Preceding item at this level has >8192 bytes or 0 bytes
78 REDEFINES of unequal fields or different levels
79 Data storage exceeds 64K bytes

81 Data description qualifier inappropriate or repeated
82 REDEFINES data-name not declared
83 USAGE must be COMP, DISPLAY or INDEX
84 SIGN must be LEADING or TRAILING
85 SYNCHRONIZED must be LEFT or RIGHT
86 JUSTIFIED must be RIGHT
87 BLANK must be ZERO
88 OCCURS must be numeric, non-zero, unsigned or DEPENDING
89 VALUE must be literal, numeric literal or figurative constant
90 PICTURE string has illegal precedence or illegal character

91 INDEXED data-name missing or already declared
92 Numeric-edited PICTURE string is too large

101 Verb not recognized or "." missing
102 IF....ELSE mismatch
103 Operand missing or has wrong type or undeclared or "." missing
104 Procedure name not unique or USE procedure duplicated
105 Procedure name same as data-name
106 Name required
107 Wrong combination of data-types
108 Conditional statement not allowed in this context

109 Malformed subscript
110 ACCEPT/DISPLAY wrong or Communications syntax incorrect

111 Illegal syntax used with I-O verb
112 Invalid arithmetic statement
113 Invalid arithmetic expression
115 Invalid conditional expression
116 IF statements nested too deep, or too many AFTERs in PERFORM statement
117 Incorrect structure of PROCEDURE DIVISION
118 Reserved word missing or incorrectly used
119 Too many subscripts in one statement
120 Too many operands in one statement

141 Inter-segment procedure name duplication
142 IF....ELSE mismatch at end of source input
143 Operand has wrong data-type or not declared
144 Procedure name undeclared
145 INDEX data-name declared twice
146 Bad cursor control : illegal AT clause
147 KEY declaration missing or illegal
148 STATUS declaration missing
149 Bad STATUS record
150 Undefined inter-segment reference or error in ALTERed paragraph

151 PROCEDURE DIVISION in error
152 USING parameter not declared in LINKAGE SECTION
153 USING parameter not level 01 or 77
154 USING parameter used twice in parameter list
155 FD missing
157 Incorrect structure of PROCEDURE DIVISION
160 Too many operands in one statement

In addition to these numbered error messages, the following message can be displayed with subsequent termination of the compilation:

FATAL I-O ERROR: file name

where file name is the erroneous file. Any intermediate code file produced in such a case is not usable. The conditions that will cause this error are:

Disk overflow
File directory overflow
File full
Impossible I-O device usage

Other operating system dependent conditions may also cause this error.

C

Run-Time Error Messages

153 Subscript bounds overflow: zero or greater than the number of occurrences of the item

154 PERFORMs nested too deep: usually results from using GO TO to jump out of the range of a PERFORM instead of jumping to an EXIT statement at the end of its range

157 Not enough program memory: may occur on initial program load or when the Run-Time System attempts to load one of its own modules to perform a function such as indexed I-O, SORT/MERGE or ACCEPT/DISPLAY on CRT—see the ON OVERFLOW clause of the CALL statement for handling sub-programs that can't be loaded

160 Overlay loading error: unable to load overlay or segment; for example, file not found, too many files open, or invalid file structure

161 Illegal intermediate code: operation not recognized by the Run-Time System—implies bad program file

162 Perform n times nested too deep: too many levels of PERFORM n TIMES. Error may be reported in processing a complex arithmetic expression

163 Program counter out of range: address in GO TO, PERFORM or ALTER lies outside the program area—implies bad program file

164 Program not found: loading error (for example, file not found, too many files open, invalid file structure)

165 Version number error: incompatible releases of Compiler and Run-Time System; the Compiler used may have generated code that will not be executed correctly

166 Recursive call illegal: attempt to CALL a COBOL module recursively (i.e., when it is already active)

167 Too many USING items: the list of items supplied in a CALL ... USING statement is longer than the Run-Time System can handle

168 Linkage Error: parameter count mismatch between CALL and PROCEDURE DIVISION USING statements, or an attempt to access a linkage section item when a program executes directly or when the item isn't included in the PROCEDURE DIVISION USING list

174 ISR file loading error: Intersegment Reference File for a segmented program cannot be loaded; for example if the file was not found, or had an invalid file structure

176 Illegal intersegment reference: illegal use of GO TO, PERFORM or ALTER across segment boundaries in a segmented program

177 Cancellation of active program. Attempt to CANCEL a COBOL module that is still active (it has been called but has not yet executed an EXIT PROGRAM statement)

178 Error during save: unable to SAVE the program successfully; for example, when not enough disk or directory space

200 to 255 Unclassified error condition: may be caused by a disk or directory structure error not checked for by the operating system—consult Technical Support if the problem is reproducible after transferring all files in use to another disk

The errors listed above are all fatal; they cause the Run-Time System to terminate the program, displaying the error number and naming the segment involved in the error. (The segment will always be the "root" segment in the case of a non-segmented program.)

File errors cause termination of program execution unless a status field has been specified for the file concerned. If a status field is specified, the character "9" is returned in the first byte and the error code is returned as a binary value in the second byte. The programmer must then check for error conditions and take corrective action or terminate the program run. See Chapter Six for the handling of this status byte.

FILE HANDLING ERRORS

Decimal Error Number	Meaning	File Organization Applicable
1	Out of Buffer space. Insufficient memory available for operating system I-O buffers.	All
4	Illegal file name. File or device name contains illegal character(s).	All
5	No such device. The device or disk specified cannot be found by the system.	All
7	Out of disk space. No space available on disk for file creation/extension.	All
9	Disk directory full. No space available in disk directory for further entries.	All
13	File not found. The file specified cannot be found by the system (in attempting to open for input a non-existent file not declared OPTIONAL).	All
14	Too many files open. Attempt to open more files (16) than can be catered to by the system; note that segment changes in segmented programs and calls to non-resident subprograms require the Run-Time System to open a file to satisfy the request. May mean that the Run-Time System can't acquire the memory it needs for I-O buffers.	All
15	Too many open ISAM files. Attempt to open more indexed files (8) than can be catered to by the system.	Indexed
16	Too many open devices. Attempt to open more devices than can be used simultaneously by the system.	All
24	Hardware I-O error. Device or disk I-O error; for example, checksum error, read after write verification failure, parity error, etc.	All
25	Operating system data error. Bad directory entry, invalid block allocation map, etc.	All

FILE HANDLING ERRORS

Decimal Error Number	Meaning	File Organization Applicable
37	File access denied. Access to file denied by operating system; for example, in an attempt to read from an output device, write to a write-protected file, etc.	All
38	Incompatible disk. Disk created under another operating system or operating system version, or clashes with one already loaded (same name, etc.).	All
39	Incompatible file. Directory entry indicates incorrect file type, device type illegal for file organization, etc.	All
41	Bad file. File corrupt or in unrecognized format. Possibly caused by opening a file with a different organization or record length from that used to create it. May occur if the file was not properly closed after a preceding update; for example, because of a hardware failure.	Relative
42	Misformed line sequential file. A text file was opened and found to contain >0 and <1024 bytes. A normal text file has 1024 bytes of operating system data at the beginning.	Line Sequential
43	File information missing. Indexed files—means that one file is missing completely or that a file is shorter than indicated by its internal control data (generally caused by a failure to close the file after an update, for example because of a hardware failure).	Indexed
47	Index structure overflow. Indexed files—means that the maximum number of levels permitted in the index tree structure has been exceeded: the file must be reorganized before further data is added.	Indexed
129	Record zero illegal. An attempt has been made to access record zero on a relative file.	*Relative

* Means the file is bad, if reported for an indexed file.

FILE HANDLING ERRORS

Decimal Error Number	Meaning	File Organization Applicable
139	Record length or key data error. Attempt to open an existing file where record length or key data differs from that used when it was created.	* Line Sequential * Relative Indexed
141	File already open. Attempt to open a file that is already open.	All
142	File not open. Attempt to close an unopened file.	All
143	Rewrite/delete not preceded by read. Rewrite or delete on a file in sequential access mode was not preceded by a successful read.	Sequential Relative Indexed
146	No current record. Sequential read attempted on a file in dynamic or sequential access mode when no current record was defined.	Relative Indexed
147	Wrong open mode for read/start. Attempt to read from or start on a file that has not been opened input or I-O.	All
148	Wrong open mode for write. Attempt to write to a file in sequential access mode that has not been opened output or extend, or attempt to write to a file in random or dynamic access mode that has not been opened input or I-O.	All
149	Wrong open mode for rewrite/delete. Attempt to rewrite or delete on a file that has not been opened I-O.	Sequential Relative

* Error may not be detected at open time but gives rise to a bad file when I-O is attempted.

D

FORMS2 Command Summary

FORMS2 operates in phases, introduced by way of menus. After the initialization phases are complete, the user enters data on the screen in edit mode or gives commands in command mode. Work on a form begins in edit mode; to get into command mode, press the RETURN key.

Initialization

The following items must be determined at the initialization of any FORMS2 run; most of these take default values if RETURN is pressed without any data entered. Only the first item must be specified:

1. DATA-NAME & FILE-NAME. Mandatory one to six character name to be used throughout the run as the base of all file names and COBOL data-names.
2. CRT lines. Default value 24; may optionally be set to 22 or 23 to assure that no records generated by FORMS2 occupy more than this number of lines on the screen.
3. CURRENCY SIGN. Default value "$".
4. DECIMAL-POINT. Default value ".".
5. Output files generated. The options are:

   A DDS file of COBOL Data-Description Statements only.

   B DDS file and CHK program to checkout the forms.

C DDS and CHK files, and Snn screen image files.

D DDS and Snn files only.

E Snn files only.

F No files output.

G DDS and Snn files, and GEN index-file program.

6. DEVICE/DIRECTORY PREFIX. Default is none. Zero to forty characters, used as a prefix to the name base.DDS for COPY statements. Used to specify a device for the output file at compile time which may differ from its location during the FORMS2 run.

Work Phase Initialization

With the release of each screen generated by a work phase, the next work phase is automatically initiated unless the O command is in effect. To initialize a work phase, you must choose one of the screen creation options. Initial default is A followed by C.

A Fixed Text on Clear Screen. Input defines a new COBOL record comprised of FILLER entries in blank spaces and PIC X(n) data with VALUE clauses defined by edit mode data.

B Fixed Text on Last Screen. Like A but the new record REDEFINES the previous screen record.

C Variable Data on Last Screen. Input "X"s, "Y"s, "8"s, "9"s and Numeric-editing characters against the background of the previous screen. Output is a record which REDEFINES the last one and can be used in ACCEPT statements protecting the rest of the screen.

D Variable Data without Redefinition. Like C, but the record is not a redefinition of the background.

General Commands

Once the screen type option is set, editing can begin for the new form. At any stage of the editing process, the general commands are made available by pressing RETURN. Several of these commands use non-alphabetic keys; these are listed here first and are then followed by the other commands in alphabetic order.

! (Exclamation point). Exit from FORMS2 run. Normally issued at the reappearance of the W01 work-phase initialization screen, to indicate that no more forms are to be generated. Used at other times to abort the run.

? (Question mark). Exhibit the (next in sequence of four) HELP screen. Initially shows H01, then H02, H03 and H04 before again showing H01.

?n For n = 1, 2, 3 or 4 shows the corresponding HELP screen.

_ (Underline). Resume edit mode. This is the default command.

* (Asterisk). Marks the boundary between key field and data in the Variable Data record generated for an Index File program.

b (Blank). Release the current form for processing, to end this work phase. This command will not be accepted when the GEN option has been selected, unless you have marked the Variable Data record with the boundary between key field and data fields (* command).

An (n = 1 to 9) Duplicate the current line n times, below the cursor; the cursor must be at the start of the line.

Cn (n = 1 to 9) Insert n blank characters on the current line; everything to the right of the cursor moves n positions to the right (dropping characters that shift off the screen.)

Dn (n = 1 to 9) Delete n characters on the current line: first the character under the cursor, then the succeeding n-1; if there are fewer than n characters past the cursor, only the rest of the current line is deleted.

F Display the Foreground/Background menu (work screen W02). Allows selection of commands FA through FJ by pressing keys A through J, returning to the W02 menu after each command, until option A (return to edit-mode) is selected. All the options may be selected without reference to the menu by the following two-keystroke commands:

- FA Return from Foreground/Background manipulation to edit-mode.
- FB Clear Foreground; erases current Foreground screen, leaving the Background unaffected.
- FC Clear Background; erases current Background screen, leaving the Foreground and any records already generated unaffected.
- FD Overlay Background data onto Foreground; places the entire Background screen into the Foreground (as if typed in again in edit-mode).
- FE Overlay Foreground data onto Background; places Foreground screen contents into the Background (without going through any COBOL record generation).
- FF Overlay Screen Image File onto Foreground. Requests name of the file, then reads it into the Foreground screen. It may then be edited further for current record generation.
- FG Overlay Screen Image File onto Background. Requests name of the file, then reads it into the Background.
- FH Display Foreground; shows Foreground contents, without merging the Background contents into it. When control returns to edit-mode, the screens are again merged.
- FI Display Background; shows Background contents not merged with Foreground. If the Background has been overwritten in edit-mode, this will restore its original appearance.
- FJ Display Screen Image File; requests the name of the file and displays it on the screen without changing either Foreground or background. Useful for "browsing" through a library of screen images before using bp or FG.

G Generate screen coordinates names; changes the names in the

- G0 COBOL record outputs, so that data-items have row and column position appended instead of sequential field number. G0 (with digit zero, not letter "oh") is synonymous with G.
- G1 (digit one, not letter "L"). Restore default naming; change names of data-items in a record to append the sequential field count, instead of the row and column position of the item.

In (n = 1 to 9) Insert n blank lines before the current line; moves the current line and subsequent lines down the screen, dropping any lines that move below the pre-defined limit (CRT lines on the first initialization screen).

Jn (n = 0 to 9) Multiple space reset; initial setting 1. All blank areas on a fixed screen between visible characters are replaced by FILLER items, whenever the number of contiguous spaces is greater than n; if n=0, all spaces become FILLER.

Kn (n = 1 to 9) Kill (delete) n lines, starting with the current line; if there are fewer than n lines, all remaining lines are deleted.

Mx (x any printable character; initial setting "_") Make "x" the "visible-space" character: use of this character in edit-mode causes creation of a blank in the COBOL value clause describing the item.

O (letter "oh") Turn on automatic Foreground to Background merging (default option).

O1 ("oh one") Switch off automatic screen preparation. With "O1" in effect, any Foreground/Background manipulation must be done by using the "F" commands.

P Show current cursor position (yyxx for row yy, column xx).

Q Quit: during initialization, returns to the first screen for revision of the parameters already selected; during a work phase, returns to the W01 work initialization, for revision of the type of record being generated.

S0 Cancel S option (S3 or S9) in effect.
S1 Suppress COBOL statement generation for current work phase (automatically cancelled in next work phase).
S2 Suppress screen image generation for the current work phase; usually used for "Variable Data" screens not needed for documentation. (Automatically cancelled in next work phase.)
S3 Request user names for all screen image files; in normal operation, a name like base.Snn is assumed.
S9 Edit pause for each COBOL line output; allows minor editing within lines of the COBOL DDS records, for example, changes to the names of data-items.

Un (n = 1 to 9) Move the cursor vertically upwards n lines, leaving it in the same column as on the starting line. If the cursor is fewer than n lines from the top, it moves to the top line.

Vn (n = 1 to 9) Move the cursor vertically downward n lines; leaving it in the same column as on the starting line. The "V" key is intended to suggest an arrow pointing down. If the cursor is fewer than n lines from the bottom limit of the screen (CRT lines parameter), it moves to the bottom.

W0 Window "home" key; positions the cursor at the start of the top line in the current window; "W" is synonymous with "W0".
W1 Define starting line of a window at the current line; shows a line of delimiters ("-"s) on the previous line.
W2 Define end line of window as the current line; shows a line of delimiters on the next line.
W3 Define starting line of a window at the current line; no delimiting line shown.
W4 Define end line of a window at the current line; no delimiting line shown.
W5 Display delimiters on the line before the current window.
W6 Display delimiters on the line after the current window.
W7 Erase any delimiter line before the current window; restores any work screen contents previously obscured by delimiters.
W8 Erase any delimiter line after the current window.
W9 Position cursor at the end (lower right-hand corner) of the current window.

X Reposition the command line (the two underlines "_ _" that appear when command-mode is entered by pressing the RETURN key in edit-mode) to appear at the current cursor position.

E

Animator Command Summary

This appendix is a summary in alphabetic order of the commands that the user can select from the CRT menu that is displayed when the Animator is invoked at the COBOL command line level. A brief summary of each command is given.

A One of the two available lOcAte commands. The A command locates the declaration of a data-name or procedure-name in a program when the required name is keyed in.

B The Breakpoint command allows the user to set breakpoints at which execution will halt automatically. The Breakpoint command offers a menu of four options:

- S Set breakpoint at statement currently pointed to by the cursor.
- U Unset the breakpoint currently pointed to by the cursor.
- C Cancel all breakpoints.
- X Examine next breakpoint. Can be used successively to examine all set breakpoints.

C The Compile command enables immediate compilation and execution of entered COBOL statement or statements. Any compilation error that may result offers a S(top) or C(ontinue) selection menu.

D The Display command enables display and/or amendment of the named data-item.

E The Execute command is used to specify the way in which the user requires execution of the program. On entry, the command displays an option menu as follows:

- X EXecutes a single COBOL statement and moves the cursor to the next statement.
- K SKips a single COBOL statement without execution and moves the cursor to the next statement.
- I Executes (without the Animation Option features) to the next If statement, halts and positions the cursor there.
- G Initiates execution (GO) with the Animation Option. As each statement is executed (at a specified speed) the cursor is moved to the next statement in the source code. Speed of execution is set by typing a digit 1 through 9, where 1 is slowest and 9 is fastest.
- Z Initiates execution without Animation activated (Zooms).
- S Stops execution after displaying the current user screen.

F The Find command searches from the current cursor position through the source text for a specified string of characters.

L The Level command allows a "threshold" level to be set at any level of nested PERFORMs such that any PERFORM subordinate to this level is treated as a single statement for Animation purposes.

M The Monitor command enables automatic repeated display of the value of a single specified data-item (without amendment) during program execution.

N The Name command specifies which programs are to be executed with the Animation Option invoked. Displays a menu with the following options:

- W Which program: displays the current program name.
- A All programs: this is the default; programs compiled with the ANIM Compiler directive set are animated.
- T This program: only the current program will run under animation; all others will execute normally.
- O Other program: the cursor is moved to type a program name followed by RETURN; the current program will complete

execution under animation; thereafter the named program will execute under animation, but all other programs will execute normally.

O The second of the two available lOcAte commands. The O command locates the declaration of the data-name or procedure-name of any occurrence on which the cursor is positioned when O is entered.

P The program-counter command provides facilities to ascertain the point at which execution will start (or resume), or to alter this point. On entry, the P command displays an option menu as follows:

- W Where : repositions the screen window as necessary to include the statement at the hexadecimal address given.
- R Resets the execution start point to the current cursor position.

Q The Query command allows display and/or amendment of the data-item pointed at by the cursor when Q is entered.

S The Screen command repositions the screen window to display a different part of the source text as follows:

- N displays Next screen from source text.
- P displays Previous screen from source text.
- T displays screen at Top of source text
- E displays screen at End of source text.
- V repositions window so that the source line on which the cursor sits is the third line. Note: the cursor must be positioned before pressing S.

- H splits the screen in half (two windows) divided by a line of hyphens. The lower window shows the start of the program text. Note: Subsequent screen commands operate in the window in which the cursor is positioned.
- F restores Full screen display (single window).

- 〉 same as the N screen command except you need not type "S" first; "." can also be used.
- 〈 same as the P command except you need not type "S" first; "," can also be used.

=n repositions the window such that the nth source line is aligned at the third screen line.
+n moves the window forward n lines.
−n moves the window back n lines.

Note: =, +, − all position the cursor for entry of a numeric quantity followed by RETURN.

If there is a split screen display and the cursor is on the dividing line of hyphens, then when S is pressed the following subordinate menu is displayed:

SCREEN DIVIDER - U(p) D(own)

These commands allow the relative size of the two windows to be altered:

U moves the screen divider Up one line.
D moves the screen divider Down one line.

T The unTil command allows specification of a condition which will cause execution to halt as follows:

S Set the condition. Enter the required COBOL conditional expression.
U Unset the previously set condition.
X Display the previously set conditional expression.

U The User command displays the current user screen until any key is pressed.

Z The Zoom command specifies continuation of execution of the program without further invocation of the Animation Option.

Note that the following sub-menu commands may actually be used at the top menu level:

〉 . 〈 , = + − V X G K I W, 1-9 (speeds).

F

COBOL File Formats

General

The disk file system used in Apple /// COBOL is the disk-based SOS system described in the appropriate Apple /// manuals. A description of file creation and management is available in those manuals.

Apple /// COBOL offers SEQUENTIAL, RELATIVE and INDEX SEQUENTIAL organizations, in accordance with the ANSI COBOL standard. This COBOL also offers a variant of SEQUENTIAL organization, LINE SEQUENTIAL, designed for variable length line files, such as printers, the console, and files output by text editors.

All file processing information is defined within an interactive Apple /// COBOL program. File organization, access method, device assignment and allocation of disk space are defined by the SELECT statement in the INPUT-OUTPUT SECTION of the ENVIRONMENT DIVISION and an FD entry in the FILE SECTION of the DATA DIVISION.

Apple /// COBOL offers fixed (compile-time) file assignment and dynamic (run-time) file assignment facilities.

Fixed (Literal) File Assignment

The SOS file name is assigned to the internal user file name at compile time as shown in the specifications that follow.

In the FILE-CONTROL paragraph the general format of the SELECT and ASSIGN TO statements is as follows:

```
SELECT  file-name

        ASSIGN TO  external-file-name-literal
                   file-identifier

                 , external-file-name-literal
                   file-identifier              .
```

parameters:

file-name	is any user-defined name
file-identifier	is Run-Time File Assignment; see later in this Appendix
external-file-name-literal	is a standard SOS pathname.

SOS pathnames have the following general format:

```
drive/          [subdirectory/] ...          file-name

volume-id/
?/
 or
device
```

where

drive	is the name of a disk device driver, for example .D1 for the built-in drive
volume-id	is the logical name assigned to a disk when it was formatted, for example /DEMO

device	is the name of a device driver
subdirectory	is a file which itself references files at a lower level in the disk hierarchy
file-name	is the name of a file containing data

For the purpose of generating derivative names (required in compilation or when handling certain types of COBOL files), the name is considered as being made up of two parts, a base and an extension (which may be null). The extension is those characters which follow the last period (if any) embedded in the file name.

If a drive or disk directory is not specified for a disk file, the pathname specified is resolved using the prefix pathname that has been set, according to normal SOS operating system conventions.

An Apple *III* COBOL extension allows the use of the two-character string "?/" in place of the directory string for a disk file pathname. This is intended primarily for handling system files and causes the RTS to search all the disks configured into the system for a file of the specified name when an OPEN is executed. The open will fail if no file is found (even on an OPEN for output); if files having the same name exist on more than one disk in the system, the first one found will be opened: however, the order in which disks are searched is indeterminate.

Examples of Fixed (Literal) File Assignment:

```
SELECT STOCK-FILE
    ASSIGN TO ".D3/INVENTORY/WAREHS.BUY".

SELECT STOCK-FILE
    ASSIGN TO "/DATA/INVENTORY/WAREHS.BUY".
```

The file name specified as above is then used in the FILE DESCRIPTION for that program; see The FILE DESCRIPTION - Complete Entry Skeleton in Chapters Five, Six and Seven of the *Apple III COBOL Language Reference Manual.*

This file name is then also used in the OPEN and CLOSE statements when the file is required for use in the program; see THE OPEN STATEMENT and THE CLOSE STATEMENT in Chapters Five, Six, and Seven of the *Apple III COBOL Language Reference Manual.*

Run-Time File Assignment

The internal user file name is assigned to a file-identifier (an alphanumeric user-defined COBOL word) which automatically defines a PIC X(85) data area to hold the external SOS file name. The external SOS file name can then be stored in this data area in the Procedure Division by the user, and can be altered during the run as required.

All SOS pathnames used in programs should be terminated with a space to identify the end of the string. The Compiler handles this requirement automatically in the case of a literal ASSIGN TO clause or a MOVE statement. However, if you redeclare a file-id in WORKING-STORAGE, this space is not automatically generated and you may get a surprising result. For example, observe the following code:

```
ENVIRONMENT DIVISION.
FILE-CONTROL.
      SELECT FILE1 ASSIGN TO FILE-NAME.

DATA DIVISION.
FILE SECTION.
      FD FILE1.
WORKING-STORAGE SECTION.
01    FILE-NAME  PIC X(10)  VALUE  "ABCDEF.GHI".
01    REST-1     PIC X(3)   VALUE  "00 ".
* NOTE THE SPACE IN "00 " ABOVE!

PROCEDURE DIVISION.
MAIN-ROUTINE.
        OPEN OUTPUT FILE1.
        CLOSE FILE1.
        STOP RUN.
```

The result of this code is to create a file named ABCDEF.GHI00! There are three totally separate ways to get this program to create a file named ABCDEF.GHI, as would be expected.

1. Change the declaration of FILE-NAME to PIC X(11).
2. Eliminate the declaration of FILE-NAME in WORKING-STORAGE and instead, MOVE the appropriate value to it in the PROCEDURE DIVISION.

3. Eliminate the declaration of FILE-NAME in WORKING-STORAGE and instead, use the statement SELECT FILE1 ASSIGN TO "ABCDEF.GHI" in FILE-CONTROL.

In the FILE-CONTROL paragraph the general format of the SELECT and ASSIGN TO statements for run-time assignment is as follows:

```
SELECT  file-name
    ASSIGN TO  file-identifier
```

Parameters

file-name is any user-defined Apple /// COBOL word.

file-identifier is any user-defined Apple /// COBOL word.

Example of Run-Time File Assignment:

```
SELECT STOCK-FILE
    ASSIGN STOCK-NAME.
```

The external SOS file name of the required file is then stored as required in the file-identifier location specified above by the user program before an OPEN for the file is executed; for example

```
MOVE".D3/INVENTORY/WAREHS.BUY" TO STOCK-NAME.
OPEN INPUT STOCK-FILE.
.
.
.
CLOSE STOCK-FILE.
.
.
.
MOVE ".D3/INVENTORY/WAREHS.SELL" TO STOCK-NAME.
OPEN INPUT STOCK-FILE.
.
.
.
```

The SOS file name could have been entered via an ACCEPT statement by an operator, or stored as any other variable data just as well as being coded literally, as in this example.

Apple III COBOL Disk File Structures Under SOS

Apple /// COBOL offers four types of file organization for use by the COBOL programmer: SEQUENTIAL, LINE SEQUENTIAL, RELATIVE and INDEX SEQUENTIAL (ISAM). Any file is a set of records; a record is a set of contiguous data bytes which are mapped into hardware sectors with which they need not coincide, that is, a record can start anywhere within a sector and can cross hardware sector boundaries. System considerations for the different file organizations are discussed below.

SEQUENTIAL

SEQUENTIAL files are read and written using fixed length records, the length used being that of the longest record defined in the COBOL program's FD.

Normally the space occupied per record is the same as the program record length and data of any type may be held on the file. This does not apply if a WRITE is done using BEFORE or AFTER ADVANCING, as extra control characters are inserted and the data cannot then be read back correctly.

The limit on file size is that imposed by SOS, which is 16 Mbytes.

LINE SEQUENTIAL

LINE SEQUENTIAL file format handles ASCII and TEXT files like those generated by editors and other similar utilities. In particular, it is designed to be compatible with both types of files handled by the Apple /// Pascal editor. This is the only type of Apple /// COBOL file format in which variable length records are supported: the single byte X"OD" (carriage return) is used as a record delimiter. On input the CR is removed and the record area padded out with spaces as necessary: on output any trailing spaces in the program's record area are ignored.

Use of ADVANCING phrases other than BEFORE 1 causes the output of additional control characters. A file created in this way can still be read by a program, but the additional control characters are not filtered out and they will either appear in the record area or cause the appearance of

spurious blank records. Note also that the behavior of output devices with files in this organization is dependent on their treatment of the ASCII CR; this character may, or may not, cause an automatic line feed.

Note that this file organization is appropriate for reading files of type ASCII or TEXT and for writing them to character devices such as the console or printers. An output file of this organization will always have file type ASCII. Apple /// COBOL does not support the TEXT file type except to be able to read it.

RELATIVE

RELATIVE file organization provides a means of accessing data randomly by specifying its position in the file. Records are of fixed length, the length used being that of the longest record defined in the program's FD. To designate whether or not a record logically exists, two bytes are added to the end of each record: these contain 0D0AH if the record logically exists on the file and 0000H if it does not. The total length of a file is determined by the highest relative record number used; the maximum file size is 16 Mbytes. Data of any type may be held on the file; the RTS uses information held by SOS to determine the precise position of the end of data.

INDEX SEQUENTIAL

An INDEX SEQUENTIAL (ISAM) file occupies two files on disk: both are in standard relative file format, one containing the data and the other all indexing and free space information.

The name for the index file is derived from the name supplied for the data file by substituting the extension ".IDX" in place of any supplied in the data file name. This means that different indexed files cannot be distinguished purely by a change in the file name extension and that it is advisable to refrain from using the extension ".IDX" in other contexts. For example:

"CLOCK.DAT" produces an index "CLOCK.IDX"

The index (each index in a multiple key file) is built up as an inverted tree structure which grows in height as records are added: the number of index file accesses required to locate a randomly selected record

depends principally on the number of records on the file and the applicable "keylengths". The number of levels in a tree (and hence the number of disk accesses required) is approximately

index levels = logarithm (base k) of the number of records,

where k = 150 / (keylength + 2)

but will vary slightly depending on the order in which records are added and deleted.

Faster response times are generally obtainable when accessing a file sequentially, particularly when reading or when processing a single-key file.

The size (in bytes) of an ISAM file is approximately related to the maximum number of records it contains as follows:

data = (record length + 2) * max. no of records

index = sum over all keys of:

256 * (no. of records) / (k — 1)

where k is defined as above

Each key is restricted to a maximum of 32 bytes and a limit of 65535 records is required on both data and index files; index records are 256 bytes long. An attempt to exceed this limit will result in a boundary error (invalid key, file status "24") being reported.

The necessity of taking regular backup copies of all types of files cannot be emphasized too strongly; this should always be regarded as the main safeguard. However, there are some situations with indexed files (such as media corruption) that can lead to only one of the two files becoming unusable. If the index file is lost in this way, it is normally possible to recover data records from just one data file (although not in key sequence) and cut down on the time lost due to a failure. As an aid to this, all unused data records are deleted at the relative file level and so the operation may be done with a simple COBOL program by defining the data file as ORGANIZATION RELATIVE ACCESS SEQUENTIAL; READ the records from this file and WRITE them to a new version of the indexed file.

Sort-Merge Files

Sort and Merge operations are done using work files of formats similar to those of indexed sequential files. Their locations and names are determined from the file name supplied in the SELECT statement, one file being given the name supplied and the names of up to two further files being derived by substituting the extensions ".SX1" and ".SX2" for any supplied. The space required (in bytes) to accommodate the main workfile is

(record length + 2) * number of data records

and to accommodate the auxiliary workfiles

up to about 80 * number of data records each,
depending on the total length of all the key fields.

There is a limit of 65535 on the number of records that can be handled by a sort or merge (assuming that disk space is available). There is no restriction on key types or lengths, although the shorter and simpler these are kept, the faster the operation will proceed.

G

Conversion from Other COBOL Systems

Apple III COBOL Limits

There are certain implementation limits inherent in any COBOL system. Some of these will rarely, if ever, be encountered by practical programs; for example, in Apple /// COBOL, PERFORM statements may not be nested to more than 50 levels. Others have more impact on applications. Users of Apple /// COBOL should be aware of the following limits:

- No more than 16 files may be open at any one time. Note that INDEX SEQUENTIAL files require an index file in addition to the data file, so each INDEX SEQUENTIAL file counts for two files in this limitation.
- An unsegmented program is limited to 64K bytes of memory, of which no more than 60K bytes may be PROCEDURE DIVISION code. For a segmented program, the root segment (code and data) and the largest independent segment, plus the .ISR file, must meet this 64K limit. An application exceeding this limit will need to be rewritten with inter-program CALL statements.
- The Compiler (and the Animator) use dictionary files which are also limited to 64K bytes in size; there will be one such file for each independent segment of a program. If this limit is exceeded, the Compiler will fail. Again the solution is to rewrite the application as several CALLed programs. Also note that for an Apple /// with less than 256K memory, it is the size of the dictionary files that limits the use of Animator. With a 128K

system, you may also notice dictionary overflow by Compiler error 06: "too many data and procedure names declared, or not enough memory".

Apple *III* COBOL follows the ANSI standard (X3.23 – 1974) quite closely; the *Apple III COBOL Language Reference Manual* has the same organization as the standard, to assist you in comparing Apple COBOL and other systems.

Standard COBOL attempts to limit system dependencies to highly visible entries in the ENVIRONMENT DIVISION; any COBOL conversion should look to this DIVISION first for necessary changes. You should read carefully through the ENVIRONMENT DIVISION references in the *Apple III COBOL Language Reference Manual.* Appendix F of this manual is also useful; it deals with the interface between COBOL files and the SOS Operating System on the Apple *III.*

Although certified at the High-Intermediate Level of Compiler acceptance by the General Services Administration, Apple *III* COBOL in fact implements many of the features of full COBOL. In particular, the full COBOL Nucleus module is implemented; thus all aspects of the data-manipulation and arithmetic verbs (MOVE, INSPECT, ADD, COMPUTE, etc.) are available in Apple *III* COBOL. You will find a detailed listing of what is implemented and what is not in Appendix I of the *Apple III COBOL Language Reference Manual;* the main omissions are

- Report Writer. This module is the only module of the COBOL standard not handled in any way by this implementation.
- Communication. This module, requiring a Message Communication System, is not implemented. Communications statements are checked and compiled, but they are not handled by the Run-Time System. Attempting to run a program with such statements will give RTS error 161—ILLEGAL INTERMEDIATE CODE.
- Tape Handling. Clauses dealing with blocking of tape records, tape labels, and other aspects of magnetic tape storage are accepted by the Compiler, but treated as comments. Similarly, the RERUN clause is accepted as documentary only. All such COBOL statements are marked by shading and an affixed "D" (for Documentary) in the COBOL syntax summary, which is Appendix F of the *Apple III COBOL Language Reference Manual.*

- The highest level of the Segmentation module, which allows overlays in the fixed segment of a segmented program, isn't implemented.
- The highest level of the Library module, allowing the REPLACING phrase in a COPY statement, isn't implemented.

Most surprises in the use of Apple /// COBOL have to do, not with partial implementation of the standard, but with extensions to it, either in the Apple system, or in the COBOL you are used to; for example:

- The Apple /// file organization LINE SEQUENTIAL will, unless you specify a WRITE AFTER ADVANCING or WRITE BEFORE ADVANCING phrase, assume a BEFORE ADVANCING 1 default. That is, a line normally ends with a carriage return; for other behavior, you must direct the Compiler to insert the necessary control characters. A SEQUENTIAL file would assume (in accordance with the standard) the phrase AFTER ADVANCING 1.
- Most COBOL Compilers have a directive which will enable you to check a code file for conformity to the ANSI standard. In the case of Apple /// COBOL, this is the directive

FLAG "A///"

- which reports on any features used in the source which are Apple /// COBOL extensions to the ANSI standard. Always make a listing of your source files with this switch set; then when you need to move the program to another system, you are less likely to have surprises.

Things to look out for:

1. Operating system features. Many file handling options in ENVIRONMENT DIVISION code are system dependent.
2. Non-standard verbs. Sometimes the syntax of COBOL statements is non-standard: SORT and MERGE frequently have operating system "features" attached to them; some systems allow a non-standard THEN in IF-ELSE sentences (IBM COBOL does this, so does Apple /// COBOL). Sometimes the verb will be totally different from anything in the standard; for example, the verb TRANSFORM in IBM OS/VS COBOL.

3. Special Compiler or listing features. Portable programs should not depend on "optimizations" or special Compiler treatment to run correctly. Check for MOVE statements and arithmetic that depends on special features of internal representation. Compiler directives can also throw you off; for example, many IBM Compilers have an EJECT directive that forces a listing to the next page—standard COBOL uses a slash ("/") character in the Indicator Area (column 7) to achieve the same effect.

4. Look carefully through your old COBOL manuals; study the "gray areas" in particular, as these will give you warning of code that is not likely to be easy to convert. In some cases, only the heading on a page or a section will be shaded when in fact the whole section is non-standard.

5. Programs containing logic depending on the EBCDIC collating sequence will require modification. For example, a flag which could contain either a numerical or an alphabetic result would perform differently in the ASCII collating sequence.

6. Apple /// COBOL limits exponents to 9999. Although it is unlikely that you will have a program using an exponent larger than this, if you do, it will have to be changed.

7. Note the previously mentioned Apple /// COBOL requirement for a trailing space in all file names. This is explained in detail in the section on Run-Time File Assignment in Appendix F and may affect conversions from other systems.

8. If you are using DECLARATIVES to handle I-O errors, remember that the second byte of FILE-STATUS needs to be decoded before you can ascertain the actual cause of the error. Please refer back to the section on File Status in Chapter 6 for a more complete explanation of this point.

9. In many cases where you might use SEQUENTIAL file organization on another system, Apple /// COBOL will more appropriately use LINE SEQUENTIAL. As explained previously, this is true of any file that you intend to transfer to a printer or CRT.

Please note once again that most of the above instances result from programming styles that people have become accustomed to on other systems, which are in fact non-ANSI standard. Apple /// COBOL *is* ANSI standard.

H

Transferring Files Using ACCESS ///

Existing COBOL programs and data files can be transferred from a host system to the Apple /// using the ACCESS /// utility. This is a separate product, not part of the Apple /// COBOL system, and it is described in detail in its own manual, *Apple Access ///*. The discussion here is intended to point out the major operational considerations in the use of ACCESS /// and some potential difficulties in the use of transferred files.

You use ACCESS /// by connecting your Apple ///, either through a modem link or by direct cable, to the system containing the files you want to transfer. If possible, that system should be set up to handle XON/XOFF protocol; that is, it should stop sending characters to a terminal when it receives an XOFF (CONTROL-S), and resume when it receives an XON (CONTROL-Q) from the Apple ///. If the system doesn't respond to this protocol, you can still use ACCESS ///, but you will not be able to transfer files at more than 1200 baud to a ProFile or other large disk, or at more than 300 baud to a floppy—greater speeds may overflow the receive buffers.

ACCESS /// allows you to use the Apple just like a terminal on the other system and in addition to record all or part of the transmission from the host system. In operation, ACCESS /// distinguishes its commands from characters sent to the host by using the OPEN-APPLE key just to the right of the ALPHA LOCK; this key sets the eighth data bit from whatever key is typed simultaneously. One of these commands, OPEN-APPLE and R, toggles the recording of transmission from the host computer. You type this, then give a command to the host to list the file you want on your terminal; as you watch it on your display, ACCESS /// records it for later

use. When the transfer is done, type OPEN-APPLE and R again to stop recording. At that stage, you can change recording files and transfer another file from the host or exit from ACCESS *///* to look at the result.
use. When the transfer is done, type OPEN-APPLE and R again to stop recording. At that stage, you can change recording files and transfer another file from the host or exit from ACCESS *///* to look at the result.

In most cases, you will have recorded some initial dialogue with the host system as well as the file you want. Possibly there will be some terminating dialogue with the host as well. You should examine the file with an editor to remove these accretions. You may also discover some extra characters throughout the file. Each line of your file probably ends with both carriage return and line feed characters; the carriage return by itself is the usual line terminator for Apple *///* ASCII files. You may use the recording filter option in ACCESS *///* to screen out superfluous control characters. Use a global editor replace command to get rid of any unnecessary additional controls.

Note that you can transfer non-character data files in this same way; ACCESS *///* will intercept unprintable bytes from the host system and not display them on the screen, but it will write them out to the recording file, unless the filter is on. However, you may have more difficulty in such a case editing out extraneous bytes like the host dialogue.

Figures and Tables

Index

G

H

I

J

K

T

U

Apple III COBOL: *Introduction and Operating System Manual*

apple computer

20525 Mariani Avenue
Cupertino, California 95014
(408) 996-1010
TLX 171-576
030-0435-A

Photographs courtesy of Vorpal Gallery, San Francisco, New York, Laguna Beach, CA.