SOFTWARE MANUAL

AlphaBASIC

XCALL SUBROUTINE USER'S MANUAL

DSS-10008-00

REV. A00

alpha micro

ALPHA MICROSYSTEMS
17881 Sky Park North
Irvine, CA. 92714

SOFTWARE MANUAL

AlphaBASIC

XCALL SUBROUTINE USER'S MANUAL

DSS-10008-00

REV. A00

alpha micro

FIRST EDITION

June 1982

REVISIONS INCORPORATED	
REVISION	DATE
A00	June 1982

This document reflects AMOS versions 4.6 and later
and AMOS/L versions 1.0 and later

ALPHA MICROSYSTEMS
17881 Sky Park North
Irvine, CA. 92714

Table of Contents

CHAPTER 1

INTRODUCTION

AlphaBASIC, the Alpha Micro BASIC Language Processor, is a powerfully enhanced version of BASIC. AlphaBASIC has the ability to access external machine language subroutines using a keyword called XCALL. Several machine language subroutines, ones that perform complex and yet frequently required tasks, are provided on your System Disk. These external subroutines, their features, abilities and restrictions, are the subject of this manual.

Because these external subroutines are machine language programs, they are much smaller and faster than equivalent AlphaBASIC programs. Machine language programs work closely with hardware and the operating system, which AlphaBASIC cannot do in some applications.

It is important to note here that, whereas you can write your own machine language subroutines and access them via XCALL, this manual does not discuss how those machine language subroutines can be written. This manual instead restricts iself to a discussion of the existing external subroutines named BASORT, COMMON, FLOCK, XLOCK, SPOOL and XMOUNT. You will find this manual useful if you are already somewhat familiar with AlphaBASIC and wish to understand, and then access, these external subroutines. You may also find this manual to be useful later as a reference guide to the various existing subroutines.

Please refer to the AlphaBASIC User's Manual, DWM-00100-01 for further information about the XCALL keyword and any other topic dealing with AlphaBASIC itself.

1.1 MANUAL ORGANIZATION

This manual is arranged in chapters. You are reading the introductory chapter, Chapter 1. Chapters 2 through 7 discuss the XCALL subroutines themselves; how, when, where and why to use them, and what special features they provide.

Chapter 2 talks about BASORT, the AlphaBASIC Sort subroutine. BASORT sorts the kinds of files called Random files and Sequential files. There is also a list of the error messages the BASORT subroutine may return.

Chapter 3 discusses COMMON, the external subroutine that enables data to be transferred into a common storage area of memory (for example, to pass variables between chained programs).

Chapter 4 details FLOCK, the File Locking subroutine that protects a data file from being accessed by more than one program in a given moment, so that the file won't be updated by two or more program users concurrently.

Chapter 5 discusses XLOCK, the subroutine used to set, test and clear "locks" on files and devices. This subroutine is similar in some respects to FLOCK, discussed in Chapter 4.

Chapter 6 talks about SPOOL (an acronym for "Simultaneous Printer Output On-Line"), the subroutine that inserts, or "spools," a file into a printer queue for immediate or eventual processing outside of the control of the job running the AlphaBASIC program.

Chapter 7 discusses XMOUNT, the subroutine used to mount a disk from within a AlphaBASIC program, as when a user must access a new disk during the course of a multi-disk file update event. You mount a disk after you have changed a hard disk cartridge or a floppy diskette, in order to inform the system that the disk in that drive has a different "bitmap," or index of free and used storage areas.

1.2 SAMPLE PROGRAMS

There are a number of sample programs in this manual, ranging in complexity from one to several dozen program lines. Remember that these samples are meant only to demonstrate the use of the AlphaBASIC XCALL subroutines, and are not intended as examples of the best or most elegant techniques to follow when creating AlphaBASIC programs.

To quickly grasp the point of these examples, remember that AlphaBASIC permits the use of labels, as well as line numbers, to identify locations in a program. A program label is composed of one or more alphanumeric characters which are not separated by a space or other delimiter. The first character is always an upper case or lower case letter. A label must be the first item on a line after the line number and must be terminated by a colon (:). The following is an example of labels (RANDOM'DIRECTION, UP, DOWN and STRAIGHT) in an AlphaBASIC program that performs a kind of simple animation.

```
10  MAP 1  DIRECTION,F

100 RANDOM'DIRECTION:
110   DIRECTION=INT(3*RND(0)+1)
120     ON DIRECTION GOSUB UP, DOWN, STRAIGHT
130   GOTO RANDOM'DIRECTION

1000 UP: PRINT "/"; TAB(-1,3); : RETURN

2000 DOWN: PRINT TAB(-1,4);"\"; : RETURN

3000 STRAIGHT: PRINT""; : RETURN
```

In the pages of this manual you will be seeing a number of program examples that use labels.

Notice that line 10 of the above program example is a level-1 MAP statement; we map the variable DIRECTION as a floating point variable (F). AlphaBASIC provides you with the ability to specify the pattern in which variables of all kinds (floating point, string, and binary) are allocated in memory. By mapping variables at different levels you may define whole groups of related information and reference single elements or an entire group as you choose. You will see MAP statements in many of the examples within this manual. For further information on interpreting and using MAP statements, see Chapter 8, "Memory Mapping System," of the AlphaBASIC User's Manual, DWM-00100-01.

1.3 USING XCALL SUBROUTINES

There are several things you should keep in mind before beginning to use XCALL subroutines:

1. All XCALL subroutines must have a .SBR extension. The subroutines supplied with your system software reside in account [7,6] of the System Disk.

   Whenever a subroutine is requested, AlphaBASIC follows a specific pattern in looking for the requested subroutine. The search sequence is as follows (where [P,pn] designates the Project-programmer number that specifies your account):

   a. System memory

   b. User memory

   c. Default disk:[User P,pn]

   d. Default disk:[User,0]

   e. DSK0:[7,6]

   Notice that AlphaBASIC checks first system, then user memory. If a subroutine is to be called a large number of times, it is wise to load it into memory to avoid the overhead of fetching the subroutine from disk.

   If the subroutine is not in memory, AlphaBASIC attempts to load the subroutine from the disk, following steps c. through e. of the search sequence above. If an AlphaBASIC program fetches a subroutine from disk, AlphaBASIC loads it into memory only for the duration of its execution; afterward it is removed from memory if it is loaded via this automatic procedure. NOTE: Subroutines loaded into system or user memory via the LOAD command remain in memory until you reset the system or until you use the monitor command DEL to delete them.

2. You will invoke a particular subroutine via the AlphaBASIC XCALL statement, and will usually need to specify several control parameters on that statement line. A typical XCALL statement line might look like this (where COMMON is the name of the subroutine you want to invoke, and SEND, "MSGNAM", and WRITE'OUT are variables that specify information to the COMMON subroutine):

```
100 XCALL COMMON,SEND,"MSGNAM",WRITE'OUT
```

3. You will need to use MAP statements to define many of the control variables you specify on the XCALL statement line. (This is because only by way of MAP statements can you define binary variables.) For information on MAP statements, refer to the AlphaBASIC User's Manual, DWM-00100-01.

4. Many of the XCALL subroutines require that you pre-load special files. For example, you must load the file DSKO:COMMON.SBR[7,6] into user or system memory before running an AlphaBASIC program that makes use of the COMMON subroutine. (For each XCALL subroutine, the documentation that follows will let you know what files need to be pre-loaded.)

   To load a file into user memory (i.e., your own memory partition), enter either of the following from AMOS or AMOS/L command level:

   ```
   .LOAD DSKO:Filename.SBR[7,6] RET
   .
   ```

   or

   ```
   .LOAD BAS:Filename.SBR RET
   .
   ```

   where Filename is the name of the subroutine you are requesting (e.g., COMMON, BASORT, etc.).

   Note the use of the ersatz name, BAS:, which indicates account [7,6] of the System Disk. After you see the monitor prompt, ., you may run an AlphaBASIC program that uses the specific subroutine.

   To load an XCALL subroutine into system memory, the System Operator must use the SYSTEM command within the system initialization command file. For more information on loading files, including subroutines, into system memory during system boot-up, see the AMOS System Operator's Guide, DSS-10001-00, or the AMOS/L System Operator's Guide, DSS-10002-00.

5. Some XCALL subroutines (namely, FLOCK, XLOCK and SPOOL) use the monitor queue. The monitor queue is a list of blocks in system memory which are linked to each other in a forward chain. Each queue block is currently eight words (16 bytes) in size (this value may change with the next release of the file system). During normal monitor operations, various functions use these queue blocks

to perform certain tasks. The monitor initially contains 20 blocks in the available queue list. This quantity is established in the system initialization command file. For information on increasing the number of available monitor queue blocks, see the AMOS System Operator's Guide, DSS-10001-00, or the AMOS/L System Operator's Guide, DSS-10002-00.

If you use an XCALL subroutine that uses the monitor queue, you must be sure that enough queue blocks are available before executing the subroutine. If not enough blocks are available when the AlphaBASIC program executes the XCALL subroutine, the system could lock up and require manual reset.

Your AlphaBASIC program can check the number of free queue blocks before you perform the XCALL subroutine by using the WORD function to read the QFREE memory location. The program should not continue if the quantity of free queue blocks is insufficient.

To find the QFREE memory location for an AMOS system, check the current SYS.MAC file. For AMOS/L systems, see the SYS.M68 file to see the location of QFREE.

The queue block requirements for each of the XCALL subroutines is discussed in the appropriate chapter.

CHAPTER 2

BASORT - XCALL SUBROUTINE FOR SORTING FILES

BASORT is an external subroutine, callable from AlphaBASIC via the XCALL keyword, which can sort both random and sequential files. A random file is one in which the records are physically grouped together in one area of the disk, and where any point within that file can thus be found immediately by calculating an offset from the file's beginning. A sequential file's records are not necessarily contiguous on the disk, but are linked in sequence by pointers in each segment that indicate where on the disk the next segment can be found. For information on creating and using files from within AlphaBASIC, refer to Chapter 15 of the AlphaBASIC User's Manual, DWM-00100-01.

You can use BASORT to sort a file into numeric order, a list of names or words into alphabetic order, and so on. BASORT permits up to three keys, or elements of the data records you wish to base your sort on. For example, say you have a list of customer names, each with an associated order date code and a purchase order number. The first key might be the customer name. If a particular customer has ordered more than once, the second key comes into play to determine which record of that customer's should go first. You can sort that customer's orders chronologically based on the date code. And if that customer has placed two or more orders in the same day, the third key will determine the final sorting placement of that customer's records based on his purchase order numbers. (An example of this kind of sort is in Section 2.2.1.1 below.)

BASORT combines two sorting methods to make it a relatively fast sort utility that can still handle very large files. If your memory partition is large enough to contain the entire file that is to be sorted, BASORT performs a memory-based heap sort. That means it sifts through and rearranges the "heap" of data in memory to bring the data into the order you specify in the BASORT command line. If there is not enough room in user memory for the entire file, BASORT does a disk-based polyphase merge-sort. That is, the data is brought into memory in small groups where it is sorted and rewritten to the disk; then the several groups are merged together on the disk.

2.1 LOADING BASORT INTO MEMORY

The BASORT package consists of three modules (or two modules on AMOS/L systems) --BASORT.SBR, AMSORT.SYS, and FLTCNV.PRG (FLTCNV is omitted on AMOS/L systems). These modules must be in memory when BASORT is used. When the XCALL BASORT command is used in an AlphaBASIC program, the AlphaBASIC program automatically loads BASORT.SBR into user memory. However, AMSORT.SYS and FLTCNV.PRG (for AMOS systems) must be loaded into either system or user memory prior to running an AlphaBASIC program using BASORT.

To load AMSORT.SYS (and FLTCNV.PRG for AMOS systems) into user memory, enter the following from AMOS or AMOS/L command level:

```
.LOAD DSK0:AMSORT.SYS[1,4] (RET)   or     .LOAD DSK0:AMSORT.SYS[1,4] (RET)
.LOAD DSK0:FLTCNV.PRG[1,4] (RET)          .
.
```

To load AMSORT.SYS and FLTCNV.PRG into system memory, you must have two lines in your system initialization command file that perform those functions. For more information on loading subroutines into system memory during system boot-up, see the <u>AMOS System Operator's Guide</u>, DSS-10001-00, or the <u>AMOS/L System Operator's Guide</u>, DSS-10002-00.

AMSORT.SYS and FLTCNV.PRG are re-entrant; BASORT.SBR is not, so you must not load it into system memory.

2.2 USING BASORT IN AN ALPHABASIC PROGRAM

You may use BASORT to sort both random and sequential files. Like all the other external subroutines discussed in this manual, you will call BASORT from the AlphaBASIC program using the XCALL keyword. Then you will supply the parameters of up to three keys you wish to sort on are provided to the AlphaBASIC program via the XCALL BASORT command line.

Using BASORT for random files requires some different parameters than does using BASORT for sequential files. The next two sections describe the specific methods of using BASORT for both random and sequential files.

2.2.1 Sorting Random Files

When you use BASORT to sort random files, BASORT sorts the file onto itself (that is, it replaces the original, unsorted file with a file containing the sorted data). Therefore, if you wish to retain a backup copy of the unsorted file, you must create a separate copy to be sorted.

BASORT for random files is called via variables or constants in this order (where the ampersand (&) means a continuation of the AlphaBASIC line statement):

```
XCALL BASORT, CHANNEL'NUMBER, RECORD'COUNT, RECORD'SIZE, &
              KEY1'SIZE, KEY1'POSITION, KEY1'ORDER, &
              KEY2'SIZE, KEY2'POSITION, KEY2'ORDER, &
              KEY3'SIZE, KEY3'POSITION, KEY3'ORDER, &
              KEY1'TYPE, KEY2'TYPE, KEY3'TYPE
```

Where:

CHANNEL'NUMBER - File channel on which file to be sorted is open for random processing.

RECORD'COUNT - Number of records in the random file you are sorting. (Unlike sequential files, the programmer must know the precise number of records in a random file.)

RECORD'SIZE - Size of the longest record in the file you are sorting. The size of a record is its byte count (including characters, spaces, etc.). Again, for a random file, you must be sure of the record size.

KEY1'SIZE - The size, in bytes, of sort key #1. Give the size of the largest instance of key #1 (i.e., if sort key #1 is the customer's name, find the longest name in any record, or perhaps allow for a very long one.)

KEY1'POSITION - The first character position occupied by key #1. If the KEY1'POSITION variable given is 50, for example, BASORT will fit the characters beginning at the fiftieth byte in the record into the sequence it is creating.

KEY1'ORDER - Sort order of key #1. Enter the digit 0 to indicate that you want key #1 of each record to be sorted in ascending sequence, or enter the digit 1 to indicate descending seqeuence. (NOTE: The order is determined using ASCII collating sequence; e.g., all upper-case letters come before lower-case letters.)

KEY2'SIZE - The size, in bytes, of sort key #2.

KEY2'POSITION - The first character position occupied by key #2.

KEY2'ORDER - Sort order of key #2. Enter a 0 or a 1. (See KEY1'ORDER, above.)

KEY3'SIZE - The size, in bytes, of sort key #3.

KEY3'POSITION - The first character position occupied by key #3.

KEY3'ORDER - Sort order of key #3. Enter a 0 or a 1. (See KEY1'ORDER, above.)

KEY1'TYPE - The data type of key #1. Key types are:

0 = String
1 = Floating Point
2 = Binary

KEY2'TYPE - The data type of key #2. (See KEY1'TYPE, above.)

KEY3'TYPE - The data type of key #3. (See KEY1'TYPE, above.)

Remember, keys are the elements of the data records you wish to base your sort on (i.e., customer name, order number, etc.). If you want to use less than three keys, all entries in the XCALL command line for the unused keys must be zero. If the key types are omitted, BASORT assumes string data type.

All arguments in the XCALL command line are numeric, but may be passed as either floating point or string values. For example, "99" is a valid entry. Arguments must not be in binary format.

The first character in a record is considered position 1.

2.2.1.1 An Example of using BASORT on a Random File

The following is the contents of an unsorted file that we'll pretend we want sorted. The file we have gathered the following customer names in is called POINFO.DAT, containing the purchase order information of the specific printed business form (we're pretending) they ordered from us.

```
ROBIN GOOD PUBLICATIONS                    1/3/81     49130
K.A.L. ENTERPRISES                         12/7/81    1207
EVANS' CLASSIC AUTOMOBILES, Inc.           1/20/81    K79876
ROBIN GOOD PUBLICATIONS                    2/14/81    49201
DE SOTO HORSE GROOMING EQUIPMENT Co.       4/7/81     1836
VIDCOM                                     8/3/81     14101
ROBIN GOOD PUBLICATIONS                    2/28/81    49393
MARTIN MICHAEL LAVELLE, CONSULTANT         6/12/81    7S729
HONEST DAVE'S CHEAP CAR PARTS              9/11/81    A00326
DE SOTO HORSE GROOMING EQUIPMENT Co.       4/9/81     1895
ROBIN GOOD PUBLICATIONS                    2/28/81    49397
EVANS' CLASSIC AUTOMOBILES, Inc.           9/11/81    L98467
```

The program that we will use to sort the above file looks like this:

```
5  ! SAMPLE PROGRAM TO SORT SMALL RANDOM DATA FILE
10 MAP1 CUSTOMER'INFO              ! DEFINITION OF RECORD:
15  MAP2 NAME,S,35                 !   35 BYTES MAXIMUM
20  MAP2 PURCHASE'DATE,S,8         !    8 BYTES MAXIMUM
25  MAP2 PURCHASE'ORDER,S,7        !    7 BYTES MAXIMUM
30 MAP1 RECORD'SIZE,F,6,50         ! RECORD IS TOTAL OF 50 BYTES
35 MAP1 RECORD'NUMBER,F,6,0        ! START WITH RECORD #0
40 MAP1 CHANNEL,F,6,100            ! FILE IS OPEN ON CHANNEL #100
45 MAP1 RECORD'TOTAL,F,6,12        ! TOTAL OF 12 RECORDS IN FILE
50 MAP1 ASCENDING,F,6,0            ! SORT IN ASCENDING ORDER
55 MAP1 STRING,F,6,0               ! ALL KEYS ARE OF TYPE "STRING"

100 START:
120   OPEN #100,"POINFO.DAT",RANDOM,RECORD'SIZE,RECORD'NUMBER
130   PRINT "Now sorting..."
140   XCALL BASORT,CHANNEL,RECORD'TOTAL,RECORD'SIZE,35,1,ASCENDING,8,36,&
         ASCENDING,7,44,ASCENDING,STRING,STRING,STRING
150   PRINT "We will sort on name, purchase date, and purchase order number"
160   FOR RECORD'NUMBER = 0 TO 11
170     READ #100,CUSTOMER'INFO
180     PRINT NAME,
190     PRINT PURCHASE'DATE,
200     PRINT PURCHASE'ORDER
210  NEXT
220   CLOSE #100
230 END
```

Note that line 120 opens the file, POINFO.DAT. Line 140 is the XCALL BASORT command line, where the variables (defined in the MAP statements of lines 15 through 55) define the BASORT parameters. The file is sorted back on itself at that point. Then it is printed as a result of lines 160 through 210. Line 220 closes the file.

The resulting printout, when running the above program, is:

```
Now sorting...
We will sort on name, purchase date, and purchase order number
DE SOTO HORSE GROOMING EQUIPMENT CO       4/7/81        1836
DE SOTO HORSE GROOMING EQUIPMENT CO       4/9/81        1895
EVANS' CLASSIC AUTOMOBILES, INC.          1/20/81       K79876
EVANS' CLASSIC AUTOMOBILES, INC.          9/11/81       L98467
HONEST DAVE'S CHEAP CAR PARTS             9/11/81       A00326
K.A.L. ENTERPRISES                        12/7/81       1207
MARTIN MICHAEL LAVELLE, CONSULTANT        6/12/81       78729
ROBIN GOOD PUBLICATIONS                   1/3/81        49130
ROBIN GOOD PUBLICATIONS                   2/14/81       49201
ROBIN GOOD PUBLICATIONS                   2/28/81       49393
ROBIN GOOD PUBLICATIONS                   2/28/81       49397
VIDCOM                                    8/3/81        14101
```

2.2.2 Sorting Sequential Files

When you sort a sequential file, you must specify both an input and an output file. If you wish to sort a file back onto itself, you may specify the same file for both input and output.

IMPORTANT NOTE: Before BASORT is called, the file must be opened for input. BASORT leaves the file open for output.

Call BASORT for sequential files via:

```
XCALL BASORT, INPUT'CHANNEL, OUTPUT'CHANNEL, RECORD'SIZE,
              KEY1'SIZE, KEY1'POSITION, KEY1'ORDER,
              KEY2'SIZE, KEY2'POSITION, KEY2'ORDER,
              KEY3'SIZE, KEY3'POSITION, KEY3'ORDER
```

Where:

INPUT'CHANNEL - The file channel on which the input file is open.

OUTPUT'CHANNEL - The file channel on which the output file is open.

RECORD'SIZE - The size, in bytes, of the largest record in the file, including the terminating carriage return/linefeed characters. NOTE: Too small a value results in truncation of data records.

KEY1'SIZE - The size, in bytes, of sort key #1. Give the size of the largest instance of key #1 (i.e., if sort key #1 is the customer's name, find the longest name in any record, or perhaps allow for a very long one).

KEY1'POSITION - The first character position occupied by key #1. If the KEY1'POSITION variable given is 50, for example, BASORT will fit the characters beginning at the fiftieth byte in the record into the sequence it is creating.

KEY1'ORDER - Sort order of key #1. Enter the digit 0 to indicate that you want key #1 of each record to be sorted in ascending sequence, or enter the digit 1 to indicate descending seqeuence. (NOTE: The order is determined using ASCII collating sequence; e.g., all upper-case letters come before lower-case letters.)

KEY2'SIZE - The size, in bytes, of sort key #2.

KEY2'POSITION - The first character position occupied by key #2.

KEY2'ORDER - Sort order of key #2. Enter a 0 or a 1. (See KEY1'ORDER, above.)

KEY3'SIZE - The size, in bytes, of sort key #3.

KEY3'POSITION - The first character position occupied by key #3.

KEY3'ORDER - Sort order of key #3. Enter a 0 or a 1. (See KEY1'ORDER, above.)

NOTE: Sequential files contain only ASCII data. For that reason, when you sort sequential files you do not have to specify the data type of the sort keys; BASORT knows that all keys in a sequential file are strings.

2.2.2.1 An Example of Using BASORT on a Sequential File

The following is the contents of an unsorted sequential file that we want to sort. Pretend this time that we are cartographers, making a map of a new suburb just being built. We want to compile an alphabetic index of all the street names laid out and defined so far, but then we want to compile an alphabetic list of the streets to the north of town center only, then one of the streets to the east, and so on for the streets to the south and west.

We use a sequential file for this data because as new streets are laid out and named, we can later add those to our sequential file and then resort the file for future maps.

We have gathered the existing street names and their relative positions from city plans. The file we have put the unsorted list of all the streets in is called STREET.DAT. The extension, .DAT, indicates to us that this is the raw data file.

We want to record the sorted, alphabetic list of all the streets in a file called STREET.LST. The street names sorted according to direction we'll place in a file called ENSW.LST.

We choose an extension of .LST to remind us that these are files we can print when we want to.

Here is the list of street names and directions we've gathered from city plans:

```
Sinbad St.                        N
John Silver Rd.                   W
Marco Polo Ave.                   E
Robinson Crusoe Dr.               S
Nimrod Cr.                        S
William Tell Ln.                  W
Achilles Dr.                      W
Pontiac Ln.                       N
Fremont St.                       E
Kublai Khan Cr.                   S
Constantine Rd.                   W
Sancho Panza Cr.                  E
Balboa Dr.                        N
John Carter Ln.                   N
Homer Ave.                        E
William Taft Ave.                 S
Edward Teach St.                  E
Cisco Kid Rd.                     S
Michael Fink Dr.                  N
Herman Melville Ln.               W
```

The first thing we need to do is load AMSORT.SYS (and FLTCNV.PRG for an AMOS system) in user memory. We do that at AMOS or AMOS/L command level, this way:

```
.LOAD DSK0:AMSORT.SYS[1,4] RET   or     .LOAD DSK0:AMSORT.SYS[1,4] RET
.LOAD DSK0:FLTCNV.PRG[1,4] RET          .
.
```

Now we create the AlphaBASIC program. The first thing we have to remember to do is open the file channel for the file that we want to sort, and two more file channels and files where we want to put the sorted data into. (We could name the same file in both lines 110 and 120 or 110 and 130 below to write one of the sorted files right over the original, unsorted data.) Our program might look like this:

```
10  ! SAMPLE PROGRAM TO SORT SMALL SEQUENTIAL DATA FILE

100 START:
110     OPEN #1,"STREET.DAT",INPUT
115     OPEN #2,"STREET.LST",OUTPUT
120     OPEN #3,"ENSW.LST",OUTPUT
125     PRINT "Now sorting all streets alphabetically."
130        XCALL BASORT,1,2,50,25,1,0,1,33,0,0,0,0
135     CLOSE #1
140     PRINT "Now sorting according to direction from town center."
145     OPEN #1,"STREET.DAT",INPUT
150     XCALL BASORT,1,3,50,1,33,0,25,1,0,0,0,0
155     PRINT "All done.  See STREET.LST and ENSW.LST for sorted files."
200     CLOSE #1
210     CLOSE #2
220     CLOSE #3
230 END
```

Line 110 opens file channel #1 and the file called STREET.DAT for input. Line 115 opens file channel #2 and the file called STREET.LST for output. Line 120 opens file channel #3 for output also.

Line 130 performs the first XCALL BASORT subroutine. Immediately following the word BASORT and the delimiting comma, we indicate the file channel open for input. Then we indicate the output file channel, 2, where we want the file sorted the first way.

Note that in the unsorted file above, a record (the data of a single street name) is confined to one line. That makes it easy to judge the approximate size of the longest record. So, being liberal, we round it up to a record-size of 50.

The size of key #1 is never more than 25 bytes in size, so next on the XCALL BASORT line we enter a 25. The position of Key #1 is the first byte in the record (column 1, as it happens), so we enter a 1. Next, we must specify a 0 or a 1 to flag whether we want to sort Key #1 in ascending or descending order. Our street index is alphabetically ordered, (starting at A and ending at Z), so we enter a 0 here to choose ascending order.

Key #2 is our direction, N, S, E or W. The size of Key #2 in this case is always 1. The position of Key #2 in our file STREET.DAT is column (or record byte number) 33. We don't really care whether our directions are ascending or descending yet, but we'll enter a 0 to indicate ascending order.

We don't have a Key #3, so we specify Key #3 size, position and order as 0, 0, and 0 respectively.

Note that we do not specify the data type of keys #1, #2 and #3 for a sequential file because they are always ASCII data, which BASORT knows.

After line 130 is executed, the file STREET.LST is created and the data in STREET.DAT is rewritten in alphabetical order. Lines 135 and 145 are in the program to close, then reopen file channel #1 and the file STREET.DAT. If those two lines are omitted, the new file ENSW.LST, though created, would be empty because no further data would be found in the file STREET.DAT. These two lines cause the BASORT subroutine to look at the beginning of the file, rather than the end.

Line 150 is the second XCALL BASORT program line in the AlphaBASIC program. This line is different than the first (line 130) because we are now specifying the direction byte (N, S, E or W) as Key #1 and the street name as Key #2.

The size of key #1 is always just 1 byte in size, so on the XCALL BASORT program line after the delimiting comma following the subroutine name, we enter a 1. The position of Key #1 is the thirty third byte (column 33), so we next enter a 33. We must specify a 0 or a 1 to flag whether we want to sort Key #1 in ascending or descending order. ASCII sequence puts E first, then N, then S and W, which is fine with us. We'll enter a 0 here.

Key #2 this time is the street name. That is, the size of Key #2 in this case is 25. The position of Key #2 is column (or record byte number) 1. We want the street names within the four direction groups alphabetically ordered, so we specify ascending order, or 0.

Again, we don't have a Key #3, so we specify Key #3 size, position and order as 0, 0, and 0 respectively.

Lines 200, 210 and 220 close our input and two output files. The program prints us a reminder of the file names, then ends.

STREET.LST, the sorted version of all the streets, contained in the file STREET.LST, would appear like this:

```
Achilles Dr.               W
Balboa Dr.                 N
Cisco Kid Rd.              S
Constantine Rd.            W
Edward Teach St.           E
Fremont St.                E
Herman Melville Ln.        W
Homer Ave.                 E
John Carter Ln.            N
John Silver Rd.            W
Kublai Khan Cr.            S
Marco Polo Ave.            E
Michael Fink Dr.           N
Nimrod Cr.                 S
Pontiac Ln.                N
Robinson Crusoe Dr.        S
Sancho Panza Cr.           E
Sinbad St.                 N
William Taft Ave.          S
William Tell Ln.           W
```

The file ENSW.LST, which is the streets first sorted according to their location relative to town center, then sorted alphabetically, would appear like this:

```
Edward Teach St.          E
Fremont St.               E
Homer Ave.                E
Marco Polo Ave.           E
Sancho Panza Cr.          E
Balboa Dr.                N
John Carter Ln.           N
Michael Fink Dr.          N
Pontiac Ln.               N
Sinbad St.                N
Cisco Kid Rd.             S
Kublai Khan Cr.           S
Nimrod Cr.                S
Robinson Crusoe Dr.       S
William Taft Ave.         S
Achilles Dr.              W
Constantine Rd.           W
Friday Dr.                W
Herman Melville Ln.       W
John Silver Rd.           W
William Tell Ln.          W
```

Remember, if you choose not to assign a third key, or perhaps even a second key, you still must place zeros in the size, position and order variables of the keys you omit.

2.3 BASORT ERROR MESSAGES

?AMSORT.SYS not found in memory
: The sort utility routine, AMSORT.SYS, must be loaded into user or system memory before calling BASORT.SBR.

?Bad channel number in XCALL BASORT
: The channel number you passed to BASORT was invalid. This error can occur if the file is not open, or if the value given as channel is not an integer.

?File improperly open in XCALL BASORT
: When you call BASORT, the file you wish to sort must be open for INPUT or RANDOM processing.

?FLTCNV.PRG not found in memory
: For an AMOS system, the floating-point conversion module, FLTCNV.PRG, must be loaded into user or system memory before calling BASORT.SBR.

?Illegal value in XCALL BASORT
: One of the arguments to the BASORT call was invalid. Check the key sizes and positions to make sure they fit into the record size which you specified. Also make sure that you have given valid key types.

?Read file error in XCALL BASORT
: An error occurred during a read operation while sorting your file.

?Write file error in XCALL BASORT
An error occurred during a write operation while sorting your file.

?Wrong record size in XCALL BASORT
The record size you specified when calling BASORT does not match the record size you specified when you OPENed the file.

2.4 SUMMARY

BASORT can sort both random and sequential files, whether or not those files can fit entirely into user memory. The data to be sorted must already be in a format where the BASORT execution line within an AlphaBASIC program can specify the position and size of up to three sort keys. The data can be sorted in ascending or descending order, each key being independent of the others.

Because BASORT combines, as needed, two sort techniques called a memory-based heap sort and a disk-based polyphase merge-sort, it is a relatively fast sort utility subroutine.

CHAPTER 3

COMMON - XCALL SUBROUTINE TO PROVIDE COMMON VARIABLE STORAGE

COMMON is an external subroutine that allows you to place data into a common storage area of either user memory or system memory. The data can be numeric variables or string variables of up to 150 bytes in length.

When this data is in user memory, it may be accessed by separate AlphaBASIC programs, as when chaining from one program to a second that requires variable information defined in the first program. When it is in user memory, the data is only common to programs run by the particular job that placed them in memory.

When in system memory, this common data can be used to pass messages between jobs, or for any other function that requires a data area that is accessible to more than one person.

The common data is placed in either user or system memory via an AlphaBASIC program. The idea is to assign a name to one or several packets of data, which can later, and at various times, be retrieved by other AlphaBASIC programs. The AlphaBASIC program assigns a name to a packet of data by using the BASIC keyword XCALL and then the name of the external subroutine, COMMON. On the same line the AlphaBASIC program must indicate whether it is sending a variable to or retrieving a variable from user or system memory. Following that, on the same line, the program must give either a string variable or a string literal (the name must be six characters or fewer) to be the name of the data packet. Finally, still on the same line, the name of the numeric or string variable containing the data of the packet (which can be up to 150 bytes in length) is specified.

3.1 LOADING COMMON INTO USER OR SYSTEM MEMORY

To insure proper results, you must load the COMMON subroutine into memory before you use it from within an AlphaBASIC program.

You may load COMMON into either system or user memory. If you load COMMON into a user's memory partition, only that user can access the data stored by COMMON. If you load COMMON into system memory (making the data accessible to all users), be sure that you assign a unique name for each packet of data.

To load COMMON into user memory, enter either of the following from AMOS or AMOS/L command level:

```
.LOAD DSK0:COMMON.SBR[7,6] (RET)
.
```

or

```
.LOAD BAS:COMMON.SBR (RET)
.
```

(BAS: is the ersatz name for ppn [7,6] of the system disk). After you see the AMOS or AMOS/L prompt, you may run an AlphaBASIC program that uses the COMMON subroutine.

To load COMMON.SBR into system memory, you must have a line in your system initialization file that performs that function. For more information on loading subroutines into system memory during system boot-up, see the AMOS System Operator's Guide, DSS-10001-00, or the AMOS/L System Operator's Guide, DSS-10002-00.

3.2 USING COMMON FROM WITHIN AN ALPHABASIC PROGRAM

There are two things that the AlphaBASIC program itself must accomplish in order to use the COMMON subroutine. The program must define certain variables that COMMON will use, and it must contain an XCALL command line using the name COMMON and certain parameter specifications.

3.2.1 Defining Variables

To use COMMON from within an AlphaBASIC program, you must first define certain binary variables that tell COMMON to send a packet to memory or to receive one from memory; and, if set to receive, to set a flag if the packet is in fact received.

You define these binary variables by using MAP statements. MAP statements are discussed at length in Chapter 8, "Memory Mapping System," of the AlphaBASIC User's Manual, DWM-00100-01. (The MAP statements you see below will be sufficient for all but the most exotic programs using COMMON.)

To send a packet of data to common memory, you must define a variable (we'll call it SEND), which must appear in the XCALL COMMON program line when you are sending, as:

```
MAP1 SEND,B,1,0
```

This one-byte binary variable always contains zero (the flag telling COMMON to send).

To receive a packet of data from common memory, you must define a two-byte binary variable (we'll call it RECEIVE) which must appear in the XCALL COMMON program line when you are receiving, to communicate two pieces of information to COMMON. The first byte must be a 1, which is the flag to

COMMON that you are going to receive a packet from memory (we'll name that byte F'RCV). The second byte (which we'll call RCVFLG) is a flag you can test after the XCALL COMMON subroutine is executed to see of you did in fact receive the packet. That two-byte binary variable is defined like this:

```
MAP1 RECEIVE
     MAP2 F'RCV,B,1,1
     MAP2 RCVFLG,B,1,0
```

Again, F'RCV always contains a one (the flag telling COMMON to receive). RCVFLG functions as a flag to indicate whether or not COMMON finds the requested packet of information. If COMMON does not find that packet, it will return a zero in this byte; otherwise it is non-zero.

3.2.2 The XCALL COMMON Command Line

You call COMMON to send data to the common area via:

```
XCALL COMMON,SEND,"MSGNAM",INFO
```

You call COMMON to receive data from the common area via:

```
XCALL COMMON,RECEIVE,"MSGNAM",INFO
```

Where:

SEND — A one-byte binary variable that contains zero.

RECEIVE — A two-byte binary variable, where the first byte must be set to one, and the second byte functions as a flag that indicates whether or not COMMON found the requested packet of information. If COMMON did not find that packet, it returns a zero in this byte; otherwise it is non-zero.

IMPORTANT NOTE: Once you use COMMON to retrieve a data packet, that data packet is gone from memory, and cannot be read again.

"MSGNAM" — A string containing from one to six characters that specifies the name of the packet to be sent or received. Note that a string literal must be enclosed in quotation marks. COMMON also can handle a string variable here (e.g., XCALL COMMON,SEND,PACKET,INFO). A string variable, of course, must be defined earlier in the program.

INFO — The variable to hold the data to be sent or received. The variable must represent data that is less than 151 bytes long.

If you load COMMON into system memory (making the data accessible to all users), be sure the 1- to 6-character name is unique for each packet.

3.3 AN EXAMPLE OF COMMON

Let's create a pair of elementary AlphaBASIC programs and put a packet into user memory, then retrieve it. We assume that after you write and compile these programs, you will load the COMMON subroutine into user memory before running them, as we discussed in Section 3.1 above.

To send a data packet to common memory, you may use a routine like this:

```
10 MAP1 SEND,B,1,0
20 MAP1 INFO,S,150
30 MAP1 PACKET,S,6
100 INPUT "Enter message (maximum of 150 characters): ",INFO
110 INPUT "Now enter name of data packet (up to 6 characters): ",PACKET
120 XCALL COMMON,SEND,PACKET,INFO
130 END
```

Line 10 defines the binary variable SEND as a zero. Line 20 defines the variable "INFO" as a string variable up to 150 characters in length (the maximum COMMON can handle). Line 30 defines a string variable called PACKET, which can be up to six characters in length. Line 100 accepts a value and assigns it to the variable INFO, which will make up the data in the packet you'll store in common memory. Line 110 accepts an input string that becomes the name of the packet. Line 120 begins with the BASIC keyword XCALL, which means the program is going to access one of the external subroutines on the system. COMMON is the name of the specific subroutine to be accessed. SEND is the variable name for the binary byte that, because it is a 0, tells COMMON to write into common memory. PACKET is the string variable just entered that names the specific packet, because several can be placed in memory via COMMON at one time. Finally, the value of the variable INFO, from line 100, is placed in common memory under the name defined as PACKET. Then, of course, the program ends in line 130. When the program is run, at this point the packet is in common memory.

To retrieve the packet under the name you input (defined as the string variable PACKET), which is now residing in common memory, you may use a routine like the following:

```
10 MAP1 RECEIVE
20      MAP2 F'RCV,B,1,1
30      MAP2 RCVFLG,B,1,0
40 MAP1 RETRIEVE,S,150
50 MAP1 PACKET,S,6
100 INPUT "Enter name of data packet: ",PACKET
110 XCALL COMMON,RECEIVE,PACKET,RETRIEVE
120 IF RCVFLG=0 PRINT "Message not found" &
      ELSE PRINT "Message is: ";RETRIEVE
130 END
```

Line 10 of the retrieving program is a level 1 MAP statement. The subsequent MAP2 statements pertain to it; when the variable RECEIVE is looked at, the associated information in lines 20 and 30 are automatically accepted as well.

Line 20 defines the binary variable F'RCV as a one, which later will tell COMMON to receive, rather than send. Line 30 contains RCVFLG, another binary byte. This one can be tested by the program following the XCALL to the COMMON subroutine. If this binary variable equals zero, the program can determine that for some reason COMMON did not find the designated packet. A non-zero means it did find it.

Line 40 defines RETRIEVE as a string variable of up to 150 characters in length.

Line 50 defines PACKET as a string variable of up to six characters in length.

Line 100 asks the program user for the name of the packet in common memory that he or she wants to retrieve.

Line 110 begins with the BASIC keyword XCALL, which means the program is going to access one of the external subroutines on the system. COMMON is the name of the specific subroutine to be accessed. RECEIVE is the variable name for the binary byte that, because it is a 1, tells COMMON to find a data packet in common memory. PACKET is the string variable that takes the string the user enters at line 100 and uses it to name the specific packet that COMMON is to find (ignoring any others that may be in memory). Finally, the variable RETRIEVE is assigned the value of the data found in that packet.

Line 120 tests the binary flag to see if the packet was found and displays the appropriate message on your terminal. If the packet is found, its contents are displayed also. Then the program ends.

Sample runs of the sample programs above could be:

```
.RUN FIRST (RET)
Enter message (maximum of 150 characters): TEMPUS FUGIT! (RET)
Now enter name of data packet (up to 6 characters): MESAG1 (RET)
.
```

and:

```
.RUN SECOND (RET)
Enter name of data packet: MESAG1 (RET)
Message is: TEMPUS FUGIT!
```

When running the second program above, if you were to enter a message name that does not represent a packet in common memory, you would see the message from line 120 of the program saying, "Message not found."

3.4 SUMMARY

COMMON is an external subroutine that allows you to place data into a common storage area in memory. This is useful for passing data between chained programs, passing messages between jobs, or any other function that requires a data area accessible to more than one program or person. By assigning a name to each packet of information within the common area, you can have several of these packets in common storage ready to be retrieved by other users or programs at various times.

CHAPTER 4

FLOCK - XCALL SUBROUTINE TO COORDINATE MULTI-USER FILE ACCESS

The name FLOCK is an acronym for "File Locking." FLOCK is an external subroutine that is callable from AlphaBASIC, and is used in a program that accesses files when it is necessary to protect a file or files from concurrent access by another user. In other words, FLOCK prevents one user from accessing information that another user is updating at the same time.

Below we describe in some detail the potential problems of multi-user file access. Then, afterward, we detail how you can use FLOCK from an AlphaBASIC program to coordinate shared file access and processing, and offer you some schemes to implement FLOCK in your AlphaBASIC programs. Finally, we discuss the hazards of "Deadlock," and how FLOCK conquers that too.

4.1 THE MULTIPLE UPDATE PROBLEM

Consider the following program:

```
10 OPEN #1,"FILE",RANDOM,6,KEY
20 KEY = 1
30 READ #1,ONE
40 ONE = ONE + 1
50 WRITE #1,ONE
60 CLOSE #1
70 END
```

The purpose of this program is to increment record 1 of 'FILE' by one. If two users execute this program concurrently, we wish the value in record one to be incremented by two, thus:

ONE	USER #1	REC #1	USER #2	ONE
-	OPEN #1,"FILE",RANDOM,6,KEY	5		
-	KEY = 1	5		
5	READ #1,ONE	5		
6	ONE = ONE + 1	5		
6	WRITE #1,ONE	6		
6	CLOSE #1	6		
6	END	6		
-		6	OPEN #1,"FILE",RANDOM,6,KEY	-
-		6	KEY = 1	-
-		6	READ #1,ONE	6
-		6	ONE = ONE + 1	7
-		7	WRITE #1,ONE	7
-		7	CLOSE #1	7
-		7	END	7

NOTE: In this example, the value in record 1 is initially 5.

However, under some circumstances it is possible for record 1 to be incremented by only 1, rather than 2, after being accessed by two users concurrently:

ONE	USER #1	REC #1	USER #2	ONE
-	OPEN #1,"FILE",RANDOM,6,KEY	5		
-	KEY = 1	5		
5	READ #1,ONE	5		
5		5	OPEN #1,"FILE",RANDOM,6,KEY	-
5		5	KEY = 1	-
5		5	READ #1,ONE	5
5		5	ONE = ONE + 1	6
5		6	WRITE #1,ONE	6
5		6	CLOSE #1	6
5		6	END	6
6	ONE = ONE + 1	6		
6	WRITE #1,ONE	6		
6	CLOSE #1	6		
6	END	6		

To prevent multiple update problems from occurring, we need some method to prevent the kind of overlap in READ-modify-WRITE sequences on shared data that is illustrated above.

4.2 THE INTERCONSISTENCY PROBLEM

Consider the following two programs:

```
10 OPEN #1,"FILE",RANDOM,6,KEY
20 KEY = 1 : READ #1,ONE
25 ONE = ONE + 1 : WRITE #1,ONE
30 KEY = 2 : READ #1,ONE
35 ONE = ONE+1 : WRITE #1,ONE
40 CLOSE #1 : END
```

```
10 OPEN #1,"FILE",RANDOM,6,KEY
20 KEY = 1 : READ #1,ONE
30 KEY = 2 : READ #2,TWO
40 PRINT ONE - TWO
50 CLOSE #1 : END
```

If the values in records one and two of 'FILE' are identical, then they should continue to be identical if the first program (which increments the values in both records by one) is executed. Hence, if the values in records one and two are identical, and we execute both of the above programs concurrently, we would like the second program to print zero, thus:

```
ONE       USER #1                   REC #1  #2      USER #2                      ONE  TWO
-----------------------------------------------------------------------------------------
 -   OPEN #1,"FILE",RANDOM,6,KEY     5   5
 5   KEY = 1 : READ #1,ONE           5   5
 6   ONE = ONE + 1 : WRITE #1,ONE    6   5
 5   KEY = 2 : READ #1,ONE           6   5
 6   ONE = ONE + 1 : WRITE #1,ONE    6   6
 6   CLOSE #1 : END                  6   6
 -                                   6   6   OPEN #1,"FILE",RANDOM,6,KEY     -    -
 -                                   6   6   KEY = 1 : READ #1,ONE           6    -
 -                                   6   6   KEY = 2 : READ #1,TWO           6    6
 -                                   6   6   PRINT ONE - TWO                 6    6
                                             0
 -                                   6   6   CLOSE #1 : END                  6    6
-----------------------------------------------------------------------------------------
```

However, under some circumstances it is possible for the second program to print 1, rather than 0:

```
ONE        USER #1                    REC #1 #2        USER #2                        ONE TWO
-----------------------------------------------------------------------------------------------
 -   OPEN #1,"FILE",RANDOM,6,KEY     5  5
 5   KEY = 1 : READ #1,ONE           5  5
 6   ONE = ONE + 1 : WRITE #1,ONE    6  5
 6                                   6  5   OPEN #1,"FILE",RANDOM,6,KEY     -   -
 6                                   6  5   KEY = 1 : READ #1,ONE           6   -
 6                                   6  5   KEY = 2 : READ #1,TWO           6   5
 6                                   6  5   PRINT ONE-TWO                   6   5
                                            1
 6                                   6  5   CLOSE #1 : END                  6   5
 5   KEY = 2 : READ #1,ONE           6  5
 6   ONE = ONE + 1 : WRITE #1,ONE    6  6
 6   CLOSE #1 : END                  6  6
-----------------------------------------------------------------------------------------------
```

The READ-WRITE-READ-WRITE sequence in the first program can be considered as steps in a single update operation. To maintain interconsistency-- that is, to eliminate the situation outlined above-- we need a mechanism to prevent access to a collection of data during any update operation. Otherwise, the collection of data we retrieve may be only partially updated, due to interference from another program which has concurrently accessed that data.

In actual applications, the loss of interconsistency described above can cause you to access nonexistent records through a faulty index file, to derive incorrect totals on reports, to create inconsistent reports, and so forth.

4.3 THE FLOCK SUBROUTINE

FLOCK exists to prevent multiple update problems, interconsistency flaws, and other file access hazards that may occur if you are not the only user on your system. FLOCK provides a way to synchronize attempts at accessing files and devices so that you and the other users can avoid partially updating or scrambling data.

4.3.1 FLOCK Program Requirements

FLOCK only functions properly if it is loaded into system memory. FLOCK resides in account DSK0:[7,6], and has a .SBR extension. If you have an AMOS system, rather than an AMOS/L system, FLOCK also requires that you have FLTCNV.PRG in system memory. FLTCNV.PRG resides in account DSK0:[1,4].

IMPORTANT NOTE: You must load FLOCK into system memory only; it will appear to work if you load it into user memory, but no file locking will actually occur.

To load FLOCK.SBR (and FLTCNV.PRG for an AMOS system) into system memory, you must have lines in your system initialization command file that perform those functions. For more information on loading subroutines into system memory during system boot-up, see the AMOS System Operator's Guide, DSS-10001-00, or the AMOS/L System Operator's Guide, DSS-10002-00.

4.3.2 FLOCK Calling Sequence

The calling sequence for FLOCK in AlphaBASIC is:

```
XCALL FLOCK,ACTION,MODE,RETURN-CODE,FILE,RECORD
```

Where:

1. Action, Mode, File, and Record are all either floating point expressions which evaluate to positive integer values, or string expressions which represent positive integer values.

2. Return-Code is a 6-byte floating point variable.

4.3.2.1 Action & Mode

Action, modified by mode, specifies the action to be performed by FLOCK. A quick-reference summary of the actions and their modes is in Section 4.6.1. The actions, and their modes:

Action 0, Mode 0: Requests permission to open 'File' for non-exclusive use (that is, other users can access the file). The request is placed in a first-come-first-served queue and the program is delayed until the request can be granted.

Action 0, Mode 2: Requests permission to open 'File' for exclusive use. The request is placed in a first-come-first-served queue and the program is delayed until the request can be granted.

Action 0, Mode 4: Requests permission to open 'File' for non-exclusive use. If the request cannot be immediately granted, Return-Code 1 is returned.

Action 0, Mode 6: Requests permission to open 'File' for exclusive use. If the request cannot be immediately granted, Return-Code 1 is returned.

Action 1, Mode 0: Informs FLOCK that 'File' has been closed. Unlocks the file. Implicitly informs FLOCK that any processing of records in 'File' has been completed (i.e., Actions 5 or 6 are performed automatically as necessary).

Action 2, Mode 0: Informs FLOCK that abnormal program termination is about to occur (e.g., during an error handling routine). Releases all locks on all files by performing Action 1 as necessary.

Action 3, Mode 0: Requests permission to read 'Record' of 'File' for non-exclusive use (i.e., record will not be used to update file). Permission to open 'File' must already be granted. The request is placed in a first-come-first-served queue and the program is delayed until the request can be granted.

Action 3, Mode 2: Requests permission to read 'Record' of 'File' for exclusive use (i.e., record will be used to update file). Permission to open 'File' must already be granted. The request is placed in a first-come-first-served queue and the program is delayed until the request can be granted.

Action 3, Mode 4: Requests permission to read 'Record' of 'File' for non-exclusive use (i.e., record will not be used to update file). Permission to open 'File' must already be granted. If the request cannot be immediately granted, Return-Code 1 is returned.

Action 3, Mode 6: Requests permission to read 'Record' of 'File' for exclusive use (i.e., record will be used to update file). Permission to open 'File' must already be granted. If the request cannot be immediately granted, Return-Code 1 is returned.

Action 4, Mode 2: Requests permission to read/write all records of 'File' for exclusive use (i.e., processing will update and possibly re-create file). Permission to open 'File' must already be granted. The request is placed in a first-come-first-served queue and the program is delayed until the request can be granted.

Action 4, Mode 6: Requests permission to read/write all records of 'File' for exclusive use (i.e., processing will update and possibly re-create file). Permission to open 'File' must already be granted. If the request cannot be immediately granted, Return-Code 1 is returned.

Action 5, Mode 0: Informs FLOCK that processing of 'Record' of 'File', for which permission was granted by Action 3, has been completed. The record is unlocked. If data has been buffered for output, it is written to disk.

Action 6, Mode 0: Informs FLOCK that exclusive processing of 'File', for which permission was granted by Action 4, has been completed. The file is unlocked. Any succeeding programs which are granted use of 'File' by Actions 3 or 4 will automatically reopen 'File'. This is done in case exclusive processing of 'File' has caused it to be re-created. If data has been buffered for output, it is written to disk.

4.3.2.2 File

File specifies a file-channel number. File is ignored by Action 2 and may be omitted if 'Record' is also omitted. The file specified may be either RANDOM or SEQUENTIAL for Actions 0 and 1, but must be a RANDOM file for all other actions.

IMPORTANT NOTE: In order for FLOCK to function properly, file-channel numbers should denote specific and unique files. This means you must systematically assign file-channel numbers to your files when designing applications programs, being careful to assign the same numbers to the same files and different numbers to different files.

File-channel numbers 1 through 999 have been reserved for use by Alpha Micro software. Although there is nothing to prevent your programs from using these numbers, we advise you not to do so in conjunction with FLOCK so that no conflict can arise between your application programs and any present or future Alpha Micro software on your system.

4.3.2.3 Record

Record specifies a logical record number. For Actions 0 through 2, 4, and 6, record is ignored and may be omitted.

4.3.2.4 Return-Code

Return-Code denotes a variable in which FLOCK places a number that indicates the success or failure of an action:

- Code 0: Successful (All actions)
- Code 1: Resource unavailable (Actions 0, 3, 4)
- Code 2: Open request has already been granted (Action 0)
- Code 3: Permission to open must first be granted (Actions 1, 3-6)
- Code 4: Duplicate request for use of some record in file (Actions 3, 4)
- Code 6: Permission to use some record in file must first be granted (Actions 5, 6)
- Code 100: Unimplemented Action
- Code 101: File-channel number is not open in AlphaBASIC for RANDOM processing (Actions 3-6)

Code 102: File-channel is already open in AlphaBASIC for an ISAM indexed file.
Code 103: For actions 0, 3 and 4: Less than 15 queue blocks are available.

A Return-Code greater than 1 is an indication of some programming error. For calls to FLOCK which do not use modes 4 or 6, you should use the following statement while debugging your program:

```
IF Return-Code>1 THEN PRINT "FLOCK Error" : STOP
```

For calls which use modes 4 or 6, Return-Code = 1 should be checked to determine if FLOCK was able to immediately satisfy the request. Modes 4 and 6 are generally used in this way to allow the user to cancel a request which may involve a lengthy delay.

4.3.3 Queue Block Requirements

The FLOCK subroutine builds its dynamic tables out of monitor queue blocks. The monitor queue is a list of blocks of system memory which are linked to each other in a forward chain. It is very important, before running any AlphaBASIC program using FLOCK, to ensure that the monitor is configured to make an adequate number of these queue blocks available. The number of queue blocks FLOCK uses varies with the number of jobs accessing files, the number of files open at one time, and the number of records open for each file. Currently, at any given moment during the use of FLOCK, the number of queue blocks being used equals:

> twice the number of different files open using FLOCK, plus
> the number of different records open using FLOCK, plus
> the number of jobs with files open using FLOCK, plus
> the total number of FLOCK opens (i.e., number of Action 0s) that haven't been closed, plus
> the total number of record uses (i.e., number of Action 3s) that haven't been released
>
> (The last two factors of this equation anticipate circumstances where the same file and/or the same record is being accessed by more than one job at a time. If two jobs are reading the same file, that is two opens or two Action 0s.)

NOTE: If FLOCK changes in the future, the above formula may also require modification.

The monitor is initially generated with 20 free blocks in the available queue. You may modify the system initialization command file to allocate more queue blocks by adding the "QUEUE nnn" command anywhere in the system initialization command file prior to the final SYSTEM command. When the QUEUE nnn command is executed, "nnn" more queue blocks will be allocated for general use. For more information on modifying the system initialization command file, see the AMOS System Operator's Guide, DSS-10001-00, or the AMOS/L System Operator's Guide, DSS-10002-00.

NOTE: You may use the QUEUE command at monitor level to determine your system's use of queue blocks. The system will respond with the current number of free queue blocks in the available queue list. For example:

```
.QUEUE (RET)
20 Queue blocks available
```

4.4 USING FLOCK

There are three levels of increasing complexity with which FLOCK subroutine calls may be incorporated into a program system:

1. Use Actions 0 through 2 to implement file-open interlocks (see Section 4.2.1).

2. Use Actions 0 through 2 to implement file-open interlocks and use Actions 3 and 5 to implement individual record-update interlocks (see Section 4.2.2).

3. Use Actions 0 through 2, 4, and 6 to implement complete file interlocks and use Actions 3 and 5 to implement individual record-processing interlocks (see Section 4.2.3).

The problems outlined in Sections 4.1.2 and 4.1.3 can be solved by using FLOCK to any of the above levels of complexity. In your design you are free to trade off complexity for performance, so long as you use a single level of complexity consistently for any given data file.

4.4.1 File-Open Interlocks

Using just Actions 0 through 2, it is possible to implement a very simple file access coordination scheme which solves the problems of Sections 4.1.2 and 4.1.3. Action 0, Mode 0 or 4, is used before opening a file for input only (that is, opening a file for RANDOM processing, upon which only READs will be performed). Action 0, Mode 2 or 6, is used before opening a file for output (e.g., a file open for RANDOM processing, upon which READs or WRITEs will be performed, or a file which may be re-created). Finally, Action 1 is used after closing any file, and Action 2 is used before any abnormal termination points in the program.

4.4.1.1 <u>The Multiple Update Problem</u>

Here is how the program of Section 4.1.2 can be rewritten to incorporate file-open interlocks:

```
10 XCALL FLOCK,0,2,RET,1000
20 OPEN #1000,"FILE",RANDOM,6,KEY
30 KEY = 1
40 READ #1000,ONE
50 ONE = ONE + 1
60 WRITE #1000,ONE
70 CLOSE #1000
80 XCALL FLOCK,1,0,RET,1000
90 END
```

The program now will function correctly in a concurrent environment. If any other programs have 'FILE' open when line 10 is executed (and have correctly informed FLOCK of the fact with Action 0), FLOCK will make the above program wait until the other program closes 'FILE'. Furthermore, no more programs will be allowed to open 'FILE' until the above program reaches line 80.

The above program has no provisions for the user typing ^C, or for other errors occurring which will abort execution. This can be corrected by further rewriting the program, as follows:

```
5      ON ERROR GOTO ABORT
10     XCALL FLOCK,0,2,RET,1000
20     OPEN #1000,"FILE",RANDOM,6,KEY
30     KEY = 1
40     READ #1000,ONE
50     ONE = ONE + 1
60     WRITE #1000,ONE
70     CLOSE #1000
80     XCALL FLOCK,1,0,RET,1000
90     END
100 ABORT:
110    XCALL FLOCK,2,0,RET
120    ON ERROR GOTO 0
```

4.4.1.2 The Interconsistency Problem

Here is how the programs of Section 4.1.3 can be rewritten to incorporate file-open interlocks. The first program:

```
10    ON ERROR GOTO ABORT
20    XCALL FLOCK,0,2,RET,1000
30    OPEN #1000,"FILE",RANDOM,6,KEY
40    KEY = 1 : READ #1000,ONE
50    ONE = ONE + 1 : WRITE #1000,ONE
60    KEY = 2 : READ #1000,ONE
70    ONE = ONE + 1 : WRITE #1000,ONE
80    CLOSE #1000
90    XCALL FLOCK,1,0,RET,1000
100   END
110 ABORT:
120   XCALL FLOCK,2,0,RET
130   ON ERROR GOTO 0
```

The second program:

```
10    ON ERROR GOTO ABORT
20    XCALL FLOCK,0,0,RET,1000
30    OPEN #1000,"FILE",RANDOM,6,KEY
40    KEY = 1 : READ #1000,ONE
50    KEY = 2 : READ #1000,TWO
60    PRINT ONE - TWO
70    CLOSE #1000
80    XCALL FLOCK,1,0,RET,1000
90    END
100 ABORT:
110   XCALL FLOCK,2,0,RET
120   ON ERROR GOTO 0
```

The above programs will now function correctly in a concurrent environment. While the first program is updating 'FILE', no other programs can have 'FILE' open. This prevents the second program from reading 'FILE' when it is in a partially updated state.

Since the second program does not update 'FILE', it requests permission to open it using Mode 0 with Action 0. This enables other programs which read but do not update 'FILE' to open and process 'FILE' simultaneously.

4.4.2 Record-Update Interlocks

Most programs open files when the programs begin, and close those files when they end. The programs may not actually need the files to be open throughout execution, but by not repeatedly opening and closing the files, the programs avoid many undesirable delays.

File-open interlocks that are set lock out the entire file; if a file is open throughout the run of a program, and thus unavailable to programs run by other users, serious or annoying delays can result.

Although file-open interlocks do prevent concurrency problems, they generally reduce concurrency far more than is necessary. Typically, file-open interlocks lock out the entire file to prevent access to the single record. Locking out an entire file to prevent access to a single record is like using a sledge hammer to drive a push-pin. All that is actually necessary is to delay any other user attempting to modify the record until the user originally accessing the record is done.

Consider an example of application in which you and several other users are interactively updating an employee record file. Assume files are kept open only where required. Once you display an employee's record, it is necessary that all the other users wait for you to finish making changes to that record before they can, in turn, access it; otherwise two users might concurrently attempt to update the same employee record. This results in the multiple update problem described in Section 4.1.2. In other words, all other users must wait for one user to enter changes to the employee's record before any other user can access and modify that record. This is called a record-update interlock, and is a far less severe restriction to all the users accessing a file than a file-open interlock is. (NOTE: You should remember, when performing a record-update interlock, that FLOCK converts logical record numbers into physical block numbers. All record locking operations are performed on physical blocks, not logical records. If both you and another user attempt to lock two separate logical records within the same physical block, you will see the error message "Record already locked.")

Actions 3 and 5 of FLOCK permit control of concurrent access to individual records. Action 3, Mode 0 or 4, is used before reading a sequence of records which will not be used for updating, in order to prevent interconsistency errors (see Section 4.1.3). Action 5 is used after the sequence of reads. Action 3, Mode 2 or 6, is used before reading records which will be used for updating. Action 5 is used again after rewriting the records.

4.4.2.1 The Multiple Update Problem

Here is how the program of Section 4.1.2 can be rewritten to incorporate Record-Update interlocks:

```
5      ON ERROR GOTO ABORT
10     XCALL FLOCK,0,0,RET,1000
20     OPEN #1000,"FILE",RANDOM,6,KEY
30     KEY = 1
40     XCALL FLOCK,3,2,RET,1000,KEY
50     READ #1000,ONE
60     ONE = ONE + 1
70     WRITE #1000,ONE
80     XCALL FLOCK,5,0,RET,1000,KEY
90     CLOSE #1000
100    XCALL FLOCK,1,0,RET,1000
110    END
120 ABORT:
130    XCALL FLOCK,2,0,RET
140    ON ERROR GOTO 0
```

4.4.2.2 The Interconsistency Problem

Here is how the programs of Section 4.1.3 can be rewritten to incorporate Record-Update interlocks:

```
10   ON ERROR GOTO ABORT
20   XCALL FLOCK,0,0,RET,1000
21   OPEN #1000,"FILE",RANDOM,6,KEY
30   KEY = 1
31   XCALL FLOCK,3,2,RET,1000,KEY
32   READ #1000,ONE : ONE = ONE + 1 : WRITE #1000,ONE
33   XCALL FLOCK,5,0,RET,1000,KEY
40   KEY = 2
41   XCALL FLOCK,3,2,RET,1000,KEY
42   READ #1000,ONE : ONE = ONE + 1 : WRITE #1000,ONE
43   XCALL FLOCK,5,0,RET,1000,KEY
50   CLOSE #1000
51   XCALL FLOCK,1,0,RET,1000
60   END
70 ABORT:
71   XCALL FLOCK,2,0,KEY
72   ON ERROR GOTO 0
```

```
10   ON ERROR GOTO ABORT
20   XCALL FLOCK,0,0,RET,1000
21   OPEN #1000,"FILE",RANDOM,6,KEY
30   XCALL FLOCK,3,0,RET,1000,1
31   XCALL FLOCK,3,0,RET,1000,2
32   KEY = 1 : READ #1000,ONE
33   KEY = 2 : READ #1000,TWO
34   XCALL FLOCK,5,0,RET,1000,2
35   XCALL FLOCK,5,0,RET,1000,1
40   PRINT ONE - TWO
50   CLOSE #1000
51   XCALL FLOCK,1,0,RET,1000
60   END
70 ABORT:
71   XCALL FLOCK,2,0,KEY
72   ON ERROR GOTO 0
```

4.4.3 Improved File Interlocks

In Section 4.2.2 we said that file-open interlocks can incur long delays upon any users trying to access a file after one user has opened it and therefore locked them out. Nevertheless, it is sometimes necessary to lock an entire file for exclusive use. For example, if file XYZ is becoming full, you might wish to copy the file XYZ into a new, larger file TEMP, and then delete XYZ and rename TEMP to XYZ. Or, as another example, you might wish to reorganize an index and data file. Obviously, during these maneuvers, you want assurance that no other user can access the file.

Action 4 obtains exclusive access to a file by obtaining exclusive access to all the records of that file. Exclusive access is relinquished by using Action 6. Action 3, Mode 0 or 4, is necessary before reading a sequence of records in order to avoid the interconsistency problem. If Action 4 is used, it is necessary to use Action 3, Mode 0 or 4, before reading individual records which won't be used for updating. This is because a user who has exclusive use of a file can re-create it, which requires that all other users with the file open must then reopen it. Action 3 performs the necessary reopenings.

4.4.3.1 Example

Here are two partial programs which illustrate the use of improved file interlocks:

```
10 !REORGANIZATION PROGRAM
15 XCALL FLOCK,0,0,RET,1001
20 XCALL FLOCK,0,0,RET,1002
25 OPEN #1001,"INDEX",RANDOM,512,KEY1
30 OPEN #1002,"DATA",RANDOM,512,KEY2
35 XCALL FLOCK,4,2,RET,1001
40 XCALL FLOCK,4,2,RET,1002
45 CALL REORGANIZE  !  REORGANIZE INDEXED DATA FILE
50 XCALL FLOCK,6,0,RET,1002
55 XCALL FLOCK,6,0,RET,1001
60 CLOSE #1001 : CLOSE #1002
65 XCALL FLOCK,1,0,RET,1001
70 XCALL FLOCK,1,0,RET,1002
75 END

100 REORGANIZE:
110  REMARK *** SUBROUTINE GOES HERE ***
120  RETURN
```

```
10  !INQUIRY PROGRAM
15  XCALL FLOCK,0,0,RET,1001
20  XCALL FLOCK,0,0,RET,1002
25  OPEN #1001,"INDEX",RANDOM,512,KEY1
30  OPEN #1002,"DATA",RANDOM,512,KEY2
35  EMPLOYEE'ENTRY:
40    INPUT "EMPLOYEE #",EMPLOYEE$
45    IF EMPLOYEE$ = "" THEN LEAVE
50    CALL LOOKUP !LOCATE EMPLOYEE$ IN INDEX FILE, &
                   RETURN EMPLOYEE REC # IN KEY2
55                !XCALL FLOCK,0,0,RET,KEY1 IS IN EFFECT &
                   WHEN LOOKUP RETURNS
60    IF KEY2 = 0 THEN ?"EMPLOYEE NOT ON FILE" : GOTO EMPLOYEE'ENTRY
65    XCALL FLOCK,3,4,RET,1002,KEY2
70    IF RET <> 1 THEN 55
75     INPUT "DO YOU WISH TO WAIT? ",ANSWER$
80     IF UCS(ANSWER$) <> "Y" AND UCS(ANSWER$) <> "YES" &
          THEN EMPLOYEE'ENTRY
85     XCALL FLOCK,3,0,RET,1002,KEY2
90    READ #1000,EMPLOYEE'RECORD
95    XCALL FLOCK,5,0,RET,1002,KEY2
100   XCALL FLOCK,5,0,RET,1001,KEY1
105   CALL DISPLAY !  DISPLAY EMPLOYEE'RECORD
110   GOTO EMPLOYEE'ENTRY
200 LEAVE:
210   CLOSE #1001 : CLOSE #1002
220   XCALL FLOCK,1,0,RET,1001
230   XCALL FLOCK,1,0,RET,1002
300 END
400 LOOKUP: REMARK **SUBROUTINE GOES HERE**
499   RETURN
500 DISPLAY: REMARK **SUBROUTINE GOES HERE**
599   RETURN
```

4.5 DEADLOCK, AND HOW TO PREVENT IT

NOTE: For the purposes of the following discussion, having permission to open a file or use a record is referred to as possessing a resource.

The possession of a resource by some job XYZ can directly or indirectly cause the execution of other jobs to be delayed. It is then possible for one of these delayed jobs to possess a resource needed by job XYZ, thus causing execution of job XYZ to be delayed also. This is known as a DEADLOCK. None of the jobs involved can proceed since each requires a resource owned by one of the other jobs involved. The situation is permanent because none of the jobs involved can proceed until one of the other jobs proceeds and relinquishes a needed resource.

DEADLOCK can only occur if a job requests more than one resource simultaneously. There is a simple way to prevent DEADLOCK, a method which, in most cases, is feasible to implement. The method is: ALWAYS request resources in the same order.

Here is a simple illustration of the principle. First we consider what can happen if resources are requested in differing order in two programs:

```
10  !PROGRAM 1
20  XCALL FLOCK,0,2,RET,1001
21  XCALL FLOCK,0,2,RET,1002
100 REMARK  ** BODY OF PROGRAM **
990 XCALL FLOCK,1,0,RET,1002
991 XCALL FLOCK,1,0,RET,1001
992 END

10  !PROGRAM 2
20  XCALL FLOCK,0,2,RET,1002
21  XCALL FLOCK,0,2,RET,1001
100 REMARK ** BODY OF PROGRAM **
990 XCALL FLOCK,0,2,RET,1001
991 XCALL FLOCK,0,2,RET,1002
992 END
```

Consider the following sequence of execution:

1. Program 1 executes lines 10 and 20, obtaining exclusive permission to open file 1001.

2. Program 2 executes lines 10 and 20, obtaining exclusive permission to open file 1002. It then executes line 21, and must be delayed because Program 1 already has exclusive permission to open file 1001.

3. Program 1 executes line 21, and must be delayed because Program 2 already has exclusive permission to open file 1002.

At this point, programs 1 and 2 have both been delayed. Since no other programs are present, the reasons for their delays will remain unchanged. DEADLOCK has occurred.

But DEADLOCK will not occur if program 2 requests permission to open files 1001 and 1002 for exclusive use in the same order as program 1. For DEADLOCK to occur, program 1 must be granted permission to open file 1001 for exclusive use, but be delayed permission to open file 1002 for exclusive use. However, if program 1 is granted permission to open file 1001 for exclusive use, the corrected program 2 (a duplicate of program 1) will not be allowed to execute lines 21-990; thus it will be unable to obtain permission to open file 1002 for exclusive use. DEADLOCK cannot occur.

4.6 SUMMARY

The FLOCK.SBR program is an external XCALL subroutine which is callable from BASIC. FLOCK locks files, and can lock records within files, to prevent concurrent access by other users running programs that access the same files. FLOCK may also be used to coordinate shared file access and processing.

FLOCK only functions properly if it is loaded into system memory via the SYSTEM command in the system initialization command file, DSK0:SYSTEM.INI[1,4]. If you have an AMOS system, FLOCK also requires that you have FLTCNV.PRG in system memory.

4.6.1 Quick Reference Summary of Actions/Modes

ACTION 0: REQUEST TO OPEN FILE
- MODE 0: Non-exclusive; delay until free
- MODE 2: Exclusive; delay until free
- MODE 4: Non-exclusive; RETURN'CODE = 1 if not free
- MODE 6: Exclusive; RETURN'CODE = 1 if not free

ACTION 1: TELLS FLOCK THAT FILE IS CLOSED. RELEASES THE LOCK. (ACTIONS 5 AND 6 PERFORMED AS NECESSARY.)

ACTION 2: TELLS FLOCK THAT A PROGRAM ABORT IS ABOUT TO OCCUR. RELEASES ALL LOCKS ON ALL FILES BY PERFORMING ACTION 1 AS NECESSARY.

ACTION 3: REQUEST TO READ RECORD.
- MODE 0: Non-exclusive; delay if not free. (Action 0 must already have been granted.)
- MODE 2: Exclusive; delay if not free. (Action 0 must already have been granted.)
- MODE 4: Non-exclusive; RETURN'CODE = 1 if not free. (Action 0 must already have been granted.)
- MODE 6: Exclusive; RETURN'CODE = 1 if not free. (Action 0 must already have been granted.)

ACTION 4: REQUEST TO READ/WRITE ALL RECORDS.
- MODE 2: Exclusive; delay if not free. (Action 0 must already have been granted.)
- MODE 6: Exclusive; RETURN'CODE = 1 if not free. (Action 0 must already have been granted.)

ACTION 5: TELLS FLOCK THAT YOU HAVE FINISHED PROCESSING THE RECORD REQUESTED BY A PREVIOUS ACTION 3 CALL. ANY BUFFERED DATA IS OUTPUT TO DISK.

ACTION 6: TELLS FLOCK THAT YOU HAVE FINISHED PROCESSING THE FILE REQUESTED BY A PREVIOUS ACTION 4 CALL. ANY BUFFERED DATA IS OUTPUT TO DISK.

CHAPTER 5

XLOCK - XCALL SUBROUTINE FOR MULTI-USER LOCKS

XLOCK is an external subroutine that your AlphaBASIC program can call to set and test "locks."

A lock is a tool to help you synchronize attempts to access devices and files. You can imagine the problems that result when you have two users trying to update the same record of the same file at the same time. A lock is an entity created by a program to help it keep track of whether a certain device, file, etc., is in use at the specific time that the program wants to access it. The general way that the locking system works is this:

1. When you want to prevent access to something (a file, a device, etc.) while your program accesses it, you create (that is, "set") a system lock on that resource.

2. Whenever you want to access a device or file, your program tries to set the lock associated with that item; if it is already set, you know that another user's program is using the device or file.

3. When you are finished accessing a device or file, you destroy (that is, "clear") the lock so that other programs can now access the resource.

Note that a system lock is <u>NOT</u> a security device-- it's a convenience. If a program wants to allow its users to write to a file without checking to see if another user is there first, it can do so (and run the risk of creating chaos). A system lock simply provides a convenient way to help a program keep its users from conflicting in their attempts to use system resources. The only job that can clear a lock is the job that originally set the lock. AlphaBASIC does not automatically clear locks when a program exits, so be careful that your program clears any locks it has set before it exits. (For more background information on why locks are necessary, see Chapter 4, "FLOCK - XCALL Subroutine to Coordinate Multi-user File Access.")

5.1 LOADING XLOCK INTO SYSTEM MEMORY

You must include the DSK0:XLOCK.SBR[7,6] in system memory before you can use an AlphaBASIC program implementing XLOCK.

To load XLOCK.SBR into system memory, you must have a line in your system initialization command file that performs that function. For more information on loading subroutines into system memory during system boot-up, see the AMOS System Operator's Guide, DSS-10001-00, or the AMOS/L System Operator's Guide, DSS-10002-00.

5.2 THE XLOCK SUBROUTINE

```
XCALL XLOCK, MODE, LOCK1, LOCK2
```

Where:

MODE The function you want to perform. These modes are:

- Mode 0: Set lock and return.
- Mode 1: Set lock. (Wait if already locked; then set).
- Mode 2: Clear lock (if set by your job).
- Mode 3: Return list of all system locks and the jobs that set them.

(See below for a discussion of each mode.)

LOCK1 The first digit of the lock code. (See below.)

LOCK2 The second digit of the lock code. (See below.)

Use MAP statements at the front of your program to define MODE, LOCK1, and LOCK2 as two-byte binary variables. (They may not be floating point or string variables.) For example:

```
MAP1 MODE, B, 2
MAP1 LOCK1, B, 2
MAP1 LOCK2, B, 2
```

Before you call XLOCK, your AlphaBASIC program must first set up the correct values for MODE, LOCK1, and LOCK2.

IMPORTANT NOTE: XLOCK parameters must be defined on even-byte boundaries in memory. (That is, the variables must begin on word boundaries.) Variable structures defined at a MAP1 level always begin on a word boundary. Therefore, the easiest way to ensure that XLOCK arguments begin on a word boundary is to define them in MAP1 statements (as in the example above). If you do define XLOCK parameters in deeper level MAP statements (e.g., MAP2 or MAP3), make sure that the variables begin on a word boundary by keeping the number of bytes defined an even number. For example, this definition:

```
MAP1  PARAMETERS
 MAP2  FILL, S, 1
 MAP2  L1, B, 2
 MAP2  L2, B, 2
 MAP2  MODE, B, 2
```

will cause XLOCK to fail; however, removing the definition for FILL (which pushed the XLOCK parameters onto an odd-byte boundary) will correct the problem.

5.3 THE LOCKS

A system lock is a two-level numeric lock; the number representing either level may be from 1 to 65535. (A value of zero in either position acts as a wildcard. That is, any number will match in that position when it comes to clearing or setting that lock.) Some typical locks are:

```
1,1
1,2
4,0
100,100
```

The numbers you choose are up to you. You may choose to assign some meaning to the numbers (for example, the first number might be the file-channel number of the file you want to lock, and the second number might be the number of the record within that file that you want to lock.)

Since both numbers in the lock may range from 1 to 65535, the actual possible number of unique locks is 65535 * 65535. But, every time you create a lock, the system sets aside a block in the monitor queue in system memory for that lock, which is not returned to the available list until the lock is released by the job that has it locked. Since there are initially only 20 queue blocks available, it's a good idea to keep the number of locks to a minimum. A good rule is that a program should not have more than two or three locks active at any one time. As you clear a lock, that queue block becomes available again. (So, in essence, every time you set a lock you create it, and every time you clear a lock, you destroy it.)

5.4 THE MODES

The MODE argument in the XLOCK call line can contain one of four values (0-3) which selects one of the four possible locking modes:

5.4.1 MODE 0 (Lock and Return)

This mode tells XLOCK to create a lock with the value LOCK1,LOCK2. If the lock already exists (i.e., some other job is accessing the file or device you want to use), XLOCK returns with MODE equal to the number of the job that set the lock. (A job number is assigned to each job in the order that the jobs were defined in the JOBS command in the system initialization command file. For example, the first job defined in the JOBS command line is Job #1. The SYSTAT command lists the jobs in this order.) If the lock does not already exist, XLOCK creates it and returns with a zero in MODE. You've now set the lock.

5.4.2 MODE 1 (Lock and Wait)

This XLOCK mode is identical to MODE 0, except that if the lock already exists, XLOCK tells the system to put your job to sleep until the lock is cleared. That means that your job will be in an inactive state (except for waking at every clock tick to test the status of the lock) until the job that originally set the lock clears it. If you use this mode, take into consideration the fact that another user may be waiting for the same lock; it's possible that the lock might be cleared and then grabbed up either by the same or another job before your job wakes up.

5.4.3 MODE 2 (Clear Lock)

XLOCK clears the lock specified by LOCK1 and LOCK2 and returns to your program. A zero returned in MODE indicates that the lock you tried to clear wasn't set by your job; a one returned indicates that you sucessfully cleared one lock; a number greater than one indicates that you cleared more than one lock (which means that LOCK1 or LOCK2 were originally set to zero--the wildcard value). You may never use XLOCK to clear a lock that was not set by your job. (NOTE: If you attach your terminal to another job, XLOCK considers you a new job.)

5.4.4 MODE 3 (List Locks)

MODE 3 returns a complete list of all the locks set on the system and the numbers of the jobs that set them. When you use MODE 3, LOCK2 must represent a mapped array large enough to hold the expected data. When XLOCK returns from a MODE 3 call, MODE contains the number of locks that are set on the system, LOCK1 contains your job number, LOCK2 contains one three-word entry for each lock that is set on the system. (You must set up this entry as three binary words in a MAP statement.) The first two bytes hold the job number; the second and third words hold the actual LOCK1 and LOCK2 values of the specified lock. The following is an example of how to set up the MAP statement for a MODE 3 call:

```
10 MAP1  MODE, B, 2
20 MAP1  MYJOB, B, 2
30 MAP1  LISTARRAY
40   MAP2  LOCKENTRY(25)
50     MAP3  JOBNUMBER, B, 2
60     MAP3  L1, B, 2
70     MAP3  L2, B, 2
80      ! Start of Program goes here
100 MODE = 3
110 XCALL XLOCK, MODE, MYJOB, LISTARRAY
120     ! Rest of program goes here
```

5.5 WILDCARDS

A system lock consists of two numbers, the values of LOCK1 and LOCK2. If either of these two numbers is a zero, that number is a wildcard and any number between 1 and 65535 will match it. (A wildcard is a symbol that is matched by any other symbol.)

You can use wildcards for various reasons. For example, suppose that you decide that the LOCK1 value is going to represent a particular file and that the LOCK2 value will represent a particular record in that file. If you want to stop all references to that file while your program is accessing it, you would set the lock with a zero in LOCK2 and the number representing your file in LOCK1. Anyone who tries to set a lock that has the same LOCK1 value as your lock won't be able to do so; the system will tell him that that lock already exists (since your wildcard in LOCK2 will match any number he may try in that position). No one (including yourself) will be able to set a lock with the same LOCK1 value until you clear the lock. Note that setting a lock with both numbers zero will prevent anyone from setting a lock, since the system will say that all possible locks are already set.

5.6 PROGRAMMING EXAMPLES

The following is a small sample demonstration program that you may want to use to experiment with XLOCK, and to get a feeling for how it works. It asks you for the values of MODE, LOCK1, and LOCK2, and then reports back on the results of the locking operation you asked for. Remember: MODE = 0 sets a lock, MODE = 1 sets the lock after waiting for it to be cleared; MODE = 2 clears the lock, and MODE = 3 displays the locks set.

```
5  !  Sample Program to Illustrate File Locking
10  MAP1  FLAG,F
15  MAP1  COUNTER, F
20  MAP1  MODE, B, 2
25  MAP1  LOCK1, B, 2
30  MAP1  LOCK2, B, 2
35  MAP1  LOCKARRAY
40   MAP2  LOCKENTRY(25)
45     MAP3  JOB, B, 2
50     MAP3  L1, B, 2
55     MAP3  L2, B, 2
60  START:
65      INPUT "MODE, LOCK1, LOCK2: ",MODE,LOCK1,LOCK2
70      FLAG = MODE
75      IF MODE = 3 GOTO DISPLAY
80      XCALL XLOCK, MODE, LOCK1, LOCK2
85      PRINT "Mode = ";MODE :
90      IF FLAG = 0 AND MODE <> 0 PRINT "Lock already set."
95      IF FLAG = 2 AND MODE = 0 PRINT "You didn't set that lock."
100     IF FLAG = 2 AND MODE = 1 PRINT "You cleared the lock."
105     IF FLAG = 2 AND MODE > 1 PRINT "You cleared more than one lock."
110     GOTO START
115 DISPLAY:
120     XCALL XLOCK, MODE, LOCK1, LOCKARRAY
125     PRINT "Your job number is: ";LOCK1
130     PRINT "Current locks in use = ";MODE
135     IF MODE = 0 GOTO LOOP
140     FOR COUNTER = 1 TO MODE
145       PRINT SPACE(5);
150       PRINT STR(L1(COUNTER))+","+STR(L2(COUNTER));
155       PRINT SPACE(4) : PRINT "(Job";JOB(COUNTER);")"
160     NEXT
165 LOOP:
170     PRINT : GOTO START
```

XLOCK is often used to lock individual records within a file so that more than one user can update that file at the same time. LOCK1 might contain a number that represents the particular file you want to open for multi-user updating (perhaps by containing the file's file-channel number). LOCK2 might hold a number that represents the specific record within the file that you want to update.

5.6.1 Calculating Record Numbers

We assume that you will usually be using XLOCK to control multi-user updating of random files. (For information on random files, see Chapter 15 of the AlphaBASIC User's Manual, DWM-00100-01.) If you are going to be locking a specific file record, you need to understand the relationship between disk blocks and file records. A record (sometimes called a "logical record") is a grouping of data that you define; you also define the length of that record. Just as an example, let's define a file record that contains 6 bytes for a customer ID number, 24 bytes for a customer name, 10 bytes for the name of the customer's sales contact, and 10 bytes for the customer phone number. This file record would then contain 50 bytes. A disk block is a physical grouping of data on the disk that is always 512 bytes long. The monitor always transfers disk information in this 512-byte block. AlphaBASIC unblocks a disk block into smaller groups-- your logical records. For example, one disk block (512 bytes) would contain 10 of the logical records we defined above (50 * 10 = 500) with 12 bytes left over. No logical record is ever larger than a disk block. NOTE: You specify the size of your logical record in the OPEN statement for the file.

The reason for our explanation above is this: if you want the LOCK2 value to contain the number of the record you are updating, it must contain the relative number of the disk block being used, and not the logical record number. When AlphaBASIC unblocks a disk block into logical records, it brings the entire disk block into your memory partition. Even if you are only updating one logical record in that disk block, the entire disk block remains in your memory area until you either close the file or read a logical record that is in a different disk block. What this means is that more than one user could try to write out the same disk block at the same time even though they are updating different logical records. So, you must prevent access, not only to the logical record that you are updating, but to the entire disk block that contains it.

You must calculate the relative disk block number yourself by dividing the logical record number by the blocking factor. (The blocking factor is the number of logical records that can fit in one disk block.) In the example above where we had logical records 50 bytes long, the blocking factor is 10. Remember that each disk block is 512 bytes long and will be blocked to contain as many logical records as will fit.

If one of your lock digits is the disk block number, you can prevent access to the entire disk block; no one can access any of the logical records in the disk block until you clear the lock.

REMEMBER: The lock wildcard symbol is a zero, so calculate your disk blocks beginning with one instead of zero. Before you unlock the lock on a disk block, force the system to write that record by reading a logical record that falls outside of that disk block. (NOTE: You may also use the RANDOM'FORCED mode in your OPEN statement to force AlphaBASIC to perform a disk read or a disk write every time you access the file. See Chapter 15 of the AlphaBASIC User's Manual for more information.) The sample program below may help to clarify the last few paragraphs.

5.6.2 Sample Program to Illustrate File Record Locking

```
10      ! Sample Program to Illustrate File Record Locking
15      ! Remember to load XLOCK.SBR before running!
20  MAP1  MODE, B, 2                         ! Define locking variables
25  MAP1  LOCK1, B, 2
30  MAP1  LOCK2, B, 2
35  MAP1  LOGICAL'RECORD                     ! Define logical record
40    MAP2  CUST'ID,F,6                      ! contents-- 50 bytes
45    MAP2  CUSTOMER,S,24                    ! of customer info.
50    MAP2  CONTACT,S,10                     ! Customer ID is actually
55    MAP2  PHONE,S,10                       ! logical record number.
60  MAP1 RECORD'SIZE,F,,50
65      ! Scratch variables:
70  MAP1  RECORDNUM,F                        ! Logical record number
75  MAP1  FLAG,F
80  MAP1  QUERY,S,1
85      ! Begin program:
100 START:
105     LOOKUP "CUSTID.DAT",FLAG             ! If file doesn't exist,
110     IF FLAG = 0 THEN GOTO FILE'ERR       !   report error and exit.
115     OPEN #100, "CUSTID.DAT",RANDOM,RECORD'SIZE,RECORDNUM
120     LOCK1 = 100                          ! "100" represents CUSTID file
125     PRINT "Welcome to the Customer Maintenance Program."
130 LOOK:
135     INPUT "Please enter customer identification number: ",RECORDNUM
140        ! Note: Customer ID is just number of that logical record.
145        ! Calculate relative disk block number (assumes logical
150        ! records begin with zero):
155     LOCK2 = INT(RECORDNUM/10)+1
160        ! Lock the disk block used by the record.
165     XCALL XLOCK,MODE,LOCK1,LOCK2
170     READ #100, LOGICAL'RECORD
175     PRINT "Customer information:"
180     PRINT TAB(5);"Customer ID#: ";CUST'ID
185     PRINT TAB(5);"Customer name: ";CUSTOMER
190     PRINT TAB(5);"Sales contact: ";CONTACT
195     PRINT TAB(5);"Phone #: ";PHONE
200 UPDATE:
205     INPUT "Do you wish to change any info? ";QUERY
210     IF UCS(QUERY) = "N" THEN GOTO LOOP
215     PRINT "Customer ID: ",CUST'ID
220     INPUT "Enter customer name: ";CUSTOMER
225     INPUT "Enter sales contact: ";CONTACT
230     INPUT "Enter phone number: ";PHONE
235     WRITE #100, LOGICAL'RECORD
240        ! Force BASIC to bring different disk block into memory.
245        ! (If we are in first disk block, since blocking factor is
250        ! 10, record number >= 10 will force in next disk block)
255     IF LOCK2 = 1 THEN RECORDNUM = 10 ELSE RECORDNUM = 0
260        ! Now bring in different disk block:
265     READ #100, LOGICAL'RECORD
```

```
270         ! Release the lock.
275     MODE = 2
280     XCALL XLOCK, MODE, LOCK1, LOCK2
285 LOOP:
290     INPUT "Do you wish to see info on another customer? ",QUERY
295     IF UCS(QUERY) = "Y" THEN GOTO LOOK
300 EXIT:
305     PRINT "Returning you to Command Level..."
310     CLOSE #100
315     END
320 FILE'ERR:   ! Oops. File didn't exist.
325     PRINT "File error.  Please see System Operator."
330     END
```

5.7 SUMMARY

XLOCK can both set and test system locks, to help users from conflicting in their attempts to use system resources. These locks are not for security; they are for the convenience of the users. A user may lock a file or a device to prevent any other user from accessing it, may test a lock to see if another user has already set a lock and is using the file or device, or may clear the lock so that the programs of other users may access the file or device.

Before running any program containing the XCALL XLOCK subroutine, you must include the XLOCK.SBR in system memory by using the SYSTEM command within the system initialization command file.

CHAPTER 6

SPOOL - XCALL SUBROUTINE FOR SPOOLING FILES TO THE LINE PRINTER

SPOOL is an XCALL subroutine that you can call from AlphaBASIC to spool a disk file to the line printer. ("SPOOL" is actually an acronym meaning "Simultaneous Printer Output On-line." To "spool" a file is to insert it into the printer queue, after which you can continue to do other things while your file waits in the queue for its turn to be printed.) You can specify to SPOOL which printer you want the file to be printed on, the number of copies to print, the form to print on, the width (measured in characters) of a page, and the lines per page. Also you can specify any combination of switches to turn on or off the banner option, the delete option (which deletes the file from the printer queue after printing), the header option, the formfeed option, or the wait option.

You do not have to load the SPOOL subroutine into system or user memory in order to access it from an AlphaBASIC program. However, if you have an AMOS system, rather than an AMOS/L system, and if you are going to use the SWITCHES feature of SPOOL, you must load FLTCNV.PRG into either user or system memory before you run an AlphaBASIC program containing the XCALL SPOOL program line.

To load FLTCNV.PRG into user memory, enter the following from AMOS command level:

```
.LOAD DSK0:FLTCNV.PRG[1,4] RET
.
```

To load FLTCNV.PRG into system memory of your AMOS system, you must have a line in your system initialization command file that performs that function. For more information on loading subroutines into system memory during system boot-up, see the AMOS System Operator's Guide, DSS-10001-00, or the AMOS/L System Operator's Guide, DSS-10002-00.

6.1 USING THE XCALL SPOOL SUBROUTINE

Call the SPOOL subroutine from within an AlphaBASIC program via:

```
XCALL SPOOL,FILE,PRINTER,SWITCHES,COPIES,FORM,WIDTH,LPP
```

where:

FILE — A string variable or expression that gives the specification of the file you want to print. If you specify a file which does not exist, SPOOL doesn't tell you that it can't find the file (but, of course, doesn't print anything).

PRINTER — A string variable or expression that gives the name of the printer you want to send the file to. If PRINTER is omitted or is a null string, SPOOL uses the default printer. If you want to use the default printer and also wish to use one or more of subsequent features (SWITCHES, COPIES, etc.), place a null string designation ("") in the PRINTER position of the program line (e.g., XCALL SPOOL,"DATA.TXT","",5).

SWITCHES — A floating point variable or expression that specifies various control switches and flags that affect the printing of the file. If you have a AMOS system (as opposed to an AMOS/L system), you must load FLTCNV.PRG into system or user memory if you are going to use the SWITCHES argument.

The switches that SPOOL uses are the same as the switches of the same names used by the monitor PRINT command. (See the AMOS System Commands Reference Manual, DWM-00100-49 or the AMOS/L System Commands Reference Manual, DSS-10004-00, for information on PRINT.)

The switches are:

1. BANNER - To print a banner (identifying) page at the front of the printout.
2. NOBANNER - So a banner will not be printed.
3. DELETE - To delete a file after it is printed.
4. NODELETE - So a file is not deleted after it is printed.
5. HEADER - To print a page header at the top of every page of the printout. Page headers give the name of the file being printed, the date,

and the current page number.

6. NOHEADER - So a page header is not printed on each page of the printout.

7. FF - To do a formfeed after a file is printed.

8. NOFF - Supresses a formfeed after a file is printed.

9. WAIT - To wait until previous entries into the printer queue are finished printing, so that the print request is not discarded if the printer queue is temporarily full. (If the file has to wait to be printed, the job running the AlphaBASIC program that performed the XCALL SPOOL subroutine waits too, and nothing else can be done until that request is inserted into the queue.)

Each switch you can use has a numeric code associated with it (see below). For example, the BANNER switch code is 1; the DELETE switch code is 4. Set control switches by putting the sum of the appropriate switch codes into the SWITCHES variable. For example, if you want to use the BANNER and DELETE switches (to tell the line printer spooler program to print a banner page and delete the file after printing it), load SWITCHES with 5 (BANNER code + DELETE code). If you omit SWITCHES, SPOOL uses the default switches for the selected printer. If you do not wish to use SWITCHES, but want to use one or more of the subsequent options (COPIES, FORM, etc.), replace the SWITCHES variable or expression with the null designation ("").

Switch codes:

BANNER	1
NOBANNER	2
DELETE	4
NODELETE	8
HEADER	16
NOHEADER	32
FF	64
NOFF	128
WAIT	256

COPIES — A floating point variable or expression that specifies the number of copies to be printed. If you omit COPIES or it is zero, the line printer spooler program prints one copy. If you want COPIES to print the default number of copies of the line printer spooler, and want to use subsequent options (FORM, WIDTH, etc.), enter the null designation ("") in place of the COPIES variable or expression.

FORM — A string variable or expression that specifies the form on which the file is to be printed. If you omit FORM or it is a null string, the line printer spooler uses the NORMAL form. If you want FORM to use the default form of the line printer spooler, and want to use subsequent options (WIDTH or LPP) enter the null designation ("") in place of the FORM variable or expression.

WIDTH — A floating point variable or expression that specifies the width (in characters) of the page. SPOOL only uses this value if you have specified the HEADER switch in the SWITCHES variable. WIDTH does <u>not</u> affect the number of characters in the print line; it only affects the text in the banner (if any) and the header, based on the width you specify. If you omit WIDTH, the spooler program uses the default value for the specified printer. If you want to omit WIDTH, but want to use LPP, the subsequent option, enter the null designation ("") in place of the WIDTH variable or expression.

LPP — A floating point variable or expression that specifies the number of lines per page. SPOOL only uses this value if you have specified the HEADER switch in the SWITCHES variable. If you omit LPP, the spooler program uses the default value for the specified printer.

6.1.1 Some Examples using SPOOL

The following examples are intended to be various modifications of the same one- or two-line programs. Each modification will affect the printing of a file in a different way.

6.1.1.1 <u>XCALL SPOOL,"FILENAME"</u>

As with all of the XCALL subroutines callable from AlphaBASIC, the SPOOL subroutine must be indicated by the XCALL keyword followed by the name of the subroutine, SPOOL. The keyword and the subroutine name, a comma, and the filename (as either a string variable or expression) to be spooled are mandatory:

```
10 XCALL SPOOL,"TEXT.LST"
```

where "TEXT.LST" is regarded as an expression by the AlphaBASIC program, and TEXT.LST is the file you want printed. (Note that the expression is enclosed in quotation marks.) This next program accomplishes the same thing because SPOOL accepts a string variable designation:

```
5  MAP1 FILENAME,S,26
10 FILENAME="TEXT.LST"
20 XCALL SPOOL,FILENAME
```

Notice in both of the above examples that no options have been specified. All the parameters are set by default.

6.1.1.2 XCALL SPOOL,"FILENAME","PRINTER"

Modifying the above examples, the XCALL SPOOL command line may specify a printer via a string variable or an expression:

```
5  MAP1 PRINTER,S,6
10 PRINTER="TI810"
20 XCALL SPOOL,"TEXT.LST",PRINTER
```

or:

```
10 XCALL SPOOL,"TEXT.LST","TI810"
```

where TI810 is the name of a printer defined by the monitor TRMDEF command. Note that the string expression TI810 must always be enclosed in quotation marks.

6.1.1.3 XCALL SPOOL,"FILENAME","PRINTER",SWITCHES

Now we'll add the SWITCHES option to our examples.

If you have an AMOS system, then before you can run an AlphaBASIC program using the XCALL SPOOL subroutine and the SWITCHES option, you must load FLTCNV.PRG into system or user memory.

The nine available switches each have a unique numeric code assigned to them. Add the numeric value of the various codes that you want to use. For example, say we wish to have a BANNER and a HEADER, and throw a formfeed when our file is done printing. Those codes, 1, 16 and 64, add up to 81. Our sample program's XCALL SPOOL command line should read:

```
10 XCALL SPOOL,"TEXT.LST","TI810",81
```

6.1.1.4 XCALL SPOOL,"FILENAME","PRINTER",SWITCHES,COPIES

Say we want to spool two copies to the printer queue. We would add the COPIES floating point variable or expression to the XCALL SPOOL line in a way something like this:

```
10 XCALL SPOOL,"TEXT.LST","","",2
```

or like this:

```
10 COPIES=2
20 XCALL SPOOL,"TEXT.LST","","",COPIES
```

NOTE: In the above examples, the PRINTER string variable or expression and the SWITCHES floating point variable or expression have been replaced by place-holding nulls (""). You must always remember to add a place-holding null in the XCALL SPOOL program line if you are not going to use the option that goes in that place but are going to use one or more subsequent options.

6.1.1.5 XCALL SPOOL,"FILENAME","PRINTER",SWITCHES,COPIES,"FORM"

The FORM option of the XCALL SPOOL command line may specify a form that you want mounted on the printer. The FORM may be either a string variable or an expression:

```
5  MAP1 FORM,S,6
10 FORM="PAYROL"
20 XCALL SPOOL,"FILENAME","PRINTER",SWITCHES,COPIES,FORM
```

or:

```
10 XCALL SPOOL,"FILENAME","PRINTER",SWITCHES,COPIES,"PAYROL"
```

where PAYROL is the name of a form defined by the monitor TRMDEF command. Note that a string expression identifying the form to use must always be enclosed in quotation marks.

When SPOOL sends a file to the printer queue, if the FORM option is selected and the form specified is different than the one mounted on the printer, the file will not print. Instead, the file will simply remain in the queue until the monitor SET command is used and the form is changed to match the one used in the XCALL SPOOL program line. See the SET reference sheet in the AMOS System Commands Reference Manual, DWM-00100-49, or the AMOS/L System Commands Reference Manual, DSS-10004-00, for more information on setting the form for the printer to use.

6.1.1.6 XCALL SPOOL,"FILENAME","PRINTER",SWITCHES,COPIES,"FORM",WIDTH,LPP

Finally, the XCALL SPOOL subroutine can use a floating point variable or expression to set the width (measured in characters) of the page. SPOOL only uses this value if you have specified the HEADER switch in the SWITCHES variable. WIDTH affects the appearance of the banner (which is only printed when using the BANNER switch of SWITCHES) and the header text; it does not affect the number of characters in the print line.

When a file is spooled to the printer, WIDTH determines how wide the banner is to be by controlling the number of characters that form the banner alphanumerics. At the top of each page, SPOOL places the header text. Part of the header text is a page number, which is oriented near the right-hand margin. That right-hand margin is determined by WIDTH.

The actual lines that are printed are not controlled by WIDTH. In other words, print lines whose lengths have previously been established are not changed via the WIDTH value.

As an example of WIDTH, to print a file with a banner that fits on an 8 1/2" X 11" page, and a header with the page number appearing toward the right of the page, you can set WIDTH to 70. Your XCALL SPOOL program line should appear something like this:

```
10 XCALL SPOOL,"TEXT.LST","TI810",17,2,"NORMAL",70
```

or WIDTH can appear as a floating point variable, like this:

```
10 CHAR'PER'LINE=70
20 XCALL SPOOL,"TEXT.LST","TI810",81,2,"NORMAL",48,CHAR'PER'LINE
```

In either case, WIDTH will not force the file you print out to start printing a new line at 70 characters.

6.1.1.7 XCALL SPOOL,"FILENAME","PRINTER",SWITCHES,COPIES,"FORM",WIDTH,LPP

To use the LPP feature of SPOOL, the HEADER switch of the SWITCHES feature must also be used. The floating point variable or expression included on the XCALL SPOOL line specifies the number of lines per page. When a full page (according to the LPP specification) is printed, the SPOOL subroutine prints a form feed and then prints the header at the top of the following page. To allow 48 lines on a page (counting the header), for example, LPP should appear something like this in the program line:

```
10 XCALL SPOOL,"TEXT.LST","TI810",81,2,"NORMAL",70,48
```

or LPP can appear as a floating point variable, like this:

```
10 LINES'PER'PAGE=48
20 XCALL SPOOL,"TEXT.LST","TI810",81,2,"NORMAL",70,LINES'PER'PAGE
```

Remember, if LPP is the only option you care to use, you must have all previous placeholders in place:

```
10 XCALL SPOOL,"TEXT.LST","","","","","",48
```

6.2 SPOOL ERROR MESSAGE

The SPOOL subroutine returns only one error message:

?No spooler allocated

If you see the message above, it means that no line printer spooler program is currently running on the system.

A note of caution: Each use of SPOOL in your AlphaBASIC program places the filename specified in the XCALL SPOOL program line into the monitor queue. The system is protected so that a certain number of monitor queue blocks (currently 15) are left unoccupied by SPOOL (or by the monitor command PRINT). However, if the total of monitor queue blocks being occupied at a given moment by all the jobs running on the system (including your AlphaBASIC program using SPOOL) exceed the total allocated, the system will lock up and require a manual reset. No error message will be generated.

6.3 SUMMARY

SPOOL inserts a file into your system's printer queue, after which your AlphaBASIC program can continue to do other things. The file spooled into the queue waits its turn to be printed.

SPOOL has a number of options that are very similar in both function and use to the options available using the PRINT command from AMOS or AMOS/L command level. The options each have specific positions on the XCALL SPOOL program line. If an option is not desired, but a subsequent option is, the preceding option must be replaced by a placeholding null string enclosed in quotes ("").

For AMOS systems (but not AMOS/L systems), one option, the SWITCHES command, requires that FLTCNV.PRG be in system or user memory.

Subsequent options (those whose positions on the XCALL SPOOL program line are to the right of the SWITCHES option) are available even if the SWITCHES option is not desired by placing a null argument ("") in the SWITCHES position. However, even if SWITCHES is null, FLTCNV.PRG must be loaded in system or user memory if its position on the program line is used.

CHAPTER 7

XMOUNT - XCALL SUBROUTINE TO MOUNT A DISK

XMOUNT is an XCALL subroutine that allows you to mount a disk from within an AlphaBASIC program without leaving AlphaBASIC. You should call it whenever you change a disk and your AlphaBASIC program is going to sort files or create new files on the newly changed disk. (You must always mount a disk after you've changed it and before you write to it; otherwise the system will think that the old disk is still in the drive. When it comes time to write information out to the new disk, the disk's bitmap will be wrong, and the system will try to write to the new disk as if it had the same areas free as the old one.) Besides bringing into memory the proper bitmap, XMOUNT also loads in the alternate track table, if any, for the specified device.

IMPORTANT NOTE: NEVER mount or unmount a disk while someone is accessing that disk. Doing so may corrupt the data on the disk.

It is not necessary to load the XMOUNT subroutine into system or user memory. However, the XMOUNT subroutine is fully re-entrant, so for increased access speed you may load it into system memory via the SYSTEM command in your system initialization command file. (See the <u>AMOS System Operator's Guide</u>, DSS-10001-00, or the <u>AMOS/L System Operator's Guide, DSS-10002-00</u> for information on the system initialization command file.)

7.1 THE XMOUNT SUBROUTINE

You can call XMOUNT to mount a disk via:

```
XCALL XMOUNT,DEVICE,VOLUME'ID
```

Where:

DEVICE — String variable or expression that represents a device specification (e.g., "DSK1:"). You may optionally follow the device specification with "/U" to unmount the device (e.g., "DSK0:/U").

VOLUME'ID String variable in which the volume ID of the mounted device will be returned. This variable must be 10 bytes long. If it is not specified the labels block will not be read. This variable is ignored if the /U option is used.

If you specify the unmount option, the "U" must be uppercase. When you unmount a disk, you prevent AlphaBASIC and most system programs from accessing that device.

7.1.1 Some Examples Using XMOUNT

As with all the XCALL subroutines callable from AlphaBASIC, the program line must begin with the keyword XCALL and the name of the subroutine, XMOUNT. The XMOUNT subroutine further requires a string variable or expression to represent the specification of the device to be mounted (or unmounted), which is separated by a comma from the word XMOUNT. For example:

```
10 XCALL XMOUNT,"HWK1:"
```

or

```
5  MAP1 DEVICE,S,9
10 DEVICE="HWK1:"
20 XCALL XMOUNT,DEVICE
```

You may similarly unmount a disk by making the /U switch part of the same expression or string variable:

```
10 XCALL XMOUNT,"HWK1:/U"
```

or

```
5  MAP1 DEVICE,S,9
10 DEVICE="HWK1:/U"
20 XCALL XMOUNT,DEVICE
```

The only option available when using XMOUNT (other than the /U switch to unmount a disk) is the ability to store the volume ID of the newly mounted disk within a string variable, perhaps to be displayed immediately after using the XMOUNT subroutine so the program user is sure he or she put the right disk in the drive.

XMOUNT recognizes this option when it sees a string variable following the device specification string or expression (and separated from it by a comma). XMOUNT returns the volume ID of the disk as that variable, which then may be displayed or tested. For example:

```
5  MAP1 VOLUME'ID,S,10
10 MOUNTING: XCALL XMOUNT,"HWK1:",VOLUME'ID
20 PRINT VOLUME'ID;" is mounted."
30 IF VOLUME'ID<>"ARCHIVE" THEN GOTO WRONG'DISK
40 GOTO CONTINUING

100 WRONG'DISK: PRINT "This is not the ARCHIVE disk."
110 PRINT "You may abort the program or place the correct"
120 PRINT "disk in the drive.  To abort type Control-C." : STOP
130 GOTO MOUNTING

200 CONTINUING: ...
```

If the volume ID string variable is omitted or is too small, or if a /U follows the device specification string variable or expression, the volume ID variable is ignored and returns a null string.

7.2 SUMMARY

The XMOUNT subroutine provides you with the ability to mount a disk without leaving an AlphaBASIC program. It is used when a new disk has been inserted in a disk drive and must be mounted in order for the bitmap to be updated. XMOUNT may also be used to unmount a disk from within an AlphaBASIC program. XMOUNT also provides the volume ID of the disk as an option, if the program user needs to identify the disk just mounted.

DOCUMENT HISTORY

Revision A00 - AMOS Release 4.6 and AMOS/L Release 1.0 - (Printed 6/82)

The information included in this manual was formerly contained as separate documents in the "BASIC Programmer's Information" section of the AMOS Software Update Documentation Packet. The contents of this manual are updated to reflect advancements in software and the inclusion of AMOS/L system information. Also, the information in this manual has been expanded and clarified in response to user requests.

Index

TECHNICAL PUBLICATIONS READERS COMMENTS

We appreciate your help in evaluating our documentation efforts. Please feel free to attach additional comments.
If you require a written response, check here: ☐

NOTE: This form is for comments on documentation only. To submit reports on software problems, use Software Performance Reports (SPRs), available from Alpha Micro.

Please comment on the usefulness, organization, and clarity of this manual:

Did you find errors in this manual? If so, please specify the error and the number of the page on which it occurred.

What kinds of manuals would you like to see in the future?

Please indicate the type of reader that you represent (check all that apply):

- ☐ Alpha Micro Dealer or OEM
- ☐ Non-programmer, using Alpha Micro computer for:
  - ☐ Business applications
  - ☐ Education applications
  - ☐ Scientific applications
  - ☐ Other (please specify):
- ☐ Programmer:
  - ☐ Assembly language
  - ☐ Higher-level language
  - ☐ Experienced programmer
  - ☐ Little programming experience
  - ☐ Student
  - ☐ Other (please specify):

NAME: ______ DATE: ______

TITLE: ______ PHONE NUMBER: ______

ORGANIZATION: ______

ADDRESS: ______

CITY: ______ STATE: ______ ZIP OR COUNTRY: ______

STAPLE

STAPLE

FOLD

FOLD

PLACE
STAMP
HERE

alpha micro

17881 Sky Park North
P.O. Box 18347
Irvine, California 92714

ATTN: TECHNICAL PUBLICATIONS

FOLD

FOLD

CUT ALONG LINE