ALPHA MICROSYSTEMS
RIGHT. FROM THE START.

# AMOS® File Locking User's Manual

DSM-00044-04

| REVISIONS INCORPORATED | |
|---|---|
| REVISION | DATE |
| 00 | March 1988 |
| 01 | September 1989 |
| 02 | April 1991 |
| 03 | June 1996 |
| 04 | September 1996 |

*AMOS File Locking User's Manual*
To re-order this document, request part number DSO-00044-00.

This document applies to AMOS version 2.3 and later.

| | | | |
|---|---|---|---|
| AMIGOS | AMOS | Alpha Micro | AlphaACCOUNTING |
| AlphaBASIC | AlphaCALC | AlphaCOBOL | AlphaDDE |
| AlphaFORTRAN 77 | AlphaLAN | AlphaLEDGER | AlphaMAIL |
| AlphaMATE | AlphaNET | AlphaPASCAL | AlphaRJE |
| AlphaWRITE | CASELODE | OmniBASIC | VER-A-TEL |
| VIDEOTRAX | | | |

| | | | |
|---|---|---|---|
| AlphaBASIC PLUS | AlphaVUE | AM-PC | AMTEC |
| AlphaDDE | AlphaConnect | DART | *in*Sight/am |
| *in*Front/am | ESP | MULTI | |

ALPHA MICROSYSTEMS
2722 S. Fairview St.
P.O. Box 25059
Santa Ana, CA 92799

# TABLE OF CONTENTS

# CHAPTER 1

# INTRODUCTION

On AMOS 2.0 and later systems, file protection is automatic. This manual describes this file protection, and how to use it in your application programs to achieve true file security on your computer. Protecting the integrity of your data is perhaps the highest priority of any file protection system. On multi-user systems, files are vulnerable to random modification and deletion unless some mechanism exists to insure the orderly access and processing of their contents.

AMOS File Locking provides system level file security and works with very little direction from application programs. It supervises the accessing of all files automatically, whether your programs contain any of the special File Locking commands described in this manual or not. For compatibility purposes, Appendices B and C describe the voluntary file locking routines XLOCK and FLOCK.

## 1.1 THE BENEFITS OF AMOS FILE LOCKING

With AMOS File Locking, you can customize a system level file security structure to the specific requirements of your installation. AMOS lets you specify special protection for your most frequently used files, and provides default security measures for other files on your computer. On AMOS 2.1 and later systems and using AlphaNET 2.1 and later, File Protection works over networks also. Some of the notable file locking features:

- File locking for application programs.
- File locking at the system level.
- Files may be locked for shared or exclusive access.
- Locking of individual records within random files.
- Locking of individual streams within USAM files.
- Checking for possible deadlocks.
- File lock and access monitoring.

- Easily maintained supervisor initialization file.
- Problem correction outside the application program.
- File locking can be disabled for individual jobs, files, or combinations of jobs and files by use of the SET command with the LOCK and NOLOCK options. This feature only works on traditional, non-extended devices.

## 1.2 GRAPHIC CONVENTIONS

This manual conforms to the other Alpha Micro publications in its use of a standard set of graphics conventions. We hope these graphics simplify our examples and make them easier for you to use. Unless stated otherwise, all examples of commands are assumed to be entered at AMOS command level.

| SYMBOL | MEANING |
|---|---|
| devn: | Device-Name. The "dev" is the three letter physical device code, and the "n" is the logical unit number. Examples of device names are DSK0:, WIN1:, and MTU0:. Usually, they indicate disks, but they can also be magnetic tape drives or video cassette recorders. |
| filespec | A file specification that identifies a specific file within an account. Here is the format and an example:<br><br>devn:filename.ext[p,pn]<br>DSK0:SYSTEM.INI[1,4] |
| **TEXT** | Bold text in an example of user/computer communication represents the characters you type. |
| `TEXT` | Text like this in an example of user/computer communication represents characters the computer displays on your terminal screen. |
| [p,pn] | This abbreviation represents an account on a disk where you can store files and data. An actual disk account number looks like this: [100,2] or [1,4]. Disk account specifications are sometimes referred to as "Project-programmer numbers." |
| { } | Braces are used in some examples to indicate optional elements of a command line. For example:<br><br>`DIR{/switch}` |

| SYMBOL | MEANING |
|---|---|
| / | The slash symbol precedes a command line switch or "option request." For example:<br><br>**`DIR/WIDE:3`**$\boxed{\text{RETURN}}$<br><br>This command requests a directory display of the disk account you are currently logged into. The switch (/WIDE:3) indicates you want the display to be three columns wide. |
| * | The LOKUTL program uses the asterisk as a prompt symbol to indicate it’s waiting for further instructions. |
| $\boxed{\text{KEY}}$ | In our examples, the key symbol appears whenever you need to press a certain key on your terminal keyboard. The name of the key you need to press appears inside the key symbol, like this: $\boxed{\text{RETURN}}$. |
| $\boxed{\text{CTRL}}$/$\boxed{\text{KEY}}$ | This indicates a control sequence you press on the keyboard. Press $\boxed{\text{CTRL}}$and hold it down while pressing the indicated key. |
| ^ | This symbol in front of a capital letter means the letter is a "control character." For example, when you press $\boxed{\text{CTRL}}$/$\boxed{\text{C}}$, it appears on your screen as `^C`. |
| ✋ | This symbol means "halt!" It indicates an important note you should read carefully before going further in the documentation. Usually, text next to this symbol contains instructions for something you MUST or MUST NOT do, so read it carefully. |
| 🤫 | This symbol means "hint." It indicates a helpful bit of information, or a "short cut" that could save you time or trouble. |
| ☝ | This symbol means "remember." It indicates something you should keep in mind while you are following a set of instructions. |

## 1.3∞READER’S COMMENT FORM

Please note the Reader’s Comment Form at the back of this manual. Your suggestions are important to us, and we’ll use them to improve later versions of this manual.

# CHAPTER 2

# WHAT IS FILE PROTECTION?

In many instances, it’s necessary for more than one user to retrieve data from the same file—often from the same record. However, when more than one user tries to use the same data resources at the same time, problems arise.

## 2.1 FILE PROTECTION PROBLEMS

The two major problems associated with multi-user systems are "Simultaneous Update" and "Deadlock."

### 2.1.1 Simultaneous Update

When two or more users try to modify the same record of a file at the same time, a unique situation arises. Here’s what can happen:

1. The first user reads the record and makes changes.
2. The second user also reads the record and makes different changes.
3. The first user rewrites the updated record back to the file.
4. The second user rewrites the same record back to the file which overlays the first user’s changes.

This series of events creates a problem because neither user knew of the other’s changes; and as a result, one set of modifications is lost. A solution to this problem is to place a temporary lock on the record or the entire file while one user is working with the data. The second user would either be informed the record was unavailable or would be placed in a wait state until the first user had rewritten or unlocked it. When user one was finished, then user two could go ahead and read the same record, incorporate the additional changes, and finally rewrite the record containing both sets of changes.

This solution solves the simultaneous update problem, but can lead to the second, more severe problem—Deadlock.

### 2.1.2 Deadlock

Deadlock occurs when two users require the same **set** of data resources. As your applications proliferate, it's not unlikely that two users might each execute programs that need to access the same files. For example, if one program places a lock on file A and tries to access file B, while at the same time, a second program places a lock on file B and tries to access file A, neither program can access the file locked by the other. They are deadlocked.

AMOS can prevent this by requiring file A always be locked before file B. Then the first program would proceed normally, while the second program either sees an error message and cancels, or waits until the first program releases file A before proceeding.

## 2.2 AMOS FILE ACCESS CO-ORDINATION

AMOS provides automatic file access coordination for all files on the system. In addition, the System Operator can force programs and users to open certain files in a certain order to prevent deadlock.

AMOS creates and maintains an internal data base containing a list of the files, records, and USAM (Unix Style Access Method) streams locked by each user on the system. AMOS checks any request to lock a file, record, or stream against its data base before taking any action. This prevents a simultaneous update situation. In addition, AMOS optionally maintains a list of files to check for deadlock, and forces all programs to lock these files in a specific order.

### 2.2.1 Locking Files and Records

AMOS lets you lock entire files or individual records within files. The locks are of two types: shared and exclusive. The type of lock AMOS places on a resource is automatically determined by two things: the file type, and the access mode of the file or record. The tables below clarify the relationships among shared and exclusive file and record locks in AlphaBASIC and AlphaBASIC PLUS:

**SEQUENTIAL FILE LOCKS**

| Access Mode | File Lock Default | Record Lock Possible? |
|---|---|---|
| INPUT | Shared | n/a |
| OUTPUT | Exclusive | n/a |
| APPEND | Exclusive | n/a |

**RANDOM FILE LOCKS**

| Access Mode | File Lock Default | Record Lock Possible? |
|---|---|---|
| RANDOM | Exclusive | n/a |
| RANDOM’FORCED | Shared | yes |
| INDEXED | Shared | yes |
| INDEXED’EXCLUSIVE | Exclusive | n/a |

**USAM FILE AND STREAM LOCKS**

| Access Mode | File Lock Default | Stream Lock Possible? |
|---|---|---|
| n/a | Shared | yes |

where:

| | |
|---|---|
| **Shared** | Shared locks permit several users to access the same file or record at the same time. |
| **Exclusive** | Exclusive locks let one user have sole access to a specific file or record. USAM streams can be locked and unlocked by means of the U.LOK and U.ULK calls described in the *USAM User’s Guide*. Stream locks are always exclusive. |

### 2.2.4∞When a Conflict Arises

AMOS also allows you to determine what happens when you request to lock a file, record, or stream already locked by another user. By including or omitting the WAIT’FILE or WAIT’RECORD clauses in AlphaBASIC programs (or by setting or clearing the F.WAT flag in M68 programs), you can either have AMOS return a status code to your program, or have AMOS place your program in a SLEEP state until the first user is finished processing and has unlocked the needed file, record, or stream.

### 2.2.5∞The File Locking Initialization File

AMOS keeps track of the files to check for deadlock by means of an initialization file. The System Operator can modify the contents of this file at any time.

When listed in the File Locking initialization file, each file is checked for a possible deadlock situation. Files checked for deadlock must then be OPENed in the same sequence they appear in in the initialization file by each application program that accesses them. The next chapter describes how to do this.

# CHAPTER 3

# AMOS FILE DEADLOCK PROTECTION

To detect a possible "deadlock" situation, you need to add a LOKINI statement to your system initialization command file and create a list of all the permanent files on your system that you want to protect.

First, log into SYS: (DSK0:[1,4]) and use the AlphaVUE text editor to create a File Locking Initialization file. You can choose any unique file name you wish, but in the following examples, we've named ours LOK.INI:

```
LOG SYS:RETURN
VUE LOK.INIRETURN
```

In this file you can list all the permanent files on your system you want to protect from "deadlock." The format is this:

```
filespec filetype
```

where `filespec` is the full file specification of the file, and `filetype` is one of three letters:

| | |
|---|---|
| R | Random file |
| S | Sequential file |
| U | USAM file |

You may only specify files that exist on your own computer (no files that exist on another computer connected by a network).

The contents of your file might look something like this:

```
DSK0:PAYROL.DAT[200,0] R
DSK0:EMPLEE.DAT[201,0] S
DSK0:GNRLEG.DAT[202,0] R
```

Note the order of entries. Programs which access these files must now open them in the order listed. If a program opens the GNRLEG.DAT file and then tries to open the PAYROL.DAT file, AMOS displays an error.

When you have listed all the permanent files you want to protect, press ESC to return to AlphaVUE Command level, and type an "F" to write the file onto your disk.

## 3.1∞MODIFYING THE SYSTEM INITIALIZATION COMMAND FILE

Use the COPY command to make a copy of your standard system initialization command file with a name such as TEST.INI. **NEVER MODIFY THE FILE DIRECTLY!**

Use AlphaVUE to edit TEST.INI and add the following statement somewhere in the file before the first SYSTEM statement:

```
LOKINI LOK.INI
```

This statement causes AMOS to read the LOK.INI file you just created and store the filespecs listed there in the system queue blocks.

If your system initialization file contains a SYSTEM statement like this: `SYSTEM LOKSER.SYS/N`, delete it. This statement was used by an older version of the file locking system and is no longer needed.

One of the first statements in your initialization file should be the QUEUE statement which allows you to specify more queue blocks if you need them. AMOS automatically reserves 80 queue blocks, and the QUEUE statement lets you increase that number if necessary.

The AMOS File Locking system uses queue blocks to keep track of your permanent files, so the more permanent files you have, the more queue blocks you need to reserve. If error messages containing the phrase `"- insufficient queue blocks"` appear, increase the number of queue blocks. See the QUEUE reference sheet in your *System Commands Reference Manual* for more information. To reserve a hundred additional queue blocks, for example, the statement is:

```
QUEUE 100
```

After you have made these changes, press ESC to return to AlphaVUE command level and enter **F**RETURN to write the test file to your disk. Use the MONTST command to reboot your computer using the TEST.INI file.

When you are confident the TEST.INI file is working correctly, rename it to the name of your system initialization command file. AMOS now protects your permanent files from "deadlock."

# CHAPTER 4

# MONITORING FILE PROTECTION

LOKUTL is a dual-purpose program that shows you how AMOS is handling files for currently running jobs, and lets the System Operator perform alterations to current file and record locks.

## 4.1∞LOKUTL MODES

LOKUTL behaves in two ways depending on which disk account it's called from. If it's run from the System Operator's account, OPR:, it replies in Privileged Mode which allows the System Operator to monitor file protection and manipulate current file locks. If it's executed from any other account, it shows you the current status of all files. To run LOKUTL, enter: LOKUTL RETURN

### 4.1.1∞Normal Mode

When you execute LOKUTL from any account other than OPR:, it displays the status of the file protection on your terminal in the following format:

```
           File Locking Data Base

Job ID
     Filespec {Exclusive} {Writer} {Read-only} {Keep}
          {Record nn {Exclusive}}
{Job ID
     Filespec {Exclusive}}
          {Stream nn-mm Exclusive}
```

LOKUTL lists each job currently active. Under each Job ID, LOKUTL lists the specifications of the files the job is currently using. ***Exclusive*** means the file is locked exclusively, and ***Writer*** means the file is open output. ***Read'Only*** means the file was opened in read-only mode, and ***Keep*** means the file is closed, but a lock is maintained (another program in the process is about to open it again).

If it's a random file, LOKUTL lists the records locked by the job. If the file is locked exclusively, individual records don't need to be locked.

The last File Specification in the example above is for a USAM file. USAM (Unix Style

Access Method) files are similar to UNIX files and are accessed randomly. After ***Stream***, the first number represents the starting byte position, and the second number the ending byte position.

### 4.1.2∞Privileged Mode

LOKUTL's Privileged mode is reserved for use by the System Operator who must be logged into the System Operator's account OPR: to run it. This is what you see when LOKUTL is ready for processing:

```
LOKUTL Version x.x
*
```

where the asterisk is the LOKUTL prompt which means it is waiting for an instruction.

## 4.2∞LOKUTL PRIVILEGED MODE COMMANDS

LOKUTL provides a variety of commands to help you monitor, analyze, and modify system file locking. To see what they are, enter **H**RETURN at the prompt. The LOKUTL command menu then appears on your screen:

```
H = List LOKUTL commands
U = List data base sorted by User ID
F = List data base sorted by File ID
M = Monitor LOKSER operation
A = Abort a user from the data base
W = Wake up a user
C = Close a file for a user
R = Unlock a record for a user
S = Unlock a stream for a user
Q = Quit from LOKUTL
X = Enable diagnostic messages
Y = Disable diagnostic messages
O = Set Lock options
L = List Lock options
```

**U = List Data Base Sorted by User ID**

Displays the currently active jobs on your system and the files they are using.

**F = List Data Base Sorted by File ID**

Displays the same information as U, sorted in a different order. F shows you which jobs are using each file. The display format is:

```
                File Locking Data Base

   Filespec fileID  {Permanent} {Random}
     {Job ID {Exclusive} {Writer}} {Read-Only} {Keep} {Wait}
         {Record nn {Wait} {Exclusive}}
   {Filespec nnnnnn   {Permanent} {USAM}}
     {Job ID {Exclusive} {Keep} {Wait}}
         {Stream nn-mm {Wait} Exclusive}
```

***fileID*** is the file identification number LOKUTL gets from the OPENx call within the active program. The file specification line includes the parameters defined for the file. The Job ID line indicates whether the job has the file locked exclusively and whether the file was opened for output, read-only, or is closed but locked (***KEEP***). If the file is a random file, LOKUTL lists the individual records each job has locked.

If the file is a USAM file, the Stream designation appears instead of the Record number. After ***Stream***, the first number is the starting byte position, and the second is the ending byte position.

## M = Monitor File Locking Operation

Dynamically displays the file accesses in the normal format. The display is updated whenever AMOS services a file access request, or every 40 seconds, to reflect the current status of each job. At the bottom of your screen you see the word: ***Waiting....***

LOKUTL prints one dot after this word every second until it renews the display. When you're done, use CTRL/C to return to LOKUTL command level. This feature should only be used while debugging an application; it has a noticeable effect on file access speed. If someone else is already monitoring the system, LOKUTL tells you so and returns you to LOKUTL command level.

## A = Abort a User from the Data Base

Cancels the program being run by a particular job and releases all its locked files, records, and streams. LOKUTL asks you for the name of the job and processes it as if it EXITed. The job itself is not aborted, and the files are not CLOSEd as far as AMOS is concerned, but the files, records, and streams it may have had locked are released for use by other programs.

## W = Wake up a User

"Wakes up" a job who is waiting for a file or a record. This command is useful for freeing jobs trapped in a deadlock.

**C = Close a File for a User**

Unlocks a file and releases all its records or streams. LOKUTL asks you for the job name and file ID (see F, above). As in the A command above, the file is not CLOSEd as far as AMOS is concerned.

**R = Unlock a Record for a User**

Unlocks a specific record in a random file. LOKUTL asks you for the job name, file ID, record number, and whether the lock is exclusive.

**S = Unlock Stream for a User**

Unlocks a specific stream in a USAM file. LOKUTL asks you for the job name, the file ID, the starting byte position, the stream length, and whether it's locked exclusively or not.

**Q = Quit**

Returns you to AMOS command level.

**X = Enable Diagnostic Messages**

LOKUTL has two diagnostic messages:

```
%File not OPEN
%File already OPEN
```

Causes these to be displayed when a program executes a conflicting instruction. You probably want to use X only when debugging a program. The first message, `%File not OPEN`, appears when a program tries to access a file before it is OPENed. This situation might occur when a program searches for a file by means of a LOOKUP request and then "OPENs" it artificially by setting the open code in the file's DDB. Unless a file is opened with a true OPENx call, AMOS has no information on the file and cannot mediate access requests. The second message, `%File already OPEN`, is caused by trying to OPEN a file more than once in the same program.

**Y = Disable Diagnostic Messages**

Turns off the display of diagnostic messages.

**L = List Lock options.**

Lists jobs, filespecs, and job/filespec combinations that have locking turned off.

**O = Set Lock options.**

Turn locking ON or OFF for jobs, filespecs or job/filespec combinations. If event logging is active, it records the event in the system event log.

This chapter has discussed the controlling elements of AMOS file protection and the supporting utility programs. In the next chapter you will find out how to incorporate some explicit AMOS file protection instructions into your programs.

# CHAPTER 5

# FILE LOCKING IN ALPHABASIC PROGRAMS

There are times when you want to allow two or more programs to have access to the same file at the same time. The File Locking system is designed so your application programs can control file access while maintaining file security. This chapter discusses the special statements available in AlphaBASIC and AlphaBASIC PLUS.

## 5.1∞PROGRAMMING WITH AMOS FILE LOCKING

For AMOS to process file and record locks successfully, it needs to know how each file and record is going to be used as soon as possible. AMOS also needs to know when to unlock these resources so other programs can access them as soon as possible. For these reasons, several different statements are used to open, close, and read files and records. They give AMOS all the information it needs to process your request.

### 5.1.1∞Coding File Locking Instructions

Keep in mind as you're using these instructions in your programs that no one else can access an exclusively locked file or record until you release it. When you are finished using a file or record be sure to free it as soon as possible so other programs that might also need to use the same resource won't have to wait.

The following instructions are all you need to protect files from within your AlphaBASIC or AlphaBASIC PLUS programs. For complete information, see the *AlphaBASIC User's Manual*, or the *AlphaBASIC PLUS User's Manual*.

#### The OPEN Statement

Several parameters are available with this instruction to support AMOS file locking. The WAIT'FILE clause puts your program to sleep if you try to open a file while another program has a conflicting lock on it. As soon as the other program releases the file, your program awakens and continues processing.

If you have a random or ISAM PLUS file opened for shared access (RANDOM'-FORCED or INDEXED), the WAIT'RECORD clause will put your program to sleep if you try to READ or READL (GET or GET'LOCKED for ISAM) a record while another program has a lock on it. When the other program writes it, your program awakens and continues processing.

There is a READ'ONLY option that lets you read the data without placing any lock on the file. If you use the READ'ONLY option, another program needing exclusive access to the file could get it, locking your program out. This can be valuable so less-important functions don't lock out more vital ones.

AMOS assumes sequential opened for input are shared, and sequential files opened for output or append are exclusive. Random and ISAM PLUS files can be either shared or exclusive depending on their access mode. Record locks only apply to random and ISAM PLUS files that are shared. The table below shows the types of locks AMOS places on files and shows where record locks are kept.

| **File Type** | **Access Mode** | **Default** | **Lock Kept?** |
|---|---|---|---|
| Sequential | INPUT | Shared | --- |
| | OUTPUT | Exclusive | --- |
| | APPEND | Exclusive | --- |
| Random | RANDOM | Exclusive | --- |
| | RANDOM'FORCED | Shared | yes |
| | INDEXED | Shared | yes |
| | INDEXED'EXCLUSIVE | Exclusive | --- |
| USAM | n/a | Shared | yes |

## The GET Statement

GET accesses an ISAM PLUS record by key value. You may modify it with 'NEXT or 'PREV for the next or previous record, and by 'LOCKED to lock the record, or by 'READ'ONLY. For other ISAM PLUS commands, see your *ISAM PLUS User's Manual*.

### The READ Statement

Use READ to input a random file record you don't want to update.

### The READL Statement

This instruction is for use with files opened in RANDOM'FORCED or INDEXED mode, and is equivalent to the READ statement for files opened in RANDOM mode. READL lets you read a random file record you plan to update. This record is locked for your exclusive use until you use WRITE or UNLOKR.

If you continue to READL records without WRITEing or UNLOKRing them, you soon fill the queue and see the error: `?File Locking queue is full`. If this happens, the System Operator must use LOKUTL to free the locked records. To debug the program, the System Operator can use LOKUTL's M command to monitor the program as it processes records.

### The READ'READ'ONLY Statement

Allows you to read a record without regard to locking. You can read a record with this statement even if another user has it locked. This command works only in AlphaBASIC PLUS, not in AlphaBASIC.

### The UNLOKR Statement

Unlocks a record input and locked by READL.

### The WRITE Statement

Outputs and unlocks a record input by READL.

### The WRITEL Statement

Creates and writes a new record to a RANDOM'FORCED or INDEXED file.

## 5.2∞ERRORS

As your AlphaBASIC programs process files and records using AMOS instructions, you might see the following messages.

### Error 35 - Unsupported function

Your program has a statement AlphaBASIC doesn't understand. For example, a READL instruction for a sequential file.

**Error 37 - File in use**

The file is locked by another user. You may want to use WAIT'FILE.

**Error 38 - Record in use**

The record is locked by another user. You may want to use WAIT'RECORD.

**Error 39 - Deadly embrace possible**

Your program has opened files out of sequence with their order on the File Locking data base. Check the proper order and open the files correctly.

**Error 42 - Record not locked**

Your program tried to WRITE a random file record it had not first locked with a READL instruction.

**Error 44 - File Locking queue is full**

AMOS has more resources locked than it can keep track of. There are two possible causes for this error: A program did not unlock files or records when it is finished using them, or: The System Operator needs to increase the system queue blocks. The System Operator can diagnose the situation by using the LOKUTL M command to monitor AMOS File Locking operation.

# CHAPTER 6

# FILE PROTECTION IN ASSEMBLY LANGUAGE

When you use AMOS file protection monitor calls in your assembly language programs, they must use the following calling sequence:

```
     Opcode     DDB{,#flags}
```

where ***flags*** is an optional argument and has these values:

| | |
|---|---|
| F.EXC | Requests exclusive use |
| F.LOK | Write without unlocking |
| F.NEX | Bypass nolock options |
| F.RON | Read-only—reads even if locked |
| F.WAT | Program waits for access |

For example, to open a random file for exclusive use, your instruction might be:

```
     OPENR      DDB,#F.WAT!F.EXC
```

In this case, if the file is already locked, the program sleeps until the file is available. The *AMOS Monitor Calls* manual has complete information on using AMOS monitor calls.

## 6.1∞FILE MONITOR CALLS

If you make an I/O request (an OPEN, INPUT, OUTPUT, etc.) and LOKSER rejects the request because the file or record you want is locked, the JCB index of the job which has the lock is returned in D.ARG of the DDB.

### OPENx Calls

If you don't specify ***flags***, files opened by OPENR and OPENI default to shared. The exception is files residing on traditional devices when locking is off. Files opened by OPENO and OPENA are exclusive, regardless of ***flags***.

**CLOSE - Close a file**

Closes the file and release lock.

**CLOSEK - Close a file and maintain lock**

Closes file but maintains lock so another program in a sequence can re-open it.

**DSKREN and DSKDEL**

These calls use the optional ***flags*** argument, but F.EXC is always forced. The file is locked only while the operation is in progress, and as soon as control is returned to the user, the file is no longer locked.

## 6.2∞RECORD MONITOR CALLS

The following commands let you read and write records from an OPEN file. Except in the case of a random file OPENed for shared access, INPUT/INPUTL and OUTPUT/OUTPTL act the same. Record locks are only kept for a shared random file.

| | |
|---|---|
| GET | Reads in a record you do not plan to update. |
| GETL | Reads and locks a record for update. |
| GETX | Reads in a record for read-only. Data may not be correct. |
| INPUT | Inputs a block. |
| INPUTL | Inputs and locks a block. |
| INPUTX | Inputs a block for read-only. Data may not be correct. |
| OUTPUT | Updates and unlocks a block. |
| OUTPTL | Writes a new block. |
| UNLOKR | Unlocks a block. |

## 6.3∞AMOS ERROR CODES

In addition to error codes defined in the *AMOS Monitor Calls* manual, several are designed specifically for file protection. AMOS returns these codes to your assembly language program in the error byte of the file's DDB.

**Error 34 - File in Use**

You may want to use F.WAT.

**Error 35 - Record in Use**

You may want to use F.WAT.

**Error 36 - Deadly Embrace Possible**

Check your LOK.INI file and open the files in the correct order.

**Error 41 - Record Not Locked**

Use INPUTL to read it or OUTPTL to write it.

**Error 43 - LOKSER Queue is Full**

Increase the system queue blocks or check the queue with LOKUTL.

# APPENDIX A

# CONVERTING EXISTING ALPHABASIC PROGRAMS

Although you can still run programs that use XLOCK and FLOCK on your system, you must convert these programs to use the correct parameters described earlier in this manual. Converting an existing program to use AMOS file locking is a three step process:

1. Make sure the file open statements use the correct mode for shared or exclusive file access.

2. Remove all FLOCK or XLOCK calls.

3. Make sure the program uses the correct READ/READL and WRITE/WRITEL instructions in the appropriate places. For example, if the program runs in a file sharing environment and uses the same WRITE statement to update existing records and create new ones, you must now differentiate between the two by using WRITE to update the existing records and WRITEL to create the new ones.

If you overlook some changes, you see error messages when you try to run the program. There are also other considerations you must take into account when you convert a program. The following sections give specific instructions for adapting your programs to use file locking.

## A.1 FROM XLOCK TO AMOS FILE LOCKING

When you convert a program that uses XLOCK, there are two major considerations: First, if an XLOCK call uses mode 1 which attempts to lock a file or record, or wait until it becomes available, you should add the WAIT'FILE and/or WAIT'RECORD clause to the OPEN statement.

Second, if the program does any checking or processing of XLOCK error codes, you must add a check for the appropriate errors to your normal AlphaBASIC error trapping routine. In addition, there are several specific considerations to keep in mind for random files and ISAM files.

### A.1.1 Random Files

For exclusive access to random files, make sure the OPEN uses RANDOM to prohibit file sharing. For shared access, make sure the OPEN uses RANDOM'FORCED to allow file sharing. If your program updates random file records, scan for XLOCK calls. There should be one before each READ, and one after each WRITE.

1. If the XLOCK occurs before a READ, delete the XCALL and change READ to READL.

2. If the XLOCK occurs after a WRITE, delete the XLOCK. If the program doesn't update a record, you MUST add an UNLOKR to unlock the record.

3. If the program is creating a new record, remove the XLOCK and change WRITE to WRITEL.

### A.1.2 ISAM Files

To convert a program that uses standard ISAM files, follow these steps. If you are going to convert to ISAM Plus, see the *ISAM Plus User's Guide*.

1. If any of the XLOCKs use mode 1, add WAIT'FILE to the OPEN.

2. Remove the mode 0 and mode 1 XLOCKs before the index is changed.

3. Next, remove the mode 2 XLOCKs which occur after the index is updated.

4. If the program updates existing records, change the READs to READL.

5. If the program creates new records, change the WRITEs to WRITEL.

If there are any sections of the program where ISAM is accessed in an incorrect order, you see error messages when you run the program.

## A.2 FROM FLOCK TO AMOS FILE LOCKING

When you convert programs that use FLOCK, two general rules apply:

1. If the FLOCK uses Action 0 with Mode 0 or Mode 2, you must add WAIT'FILE to the OPEN.

   If the FLOCK uses Action 3 with Mode 0 or Mode 2, you must add WAIT'-RECORD to the OPEN.

2. If your program does any checking or processing of FLOCK errors, you must add a check for the appropriate error to your normal AlphaBASIC error trapping routine.

Modifying programs that use FLOCK is easy since there is almost a one-to-one relationship between FLOCK calls and the changes needed to convert them to use AMOS file locking. The following table compares the two file locking techniques.

| FLOCK | | AMOS FILE LOCKING | |
|---|---|---|---|
| Action | Mode | Random File | ISAM File |
| 0 | 0 | OPEN WAIT'FILE, RANDOM'FORCED | OPEN - WAIT'FILE, INDEXED |
| 0 | 2 | OPEN WAIT'FILE, RANDOM | OPEN - WAIT'FILE, INDEXED'EXCLUSIVE |
| 0 | 4 | OPEN RANDOM'FORCED | OPEN INDEXED |
| 0 | 6 | OPEN RANDOM | OPEN INDEXED-'EXCLUSIVE |
| 1 | 0 | CLOSE | CLOSE |
| 2 | 0 | END or exit | END or exit |
| 3 | 0 | OPEN WAIT'RECORD | OPEN WAIT'RECORD |
| 3 | 2 | OPEN WAIT'RECORD, READL | OPEN WAIT'RECORD, READL |
| 3 | 4 | no change | no change |
| 3 | 6 | READL | READL |
| 4 | 2 | not applicable | LOCK/automatic |
| 4 | 6 | not applicable | not applicable |
| 5 | 0 | WRITE/UNLOKR | WRITE/UNLOKR |
| 6 | 0 | not applicable | UNLOKR/automatic |

There are several specific considerations to keep in mind when you convert programs using random files and ISAM files.

## A.2.1 Random Files

For exclusive access to a random file, remove all FLOCKs and make sure the OPEN uses RANDOM to prohibit file sharing. For shared access, make sure the OPEN uses RANDOM'FORCED to allow file sharing. If the program updates random file records, scan the program for FLOCKs. There should be one before each OPEN or READ, and one after each WRITE or CLOSE.

1. Remove all FLOCKs before OPENs. Add WAIT'FILE to the OPEN if the FLOCKs within the body of the program used Modes 0 or 2. You can also remove all FLOCKs after CLOSEs.

2. If the FLOCK occurs before a READ, delete the XCALL and change READ to READL. Remember to add WAIT'RECORD to the OPEN if the FLOCK used modes 0 or 2.

3. If the FLOCK occurs after a WRITE, delete the FLOCK. If the program doesn't update a record, you MUST add UNLOKR to unlock the record.

4. If the program is creating a new record, remove the FLOCK and change WRITE to WRITEL.

### A.2.2 ISAM Files

Follow these steps to convert a program that uses standard ISAM files. If you are going to convert to ISAM PLUS, see the *ISAM PLUS User's Guide*.

1. If any of the FLOCKs use Modes 0 or 2, add WAIT'FILE to the OPEN.

2. Remove the Action 0, Mode 2 or 4 FLOCKs before OPENs.

3. Remove the Action 4, Mode 2 and 6 FLOCKs before the index is changed.

4. Remove the Action 6, Mode 0 FLOCKs after the index is updated.

5. Remove the Action 1, Mode 0 FLOCKs after CLOSEs.

6. If you are using multiple keys, add a 'LOCK statement before you locate a record on a secondary index.

7. If the program updates existing records, remove the FLOCKs (Action 3, Mode 2 or 6) before each READ, remove the FLOCKs (Action 5, Mode 0) after each UPDATE'RECORD, and change the READs to GET'LOCKED.

8. If the program creates new records, remove the FLOCKs (Action 3, Mode 2 or 6) before each WRITE, remove the FLOCKs (Action 5, Mode 0) after each WRITE, and change the WRITEs to CREATE'RECORD.

9. If the program deletes records, add a GET'LOCKED to lock the data record.

If there are any sections of the program where ISAM is accessed in an incorrect order, you see error messages when you run the program.

# APPENDIX B

# FLOCK

FLOCK is an external subroutine AlphaBASIC programs can call to protect one or more files from concurrent use by other programs that use FLOCK. Its name stands for "File Locking" and it is a convenience feature—NOT a security device like the AMOS File Locking system. For FLOCK to be effective, all your AlphaBASIC programs must use it.

## B.1 LOADING FLOCK INTO SYSTEM MEMORY

FLOCK normally resides as FLOCK.SBR in account DSK0:[7,6]. FLOCK also requires the file FLTCNV.LIT be loaded into system memory. This file resides in account DSK0:[1,4].

You can use FLOCK successfully only if it is loaded into system memory. It appears to work if it's loaded into user memory, but no file locking actually occurs.

To load FLOCK.SBR and FLTCNV.LIT into system memory, add the appropriate SYSTEM commands to your system initialization command file. For example:

```
SYSTEM FLOCK.SBR[7,6]       ; load FLOCK subroutines
SYSTEM FLTCNV
```

Make a copy of your system initialization command and modify the copy. For complete information on altering this command file, see the *AMOS System Operator's Guide to the System Initialization Command File.*

## B.2 USING FLOCK IN ALPHABASIC PROGRAMS

You can put FLOCK to work at any of three levels of increasing complexity. You can:

- Implement file-open interlocks.
- Implement file-open and individual record-update interlocks.
- Implement complete file interlocks and individual record processing interlocks.

In your programs you are free to use any level of complexity, so long as you use the same level consistently for any given file within your program. The level of complexity you use is determined by the parameters given when you call FLOCK from your program. Here is the calling sequence:

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

where `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. `RETURN-CODE` is a 6-byte floating point variable.

## B.2.1∞ACTION & MODE

Action, modified by Mode, requests the file access you want FLOCK to do. The list below describes the combinations and the function of each.

| | |
|---|---|
| Action 0, Mode 0: | Asks to open the file for non-exclusive use. The request goes in a first-come, first-served queue, and the program is delayed until the request is granted. |
| Action 0, Mode 2: | Asks to open the file for exclusive use. The request is placed in a first-come, first-served queue, and the program is delay until the request is granted. |
| Action 0, Mode 4: | Asks to open the file for non-exclusive use. If the request is not granted immediately, `RETURN-CODE` is set to 1. |
| Action 0, Mode 6: | Asks to open the file for exclusive use. If the request is not granted immediately, `RETURN-CODE` is set to 1. |
| Action 1, Mode 0: | Informs FLOCK the file is closed so FLOCK can unlock it. Implicitly informs FLOCK any processing of records in the file is done so FLOCK can do Actions 5 or 6 as necessary. |
| Action 2, Mode 0: | Informs FLOCK abnormal program termination is about to occur (for example, during an error handling routine). Releases all locks on all files by Action 1 as necessary. |
| Action 3, Mode 0: | Asks to read the record of the file for shared use. Permission to open the file must already be granted. The request goes in a first-come, first-served queue, and the program is delayed until the request is granted. |
| Action 3, Mode 2: | Asks to read the record of the file for exclusive use. Permission to open the file must already be granted. The request goes in a first-come, first-served queue, and the program is delayed until the request is granted. |

| | |
|---|---|
| Action 3, Mode 4: | Asks to read the record of the file for shared use. Permission to open the file must already be granted. If the request is not granted immediately, `RETURN-CODE` is set to 1. |
| Action 3, Mode 6: | Asks to read the record of the file for exclusive use. Permission to open the file must already be granted. If the request is not granted immediately, `RETURN-CODE` is set to 1. |
| Action 4, Mode 2: | Asks to read/write all records of the file for exclusive use. Permission to open the file must already be granted. The request goes in a first-come, first-served queue, and the program is delayed until the request is granted. |
| Action 4, Mode 6: | Asks to read/write all records of the file exclusively. Permission to open the file must already be granted. If the request is not granted immediately, `RETURN-CODE` is set to 1. |
| Action 5, Mode 0: | Informs FLOCK processing of the record and the file (permission granted by Action 3) is done. The record is unlocked. If data was buffered for output, it is written to disk. |
| Action 6, Mode 0: | Informs FLOCK exclusive processing of the file (for which permission was granted by Action 4) is done. The file is unlocked. Any succeeding programs granted use of the file by Action 3 or 4 automatically re-open the file. This is done in case exclusive processing of the file caused it to be re-created. If data is buffered for output, it is written to disk. |

## B.2.2∞FILE

The FILE argument 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 random for all other actions.

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 your application programs, being careful to assign the same numbers to the same files.

File-channel numbers 1-999 are 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. Then no conflict can arise between your application programs and any present or future Alpha Micro software on your computer.

### B.2.3∞RECORD

The RECORD argument specifies a logical record number. For Actions 0, 2, 4, and 6, RECORD is ignored and may be omitted.

### B.2.4∞RETURN-CODE

The RETURN-CODE argument denotes a variable in which FLOCK places a numeric code to indicate the success or failure of an action:

| | |
|---|---|
| 0 | Successful (All actions) |
| 1 | Resource unavailable (Actions 0, 3, 4) |
| 2 | Open request has already been granted (Action 0) |
| 3 | Permission to open must first be granted (Actions 1, 3-6) |
| 4 | Duplicate request for use of some record in file (Actions 3, 4) |
| 6 | Permission to use some record in file must first be granted (Actions 5, 6) |
| 100 | Unimplemented Action |
| 101 | File-channel number is not open in AlphaBASIC for random processing (Actions 3-6) |
| 102 | File-channel is already open in AlphaBASIC for an ISAM indexed file |
| 103 | For Actions 0, 3, and 4: Fewer 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 might want to include the following statement to help you debug your program:

```
IF Return-code > 1 THEN &
     PRINT "FLOCK Error - ";Return-code : STOP
```

For calls which use modes 4 or 6, RETURN-CODE = 1 should be checked to see if FLOCK was able to satisfy the request immediately. Modes 4 and 6 are generally used in this way to let the program cancel a request which may involve a lengthy delay.

## B.3∞QUEUE BLOCK REQUIREMENTS

FLOCK builds its dynamic tables out of monitor queue blocks. The monitor queue is a list of blocks of system memory linked to each other in a forward chain.

Before you run any AlphaBASIC program that uses FLOCK, ensure the monitor is configured to make an adequate number of these queue blocks available.

The number of queue blocks used by FLOCK 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 individual files open using FLOCK, plus
The individual records open using FLOCK, plus

The jobs with files open using FLOCK, plus
The total unclosed FLOCK opens (Action 0s), plus
The total unreleased records (Action 3s)

The last two factors of this sum 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's two opens or two Action 0s.

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

```
QUEUE RETURN
20 Queue blocks available
```

The monitor is initially generated with 80 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 command file prior to the final SYSTEM command. When the QUEUE nnn command is executed, "nnn" more queue blocks are allocated for general use. For more information on modifying the system initialization command file, see the *AMOS System Operator's Guide to the System Initialization Command File*.

## B.4∞FILE LOCKS

File-open interlocks can cause long delays for any users trying to access a file after one user has opened it and 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 operations, you want assurance no other user can access the file.

Action 4 gets exclusive access to a file by gaining 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 all other users with the file open must then re-open it. Action 3 performs the necessary re-openings.

### B.4.1∞Programming Example

Here is a partial program which illustrates the use of FLOCK 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
```

## B.5∞RECORD-UPDATE INTERLOCKS

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 is not used for updating, in order to prevent interconsistency errors. Action 5 is used after the sequence of reads. Action 3, Mode 2 or 6, is used before reading records which are used for updating. Action 5 is used again after rewriting the records.

## B.6∞DEADLOCK, AND HOW TO PREVENT IT

FLOCK does not directly solve the problem of Deadlock. You must observe the necessary precautions as you write your programs to insure Deadlock does not occur.

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.

# APPENDIX C

# XLOCK

XLOCK is an external subroutine your AlphaBASIC program can call to set and test "locks." 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 the program wants to access it. Generally, XLOCK works like this:

- When you want to prevent other users from accessing a file or a device while your program accesses it, you create a system lock on that resource.
- Whenever you want to access a device or file, your program tries to set the lock associated with that resource; if it is already locked, you know another program is currently using that device or file.
- When you are finished accessing a resource, you unlock it so other programs can access the resource.

Remember, a system lock is not a security device—it's a convenience. The only job that can unlock a resource is the job that originally locked it. AlphaBASIC does not automatically remove locks when a program exits, so before your program exits, it must be sure to clear any locks it has set.

## C.1 LOADING XLOCK INTO SYSTEM MEMORY

To call XLOCK from you AlphaBASIC programs, you must first load it into system memory. The XLOCK.SBR subroutine file resides in account DSK0:[7,6], and you must add a SYSTEM command to your system initialization command file to include XLOCK. For example:

```
SYSTEM XLOCK.SBR[7,6]       ; load XLOCK subroutine
```

Do not alter the system initialization command file directly—make a copy and alter that. For complete information, see your *AMOS System Operator's Guide to the System Initialization Command File*. After the change is made, XLOCK automatically is loaded into system memory whenever you boot your computer.

## C.2∞USING XLOCK IN ALPHABASIC PROGRAMS

The calling sequence for XLOCK is:

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

where MODE is the function you want it to do and LOCK1 and LOCK2 are the first and second digits of the lock code.

Use MAP statements 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
```

XLOCK parameters must be defined on even-byte boundaries in memory. Variable structures defined at a MAP1 level always begin on a word boundary.

If you do define XLOCK parameters in deeper level MAP statements (e.g., MAP2 or MAP3), make sure the variables begin on a word boundary by keeping the number of bytes defined an even number.

### C.2.1∞MODE

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:

**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 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.

**MODE 1 (Lock and Wait)**

This XLOCK mode is identical to MODE 0, except if the lock already exists, XLOCK tells the system to put your job to sleep until the lock is cleared. That means your job is 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 another program may be waiting for the same lock; it's possible the lock might be cleared and then grabbed up either by the same or another job before your job wakes up.

**MODE 2 (Clear Lock)**

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

**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 set on the system, LOCK1 contains your job number, LOCK2 contains one three-word entry for each lock set on the system. The first two bytes hold the job number; the second and third words hold the actual LOCK1 and LOCK2 values of the specified lock.

## C.2.2∞LOCK1 and LOCK2

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—any number matches in that position when it comes to clearing or setting that lock.

## C.3∞QUEUE BLOCK REQUIREMENTS

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 80 queue blocks available, it's a good idea to keep the number of locks to a minimum.

A good rule is, 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.

# Document History

**Revision A00 - AMOS/L Release 1.1 - (Printed 3/83)**

New Document: part number DSS-10034-00.

**Revision 00 - AMOS Release 2.0 - (Printed 3/88)**

New part number assigned: DSO-00044-00. This document was revised and rewritten to reflect the incorporation of file locking into the AMOS operating system. LOKGEN utility deleted, options added to LOKUTL, etc.

**Revision 01 - AMOS Release 2.1 - (Printed 9/89)**

Re-written to include new file locking enhancements.

**Revision 02 - AMOS Release 2.2 - (Printed 4/91)**

Added SET NOLOCK information, data on READ’READ’ONLY and new LOKUTL options L and O.

**Revision 03 - AMOS Release 2.3 - (Released 6/96)**

Corrected description of READ’READ’ONLY.

**Revision 04 - AMOS Release 2.3 - (Released 9/96)**

Added note that LOKSER returns JCB of job holding a lock in D.ARG.

# INDEX