CLARION TECHNICAL BULLETIN

Bulletin #113

Processing Transactions

Overview

This document is an expansion of material covered in the "Processing Transactions" section of Chapter 11 of the Language Reference manual. It includes a discussion of the following topics: file sharing, record locking, file locking, transaction framing, and multi-user system design. Emphasis is placed on the explanation of related terms, concepts and keywords. Examples are also provided.

Note: These topics are advanced and presume knowledge of the Clarion language. The features described in this document require Batch Release 2007 or higher.

Processing Transactions

System Integrity

A system has integrity when its data records contain valid data and its keys accurately express relationships between records and files. Systems maintain integrity only through careful up-front design and programming. The old rule "if anything can go wrong it probably will" must be taken to heart in software design:

- If a data value is important, the entry field edit must exclude bad data.
- If files are related, routines must internally verify the relationships.
- If the "master" record must not be deleted while "detail" records still exist, logic must prevent the deletion.
- If data must be secure then passwords and encryption should be used.

Clarion helps programmers by supplying tools that simplify the development of systems with integrity. One tool in the Clarion programmer's arsenal is support for "transaction logging."

Transactions

A transaction is a single logical event during which more than one record is updated. It can involve several records in one file, or one or more records in more than one file. The important characteristic of a transaction is that all of the files and records should be updated together, or none of them should be updated at all. A failure in the middle of a transaction compromises the integrity of the system.

Transaction Logging

Transaction logging is also known as "pre-imaging" or as an implementation of "commit boundaries." Transaction logging provides a means of undoing a partial transaction. If a transaction is "undone," data is restored to its original form as if the data has never been updated. During transaction logging, as Clarion files are updated by the ADD, PUT, DELETE, and APPEND statements, the original data and key information for the affected records is saved in pre-image log files. The name of the file is saved in a transaction logout file.

- If the transaction is **successful**, it can be "committed." In a Clarion program, when a transaction is committed, the pre-image logging process ends, and the pre-image log files and the transaction logout file are deleted.

- If the transaction is **unsuccessful**, it can be "rolled back." When a transaction is rolled back, the information saved in the pre-image log files is restored to the data and key files, "undoing" all updates, and resetting the contents of the files back to their states at the beginning of the transaction. After the information has been rolled back, the pre-image log files and the transaction logout file are deleted.

Pre-image Log Files

A "pre-image log file" contains the image of records and keys as they existed before they were updated. Language statements do not designate pre-image log files. Pre-image log files are created, accessed, and deleted automatically by the transaction logging statements: LOGOUT, COMMIT, and ROLLBACK. The Clarion Sorter, Filer, and Scanner utility programs detect the presence of pre-image log files and inform the user that a rollback is needed.

In the event of a system failure during a transaction, pre-image log files will exist on disk. If the Clarion Sorter, Filer, or Scanner utility programs are run, they will detect the presence of the pre-image log files and inform the user that a rollback is needed.

A pre-image log file is created for each Clarion file accessed during transaction logging. This file has the same file specification as the data file with an extension of ".LOG." In other words, it is created in the same directory as the data file, and with the same name as the data file, but with an extension of ".LOG."

Pre-image log files support a single logical transaction only. If a transaction logging process begins and a pre-image log file already exists (it has not been committed or rolled back), the saved information in the existing pre-image file will be deleted.

The pre-image log files of a transaction maintain the relationship with each other by each naming a common file called a "transaction logout file."

Transaction Logout File

Transaction logout files should not be confused with pre-image log files. A transaction logout file is a binary DOS file which contains a list of the Clarion files accessed during the current transaction. Each Clarion file that is pre-imaged will be noted in the transaction log file. Each pre-image log file in the transaction contains the name of the common transaction logout file. Only one transaction logout file can be active at a time.

To begin transaction logging, a transaction logout file must be specified by the LOGOUT statement.

LOGOUT Statement

Form: **LOGOUT** *(transaction logout file)*

LOGOUT initiates transaction logging. During transaction logging, as Clarion files are updated by the ADD, PUT, DELETE, and APPEND statements, the original data and key information for the records is saved in pre-image log files, and the file names are saved in a transaction logout file.

The *transaction logout file* parameter is a constant, variable, or expression that specifies a transaction logout file. This parameter is required. If no drive or path is specified, the current directory drive and path are used; there is no default extension. If the transaction logout file already exists, the program will halt with the message: "UNABLE TO LOG TRANSACTION."

Only one transaction can be processed at a time. If transaction logging is already in progress and a LOGOUT statement is executed, the program will halt with the message: "LOGOUT ALREADY ACTIVE." The halt will occur even if a different transaction logout file is specified.

If the LOGOUT statement is successful, transaction logging continues until a COMMIT or ROLLBACK statement is issued.

COMMIT Statement

Form: **COMMIT**

COMMIT turns off transaction logging and makes a transaction permanent by deleting the pre-image log files and the transaction log file. An implied COMMIT occurs when the program goes to a normal end of job, or when a RUN, CHAIN, or CALL statement is executed.

ROLLBACK Statement

Form 1: **ROLLBACK** *(transaction logout file)*

Form 2: **ROLLBACK**

ROLLBACK turns off transaction logging, restores the files updated during the transaction to their original states, and deletes the transaction log file and the pre-image log files.

During transaction logging, as Clarion files are updated by the ADD, PUT, DELETE, and APPEND statements, the original data and key information for the records is saved in a pre-image log file for each data file, and a list of the updated files is kept in the transaction logout file. ROLLBACK refers to the transaction log file to see which files have been updated during the transaction, then each pre-image log file is processed, restoring all updated files to their condition prior to the LOGOUT statement.

- In Form 1, the *transaction logout file* parameter is a constant, variable, or expression that specifies a transaction logout file. This parameter is optional. When it is used and no drive or path is specified, the current directory drive and path are used. There is no default extension.

- In Form 2, no transaction logout file is specified. This form presumes transaction logging is active and uses the current transaction logout file.

- If transaction logging is **active**, the current transaction logout file specified by LOGOUT is used, even if ROLLBACK specifies a different transaction logout file.

- If transaction logging is **inactive**, ROLLBACK must specify a transaction logout file. If the transaction logout file does not exist, the ROLLBACK statement is ignored.

In general, Form 1 should be used when transaction logging is not in progress (i.e., at the start of a program). Form 2 should be used while transaction logging is in progress.

If the RECOVER statement has been used to arm a recovery process, an implicit ROLLBACK may be executed. If the recovery process is invoked to unlock a locked file, and if transaction logging is not in progress, and if a pre-image log file exists for that file, then pre-image data will be rolled back for that file. Other files may also be rolled back if the pre-image log file points to a transaction logout file that lists other files.

Note: Versions of Clarion prior to Batch 2008 may exhibit an error that requires all of the files named in the transaction logout file to be open at the time a ROLLBACK is issued.

Are there "windows of vulnerability"?

While transaction logging is active, before a file update is applied, the transaction logout file is updated first, followed by the pre-image log file, and then the data file. The transaction logout file and the pre-image log files are kept logically closed.

During a commit, the transaction logout file is deleted first, followed by the pre-image log files in the same order in which they were created.

While a transaction is being rolled back, pre-image data is read from the end of a pre-image log file (the last record updated is rolled back first.) After the record is rolled back, the pre-image information for that record is deleted from the pre-image log file. After all of the records have been rolled back from a given pre-image log file, ROLLBACK moves to the next pre-image log file listed in the transaction logout file. When all the records have been rolled back from all of the pre-image log files, ROLLBACK deletes the pre-image log files and the transaction logout file.

All of the steps described on the previous page insure that there are no "windows of vulnerability" while using transaction logging.

Statement Summary

Only Clarion files can be pre-imaged and rolled back. DOS files are not affected by LOGOUT, ROLLBACK, or COMMIT. During transaction logging, the following statements update Clarion data files and their effects can be rolled back:

```
ADD     PUT     DELETE     APPEND
```

The following statements will cause an implicit COMMIT:

```
RETURN     (to end the program, or any normal end-of-job)
RUN
CALL
CHAIN
```

Ctrl-Brk

Note: In batch releases prior to Batch 2008, a Ctrl-Break exit is considered a normal exit. An implicit COMMIT occurs.

In batch releases 2008 and higher, Ctrl-Break exit is not considered a normal exit, and an implicit COMMIT does not occur. It is up to the program to detect the situation and to perform the rollback as needed.

The following statements may be executed within a transaction frame, but the results of their execution are not rolled back if a ROLLBACK is executed:

```
OPEN        CLOSE       SHARE       BUFFER
CACHE       STREAM      FLUSH       SET
NEXT        PREVIOUS    GET         SKIP
COPY
```

Note: Version of Clarion prior to Batch 2008 may exhibit an error that requires all of the files named in the transaction logout file to be open at the time a ROLLBACK is issued.

The following statements will cause the program to halt with the message "LOGOUT ALREADY ACTIVE" if executed while transaction logging is active:

```
LOGOUT      BUILD       PACK        EMPTY
CREATE      RENAME      REMOVE
```

The RESTART statement may be executed during transaction logging, but the program will halt if the result is that LOGOUT is executed again while transaction logging is still in progress.

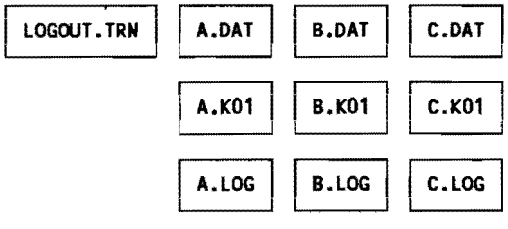
If the RECOVER statement has been used to arm a recovery process, and if the recovery process is invoked to unlock a locked file, and if transaction logging is not in progress, and if a pre-image log file exists for that file, then pre-image data will be rolled back for that file and any other file named in the transaction logout file.

A Simple Example

Consider the following example:

```
LOGOUT('LOGOUT.TRN')

ADD(A)
ADD(B)
ADD(C)
```

Assume files A, B, and C each has one key. After the above statements, the following files exist on disk:



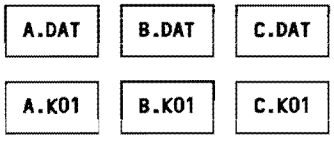
LOGOUT.TRN is the transaction logout file named by the LOGOUT statement. It contains the names of files A, B, and C which were involved in the transaction.

A.LOG, B.LOG, and C.LOG are the pre-image log files for files A, B, and C. Each of the pre-image log files contain images of data file records as they existed before they were updated. .

The following statement completes the transaction:

```
COMMIT
```

After the COMMIT statement is executed, the following files exist on disk:



The transaction logout file (LOGOUT.TRN) and the pre-image log files (A.LOG, B.LOG, and C.LOG) have been deleted. Transaction logging is turned off, and the transaction is permanent.

Single-User versus Multi-user

The basic pieces of transaction logging have now been described. Putting the pieces of the puzzle together is a different matter. There are two different pictures that must be constructed: "single-user" and "multi-user" applications. Implementing transaction logging with multi-user applications is much more complex than with single-user applications, so the single-user side will be covered first.

Single-User Transaction Framing

Single-user applications must only guard against failure in the middle of a transaction. There is no conflict over shared resources as in a multi-user environment. Transaction logging can be implemented by following a few simple rules or guidelines. (Some of these rules or guidelines will also apply to multi-user.)

Rule S1: Only use one transaction logout file.

Only one transaction logout file can be open at a time, therefore, the best strategy is to only name one transaction logout file in the course of a program. LOGOUT will halt if logging is already in progress or if it detects the existence of the named transaction logout file. When logging begins, however, an existing pre-image log file will be created fresh, deleting an existing copy of the file if one exists. By only naming one transaction logout file, it will be unlikely that pre-image data will be lost when a rollback was needed.

Rule S2: Begin a program with a ROLLBACK.

If a program halts while transaction logging is in process, the partial transaction needs to be rolled back. The transaction logout file and pre-image files will exist on disk. If an attempt is made to begin logging again, LOGOUT will halt the program.

There are only two ways to get the program up and running again:

1. **Manually delete the transaction logout file.**

   This gets around the LOGOUT halt, but it defeats the purpose of transaction logging. The integrity of the data files is now suspect.

2. **Rollback the partial transaction.**

   The ROLLBACK statement can name a transaction logout file. If the transaction logout file exists, the data is rolled back and the program continues. If the transaction logout file does not exist, the ROLLBACK statement is ignored. If a single-user program that uses transaction logging begins with a ROLLBACK statement, it ensures that any partial transaction will be rolled back.

Note: Versions of Clarion prior to Batch 2008 may exhibit an error that requires all of the files named in the transaction logout file to be open (via OPEN or SHARE) at the time a ROLLBACK is issued.

Rule S3: Keep transaction frames small.

Do not use transaction logging simply as an "oops!" feature. Use logging only around those record updates that must handled in an "all or none" manner. Transaction logging will cause some degradation in performance and requires disk space to create the logging files.

Rule S4: Check for errors.

Use the ERROR( ) or ERRORCODE( ) function to check for errors after every file update -- especially in those cases where it seems unlikely that an error would be returned. Checking errors at every opportunity will catch many program logic errors and help prevent file corruption.

A Single-User Example

The following program fragment shows an example of how a simple transaction involving two files could be coded. In this example, master records in one file maintain a count and total dollar amount of the related detail records. When a detail record is added, the master file is updated as well. If the detail record cannot be added, the master record should not be updated. If the master record cannot be updated, the detail record must be rolled back.

```
            PROGRAM

MASTER_FILE FILE,PRE(MST)           !MASTER FILE
MASTER_KEY    KEY(MST:ACCOUNT)
MASTER_REC    RECORD
ACCOUNT         DECIMAL(10)         !ACCOUNT NUMBER
COUNT           LONG                !CURRENT DETAIL COUNT
AMOUNT          DECIMAL(14.2)       !CURRENT DETAIL TOTAL AMOUNT
            . .

DETAIL_FILE FILE,PRE(DTL)           !DETAIL FILE
DETAIL_KEY    KEY(DTL:ACCOUNT,DTL:LINENO)
DETAIL_REC    RECORD
```

```
ACCOUNT          DECIMAL(10)          !ACCOUNT NUMBER
LINENO           LONG                 !DETAIL LINE NUMBER
ITEM_NO          DECIMAL(7)           !ITEM NUMBER
QUANTITY         LONG                 !QUANTITY ORDERED
AMOUNT           DECIMAL(9.2)         !ORDER AMOUNT
                . .

                     :

                CODE

                OPEN(MASTER_FILE)                   !OPEN THE MASTER FILE
                IF ERROR() THEN STOP(ERROR()).
                OPEN(DETAIL_FILE)                   !OPEN THE DETAIL FILE
                IF ERROR() THEN STOP(ERROR()).

                ROLLBACK('LOGOUT.TRN')              !UNDO DANGLING TRANSACTIONS

                     :

!ACCESS A MASTER RECORD
!   USER ENTERS ACCOUNT NUMBER
                     :

!ALLOW ENTRY OF INFORMATION FOR NEW DETAIL RECORD
!   USER ENTERS ITEM NUMBER, QUANTITY, AMOUNT
!   ACCOUNT NUMBER OF THE DETAIL IS THE MASTER ACCOUNT NUMBER
                     :

!NOW ADD A NEW DETAIL RECORD AND UPDATE THE MASTER RECORD

                LOGOUT('LOGOUT.TRN')                !BEGIN THE TRANSACTION

                DTL:LINENO = MST:COUNT + 1          !INCREMENT DETAIL LINE NUMBER
                ADD(DETAIL_FILE)                    !ADD A DETAIL RECORD
                IF ERROR()                          !IF THE ADD FAILED
                  ROLLBACK                          !  END THE TRANSACTION
                ELSE                                !IF THE ADD WAS SUCCESSFUL
                  MST:COUNT += 1                    !  INCREMENT DETAIL COUNT
                  MST:AMOUNT += DTL:AMOUNT          !  UPDATE DETAIL AMOUNT
                  PUT(MASTER_FILE)                  !  UPDATE MASTER RECORD
                  IF ERROR()                        !  IF THE PUT FAILED
                    ROLLBACK                        !    UNDO THE TRANSACTION
                  ELSE                              !  IF THE PUT SUCCEEDED
                    COMMIT                          !    END THE TRANSACTION
                . .
```

Multi-user Environments

Clarion defines a "multi-user" environment to be a hardware/software configuration in which multiple Clarion programs (or multiple copies of a Clarion program) can simultaneously view and change a common set of Clarion files. There are three prime examples of multi-user environments:

1. The most common multi-user environment consists of several PC's (workstations) connected to a Local Area Network (LAN). The data files may reside on the disk drive of a separate computer called a "file server," or the files may be distributed between various "nodes" on the network. Clarion applications have been successfully implemented with several brands of networks, including Novell, 3-Com, 10-Net, and Banyan.

2. Multiple Clarion programs can also be run on a single machine with "dumb" terminals. This environment uses a multi-user operating system to run several programs at once in separate DOS partitions. Clarion programs for environments of this type have been developed and run under Concurrent-DOS, PCMOS-386, and Multi-Link.

3. The third example is a hybrid of the previous two. A system of multiple processor (slave) cards plugged into a server machine has characteristics of both a network and a multi-user environment. The Alloy system is the best example in this category and Clarion programs work well in this configuration.

Clarion itself makes no distinction between, and provides no specific support for, any of the products mentioned. The feature of Clarion that allows it to work with all of those products (without specifically supporting any of them) involves the DOS file sharing feature.

DOS File Sharing

DOS versions 3.0 and up allow a file to be opened in "shared" mode. A file that is shared can be accessed by more than one user. The Clarion SHARE statement opens a Clarion file in "share" mode with the attributes of "deny none" and "permit read and write." Multi-user Clarion programs should be able to be written in any environment that transparently emulates a 3.0 or greater DOS version.

Multi-User Transaction Framing

A multi-user application must guard against more dangerous situations than a single-user application. In addition to the possibility of system failure during a transaction, a programmer must also design the system to deal with conflicts over shared resources. The following rules or guidelines will aid in the design of multi-user applications using transaction logging.

Rule M1: Use a UPS (Uninterruptable Power Supply).

The Clarion file system keeps the transaction logout file and the pre-image log files "logically closed." This means that internal buffers are flushed with every update so that the operating system will keep the data and the directory on disk current. This feature can also be used optionally with Clarion data files.

Flushing may or may not work, however, when using a network operating system that is emulating DOS. Some networks have memory caches that are written to disk only periodically, not necessarily when DOS buffers are flushed. Other networks may flush the data, but may not write out file allocation tables. Data will be written to files, but directory information will not be written at the same time.

Therefore, in these cases, do not bother attempting to implement transaction logging without an uninterruptable power supply. Logging serves no purpose if the log files themselves are not written to disk.

Quality power supplies are now available for only a few hundred dollars. Most UPS units can continue operation through a power outage for at least fifteen minutes. This provides plenty of time for an orderly shutdown to prevent file corruption.

Rule M2: Use one transaction logout file per union of files.

This is a qualification of Rule S1 above. This rule is especially important in multi-user applications so that one user does not destroy the pre-image log files of another user. Also, this rule is necessary so that when simultaneous transactions are attempted, programs will wait instead of halting.

This rule has two important implications:

1. A second transaction logout file can be used if (and only if) the files updated in the transaction are exclusive to that transaction (i.e., they are not involved in any other transaction's set of updated files).

2. If a set of transactions have some files in common, then all of the transactions must have at least one file in common.

For example, suppose there are four files: A, B, C, and D. Suppose there are three different transactions and each involves a different set of files:

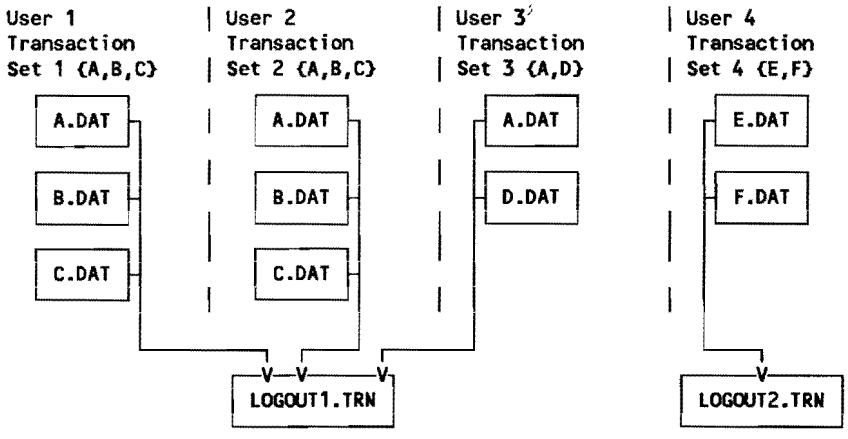
```
Set 1: {A,B,C}

Set 2: {B,C}

Set 3: {A,D}
```

Set 2 and Set 3 do not have files in common. However, Set 3 does have a file in common with Set 1, namely, A.

If Set 2 and Set 3 use a common transaction logout file, a halt may occur because the second user cannot detect that logging is in progress. A program halt will result. To avoid a halt, a second transaction logout file could be used. However, since Set 3 has a file in common with Set 1, Set 1 and Set 3 must use the same transaction logout file. Also, since Set 1 and Set 2 have files in common (B and C), Set 1 and Set 2 must use the same transaction logout file. Therefore, all three sets must use the same transaction logout file, but this presents problems for Sets 2 and 3.

The solution to the problem is that the sets must be changed so that they have at least one file in common. In this case, file A could be added to Set 2. As you will see in Rule M3 below, file A need not actually be updated in the transaction, but must be one of the files to be locked.

Only when there is no overlap of files can a second transaction logout file (as shown here) be used:



Rule M3: Lock files to gain exclusive control.

Clarion multi-user applications use record holding to gain control over records during updates. If transaction logging is used, file locking must be used around a transaction.

Locking the Clarion files that are to be updated in the transaction prevents two situations:

1. Only one transaction should be processed at a time. The transaction logout file and the pre-image log files are opened in "exclusive" mode. If another user attempts to begin logging, that user's program will halt. Locking provides a means of making the program wait, rather than halt.

2. Users should not be able to query data during a transaction. Transaction results are supposed to be "all or none." Locking a file restricts access during a transaction. Held records can be read by other users, but not records in a locked file.

Rule M4a: Be careful mixing record holding and file locking.

Some file updates are not really transactions. That is, if only one record in a file is updated, is that a transaction? Does such an operation require logging? Perhaps not, but a multi-user application requires exclusive access to the record during an update. This means that the record must, at least, be held. We know from Rule M3 that file locking needs to be used to secure a transaction. There are some problems that must be considered, however, if file locking and record holding are to be mixed.

There is nothing to prevent one station from locking a file while another station has a record held in the same file.

Consider the following situation:

Station "A" holds a record, then Station "B" locks the file. When Station "A" goes to release the held record, it will have to wait until Station "B" unlocks the file. If, by chance, Station "B" must also update a situation called "deadly embrace." (Deadly embrace is also discussed in the section on Rule M7.) Station "A" cannot release the record because Station "B" has the file locked. Station "B" cannot complete the transaction because Station "A" has the record held. Deadly embrace can be anticipated, detected, and handled. The best bet, however, it to avoid it altogether, if possible.

Rule M4b: Hold records to be updated before locking.

Before locking the files and enabling logging, first hold any records that are to be updated. This prevents another user from gaining exclusive control over the record before the file is locked.

There is an important qualification to the rule:

Only one record at a time in a given file can be held by a given station. If a HOLD/GET operation is followed by another HOLD/GET to the same file, the first held record is released. If multiple records are to be updated in a file, it is not possible to hold all of the records before locking the file.

In this case, Station "B" must first lock the file, and then also attempt to hold any record to be updated. (The hold operation should utilize a time-out and error checking in order to detect the existing hold.) The purpose of the hold is not to gain exclusive access (the file lock should provide this), but to detect records that might already be held and thus avoid "deadly embrace."

There are two alternate solutions to this problem:

One is to **never allow an application to hold a record outside of a transaction frame.** If this is not feasible, then consider designing the application so that a common "control" file is always locked and unlocked around any file updates. By first locking the control file before holding a record, the application insures that it will not conflict with another station. If the control file only contains one record, then the same results can be obtained by holding that one record.

Based on these considerations, the following "alternate" rule should be stated:

Rule M4c: Regard every update as a transaction.

If transaction logging is used in a multi-user situation, treat every update (even those involving only one file) as a transaction. Treat all of the files in an application as together comprising a database. Only one user at a time should update the database. Prevent simultaneous updates by holding or locking a shared resource, such as a control file.

If this rule is used, some further thought shows that records never need to be held, and only one file ever needs to be locked.

This rule is not mandatory, but it simplifies the problems of exclusive control on transaction framing and mixing record holds with file locks. This may be desirable for some applications. It is certainly easier to implement. It does, however, impose a single-user processing restriction onto a multi-user situation, and may have a negative affect on overall system response.

Rule M5: Verify that the held record has not been changed.

Next (also before locking the files and enabling logging), some situations may require checking to ensure that another user has not changed a record before it is updated. Save a copy of the record when it is first accessed before user entry. Then after holding the record, before updating, compare the record to the saved copy.

Rule M6: Establish a file update hierarchy.

A hierarchy of access should be established in the application to provide levels of success in completing the transaction. This order should be followed wherever possible. Only the programmer can decide the best order for a given application. This is an area where experience helps. This rule is especially useful when locking the files before beginning to log the transaction. Decide on an order for locking the files and maintain that order.

For example, assume there are four files: A, B, C, and D.

- If one transaction involves A, B, and C, then lock the files in that order: A, B, then C.
- If a second transaction involves B, C, and D, then lock the files in that order: B, C, then D.

Adhering to this rule will help with the next rule.

Rule M7: Detect and avoid "deadly embrace."

Assume Program One locks File A and Program Two locks File B. Then Program One attempts to lock File B and waits. Then Program Two attempts to lock File A and waits. Both programs are now frozen, waiting for the other to release a locked file. This situation is called "deadly embrace."

To avoid deadly embrace, use Rule M6 and the time-out parameter on the LOCK statement. Chapter 11 of the Language Reference manual provides coding examples of this, in the section entitled "Avoiding the Deadly Embrace."

Rule M8: Understand RECOVER thoroughly before using it.

File locking is used to ensure that only one transaction can be processed at a time. This fact makes it likely that eventually a program will fail and leave one or more files locked. The only way a program can unlock a locked file is by arming the RECOVER process.

The RECOVER statement arms a recovery process that unlocks files or releases records that may have been left in a locked or held state by system failure. If a language statement is stopped by a locked file or held record and RECOVER is armed, the locked file is unlocked or the held record is released.

The RECOVER statement has the following forms:

Form 1: label **RECOVER***(seconds)*

Form 2: label **RECOVER**( )

- In Form 1 of the RECOVER statement, the *"seconds"* parameter is a numeric constant or variable that specifies the recovery wait period in seconds. This is the time in seconds after which a file or record is considered to be locked due to system failure, and the recovery process is invoked. If this value is zero, RECOVER begins immediately. If it is less than zero, RECOVER is ignored.

- Form 2 of RECOVER disarms the recovery process.

For programs that use shared files, the recovery wait period should be set greater than the longest duration of exclusive control --- the longest time it takes any user to update a record, or log a transaction, or the longest time a file can be locked, or a record can be held.

It is possible to wait longer than the recovery period if a file is locked and being updated. Each time activity is detected on the locked file, the recovery wait period is started again.

If the recovery process is invoked to unlock a locked file, and if transaction logging is not in progress, and if a pre-image log file exists for that file, then pre-image data will be rolled back for that file. Other files may also be rolled back if the pre-image log file points to a transaction logout file that lists other files.

The RECOVER statement can be issued any number of times, arming or disarming the recovery process, or changing the recovery wait period. RECOVER does not post any error messages.

If the RECOVER process is called to unlock a file and a pre-image file is detected for that file, a ROLLBACK is attempted. If logging is in progress, this will result in a program halt. If RECOVER is avoided, however, this still leaves the problem of recovering locked files.

Probably the best solution is to use RECOVER with a long timeout at the beginning of a program when the data files are being opened with the SHARE statement. Once all files are opened successfully in shared mode, then disable the recovery process.

The rest of the program should use LOCK with a timeout parameter, detect the lock error, and inform the user to please try again. If problems persist due to a locked file, exit and begin the program again, letting RECOVER handle the situation.

Rule M9: Be careful beginning a program with a ROLLBACK.

This is a qualification of Rule S2 above. Users will probably not begin their programs at the same time. One user might be logging a transaction when the second user begins. If the second user executed a ROLLBACK, the second user's program will halt. Yet, in the case of a failure, someone must rollback the failed transaction.

If Rule M8 is followed, the RECOVER process should handle this problem. Another possibility is to use a shared control file to note how many users are currently active, and then only allow the first program to run to issue the ROLLBACK. This technique is illustrated in the following example.

Note: Versions of Clarion prior to Batch 2008 may exhibit an error that requires all of the files named in the transaction logout file to be open at the time a ROLLBACK is issued.

Rule M10: Keep the "windows of exclusive control" small.

This is a version of rule S3. Transaction frames are periods of exclusive control. Records must be held and files must be locked for only the briefest possible time, or system-wide response will be unacceptable. Only hold records or lock files around updates. Do not hold records or lock files in field edits. It may seem to be a less efficient way of coding to retrieve the record twice, but it is necessary in a multi-user situation. A user must not be allowed to hold a needed record (preventing updates by other users) and then go away on a lunch break.

Rule M11: Check for errors.

Same as Rule S4. Check for errors after every file update -- especially in those cases where it seems unlikely that an error would be returned. This cannot be over-emphasized. Checking errors at every opportunity will catch many program logic errors and help prevent file corruption.

A Multi-user Example

The following program fragments show the earlier example enhanced for a multi-user environment. This example shows:

- How RECOVER is used at the start of the program to handle locked files.
- How a control file is used to see if a program is permitted to execute.
- How that same control file is also used to see if this is the first active user and, if so, to execute ROLLBACK to handle any dangling incomplete transactions.
- How a function is used to do the transaction, with error conditions set by the returned string.
- How records to be updated are held before the files are locked.
- How records to be updated are checked to be sure they were not changed by another station.
- How LOCK is used to insure this is the only transaction begin entered.

```
            PROGRAM

CONTROL     FILE,PRE(CTL)             !CONTROL FILE
CONTROL_REC   RECORD
PERMITTED       BYTE                  !PERMISSION SWITCH (NOT ZERO = OK)
USERS_ACTIVE    SHORT                 !HOW MANY USERS ARE ACTIVE NOW
            . .

MASTER_FILE FILE,PRE(MST)             !MASTER FILE
MASTER_KEY    KEY(MST:ACCOUNT)
MASTER_REC    RECORD
ACCOUNT         DECIMAL(10)           !ACCOUNT NUMBER
COUNT           LONG                  !CURRENT DETAIL COUNT
AMOUNT          DECIMAL(14.2)         !CURRENT DETAIL TOTAL AMOUNT
            . .

SAVE_MAST   GROUP                     !SAVE AREA FOR MASTER RECORD
              BYTE,DIM(SIZE(MST:MASTER_REC))
            .

DETAIL_FILE FILE,PRE(DTL)             !DETAIL FILE
DETAIL_KEY    KEY(DTL:ACCOUNT,DTL:LINENO)
DETAIL_REC    RECORD
ACCOUNT         DECIMAL(10)           !ACCOUNT NUMBER
LINENO          LONG                  !DETAIL LINE NUMBER
ITEM_NO         DECIMAL(7)            !ITEM NUMBER
QUANTITY        LONG                  !QUANTITY ORDERED
AMOUNT          DECIMAL(9.2)          !ORDER AMOUNT
            . .
ERR_MSG     STRING(50)                !ERROR MESSAGE

            MAP
              FUNC(ADD_DETAIL),STRING
            .

                 :

            CODE
            RECOVER(120)                      !WAIT 2 MINUTES

            SHARE(CONTROL)                    !OPEN CONTROL FILE SHARED
            IF ERROR() THEN STOP(ERROR()).
            HOLD(CONTROL)
            GET(CONTROL,1)
```

```
            IF ERROR() THEN STOP(ERROR()).
            IF NOT CTL:PERMITTED                !IF ZERO, CANNOT ACCESS
              RELEASE(CONTROL)
              BLANK
              SHOW(12,1,'ACCESS IS NOT PERMITTED AT THIS TIME')
              SHOW(13,1,'PLEASE TRY AGAIN LATER')
              SHOW(14,1,'PRESS ANY KEY TO EXIT')
              ASK
              RETURN
            .
            IF CTL:USERS_ACTIVE = 0             !FIRST USER?
              OPEN(MASTER_FILE)                 !OPEN FILES
              IF ERROR() THEN STOP(ERROR()).
              OPEN(DETAIL FILE)
              IF ERROR() THEN STOP(ERROR()).
              ROLLBACK('LOGOUT.TRN')            !YES, TRY ROLLBACK
              CLOSE(MASTER_FILE)                !CLOSE FILES
              CLOSE DETAIL_FILE)
            .
            CTL:USERS_ACTIVE += 1               !INCREMENT USERS ACTIVE
            PUT(CONTROL)                        !RESTORE CONTROL RECORD
            IF ERROR() THEN STOP(ERROR()).

            SHARE(MASTER_FILE)                  !OPEN MASTER FILE SHARED
            IF ERROR() THEN STOP(ERROR()).

            SHARE(DETAIL_FILE)                  !OPEN DETAIL FILE SHARED
            IF ERROR() THEN STOP(ERROR()).

            RECOVER                             !DISARM RECOVER PROCESS

                 :

!ACCESS A MASTER RECORD
!   USER ENTERS ACCOUNT NUMBER
!   THEN RETRIEVE AND SAVE THE MASTER RECORD

                 :
            GET(MASTER_FILE,MST:MASTER_KEY)     !RETRIEVE MASTER RECORD
            IF NOT ERROR()
              SAVE_MAST = MST:MASTER_REC        !SAVE FOR LATER CHECK

                 :

!ALLOW ENTRY OF INFORMATION FOR NEW DETAIL RECORD
!   USER ENTERS ITEM NUMBER, QUANTITY, AMOUNT
!   ACCOUNT NUMBER OF THE DETAIL IS THE MASTER ACCOUNT NUMBER
                 :
!CALL THE FUNCTION TO DO THE TRANSACTION

            ERR_MSG = ADD_DETAIL()              !GENERATE TRANSACTION
            IF ERR_MSG                          !IF TRANSACTION FAILED
              ...                               !  SHOW MESSAGE TO USER


!**************************
!* ADD A NEW DETAIL RECORD *
!**************************

ADD_DETAIL  FUNCTION
            CODE
                                                !GET EXCLUSIVE CONTROL
```

```
            HOLD(MASTER_FILE,10)                !OF THE MASTER RECORD
            GET(MASTER_FILE,MST:MASTER_KEY)     !WAIT UP TO 10 SECS
            IF ERRORCODE() = 43                 !IF ALREADY HELD
              RETURN('PLEASE TRY AGAIN')        !  RETURN ERROR MESSAGE
            .
            IF ERROR() THEN RETURN(ERROR()).    !ANY OTHER ERROR

            IF MST:MASTER_REC <> SAVE_MAST      !CHANGED BY OTHER STATION?
              RELEASE(MASTER_FILE)
              RETURN('CHANGED BY ANOTHER STATION')
            .


            UPDATE                              !RE-UPDATE THE MASTER
                                                !RECORD FROM THE SCREEN
!NOW GET EXCLUSIVE CONTROL OF THE MASTER FILE

            LOCK(MASTER_FILE,10)                !WAIT UP TO 10 SECS
            IF ERROR()                          !IF UNABLE TO LOCK MASTER
              RELEASE(MASTER_FILE)              !  RELEASE HELD RECORD
              RETURN('PLEASE TRY AGAIN')        !  RETURN ERROR MESSAGE
            .

!NOW GET EXCLUSIVE CONTROL OF THE DETAIL FILE

            LOCK(DETAIL_FILE,10)                !WAIT UP TO 10 SECS
            IF ERROR()                          !IF UNABLE TO LOCK DETAIL
              UNLOCK(MASTER_FILE)               !  UNLOCK MASTER FILE
              RELEASE(MASTER_FILE)              !  RELEASE HELD MASTER REC
              RETURN('PLEASE TRY AGAIN')        !  RETURN ERROR MESSAGE
            .

!NOW BEGIN THE TRANSACTION

            LOGOUT('LOGOUT.TRN')                !INITIATE TRANSACTION LOGGING

            DTL:LINENO = MST:COUNT + 1          !SET A NEW DETAIL LINE NUMBER

            ADD(DETAIL_FILE)                    !ADD A DETAIL RECORD
            IF ERROR()                          !IF UNABLE TO ADD DETAIL
              ROLLBACK                          !  END THE TRANSACTION
              UNLOCK(DETAIL_FILE)               !  UNLOCK DETAIL FILE
              UNLOCK(MASTER_FILE)               !  UNLOCK MASTER FILE
              RELEASE(MASTER_FILE)              !  RELEASE HELD RECORD
              RETURN('UNABLE TO ADD DETAIL RECORD')
            ELSE
              MST:COUNT += 1                    !UPDATE DETAIL COUNT
              MST:AMOUNT += DTL:AMOUNT          ! AND ORDER AMOUNT
              PUT(MASTER_FILE)                  ! IN MASTER RECORD
              IF ERROR()                        !IF UNABLE TO PUT MASTER
                ROLLBACK                        !  UNDO THE TRANSACTION
                UNLOCK(DETAIL_FILE)             !  UNLOCK DETAIL FILE
                UNLOCK(MASTER_FILE)             !  UNLOCK MASTER FILE
                RELEASE(MASTER_FILE)            !  RELEASE HELD RECORD
                RETURN('UNABLE TO ADD DETAIL RECORD')
              ELSE                              !TRANSACTION SUCCESSFUL
                COMMIT                          !END THE TRANSACTION
                UNLOCK(DETAIL_FILE)             !UNLOCK DETAIL FILE
                UNLOCK(MASTER_FILE)             !UNLOCK MASTER FILE
                RETURN('')                      !EMPTY STRING MEANS O.K.
            . .
```