Chapter 4 – A Few Words About Files

This chapter begins the discussion of a very important topic, namely how InternetBasic programs store and retrieve data. Here are the types of data sources that are compatible with IB programs:

- Text files
- Sequential files
- Keyed files
- ODBC data sources

Text files are standard ASCII data files containing text records (up to 1,024 bytes per record). These records are stored in the order in which they are written to the file, and are accessible in the same order (i.e., sequential order). Text files are typically used to import and export standard ASCII data. They are also used to store IB source programs.

Sequential files are fixed record size files (up to 1,024 bytes) with fixed field sizes. Records are stored in the order they are written, and may be retrieved in sequential order (first record to last) as well as direct order (using a numeric index to represent the record number in the file). Hence, sequential files are sometimes referred to as "indexed sequential" files. Sequential files are commonly used as temporary transaction files.

Keyed files provide direct access to data records. Keyed files have fixed record sizes (up to 1,024 bytes) with fixed field sizes. The key for a keyed file is a string field up to 64 characters long. Records are stored in the order they are written, while the keys are stored in ASCII order. Each key includes a pointer to the associated data record. Records may be retrieved in random order (using the key field) or in key-sequential (ASCII) order.

ODBC data sources include Microsoft Access, SQL Server, Dbase, Microsoft FoxPro, Microsoft Excel, Paradox, Oracle, and other databases that are compatible with the Open Database Connectivity (ODBC) standard. IB programs interact with these data sources via an ODBC driver (see Chapter ???).

IB includes several input/output statements designed for working with text, sequential, and keyed files. The primary statements are OPEN and CLOSE. The syntax for these is:

```
OPEN (logical-unit-number) filename,DIR=directory,EXCP=statement-label

CLOSE (logical-unit-number)
```

The logical-unit-number (a value from 1 through 49) identifies the file throughout the program. The filename is a string constant or variable that contains the name of the text, sequential, or keyed file to be opened. The directory (an optional parameter) is the 3-character shortcut name for the directory where the file is stored. If this parameter is omitted, the IB runtime system searches through its configured directories in configuration order, attempting to find the specified filename. The statement-label (an

optional parameter) is the place where the program branches in case of a runtime exception.

What could cause an exception when a program tries to open a file? The most likely reason is that the specified filename does not exist, or does not exist in the specified directory. Another possibility is that another IB user or IB program has the file opened and locked, which would cause an exception for another IB program trying to open the file. (Note: While it is possible to lock an entire file, most IB programmers prefer to lock individual records within a file. See below for more information.)

Here's an example:

```
OPEN (1) "TEXTFILE.TXT",DIR="BAS"
```

In English, this is:

> Using logical-unit-number 1, open a text file named "TEXTFILE.TXT" that is stored in the IB directory named "BAS."

To read a record from this data file, use the READ statement. The syntax is:

> READ (logical-unit-number,format-label),EXCP=statement-label

The format-label identifies the FORMAT statement that contains the data item(s) that are in each record of the data file. In our example, each record contains one string field, which we call TEXT$. Here are the FORMAT and READ statements:

```
TEXTLINE: FORMAT TEXT$

READ (1,TEXTLINE),EXCP=AllDone
```

In English, this is:

> Read data from logical-unit-number 1, using the FORMAT statement identified by the format-label TEXTLINE, which contains a string field named TEXT$ (which presumably has been declared earlier in the data section). Remember, the FORMAT statements are written in the data section of the program.
>
> If a runtime exception happens while reading data from the file, the program branches to the AllDone statement-label. The most common exception in this case is when the program finishes reading all of the records and reaches the end of the file.

At this point, an obvious question comes to mind: "When you use the READ statement, which record is retrieved?" The answer to that question depends on the type of file you're reading, and whether or not you are reading in sequential order or using direct access.

Let's start with sequential reading. The above READ statement shows how to read a file in sequential order, starting with the first record and moving towards the last. The first time the program executes a READ statement it retrieves the first record in the file. Each subsequent READ retrieves the next record.

Behind the scenes, the IB runtime system keeps track of your program's current position (offset) within a data file, which it does by using an internal value called a record pointer. When a program opens a data file, the record pointer points to the beginning of the file. After the program reads the first record, the record pointer points to the second record. Subsequent reading by this program moves this pointer to subsequent records. If the program reaches the end of the file and tries to read another record, the IB runtime system returns an exception (one that can be trapped by the EXCP= portion of the READ statement).

If the program closes the file and opens it again, the pointer is positioned at the first record. (There is another way to move the pointer to the beginning of a file. It's the FILE statement, which is covered in Chapter ???.)

It's possible for more than one IB program/user to open a file. Each program has its own record pointer, which means that multiple programs/users can perform independent actions on the same file at the same time.

Direct access can be performed on sequential files and keyed files (not text files). This is explained in later in this chapter and in the next chapter.

Here's a sample program that demonstrates sequential reading from a text file. This program opens "TEXTFILE.TXT" from the "BAS" directory. It reads a record and prints its contents to the virtual printer named LPH, and repeats this process until all of the records have been read/printed. Admittedly, this is a very simple program, but it shows one of the most common algorithms in data processing.

Try running this program, which is named SAMPLE9 (source name is SAMPLE9.MTB), and you'll see the results.

This program demonstrates the DO/LOOP structure, one of the various ways to create a loop in IB. In this structure, the statements between DO and LOOP are repeated until some condition causes the program to exit from the loop. In this program, that condition occurs when the READ statement encounters an exception, at which time the program branches to the AllDone statement-label.

```
! //MTB// Src(SAMPLE9.MTB,BAS) Obj(SAMPLE9,BAS)

! Define the variable

      LENGTH 254
      LOCAL TEXT$

! Define the format for the text file
```

```
      TEXTLINE: FORMAT TEXT$

! Here's the code

      OPEN (1) "TEXTFILE.TXT",DIR="BAS"   ! Open text file
      OPEN (2) "LPH"                      ! Open HTML printer

DO                                        ! Start loop
      READ (1,TEXTLINE),EXCP=AllDone      ! Read text file
      PRINT (2) TEXT$                     ! Print text line
LOOP                                      ! Loop

AllDone:                                  ! When done,
      CLOSE (1)                           ! Close text file
      CLOSE (2)                           ! Close HTML printer
      RUN "QMONITOR"                      ! Exit

END
```

Questions and answers

At this point, you probably have a few questions about files. Here are some answers to the main ones you may be thinking about. Look for further details and statement syntax later in this chapter.

How do you create a data file in IB?

The CREATE statement takes care of this. You need to specify the filename, the IB directory where the file is to be created, the file type, the record size, and (if the file is a keyed file) the key size.

How to you add records to a file in IB?

This is done with the WRITE statement. The syntax varies depending on the type of file you're dealing with (i.e., text, sequential, keyed).

Can you delete records from a file?

You can delete records from a keyed file using the DELETE statement. You can't directly delete records from text and sequential files (although, you can do this indirectly by writing a program that copies the desired records to another file, skips those records you want to delete, and renames/erases the files at the end of the process).

How to you rename a file?

Use the RENAME statement.

How do you erase a file?

Use the ERASE statement.

How do you lock a file?

IB includes a LOCK statement for this purpose. Here's how it works: You open the file and then lock its logical-unit-number. This prevents other IB programs/users from opening that file.

When does the file get unlocked?

Either when the logical-unit-number is closed (with the CLOSE statement) or unlocked (with the UNLOCK statement).

How do you lock a single record?

This is a feature for keyed files and is done with the EXTRACT statement, which has a similar syntax to the READ statement. When a record is extracted, other IB programs/users cannot read or update this record (they will get a runtime exception if they try). A record stays extracted until the program rewrites it, extracts another record from the same file, closes the file, or exits from the IB runtime system.

If there any way for another IB program/user to see an extracted record?

Yes, that's what the INQUIRE statement does. INQUIRE allows the other program/user to view an extracted record, but that's all. They cannot update the record while someone else has it extracted.

Can you read a file in "reverse" order?

Yes. The FILE statement can be used to set the direction of the record pointer. If you set the direction to REVERSE, you can read through a file in reverse sequential order. Of course, you can also set the direction to FORWARD (which is the default direction when you open a file).

Can you move the record pointer to a specific place in a file?

Yes. You can use the FILE statement to move the pointer to the beginning or end of a file. If the file is a keyed file, you can use the POSITION statement to move the pointer to a specific key, use the KEY function to find the value of the next key, and use the PREV function to find the value of the previous key.

Are there any other input/output statements that are used on data files?

INSERT, REWRITE, and UPDATE are special cases of the WRITE statement for keyed files. INSERT adds a new record (and its associated key), but returns an exception if you try to update an existing record. REWRITE updates an existing record (and key), but

returns an exception if you try to add a new record. UPDATE extracts a record and writes it back to the file while updating specific portions of the record.

Formatted data records

The SAMPLE9 program from above retrieves data from a file where each record contains one string field. It is much more typical for each record in a data file to contain several fields. For example, a record is a customer file might include customer number, name, address, city, state (or province), ZIP (or postal) code, phone number, fax number, e-mail address, credit card number, and other necessary fields.

The question then becomes, "How does an IB program retrieve all of these fields from such a data record?" The answer is: "By using the FORMAT statement."

Here's an example that demonstrates the point. Suppose you have a file that contains data about an organization's inventory (e.g., items for sale, items in a warehouse). The following table describes each field in the record of a sample file:

Field	Type	Size
Item number	String	8
Description	String	30
Unit of measure	String	4
Product type	String	4
Base price	Numeric	12.3
Standard cost	Numeric	12.3
Last cost	Numeric	12.3
Average cost	Numeric	12.3
Primary vendor	String	8
Vendor item number	String	18
Minimum quantity	Numeric	12.3
Maximum quantity	Numeric	12.3
Reorder quantity	Numeric	12.3
Taxable (Y/N)	String	1
Commissionable (Y/N)	String	1
Miscellaneous comment	String	20

Here's how these fields could be defined the data section of a program:

```
LENGTH     8     &     LOCAL    INITNO$     ! ITEM NUMBER
LENGTH    30     &     LOCAL    INDESC$     ! ITEM DESCRIPTION
LENGTH     4     &     LOCAL    INUM$       ! UNIT OF MEASURE
LENGTH     4     &     LOCAL    INPTYP$     ! PRODUCT TYPE
LENGTH    12.3   &     LOCAL    INBPRIC     ! BASE PRICE
LENGTH    12.3   &     LOCAL    INSTDCST    ! STANDARD COST
LENGTH    12.3   &     LOCAL    INLSTCST    ! LAST COST
LENGTH    12.3   &     LOCAL    INAVGCST    ! AVERAGE COST
LENGTH     8     &     LOCAL    INVEND$     ! PRIMARY VENDOR
LENGTH    18     &     LOCAL    INVITM$     ! VENDOR ITM NUMBER
```

```
LENGTH    12.3   &     LOCAL    INMINQTY    ! MINIMUM QUANTITY
LENGTH    12.3   &     LOCAL    INMAXQTY    ! MAXIMUM QUANTITY
LENGTH    12.3   &     LOCAL    INREORD     ! REORDER QTY
LENGTH     1     &     LOCAL    INTXBL$     ! TAXABLE (Y/N)
LENGTH     1     &     LOCAL    INCOMM$     ! COMMISSIONABLE(Y/N)
LENGTH    20     &     LOCAL    INMISC$     ! MISC COMMENT
```

Here is the FORMAT statement that describes the record layout in this data file:

```
INV: FORMAT _
     INITNO$     ;_     ! ITEM NUMBER
     INDESC$     ;_     ! ITEM DESCRIPTION
     INUM$       ;_     ! UNIT OF MEASURE
     INPTYP$     ;_     ! PRODUCT TYPE
     INBPRIC     ;_     ! BASE PRICE
     INSTDCST    ;_     ! STANDARD COST
     INLSTCST    ;_     ! LAST COST
     INAVGCST    ;_     ! AVERAGE COST
     INVEND$     ;_     ! PRIMARY VENDOR
     INVITM$     ;_     ! VENDOR ITEM NUMBER
     INMINQTY    ;_     ! MINIMUM QUANTITY
     INMAXQTY    ;_     ! MAXIMUM QUANTITY
     INREORD     ;_     ! REORDER QUANTITY
     INTXBL$     ;_     ! TAXABLE (Y/N)
     INCOMM$     ;_     ! COMMISSIONABLE(Y/N)
     INMISC$            ! MISC COMMENT
```

Notice that the above lines contain comments that help describe the variables when they are defined and when they used in the FORMAT statement.

Now, it's pretty easy to write a program that reads data from the inventory file. The program starts by opening the data file, which in this example is named INVSEQ (it's a sequential file). Here's the statement:

```
OPEN (1) "INVSEQ",DIR="BAS"
```

Next, the program reads a record from the file, as follows:

```
READ (1,INV),EXCP=AllDone
```

In English, this is:

> Read a record from the file opened on logical-unit-number 1, using the FORMAT statement identified by the INV label. If a runtime exception is encountered, branch to the AllDone statement label.

Now, let's print some of the fields from the inventory record. Here's a sample FORMAT statement that shows how:

```
PrintLine: FORMAT _
      INITNO$,@(0);_
```

```
      INDESC$,@(10);_
      INUM$,@(45);_
      INPTYP$,@(52);_
      INMINQTY,@(58);_
      INMAXQTY,@(72)
```

In English, this is:

> Item number at position 0, description at position 10, unit of measure at position 45, product type at position 52, minimum quantity at position 58, and maximum quantity at position 72.

Of course, the FORMAT statement by itself is inert. The following PRINT statement does the actual printing:

```
PRINT (1,PrintLine)
```

Next, we can create a loop that reads and prints all of the records in the file (there are only 50 records in the INVSEQ file, so no need to worry about a long printout).

Another good idea might be to add a heading line that prints before the data from the file. Here's an example of the FORMAT, OPEN and PRINT statements:

```
PrintHeader: FORMAT _
      "Item #",@(0);_
      "Description",@(10);_
      "Units",@(43);_
      "Type",@(52);_
      "Minimum Qty",@(60);_
      "Maximum Qty",@(74)
.
.
.
      OPEN (2) "LPH"                        ! Open HTML printer
      PRINT (2,PrintHeader)                 ! Print heading
```

All of these ideas are demonstrated in the SAMPLE10 program (source name is SAMPLE10.MTB). Take a look at the source code and try running the program.

Although not demonstrated in the sample programs, it's possible to read a specific record from the sequential file (this is called "direct access"). To do this, include an index as part of the READ (or associated input) statement. The index in this case would be a specific record number (e.g., the number 35 would represent the 35th record in the file).

Example:

```
      READ (1,INVSEQ) KEY=35
```

The direct access to a record moves the record pointer, so that subsequent sequential access would take place from the new record pointer location, rather than from the beginning of the file.

Include files

This is a good time to tell mention a feature that saves time when you are writing IB programs. That feature is the INCLUDE statement, which merges the contents of another source file into your program as its being compiled.

For example, suppose your source program contained the following INCLUDE statement:

```
.
.
.
INCLUDE "ABC123"
.
.
.
```

When your program is compiled, the contents of the "ABC123" source file are compiled into the object code at the exact place where the INCLUDE statement is written.

To demonstrate this feature, we took the data section and FORMAT statement (showing the record layout in the inventory file) from the SAMPLE10 program and placed them into a file named "INV.INC".

Look at the source code for SAMPLE11 (source name is SAMPLE11.MTB), and you'll see the following line:

```
INCLUDE "INV.INC"
```

When SAMPLE11.MTB is compiled, the contents of "INV.INC" are merged into the object code.

Now, since it's very likely that you will write other programs that want to use the inventory file, you can simply add an INCLUDE statement to those other programs. You don't need to redefine the fields and record layout each time. Obviously, it makes sense to build a library of include files, one for each data file in your IB database.

Even though the above example shows an include file with the data section and FORMAT statement, include files may also contain executable code. Thus, you can build a library of common executable routines and let the IB compiler merge them into your object programs.

Here's another interesting note. You can nest INCLUDE statements. In other words, an include file can contain INCLUDE statements. We leave it to you to think of creative ways to use this feature.

Overview of keyed data files

One of the most valuable features in InternetBasic is its ability to interface with keyed data files. Keyed files provide direct access to data records. This allows an IB program to retrieve, update, or delete one record at a time. As you shall see, access speeds are very fast.

For example, suppose you have a keyed file that contains data about inventory items for a company. The most likely key for this file would be the inventory part number. When an IB program writes a record to the file, it also writes the key to an index file. Later, when an IB program retrieves a record from this file, it can use the key to directly access a specific record. (Alternatively, the program can read through the files in key-sequential order.)

Behind the scenes, when you create a keyed file, the IB file system creates two Windows files, one for the data records and the other for an index of the key values. When your program writes or deletes a record in a keyed file, the IB runtime system automatically updates the index file. The index file includes the key values and the associated pointers to the data records. The index file, also referred to as the "key tree," is organized in tree format, which allows for very fast searching and updating.

The following sample program uses a keyed file named INVDATA. If you look at the folder where this file is stored (using Windows Explorer), you will see two file names: INVDATA and INVDATA.I00. The first file contains the data records, while the I00 file contains the key tree. Your IB program does not need to access or maintain the I00 file separately; this is done automatically by the IB runtime/file system when your program writes or deletes a record in the INVDATA file.

The SAMPLE12 program (source name is SAMLEE12) demonstrates how to read a keyed file in key-sequential order, and is very similar to the previous programs in this chapter. INVDATA is a keyed file that contains inventory data. There are 1,000 records in this file. Rather than processing the entire file, the sample program reads and prints the first 50 keys. Here's the code that makes this happen:

```
OPEN (1) "INVDATA",DIR="BAS"        ! Open file
OPEN (2) "LPH"                      ! Open HTML printer

PRINT (2,PrintHeader)               ! Print heading

FOR I = 1 TO 50                     ! Start loop
  READ (1,INV),EXCP=AllDone         ! Read file
  PRINT (2,PrintLine)               ! Print line
NEXT I                              ! Loop
```

```
AllDone:                                    ! When done,
      CLOSE (1)                             ! Close text file
      CLOSE (2)                             ! Close HTML printer
      STOP                                  ! Stop the program
```

When you run this program, you won't notice any difference from the previous program (the ones that read the sequential file). That's because the sample program is reading in key-sequential order. The real power of a keyed file is directly accessing a specific key/record. This feature is demonstrated in the inquiry and maintenance programs in the next chapter.

Application notes

When an IB program opens a keyed file, the IB runtime system assigns a unique file pointer to the file. Each user or program opening the file is assigned a unique pointer, allowing multiple IB users to access data from the same file at the same time. To avoid data integrity problems when more than one user is accessing a file, IB provides a record locking mechanism. The EXTRACT statement reads and locks individual data records (it's syntax is the same as READ).

As a user accesses records in a keyed file, the file pointer keeps track of the current location. When the file is opened, the pointer is located at the first key (lowest ASCII order). If the records are retrieved without an index, the pointer moves along in ascending ASCII order of the key values (i.e., in key-sequential order).

When a record is accessed directly using an index, the pointer is moved to the associated position. If no record is found matching the specified index, an exception occurs, but the pointer is located at the index following the requested index value. From that point, the program could retrieve records in sequence (without an index), or retrieve another record from the file using another index value.

When a record is deleted from a keyed file, the key is removed from the key tree and the record space is reused by the file to store subsequent records.

A special type of keyed file can be used to create secondary indexes for any keyed file. A **key-only file** is a keyed file with a record size of zero bytes and a specified key size from 1 to 64 bytes. Typically, a key-only file would contain a composite key value made up of two or more fields from a related keyed file.

For example, suppose a keyed file contains customer names and addresses. Typically, customer number would be the key this file. A secondary key could be created for this file and saved in a separate key-only file. The secondary key could be composed of customer name and customer number (concatenated into a single key value). Thus, with the appropriate programming, an application could access the customer file via either the primary key (customer number) or the secondary key (customer name).

Please note that application programs must maintain the contents of key-only files. Likewise, application programs must provide the required lookup routines (i.e., routines that search through the key-only file, find a specific value, and then use a portion of that value to serve as the primary key).

Summary of file-related statements

The following information is a summary of the file-related statements in InternetBasic. For complete documentation, including coding examples, see the InternetBasic web site (www.internetbasic.com).

CREATE

To create an IB data file, use the CREATE statement. For sequential and keyed files, the syntax is:

```
CREATE filename, record-size, file-type, [key-size,] DIR=directory
[,EXCP=statement-label]
```

Where:

- **filename** is the name of the data file (a string in 8.3 format)
- **record-size** is a value up to 1,024 bytes
- **file-type** is S for sequential files and K for keyed files
- **key-size** (for keyed files only) is a value from 1 to 64 bytes
- **directory** is the 3-character shortcut for the path where the file is to be created (these shortcuts are assigned when the directory is created and configured)
- **statement-label** is the statement where the program branches if there is a runtime exception when this statement is executed (note: the EXCP= parameter is available for all file-related statements, and is always an optional parameter)

For text files, the syntax is:

```
CREATE text-file-name, DIR=DIR=directory [,EXCP=statement-label]
```

ERASE

To erase an IB data file, use the ERASE statement. The syntax is:

```
ERASE filename [,DIR=directory] [,EXCP=statement-label]
```

RENAME

To rename an IB data file, use the RENAME statement. The syntax is:

```
RENAME filename-1 [,DIR=directory-1] ,filename-2 [,DIR=directory-2]
[,EXCP=statement-label]
```

Where **filename-1** and **directory-1** describe the original name, and **filename-2** and **directory-2** describe the renamed file.

OPEN

To open an IB file, use the OPEN statement. The syntax is:

```
OPEN (lun) filename [,DIR=directory] [,EXCP=statement-label]
```

(Note: “lun” is an abbreviation for logical unit number.)

CLOSE

To close an IB file, use the CLOSE statement. The syntax is:

```
CLOSE (lun)
```

To close all logical unit numbers (0 through 49), use the following syntax:

```
CLOSE
```

Note that this version of the CLOSE statement closes logical unit number 0, which is the logical unit number that is used for input/output with the IB client window. To re-open logical unit number 0 following the closure of all logical unit numbers, use the following statement:

```
OPEN (0) TERM$
```

TERM$ is a system variable containing the number of your IB client window. See Chapter ??? for more information about system variables.

LOCK

The LOCK statement prevents another IB program/user from opening a file that your program has opened. To lock a file, open it and then use the LOCK statement. The syntax is:

```
LOCK (lun) [,EXCP=statement-label]
```

UNLOCK

To remove the lock from an open file, use the UNLOCK statement. The syntax is:

```
UNLOCK (lun) [,EXCP=statement-label]
```

WRITE

To write data records to an IB file. Use the WRITE statement. For text files and sequential files, the syntax is:

```
WRITE (lun, format-statement-label) [,EXCP=statement-label]
```

Where **format-statement-label** is the label of the FORMAT statement that describes the record layout in the file.

For keyed files, the syntax is:

```
WRITE (lun, format-statement-label) KEY=index [,EXCP=statement-label]
```

Where **index** is a string that represents the key (index) for the record being written.

INSERT

The INSERT statement is an alternative way to write a record to a keyed file. The syntax is:

```
INSERT (lun, format-statement-label) KEY=index [,EXCP=statement-label]
```

If the specified index already exists in the key tree, the INSERT statement encounters a runtime exception. Thus, this statement prevents a program from overwriting an existing key/record.

REWRITE

The REWRITE statement is another alternative way to write a record to a keyed file. The syntax is:

```
REWRITE (lun, format-statement-label) KEY=index [,EXCP=statement-label]
```

If the specified index does **not** exist in the key tree, the REWRITE statement encounters a runtime exception, making this statement the counterpart of the INSERT statement.

READ

There are several statements that read records from IB files. One of those is the READ statement. For non-indexed reading (i.e., reading records from a text file, sequential file, or keyed file in key-sequential order), the syntax is:

```
READ (lun, format-statement-label) [,EXCP=statement-label]
```

For indexed reading, the syntax is:

```
READ (lun, format-statement-label) KEY=index [,EXCP=statement-label]
```

Where **index** is the key in a keyed file (string value) or the record number in a sequential file (numeric value).

If the READ statement encounters an extracted (locked) record, it will generate a runtime exception.

EXTRACT

The EXTRACT statement reads and locks a record so another IB programs/user cannot update it. This statement is valid for keyed and sequential files only. The syntax is:

```
EXTRACT (lun, format-statement-label) KEY=index [,EXCP=statement-label]
```

An extracted record stays extracted until the program (1) reads or extracts another record using the same lun, (2) writes the extracted record back to the file, or (3) closes the file.

Unlike the READ statement, EXTRACT does not automatically advance the file pointer to the next key in the file. This allows a program to extract a record, modify the data, and write it back to the file without specifying a key index value on the WRITE statement.

INQUIRE

The INQUIRE statement works just like a READ statement, but does not encounter an exception when attempting to read an extracted record. Thus, the INQUIRE statement can be used to retrieve a record that is locked by another IB program/user. The syntax is:

```
INQUIRE (lun, format-statement-label) KEY=index [,EXCP=statement-label]
```

UPDATE

The UPDATE statement modifies only specified fields in a data record of a file. The syntax is:

```
UPDATE (lun, format-statement-label) KEY=index [,EXCP=statement-label]
```

The UPDATE statement extracts the specified record (placing it in the user's buffer), moves the field(s) being updated into the correct position(s) in the user's buffer (as specified with a FORMAT statement), and rewrites the data in the directory with an unindexed WRITE statement.

Note: Because the record is temporarily extracted during an UPDATE operation, other programs that access the same file must be prepared for the possibility of encountering an extract exception.

DELETE

To delete a record from a keyed file, use the DELETE statement. The syntax is:

```
DELETE (lun) KEY=index [,EXCP=statement-label]
```

When the DELETE statement is executed, it automatically advances the file pointer to the next available record in sequential (key) order, marks the record for deletion, and removes the specified key from the key list.

FIRST

To ascertain the value of the first key in a keyed file, use the FIRST function. The first key is the lowest ASCII valued key contained in the key tree. The syntax is:

```
FIRST(lun [,EXCP=statement-label])
```

For example:

```
OPEN (1) "INVDATA",DIR="BAS"
A$ = FIRST(1,EXCP=ReadException)
```

LAST

To ascertain the value of the last key in a keyed file, use the LAST function. The syntax is:

```
LAST(lun [,EXCP=statement-label])
```

For example:

```
OPEN (1) "INVDATA",DIR="BAS"
A$ = LAST(1,EXCP=ReadException)
```

KEY

To ascertain the value of the next key in a keyed file (based on the location of the file pointer), use the KEY function. The syntax is:

```
KEY(lun [,EXCP=statement-label])
```

PREV

To ascertain the value of the previous key in a keyed file, use the PREV function. The syntax is:

```
PREV(lun [,EXCP=statement-label])
```

POSITION

To move the file pointer to a specific key in a keyed file, use the POSITION statement. The syntax is:

```
POSITION (lun) KEY=argument [,EXCP=statement label]
```

Note: If the specified key is not found, the file pointer is moved to the next higher key in ascending ASCII order.

FILE

The FILE statement serves several purposes, one of which is moving the file pointer to the beginning of the file (BOF) or end of the file (EOF). The syntax is:

```
FILE (lun) POS=BOF            ! move to beginning of file
FILE (lun) POS=EOF            ! move to end of file
```

The FILE statement also sets the direction in which the file pointer moves. The syntax is:

```
FILE (lun) FORWARD            ! move pointer in forward order
FILE (lun) REVERSE            ! move pointer in reverse order
```

Note: When a file is opened, the direction of the pointer is forward order (first through last record in a text or sequential, low through high ASCII order of keys in a key tree).

Summary

Believe it or not, we're not done with the discussion of files. While this chapter covers most of the necessary background information, it's now time to put that information into effect. The next chapter does just that by providing examples of inquiry and maintenance programs, two of the most useful applications around.