Chapter 5 – Inquiry and Maintenance Programs

So far we have we covered many of the essential elements in InternetBasic. In this chapter, we present two very useful applications that utilize those features. Inquiry programs let you retrieve data from a particular data source, while maintenance programs let you change the contents of that data source by adding, changing, or deleting records.

The sample programs in this chapter work with the INVDATA file. This file was introduced in Chapter 4. INVDATA is a keyed file containing 1,000 records of inventory data. The key to this file is the item number, an 8-byte string. Each record contains 16 data fields, including item number, description, quantity data, cost data, and several flags – for a total of 192 bytes per record.

The data definitions and record layout for INVDATA are stored in an include file named INV.INC.

Inquiry program

The SAMPLE13 program (source name is SAMPLE13.MTB) is an inquiry program for the INVDATA file. Here's the screen display:

```
INVENTORY FILE INQUIRY

Enter item number or (F)irst,(L)ast,(N)ext,(P)revious: ________

 1  Item number
 2  Item description
 3  Unit of measure
 4  Product type

 5  Base price
 6  Standard cost
 7  Last cost
 8  Average cost

 9  Primary vendor
10  Vendor item number

11  Minimum quantity
12  Maximum quantity
13  Reorder quantity

14  Taxable (Y/N)
15  Commissionable (Y/N)
16  Miscellaneous comment
```

Notice the option line towards the top of the screen. According to the prompt, the user can enter an item number or one of the four options (F, L, N, or P). Though not stated, the user can also enter a null (by pressing the Enter key without having first typed any data into the data entry area), in which case this program stops.

Here's how the program handles each of the inputs. If the input is an item number, the program attempts to read that key from the INVDATA file. If the read is successful, the data fields are displayed on the screen. For example, here's the display for item number LAB-774:

```
INVENTORY FILE INQUIRY

Enter item number or (F)irst,(L)ast,(N)ext,(P)revious:

 1  Item number             LAB-774
 2  Item description        Liquid acrylic binding
 3  Unit of measure         GR
 4  Product type            ADRL

 5  Base price                     36.000
 6  Standard cost                  23.400
 7  Last cost                      24.570
 8  Average cost                   22.932

 9  Primary vendor          LITE
10  Vendor item number      L-3313

11  Minimum quantity                4.000
12  Maximum quantity               14.000
13  Reorder quantity                2.000

14  Taxable (Y/N)           Y
15  Commissionable (Y/N)    N
16  Miscellaneous comment   Use immediately
```

The read could also encounter a runtime exception. There are several possible explanations for an exception at this point. The two most likely reasons are:

- The input value does not exist in the file
- The input value is beyond the end of the file (i.e., its value is higher in the ASCII sequence than the last key in the key tree)
- While the pointer is positioned at the first key, the user could select the previous key, which would result in an "end of file" condition.

There's another possibility, but only if there is another IB program accessing the INVDATA file while this program is. In that case, it's possible for a specific record to be extracted (locked) by the other program. In order to avoid that particular exception at all, this sample program uses the INQUIRE statement rather than the READ statement.

Here's how the program deals with the two most likely exceptions:

- If the key does not exist, the program reads the next key in the file (in key-sequential order from where the file pointer was positioned following the initial INQUIRE).

- If the user chooses a key past the end of the file, the program reads the last key in the key tree. If the user chooses the N option when the pointer is positioned at the last key, the program retrieves the first key.
- If the user chooses the previous key from the beginning of the file, the program retrieves the last key in the file.

The program then displays the data fields.

Thus, the user can enter a complete item number or a partial item number and rely on the program to find an exact match or the next closest match. This provides a simple and effective way to find a given record in this file.

The four additional options give the user a way to position the pointer at the first or last key, or move through the file in forward or reverse order. Here's how these options are processed:

Option	Program actions
F	perform a FIRST function, then INQUIRE
L	perform a LAST function, then INQUIRE
N	perform a KEY function, then INQUIRE
P	perform a PREV function, then INQUIRE

Of course, each of these functions needs to be prepared for a possible runtime exception. It's no coincidence that the most likely exceptions are the same as for the INQUIRE statement (i.e., the key doesn't exist, or the pointer is positioned past the end of the key tree).

To provide helpful information for the user, this program displays several status messages. They are:

** First record **
** Last record **
** Closest match **

The specific message depends on the user's input and what's in the data file. No message is displayed when the user enters an item number that exactly matches a key in the file.

Let's take a look at the code for the sample program. Here's the data section from the sample program:

```
INCLUDE "INV.INC"

LENGTH 8
LOCAL OPTION$,DATA$

LENGTH 20
LOCAL MSG$
```

The first statement merges the contents of the include file into program being compiled. The other statements define local string variables named OPTION$, DATA$, and MSG$.

Here's the first part of the code section:

```
      CLEAR                                ! Clear all data fields

      OPEN (1) "INVDATA",DIR="BAS",EXCP=OpenExcp      ! Open the data
file

      ! Clear the screen and display the prompts

      Print (CS);_
            "INVENTORY FILE INQUIRY",@(0,0);_
            "Enter item number or
(F)irst,(L)ast,(N)ext,(P)revious:",@(0,2);_
            " 1  Item number",      @(0,4);_
              " 2  Item description",     @(0,5);_
              " 3  Unit of measure",      @(0,6);_
              " 4  Product type",   @(0,7);_
              " 5  Base price",     @(0,9);_
              " 6  Standard cost",  @(0,10);_
              " 7  Last cost",      @(0,11);_
              " 8  Average cost",   @(0,12);_
              " 9  Primary vendor", @(0,14);_
              "10  Vendor item number",@(0,15);_
              "11  Minimum quantity",     @(0,17);_
              "12  Maximum quantity",     @(0,18);_
              "13  Reorder quantity",     @(0,19);_
              "14  Taxable (Y/N)",  @(0,21);_
              "15  Commissionable (Y/N)",@(0,22);_
              "16  Miscellaneous comment",@(0,23)
```

These statements initialize the variables, open the data file, and display the prompts on the IB client screen.

Notice the exception path on the OPEN statement. If a runtime exception occurs on this statement, the program branches to the OpenExcp label. The code for this exception processing routine is contained at the end of the source program.

Here's the next portion of the program:

```
Top:  MSG$ = ""                            ! Set message to null

      Print (BF),@(55,2)                   ! Display blanks on option
line

      Input @(55,2),OPTION$                ! Input item number or option

      IF OPTION$ = "" THEN                 ! If null input,
        CLOSE (1)                          ! Close data file
        STOP                               ! Stop
      ENDIF
```

The first statement sets the MSG$ variable to null. While this is redundant on the first time through this part of the program, it becomes necessary when the program returns to the Top statement label.

The second statement displays blank spaces on the option line, starting a position 55,2. The (BF) control code "blank fills" from the specified position to the end of the line. This is done to clear the previous option from the screen when the program returns to the Top statement label.

The INPUT statement lets the user enter an item number or an option.

Immediately after the INPUT statement, the program checks for a null input, in which case the file is closed and the program is stopped.

The code continues…

```
OPTION$ = STRIP(UCASE(OPTION$))      ! Strip blanks and convert
                                     ! OPTION$ to upper case

SELECT CASE OPTION$                  ! Select the option
CASE "F"
  DATA$ = FIRST(1,EXCP=READEXCP)     ! F = get the first key
CASE "L"
  DATA$ = LAST(1,EXCP=READEXCP)      ! L = get the last key
CASE "N"
  DATA$ = KEY(1,EXCP=READEXCP)       ! N = get the next key
CASE "P"
  DATA$ = PREV(1,EXCP=READEXCP)      ! P = get the previous key
CASE ELSE
  DATA$ = OPTION$                    ! Otherwise, OPTION$ is a key
ENDSELECT
```

The first line removes leading and trailing blanks from the input string (if any), and converts it to upper case (needed because the item numbers start with upper case characters).

Next, the SELECT structure determines which key to read. The result of these lines is that the DATA$ variable contains one of five possible values: the first key, the last key, the next key, the previous key, or the user's input value.

Because this program uses one variable (OPTION$) for two purposes (an item number or an option code), the user cannot use the letters F, L, N, or P as partial item numbers. For example, if the user types G (and nothing else) at the option prompt, the program will find the first item number starting with the letter G (or the next highest value in ASCII order). But, if the user types F at the option prompt, the program treats it as the "first" option and reads the first key in the file.

Here's the code that retrieves the data from the file and displays the data fields on the screen:

```
ReadRecord:

      INQUIRE (1,INV) KEY=DATA$,EXCP=ReadExcp         ! Read the record


      Print INITNO$,    @(28,4);_          ! Display the fields
            INDESC$,    @(28,5);_
            INUM$,      @(28,6);_
            INPTYP$,    @(28,7);_
            INBPRIC,    @(28,9);_
            INSTDCST,   @(28,10);_
            INLSTCST,   @(28,11);_
            INAVGCST,   @(28,12);_
            INVEND$,    @(28,14);_
            INVITM$,    @(28,15);_
            INMINQTY,   @(28,17);_
            INMAXQTY,   @(28,18);_
            INREORD,    @(28,19);_
            INTXBL$,    @(28,21);_
            INCOMM$,    @(28,22);_
            INMISC$,    @(28,23);_
            "@00@"

            ! Note: When displaying multiple fields with a single PRINT
            ! statement, blanks are stripped from the last field.
            ! To prevent this feature from affecting the last displayed
            ! data field, put an extra null field at the end of the
            ! list of print items, as shown above, using hex 00
            ! ("@00@").
```

Notice that the INQUIRE statement contains an exception path. We will discuss that shortly.

Also notice that the PRINT statement contains an extra null field at the end of the list of print items. This prevents trailing blanks from being stripped from the last string field on the list (see Chapter 2 for more information on this rule).

The next part of the program determines which message is displayed (if any). If the user enters a value equal to the first key in the file, the "** First record **" message is selected. The last key results in the "** Last record **" message.

```
MessageRoutine:

      ! Check to see if we're at the first or last key in the file

      SELECT CASE INITNO$                        ! Select item number
        CASE FIRST(1,EXCP=READEXCP)              ! Check for first key
          MSG$ = "** First record **"            ! Set message
        CASE LAST(1,EXCP=READEXCP)               ! Check for last key
          MSG$ = "** Last record **"             ! Set message
```

```
      ENDSELECT

      ! Display blanks over the previous message, then display the new
message

      Print (BF),@(60,4);_                        ! Blank fill message
line
            MSG$,@(60,4)                          ! Display new message

      GOTO TOP                                    ! Go back to the top
```

In order to clear a previous message before displaying the current one, the program "blank fills" the message line and then displays the MSG$ variable.

The last line of code branches back to the Top statement label, which re-displays the option prompt.

The remainder of the code contains exception processing routines. The first one these is the ReadExcp routine, which contains the code that handles exceptions for the INQUIRE statement and the FIRST, LAST, PREV, and KEY functions:

```
ReadExcp:                                     ! Read exceptions

      ! Test to see if the exception is "key not found"

      IF EXCP = 32 THEN                       ! If key not found
        MSG$ = "** Closest match **"          ! Set message
        DATA$ = KEY(1,EXCP=ReadExcp)          ! Get the next key
        GOTO ReadRecord                       ! Read the record
      ENDIF

      ! Test to see if the exception is "end of file"

      IF EXCP = 2 THEN                        ! If end of file,
        SELECT CASE OPTION$                   ! Which direction?
          CASE "N"                            ! "NEXT" option?
            DATA$ = FIRST(1,EXCP=READEXCP)            ! then get the
first key
          CASE ELSE                           ! Otherwise,
            DATA$ = LAST(1,EXCP=READEXCP) ! then get the last key
        ENDSELECT
        GOTO ReadRecord                       ! Read the record
      ENDIF

      ! Otherwise, fall through to the runtime exception reporting
program

      ERROR
```

This code contains valuable information about how IB determines which exception has occurred. Simply stated, when an exception happens, the IB runtime system assigns a number to the event. The exception numbers are reported to the program via a system variable named EXCP.

Looking at the above code, you'll see that exception number 32 happens when a file input statement (READ, INQUIRE, EXTRACT) attempts to read a key that is not in the key tree. As described earlier, this sample program recovers from that exception by retrieving the next record in the key tree. (It also sets the message to "** Closest match **".)

The other exception that is explicitly tested in this program is 2, which means "end of file." Better stated, it means that the pointer has moved past the highest ASCII ordered key in the key tree. Recovering from this is a bit more complex than the first exception, because the specific type of recovery depends on where the pointer was before the INQUIRE statement was executed, and in which direction the user was attempting to move the pointer. The program determines this and performs a FIRST or LAST function to retrieve the appropriate key.

If the exception is neither 32 nor 2, the program falls through to an ERROR statement, which runs the IB error reporting program. (See the next chapter for more information.)

The final routine in this program is the exception routine for the OPEN statement:

```
OpenExcp:

      ! Perform this routine if there is an exception when opening the
data file

      Print (CS)
      Print "This program was unable to open the INVDATA file.",@(0,0)
      Print "Please check to make sure this file is available.",@(0,2)
      Print @(0,10);"Press ENTER to continue..."
      Input @(30,10),""
      STOP
```

Since this is an exception from which there is no graceful recovery possible, the program displays a message for the operator and then stops.

Maintenance program

The SAMPLE14 program (source name is SAMPLE14.MTB) is a maintenance program for the INVDATA file. The program builds on the previous example by allowing the user to add new records to the data file, change existing records, and delete records.

Here's the top section of the initial screen display:

```
INVENTORY FILE MAINTENANCE

Item number or (F)irst,(L)ast,(N)ext,(P)revious,(A)dd: ________
```

This is the same as the inquiry program, except for the addition of the A option, which allows the user to add a new record.

When the user retrieves an existing inventory record, the option line is replaced with the following line. Notice the C and D options, which allow the user to change or delete the record, respectively.

```
INVENTORY FILE MAINTENANCE

Item number or (F)irst,(L)ast,(N)ext,(P)revious,(A)dd,(C)hange,(D)elete: ______
```

The option line is replaced in three other situations. First, when the user chooses the A option, the option line is:

```
INVENTORY FILE MAINTENANCE

Add new record:
```

When the user chooses the C option, the option line is:

```
INVENTORY FILE MAINTENANCE

Action: (S)ave, (C)ancel, field number 1-16:
```

When the user choose the D option, the option line is:

```
INVENTORY FILE MAINTENANCE

Confirm delete (Y/N)? __
```

Adding a new record

When the user chooses to add a new record (the A option), the program creates an input field next to the "Item number" prompt. The user's input is validated in several ways. First, if the input is null, the program returns to the initial options prompt. If the input matches a key that's already in the INVDATA file, the program displays a message that reads "** Duplicate key **" and returns to the input field for another value (in other words, this program does not permit data entry using a key that's already in the file). Otherwise, the input is accepted as a new item number, and the program continues to input the other fields. Here's a sample of several of these INPUT statements:

```
      Input @(28,5),INDESC$
      Input @(28,6),INUM$
      Input @(28,7),INPTYP$
      Input @(28,9),INBPRIC
      Input @(28,10),INSTDCST
      Input @(28,11),INLSTCST
      Input @(28,12),INAVGCST
      Input @(28,14),INVEND$
      Input @(28,15),INVITM$
      Input @(28,17),INMINQTY
      Input @(28,18),INMAXQTY
      Input @(28,19),INREORD
```

Two of the input fields require special processing, because they need to be validated. These are the "taxable" and "commissionable" fields (fields 14 and 15 on the inquiry screen). The validation is handled in a subroutine, which is executed via a GOSUB statement, as follows:

```
      GOSUB Taxable
```

Here's the subroutine:

```
Taxable:
      Input @(28,21),INTXBL$

            INTXBL$ = UCASE(INTXBL$)
            IF INTXBL$ NE "Y" AND INTXBL$ NE "N" THEN
              Print "** Must be Y or N **",@(30,21)
              GOTO Taxable
            ENDIF
            Print INTXBL$,@(28,21)
            Print (BF),@(30,21)

      RETURN
```

As you can see, this subroutine inputs the data and checks to make sure it is "Y" or "N." If it isn't, the program displays a message ("** Must by Y or N **") and repeats the input step. Once a valid value is entered, the program re-displays it (after having converted the input to upper case), blank fills the message line, and returns to the statement following the GOSUB statement.

A similar GOSUB statement and subroutine exist for the "commissionable" field.

After all of the new fields have been entered, the program writes the new record to the INVDATA file and returns to the initial options line.

Changing an existing record

When the user elects to change an existing record (the C option), the program branches to an update routine (in fact, the statement label is UpdateRoutine). Here the "(S)ave, (C)ancel, field number 1-16:" option is displayed.

If the user chooses S (to save the record), the record is written back to the file (following some important validation steps which are explained below).

If the user chooses C (to cancel the changes), the program re-reads the original record and displays the original fields. In other words, it cancels any changes that were made by the user.

The numbers 1 through 16 correspond with the fields on the screen. Here's a segment of the code that updates these fields:

```
        CASE "4"                          ! Input field 4
          Input @(28,7),INPTYP$
        CASE "5"                          ! Input field 5
          Input @(28,9),INBPRIC
        CASE "6"                          ! Input field 6
          Input @(28,10),INSTDCST
        CASE "7"                          ! Input field 7
          Input @(28,11),INLSTCST
        CASE "8"                          ! Input field 8
          Input @(28,12),INAVGCST
        CASE "9"                          ! Input field 9
          Input @(28,14),INVEND$
        CASE "10"                         ! Input field 10
          Input @(28,15),INVITM$
```

Obviously, some code is missing, namely the SELECT CASE statement at the beginning of the structure, some of the other CASE values, and the ENDSELECT statement at the end, but we think this segment demonstrates the way that the fields are updated.

The "taxable" and "commissionable" fields are validated using the subroutines described above. Here's the CASE code that calls those subroutines:

```
        CASE "14"                         ! Input field 14
          GOSUB Taxable
        CASE "15"                         ! Input field 15
          GOSUB Commission
```

Validation for field 1 (the item number) is the most involved. The program allows the user to change the item number, but does not allow it to be changed to an existing key. The program also does not allow a null item number to be entered.

Validation for the S option also involves several steps. First, the program rejects null item numbers. Second, the program checks to see if the item number was changed. If so, the old key/record is deleted from the file before the new one is written.

Along the way, various messages are set, displayed, and cleared. Also, a flag field is employed to determine which option message to display. If the flag equals one value, a certain message is displayed; if it equals another value, another message is displayed. See the source code for details.

Deleting a record

When the user chooses to delete a record (the D option), the program displays the "Confirm delete (Y/N)?" prompt. If the user responds with a Y, the program deletes the record. Any other response simple re-displays the initial option (without deleting the record).

Programming techniques

This sample program introduces a few new features, including the use of the FORMAT statement to display information on the screen. The beginning section of the source program, the data/format section, includes the following statement:

```
MSG: FORMAT MSG$,@(60,4)
```

In English, this is:

> When called by a PRINT statement, this FORMAT statement displays the MSG$ variable at position 60,4.

Here's an example of how this FORMAT statement is called:

```
MSG$ = "** Last record **"
Print (0,MSG)
```

In English, this is:

> Set the MSG$ variable to "** Last record **", then print the FORMAT statement label MSG on logical unit number 0, which is the IB client window.

You will see this technique throughout the sample program.

You will also see a related statement (and FORMAT statement) that "cleans up" the message line. Here are the statements:

```
ClearMessage: FORMAT (BF),@(60,4)
.
.
.
Print (0,ClearMessage)
```

In English, this is:

> Blank fill the message line, starting at position 60,4.

Here's how the "clean up" process works:

- MSG$ is set to a certain value or to null
- The ClearMessage format is printed
- The MSG format is printed

The inventory data fields are also displayed using a FORMAT statement:

```
DisplayData: FORMAT _
             INITNO$,    @(28,4);_
             INDESC$,    @(28,5);_
             INUM$,      @(28,6);_
             INPTYP$,    @(28,7);_
             INBPRIC,    @(28,9),(BZ);_
```

```
            INSTDCST,   @(28,10),(BZ);_
            INLSTCST,   @(28,11),(BZ);_
            INAVGCST,   @(28,12),(BZ);_
            INVEND$,    @(28,14);_
            INVITM$,    @(28,15);_
            INMINQTY,   @(28,17),(BZ);_
            INMAXQTY,   @(28,18),(BZ);_
            INREORD,    @(28,19),(BZ);_
            INTXBL$,    @(28,21);_
            INCOMM$,    @(28,22);_
            INMISC$,    @(28,23);_
            "@00@"
```

A couple of items need some explanation:

- Notice the (BZ) control code that is used with the numeric fields. This modifier means "blank if zero." This modifier may only be used within a FORMAT statement.
- The trailing "@00@" ensures that trailing blanks are not stripped from the INMISC$ field.

Of course, this FORMAT statement is accompanied by a matching PRINT statement, which it is:

```
      Print (0,DisplayData)
```

This technique is used because there are several places in the code where the data fields are displayed. Rather than write a long PRINT statement each time, it's easier to code a short PRINT statement that refers to a long FORMAT statement.

This program includes a routine for handling runtime exceptions that occur when a record is written to the INVDATA file. As the following code shows, the only exception that is tested for is 33, which means that the record is locked. If this happens, the program displays a message to that effect, and then returns to the initial options prompt. If any other exception occurs during a write, the program runs the IB exception reporting program.

```
WriteExcp:

      IF EXCP = 33 THEN                          ! Record locked?

        Print (0,Blanks)                         ! Clear the option line

        ! Display a warning message

        Print "That record is locked. Your changes were not
saved.",@(0,2)
        Print " Press Enter to continue."

        Input @(79,2),""                         ! Dummy input
        FLAG$ = "Y"                              ! Set FLAG$ to "Y"
        GOTO Top                                 ! Back to the top
      ENDIF
```

```
      ! Otherwise, fall through to the runtime exception reporting
program

      ERROR
```

Summary

This chapter includes many practical examples of how to work with data files. As you will find out when you write your own IB applications, keyed files provide an excellent combination of features. Not only are they easy to work with from a coding standpoint, they provide very fast access speeds.

In the next chapter, we will take an in-depth look at runtime exceptions.