Chapter 7 – Control Structures

This chapter presents some reference information about the control structures in InternetBasic. While you've already seen many of these in action in the sample programs, this chapter provides the formal rules for these structures.

IF/THEN

There are two forms of the IF/THEN statement, the simple form and the form that involves the ELSE clause. Here's the simple form:

```
IF relational-expression [THEN] conditional-statement(s)
```

Notice that the word "THEN" is optional. This statement evaluates the relational-expression (for example "A GT 25," which means "if the value of A is greater than 25"). If this expression is true, the program executes the conditional-statement(s). (Notes: If multiple conditional-statements are used, they must be separated by the ampersand character (&). If these multiple statements exceed the maximum line length, the underscore character (_) may be used to continue the statement(s) to additional source lines.

This statement may be written as a structure, which looks like this:

```
IF relational-expression [THEN]
conditional-statement(s)
ENDIF
```

Notice that the conditional-statement(s) are not written on the same line as the IF statement, but on subsequent lines. Also notice that the structure ends with the ENDIF statement.

The advanced form of the IF/THEN structure looks like this:

```
IF relational-expression [THEN]
conditional-statement(s)
ELSE
conditional-statement(s)
ENDIF
```

This structure evaluates the relational-expression. If it is true, the program executes the subsequent conditional-statement(s). However, if it is not true, the program executes the conditional-statement(s) following the word "ELSE."

Note that this structure allows for nested IF/THEN statements (i.e., IF/THEN statements contained within the conditional-statement(s).)

A relational-expression consists of three elements: a variable name, a relational operator, and a comparison value. For example, to determine if a variable named CRLIMIT has a value greater than 500, you would use the following relational-expression:

```
CRLIMIT GT 500
```

In relational-expressions, the variable name always appears first, followed by a relational-operator from the following list:

Relational operator	Syntax
Equal to	EQ or =
Not equal to	NE or NOT=
Greater than	GT or >
Greater than or equal to	GE or >=
Less than	LT or <
Less than or equal to	LE or <=

The comparison value is the third item in a relational-expression. It may be a constant or variable and must agree in data type with the variable name (i.e., strings must be compared to strings, numbers to numbers).

Compound relational-expressions may be defined with the AND and OR parameters. These link individual relational-expressions; the outcome is based on the joint outcome as follows:

Parameter	Conditions	Outcome
AND	both conditions are true	result is true
AND	at least one condition is false	result is false
OR	either condition is true	result is true
OR	both conditions are false	result is false

In compound expressions containing AND and OR, the AND parameter takes precedence (much like multiplication and division take precedence over addition and subtraction in a compound numeric expression).

SELECT

The select structure, sometimes called a “case structure,” evaluates one or more conditions and takes action based on the value of the expression being evaluated. Here is the syntax:

```
SELECT CASE test expression

  CASE expression list 1
       [statement block 1]

  CASE expression list 2
       [statement block 2]
```

```
  .
  .
  .
  CASE expression list n
       [statement block n]

  [CASE ANY MATCH]
       [statement block]

  [CASE ELSE]
       [statement block]

ENDSELECT
```

The indentation shown above is not required, but helps clarity the statements in the structure.

The test expression may be any numeric or string expression.

The expression list elements may have any of the following forms:

1. expression or expression list (expression list elements are separated with a semicolon)
2. FROM expression TO expression
3. FROM expression THRU expression
4. IS relational-operator expression

The expressions and operators in the expression list must be compatible with the test expression. The word "IS" must precede any relational operators. The word "FROM" must precede any range specification. The word "TO" means up to but not including. The word "THRU" means up to and including.

The selected expression is tested in sequence for each of the CASE clauses. Upon reaching the first successful test, the statement block following the CASE is executed, and control is then passed to one of several places:

1. If the structure does not contain a CASE ANY MATCH clause or a CASE ELSE clause (both clauses are optional), control is passed to the statement following the ENDSELECT statement.
2. If the structure contains a CASE ANY MATCH clause, control is passed to it. Following the execution of the CASE ANY MATCH statement block, control is passed to one of two places:
    a. If the structure does not contain a CASE ELSE clause, control is passed to the statement following the ENDSELECT statement.
    b. If the structure contains a CASE ELSE clause, control is passed to it. The CASE ELSE statement block is executed and control is passed to the statement following the ENDSELECT statement.
3. If the structure contains a CASE ELSE and does not contain a CASE ANY MATCH clause, control is passed to the CASE ELSE clause. The CASE ELSE

statement block is executed and control is passed to the statement following the ENDSELECT statement.

A BREAK statement may be used in the CASE structure. If used, the BREAK statement must be imbedded within an IF/THEN or IF/THEN/ELSE structure. The BREAK statement transfers control to the statement following the ENDSELECT statement.

Here is an example:

```
SELECT CASE TEMP                      ! use the TEMP variable

CASE IS GT 102                        ! a relational operator clause
PRINT (0) "It's over 102 degrees"     ! conditional PRINT

CASE 100;101;102                      ! multiple expressions
PRINT (0) "It's a bit above 100"      ! conditional PRINT

IF TEMP LT 102 BREAK                  ! if true, jump to end
   PRINT (0) " and heading higher"    ! otherwise, PRINT
ENDIF                                 ! end of IF structure

CASE FROM 80 THRU 99                  ! range of values
PRINT (0) "It is mighty warm today"   ! conditional PRINT

CASE ANY MATCH                        ! if any of above match
PRINT (0) "Hope is it warm tomorrow" ! conditional PRINT

CASE ELSE                             ! if none of above match
PRINT (0) "It is below 80 degrees"    ! conditional PRINT

ENDSELECT                             ! end of CASE structure
```

The SELECT structure encompasses several IB keywords, including SELECT, CASE, ENDSELECT, FROM, TO, THRU, IS, ANY, and MATCH.

The expression list elements in the CASE statement may have any of several forms. The first form is simply an expression or constant value. For example:

```
SELECT CASE VALUE           ! test the numeric variable VALUE
CASE 50                     ! if VALUE = 50,
[statement block]           ! execute this code
CASE 75                     ! if VALUE = 75,
[statement block]           ! execute this code
ENDSELECT                   ! end the structure
```

Multiple expression list elements may be included in a single CASE statement. The elements are separated with a semicolon. For example:

```
SELECT CASE VALUE           ! test the numeric variable VALUE
CASE 50;51;52;53            ! if VALUE is 50, 51, 52 or 53,
[statement block]           ! execute this code
ENDSELECT                   ! end the structure
```

The CASE statement may also indicate a range of values, as follows:

```
CASE expression TO expression         ! up to but not including
CASE FROM expression THRU expression  ! up to and including
```

An example of this is:

```
SELECT CASE VALUE           ! test the numeric variable VALUE
CASE 50 TO 60               ! if VALUE is 50 to (but not
                            ! including) 60,
[statement block]           ! then execute this code
                            !
CASE FROM 80 THRU 90        ! if VALUE is 80 to (and including)
[statement block]           ! 90, then execute this code
                            !
ENDSELECT                   ! end the structure
```

The CASE statement may also include a relational-operator clause, as follows:

```
CASE IS relational-operator expression
```

An example of this is:

```
SELECT CASE VALUE           ! test the numeric variable VALUE
CASE IS GT 102              ! if VALUE is greater than 102,
[statement block]           ! then execute this code
ENDSELECT                   ! end the structure
```

The CASE ANY MATCH clause is executed if one of the specific CASE values is matched. This statement is optional, but if included in the structure, must be written after the individual CASE clauses and before the CASE ELSE clause (if one is present). For example:

```
SELECT CASE VALUE           ! test the numeric variable VALUE
CASE 50                     ! if VALUE = 50,
[statement block]           !         execute this code
CASE 75                     ! if VALUE = 75,
[statement block]           !         execute this code
CASE ANY MATCH              ! if either of above cases is
[statement block]           ! matched,execute this code, too
ENDSELECT                   ! end the structure
```

The CASE ELSE clause may be included in the structure to handle cases that are not matched by any of the individual CASE clauses. This clause is optional, but if included in the structure, must be written as the final clause (just prior to the ENDSELECT statement). For example:

```
SELECT CASE VALUE           ! test the numeric variable VALUE
.
.
.
CASE ELSE                   ! if none of the cases produce a
```

```
[statement block]           ! match, execute this code
ENDSELECT
```

The CASE structure may also include a BREAK statement. A BREAK statement, which must be imbedded within an IF/THEN or IF/THEN/ELSE structure, transfers control to the statement following the ENDSELECT statement. For example:

```
SELECT CASE VALUE           ! test the numeric variable VALUE
.
CASE ________               ! test one of the specific cases
.
   IF ________ THEN BREAK   ! if a condition is true, then break
.
ENDSELECT
```

FOR/NEXT

The FOR and NEXT statements mark the beginning and ending, respectively, of a group of statements that are to be repeated. This structure provides looping capability where the iterations are counted. Here is the syntax:

```
FOR control-variable = start-count TO stop-count [STEP increment]
.
.
.
NEXT control-variable
```

The control-variable is a previously-defined numeric variable with sufficient length and precision to store the values specified by the start-count, stop-count, and increment parameters.

Start-count is the initial value assigned to the control-variable. It may be a numeric constant or numeric variable.

Stop-count is the maximum value assigned to the control-variable. Stop-count may be a numeric constant or numeric variable. When the maximum value has been reached, the statements inside the loop will be executed for a final time and program flow will continue at the statement immediately following the NEXT statement. (The control-variable will retain the maximum value assigned.)

During the looping process, the value of the control-variable is incremented according to the increment parameter. If not specified, the increment is 1. The increment may be a numeric constant or numeric variable.

FOR/NEXT loops may be nested to any level.

Here is an example:

```
SUM = 0                       ! Initialize the sum
FOR ROW = 1 TO 50             ! Start outer loop (count 1 to 50)
```

```
  FOR COLUMN = 1 TO 200        ! Start inner loop (count 1 to 200)
    SUM=SUM+DATA(COLUMN,ROW)   ! Add up array elements
  NEXT COLUMN                  ! Continue inner loop
NEXT ROW                       ! Continue outer loop
```

DO/LOOP

The DO and LOOP statements provide another loop structure. There are three ways to write these statements.

```
DO {WHILE | UNTIL} relational-expression
[conditional statement block]
LOOP
```

```
DO
[statement block]
LOOP {WHILE | UNTIL} relational-expression
```

```
DO
[statement block]
LOOP
```

The DO and LOOP statements execute a block of code repeatedly, either (1) while a specified condition is true, or (2) until a specified condition is true.

The relational expression may be any expression containing relational operations such that it can be evaluated as true or false. (For more information, see the IF/THEN discussion.)

DO/WHILE evaluates the relational expression and executes the statement block if the expression is true. At the LOOP statement, control returns to the expression evaluation. DO/UNTIL functions in the same manner, except that the code is executed if the expression is not true. In both cases, the evaluation of the expression is at the top of the loop.

DO followed by LOOP/WHILE or LOOP/UNTIL causes the statement block to be executed at least once, no matter whether the expression is true or false. The test occurs at the bottom of the loop, and the statement block is executed again if the expression is true, for the WHILE case, or false, for the UNTIL case.

DO/LOOP with no conditional clause should be used with care. The test that determines whether the loop should be executed or not must occur in the code itself.

Example 1:

```
DO WHILE TOTQUAN LT LIMIT
   TOTQUAN=TOTQUAN+QUAN
   GOSUB GETNXT
LOOP
```

Whenever TOTQUAN is greater than or equal to LIMIT, control will pass immediately to the statement following the LOOP statement.

The same result will occur if the first statement is:

```
DO UNTIL TOTQUAN GE LIMIT
```

Example 2:

```
DO
   TOTQUAN=TOTQUAN+QUAN
   GOSUB GETNXT
LOOP WHILE TOTQUAN LT LIMIT

(or)

LOOP UNTIL TOTQUAN GE LIMIT
```

In this example, the statement block is executed at least once.

Example 3:

```
DO
.
IF X=25 [THEN] BREAK
.
LOOP
```

In this example, the loop is terminated if the variable X equals 25.

The DO/LOOP structure encompasses several specialized keywords, including DO, LOOP, WHILE, UNTIL, BREAK, and CONTINUE. Of these, only two are required to define a loop: the DO statement at the beginning and the LOOP statement at the ending.

Two program control statements can be used to modify the statement execution sequence in a DO/LOOP structure. These statements are BREAK and CONTINUE. The BREAK statement stops execution of the conditional statement block within and passes control to the statement following the LOOP statement. For example:

```
DO WHILE I LE 10
.
.
IF A=I [THEN] BREAK      ! jump out of the loop
.
.
LOOP
```

The CONTINUE statement passes control to the LOOP statement in a DO/LOOP structure. For example:

```
DO WHILE A LT 100
.
.
IF A=B
     CONTINUE     ! branch to the LOOP statement
ELSE
.
.
ENDIF
.
.
LOOP
```

Note: The BREAK and CONTINUE statements may also be used in the FOR/NEXT structure. BREAK passes control to the statement following the NEXT statement, while CONTINUE passes control to the NEXT statement.

In addition, the BREAK statement may be used in the SELECT/CASE structure.

Branching statements

There are several branching statements in IB. The unconditional statement is GOTO, which may be written in the following ways:

```
GOTO statement-label
GO TO statement-label
GO statement-label
```

The GOTO statement may also be written as a conditional statement, for example within an IF/THEN statement or CASE statement (see the sample programs for examples).

Another type of conditional branching statement is the ON/GOTO statement, whose syntax is:

```
ON numeric-argument GOTO statement-label-list
```

The ON/GOTO statement branches to a different statement-label, depending on the value of the numeric argument. If the numeric-argument has a negative value (i.e., less than zero), the program branches to the first statement-label. If the numeric-argument equals 0, the program branches to the second label. If the numeric-argument is greater than zero, the program skips the first two labels and branches to the label corresponding to its value (if the value equals 1, it branches to the third label; if the value equals 2, it branches to the fourth label, etc.; if the value exceeds the number of corresponding positive labels, the program branches to the last label on the list).

Subroutines

InternetBasic includes a provision for writing subroutines within a source program. Here’s how to branch to a subroutine:

```
GOSUB statement-label
```

When this statement is executed, the program branches to the specified statement-label, which is considered to be the beginning of the subroutine. The RETURN statement is used at the logical end of the subroutine. Its syntax is:

```
RETURN
```

The RETURN statement branches to the instruction immediately following the GOSUB statement that was executed.

Note that subroutines can be nested up to 16 levels. If a program attempts to exceed that limit, a fatal runtime exception will happen (exception 93: GOSUB Stack Overflow).

Likewise, if a program attempts to perform a RETURN statement while the program is not in a subroutine, a fatal runtime exception will occur (exception 92: GOSUB Stack Underflow).

IB provides additional control over the subroutine stack. The POP instruction clears the most recent subroutine address from the stack, which allows the program to exit from a subroutine with a branching instruction (GOTO) or exit to a higher level subroutine. The POPALL instruction clears all of the subroutine addresses from the stack. The syntax for these statements is:

```
POP
POPALL
```

IB supports inter-process communication, which means that one IB program can interact with another IB program running simultaneously in the same computer. The program that initiates the interaction uses the INTERRUPT statement (see Chapter ??? for more information), while the program that is being interrupted can use the following statement:

```
MESSAGESUB statement-label
```

The MESSAGESUB statement sets a branch-to statement-label, so that if this program is interrupted, it immediately branches to the specified subroutine. Such a subroutine would handle the interruption from the other program, and could logically end with a RETURN statement to go back to the statement following the place where the interruption occurred.

IB also supports subprograms, which are self-contained object programs. In a way, a subprogram is similar to a subroutine, however subprograms exist as separate object programs, which means they can be called from any IB program via a single statement; the source lines do not need to be included in each program that wants to use the subprogram. See Chapter ??? for more information about subprograms.

ESCAPETO and ESCAPESUB

The IB client provides a way for the user to escape from the current program (something similar to the familiar Ctrl Alt Del sequence used to control tasks within Windows). In IB, the F3 and F10 function keys, pressed at the same time, escapes from the program that is running and returns to the IB monitor program.

You can prevent these function keys from having that effect by using one of the following instructions:

```
ESCAPETO statement-label
ESCAPESUB statement-label
```

ESCAPETO sets a statement-label where the program branches when the user presses F3 and F10. ESCAPESUB sets a subroutine statement-label where the program branches when the user presses F3 and F10.

Thus, you can write an "escape routine" that is executed when the user tries to escape from the program. Using ESCAPESUB lets you include a RETURN statement that branches back to the place where the user tied to escape.

When the ESCAPETO or ESCAPESUB statement is invoked, the IB runtime system pops the statement-label that was set. This means that your escape routine needs to reset the label, as shown in this example:

```
ESCAPESUB F3TRAP
.
.
.

F3TRAP: ESCAPESUB F3TRAP
        PRINT "PROGRAM INTERRUPTION -- F3 PRESSED."
        PRINT "DO YOU WISH TO CONTINUE (Y/N)?"
        INPUT ANSWER$
        IF ANSWER$ = "Y" THEN
          RETURN
        ELSE
          STOP
        ENDIF
```