Malcolm Bull
Training and Consultancy Publications

MB-Guide to

Basic programming topics

Malcolm Bull

MB-Guide
to
Basic programming topics

MB-Guide

to

Basic programming topics

by

Malcolm Bull

MALCOLM BULL Training and Consultancy Services

Malcolm Bull
Training and Consultancy Publications
19 Smith House Lane
BRIGHOUSE
HD6 2JY
West Yorkshire
United Kingdom

Telephone: 0484-713577

ISBN: 1 873283 13 8

Edition: 2

Introduction

The *MB-Guide to Basic programming topics* discusses a number of important topics and presents a number of tips and techniques concerning the use of the Basic language on the Pick operating system. This *MB-Guide* follows on from the general presentation which is to be found in the *MB-Guide to the Basic Language*.

We assume that the reader is familiar with the facilities and features of the Basic language and that he/she has a technical knowledge of the operating system.

The material will be of interest to analysts and programmers who have had some experience in the design and development of applications for use on the Pick operating system.

You may find the following titles in the *MB-Guide* series useful in conjunction with the present volume:

File design
Program design
The Basic symbolic debugger
The Basic language

and you may also find the following *MB-Master* self-tuition courses of interest in conjunction with the material presented in this *MB-Guide*:

BASIC1: Programming in Basic

BASIC2: Advancing in Basic. Much of the material in this *MB-Guide* is drawn from the Student's Workbook for this course.

BASIC3: Moving to Basic

This *MB-Guide* is not intended to present a complete description of the subject but merely to place it in context and give the reader enough information to use the facilities and to survive.

Best use can be made of this *MB-Guide* if it is read in conjunction with the reference literature which is provided for your system. You should amend your copy of this guide so that it accurately reflects the situation and the commands which are used on the implementation which you are using. By doing this, your *MB-Guide* will become a working document that you can use in your daily work.

I hope that you enjoy reading and using this *MB-Guide* and the others in the series, and welcome your comments.

Your suggestions for further topics to be included in future editions of this *MB-Guide* are of particular interest.

1 Program structures

Throughout the discussion, we shall assume that you are familiar with the form and function of the Basic program statements and structures. In this section, we shall look at the various program structures which the language offers, looking particularly at those for:

* Selecting the course of action to be taken by the processing,

* Repeating a section of processing.

We shall look at modular programming and external subroutines later.

1.1 IF structure

The IF statement is the simplest means of testing a condition and then taking either of two possible courses of action, according to whether the condition is true or false.

The following points are of interest:

* The statement may contain one THEN clause or one ELSE clause, or both a THEN and an ELSE clause.

```
IF A=B THEN A=0
IF A=B ELSE B=0
IF A=B THEN A=0 ELSE B=0
```

* In either clause, the course of action may be a single statement or several statements:

```
IF A=B THEN A=0; B=0; C=0 ELSE A=1; B=1
```

* The statement may be written as a single-line statement:

```
IF A=B THEN A=0 ELSE B=0
```

or as a multi-line statement.

```
IF A=B THEN
     A=0
END ELSE
     B=0
END
```

* The multi-line form is easier to read and to maintain.

* All statements - including disk-file operations - which

comprise a THEN and/or ELSE clause can be written as a multi-line statement. For example:

* The same structure can be used on any Basic statement which offers the THEN ... ELSE clauses:

```
OPEN
READ
READV
MATREAD
READNEXT
LOCATE
```

* When there are multiple conditions, as in statements such as:

```
IF A=B AND C=D THEN GOSUB 100
```

```
IF A=B OR C=D THEN GOSUB 200
```

then *both* conditions are evaluated, even though the result of the first condition might be sufficient to pre-empt the outcome (as would be the case if A were not equal to B in the first example, or if A were equal to B in the second example). This suggests that it may be faster to organise these statements in forms such as:

```
IF A=B THEN IF C=D THEN GOSUB 100
```

```
IF A=B THEN
     IF C=D THEN
          GOSUB 100
     END
END
```

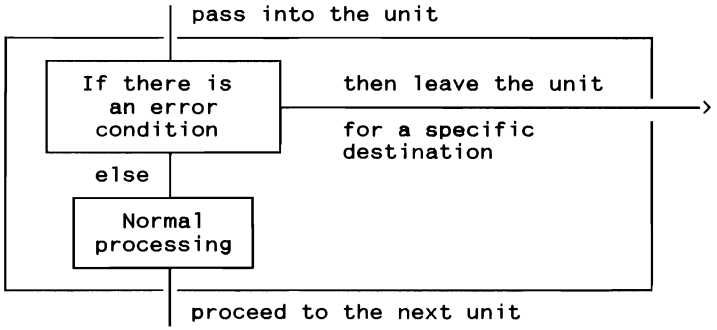
```
IF A=B THEN
     GOSUB 200
END ELSE
     IF C=D THEN GOSUB 200
END
```

These and other fundamental points are summarised in the *MB-Guide to the Basic language.*

1.2 GO TO statement

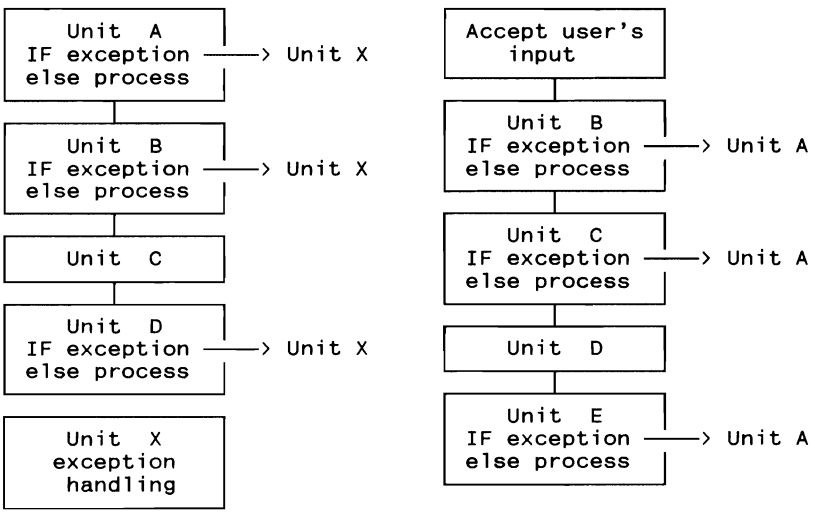
The GO / GO TO / GOTO statement has generally had a rather bad press. It can, however, justifiably be used in the situation where an escape route is required. For example, an error may have been detected in the data or the user may have indicated that he/she wishes to abandon the process. This situation is known as an *exception condition.*

If we adhere strictly to structured programming principles, then, on detection of an exception condition, we should set a switch and only carry out the processing of subsequent units (that is subroutines or statements) according to the setting of this switch. As a result, the coding could become very messy and the execution time unnecessarily protracted. Such exceptions may be handled more neatly by jumping out of the main processing flow, like this:



An exception condition will normally be handled by the use of a GO TO statement. A GOTO statement is often much easier to code (and to read) than would be a complicated set of switches testing whether or not all subsequent processing is to be carried out or not.

The destination on leaving the unit depends upon the logic of the program. Processing may pass to some exception-handling unit within the program, as shown on the left in the diagram below, or it may return to the start of the program, as shown on the right.



At this point, we might also remind ourselves of the statements:

```
ON CODE GOTO 100, 200, 300, 445, 123, 345
ON CODE GO 100, 200, 300, 445, 123, 345
```

and the associated statement:

```
ON VALUE+1 GOSUB 100,200,100,200,300,300,100
```

These and other fundamental points are summarised in the *MB-Guide to the Basic language.*

1.3 CASE structure

The CASE structure is much more flexible than the equivalent structure in other languages such as Pascal.

The following points are of interest:

* Any conditional (or logical) expression may be used in the CASE statement.

* If there are any syntax errors anywhere within the CASE structure, the compiler may flag these are being on the:

  ```
  BEGIN CASE
  ```

  statement. A process of inspection and selectively commenting out each individual CASE should be used to locate the error.

* There must be an END CASE for each BEGIN CASE.

* The BEGIN CASE and the END CASE statements should be on separate lines enclosing the rest of the structure.

* CASE structures may be nested.

* The CASE structure is more flexible than the IF statement if the number of possible alternatives is likely to be more than 2 as the program develops.

* The

  ```
  CASE 1
  ```

  statement is always true and is used as a *catch-all* to trap any situation which fails with all the preceding CASE statements.

These and other fundamental points are summarised in the *MB-Guide to the Basic language.*

1.4 FOR / NEXT structure

The FOR/NEXT structure has the general form:

```
FOR variable=initial TO terminal {STEP increment}
      :
      :
NEXT variable
```

and the extended forms:

```
FOR var=init TO term {STEP incr} {UNTIL condition}
```

in which case the loop will abandon when either the value of *var* reaches (or exceeds) *term* or the *condition* is true.

```
FOR var=init TO term {STEP incr} {WHILE condition}
```

in which case the loop will abandon when either the value of *var* reaches (or exceeds) *term* or the *condition* is false.

The elements in braces may be included or omitted, as required.

In all cases the NEXT statement has the same form.

The following points are of interest:

* The values for the initial value, for the terminal value, or for the STEP value may be specified as a literal or as an arithmetic expression.

* The initial value is evaluated only when the loop is first entered.

* The terminal value is evaluated before the start of each iteration.

  For this reason, it is possible to construct an indefinite loop of the form:

```
FOR X=1 TO X+1 UNTIL RECORD<X>=''
    PRINT RECORD<X>
NEXT X
```

  with the terminal value (X+1) being *pushed ahead* of the current value for X.

* Either the initial, terminal and/or increment values may be positive/negative integer/fractional numbers.

* An omitted STEP value is assumed to be +1.

* Ascending and descending movement is achieved with positive or negative STEP values.

* There must be variable on the NEXT statement; although the compiler does not check that the two are the same!

* You may have loops within loops, provided that the various loops have different counters.

  Loops may be nested to any number of levels, and a nested loop must be completely contained within the range of the outer loop, that is, the ranges of the loops must not cross.

* Intersecting loops are not acceptable.

* It is possible, though bad practice, to change the value of the counter inside the loop.

* It is possible, though bad practice, to jump into or out of a loop.

These and other fundamental points are summarised in the *MB-Guide to the Basic language*.

1.5 Loop structure

The LOOP structure offers powerful iteration facilities representing the design requirements:

* Repeatedly execute a set of statements *until* a specific condition is satisfied, or

* Repeatedly execute a set of statements *while* a specific condition exists,

and also:

* Test for the condition *before* entering the iteration, or

* Test for the condition *after* the iteration.

The general form of the structure is:

```
LOOP
     statement(s)
UNTIL condition DO
     statement(s)
REPEAT
```

```
LOOP
     statement(s)
WHILE condition DO
     statement(s)
REPEAT
```

The action of the statement is:

1) To execute the first set of statement(s), and then

2) To test the condition, and if the condition there obtains, then

3) To execute the second set of statement(s), and then

4) To return to step (1).

Here are some examples:

```
LOOP
     PRINT 'CONTINUE? Y OR N: ':
     INPUT RESPONSE,1
UNTIL RESPONSE='Y' OR RESPONSE='N' DO
     PRINT 'ENTER Y TO CONTINUE / N TO STOP
REPEAT
```

```
LOOP
     PRINT 'ENTER A NUMBER: ':
     INPUT NUMBER
WHILE NUMBER MATCH '1N0N' DO
     TOTAL = TOTAL + NUMBER
REPEAT
```

To simplify the coding of the illustrations below, we have shown only one of UNTIL or WHILE.

Either of the *statement(s)* blocks may be omitted:

```
LOOP
UNTIL condition DO
     statement(s)
REPEAT
```

```
LOOP UNTIL condition DO
     statement(s)
REPEAT
```

both of which afford the *test before iteration* structure, and:

```
LOOP
     statement(s)
UNTIL condition DO
REPEAT
```

```
LOOP
     statement(s)
UNTIL condition DO REPEAT
```

both of which afford the *test after iteration* structure.

The *condition* in the UNTIL or the WHILE clause may be any

combination of logical conditions. For example:

```
UNTIL X>30 OR LIST<X>='' DO
WHILE TOTAL<1000 AND INVAL NE 'END' DO
```

Some examples are:

```
COUNTER=1; INITIALISE COUNTER
LOOP
     THIS1=RECORD<COUNTER>
UNTIL THIS1=TAG DO
     *
     * PROCESS THIS1
     *
     COUNTER=COUNTER+1
REPEAT
```

```
WORD.CTR=1
LOOP
WHILE WORD.CTR<=WORDS DO
     WORD=FIELD(SENTENCE,' ',WORD.CTR)
     *
     * PROCESS WORD
     *
     WORD.CTR=WORD.CTR+1
REPEAT
```

An indefinite LOOP illustrated by the final example above is not generally available. If required, it can be achieved by a structure such as:

```
LOOP
     *
     * MAIN BODY OF LOOP
     *
REPEAT
```

```
LOOP
     *
     * MAIN BODY OF LOOP
     *
     EXIT
     *
REPEAT
```

```
LOOP
     *
     * PROCESSING HERE
     *
UNTIL 0 DO REPEAT
```

```
LOOP
     *
     * PROCESSING HERE
```

```
     *
WHILE 1 DO REPEAT
```

The first two of these structures are available on Ultimate and Advanced Pick. The EXIT statement, which is used in the second example, offers an elegant means of leaving the loop and avoids the use of a GOTO statement to leave the loop.

The third structure *works* because the value 0 will never be *true*, that is, will never reach the value 1; the final structure works for the same reason, that 1 is always true. Such a structure avoids the use of a GOTO statement.

The following points are of interest:

* The WHILE and the DO (and the UNTIL and the DO) must appear on the same line:

```
LOOP UNTIL condition DO
    statement(s)
REPEAT
```

```
LOOP statement(s) UNTIL condition DO
    statement(s)
REPEAT
```

```
UNTIL condition DO
```

```
UNTIL condition DO REPEAT
```

```
UNTIL condition DO statement(s) REPEAT
```

```
LOOP stmt(s) UNTIL condition DO stmt(s) REPEAT
```

  The other parts of the statement may be arranged on the same line or on separate lines, as illustrated by the examples.

* Although the entire statement may be entered as a single line, as illustrated in these latter examples, it is always clearer to see the action of the statement if you write the statement as a multi-line structure, as in our earlier illustrations.

* There is no END statement to mark the end of either set of statements. The UNTIL (or WHILE) marks the end of the first set of statements, and the REPEAT statement marks the end of the second set of statement.

* If you do write the statement on a single line, then semi-colons are used to separate multiple statements, like this:

```
LOOP A=A+1; PRINT A UNTIL A>10 DO REPEAT
```

```
LOOP WHILE A<10 DO A=A+1; PRINT A REPEAT
```

* There are no semi-colons around the LOOP, the WHILE, the UNTIL, the DO, or the REPEAT.

* For simple iteration, a LOOP structure such as:

```
CTR=1
LOOP UNTIL CTR>=10 DO
    * PROCESSING HERE
    CTR=CTR+1
REPEAT
```

is less efficient than the equivalent FOR/NEXT loop:

```
FOR CTR=1 TO 10
    * PROCESSING HERE
NEXT CTR
```

These and other fundamental points are summarised in the *MB-Guide to the Basic language.*

1.6 Logical expressions

Logical expressions are those which return a value of *true* (represented by the numeric value 1) or *false* (represented by the numeric value 0), and which are used in contexts such as:

```
IF logical-expression THEN/ELSE
LOOP ... UNTIL logical-expression DO ... REPEAT
LOOP ... WHILE logical-expression DO ... REPEAT
FOR ... UNTIL logical-expression
FOR ... WHILE logical-expression
CASE logical-expression
```

The logical expression may be a conditional test:

```
IF VALUE > COUNT THEN ...
IF AGE > 65 AND SEX > 'M' THEN ...
IF A < B OR C > D THEN ...
IF (A > B AND A > C) OR (A < B AND A < C) THEN ...
IF TOTAL = 0 THEN ...
```

Unlike some programming languages which have Boolean (or logical) variables and Boolean values *true* and *false*, Basic uses 1 and 0 to represent these values. If a variable contains a value of 1 or 0 then it may appear as the logical expression:

```
IF OAP THEN ...
```

The value of a logical expression can be assigned to a variable and this variable then used as a logical expression:

```
OAP = (AGE>=65 AND SEX='M') OR (AGE>=60 AND SEX='F')
```

and may be used in a context such as:

```
IF OAP THEN ...
```

This is much better than a sequence such as:

```
IF (AGE>=65 AND SEX='M') OR (AGE>=60 AND SEX='F') THEN
     OAP='Y'
END ELSE
     OAP='N'
END
```

and the corresponding:

```
IF OAP='Y' THEN ...
```

The format of logical assignment statements is:

```
variable = logical expression
```

and this explains why the Basic compiler accepts a statement such as:

```
A = B = C
```

which is interpreted as:

```
A = (B=C)
```

in which the variable A is set to 1 (if the variables B and C are equal) or 0 (if B and C are not equal).

Care should be taken if a variable - such as OAP in the previous examples - is used as a logical expression and its value may possibly be assigned values other than 1 or 0. A value of 0 is interpreted as *false*, all other numeric values (positive or negative) are interpreted as *true*. If a non-numeric field is used, such as:

```
IF 'A' THEN ...
```

a non-fatal error will result and the error message:

```
[B16] NON-NUMERIC DATA WHEN NUMERIC REQUIRED; ZERO USED!
```

will be displayed. As a consequence of 0 being used, the result will be *false*.

These and other fundamental points are summarised in the *MB-Guide to the Basic language*.

2 Data structures

We assume that you are familiar with the nature and organisation of the variables and arrays used in a Basic program.

The following points are of interest:

* There may be up to 3223 variables (including elements of dimensioned arrays) in any one program.

* A variable may hold a number or a string.

* A numeric variable may hold any value in the range:

  -140,737,488,355,327 to +140,737,488,355,327

* Numbers are normally held with 4 places of decimals, but you may use a PRECISION statement to specify a precision of 0, 1, 2, or 3 places of decimals (or greater on some implementations).

* A string may be up to 32K bytes in size.

* Each variable is identified by name.

* Each element of a dimensioned array is identified by the name of the array and a subscript specifying the position within the array.

* A dynamic array is simply a string which contains the system delimiters (attribute mark, value mark and/or subvalue mark).

* A file variable (as used in OPEN, SELECT, READ, WRITE and DELETE statements) cannot be used for any other purpose.

These and other fundamental points are summarised in the *MB-Guide to the Basic language.*

The section below where we discuss the Descriptor Table is also relevant in this context.

2.1 Dimensioned arrays

Dimensioned arrays are handled as in other programming languages.

The following points are of interest:

* All dimensioned arrays must be declared in the program where they are used.

* The DIMENSION (or DIM) statement need not physically precede uses of the array in the program.

* An array may have one or two subscripts. For example:

```
DIMENSION SALES(9)
DIMENSION STATISTICS(10,15)
```

* The number of elements in the DIMENSION statement must be an integer, as shown above. It cannot be a variable or an equated symbol:

```
DIMENSION SALES(SALESIZE)
```

is invalid and would be rejected by the compiler.

* If an array is to be passed as a parameter to a subroutine, it will be specified on statements of the form:

```
CALL SUB003(NAME, MAT SALES)

SUBROUTINE SUB003(NAME, MAT SALES)
```

The array must also be declared by means of a DIMENSION statement in the subroutine. The DIMENSION statement must specify the number of elements as an integer, as described above.

* If an array is specified on a COMMON statement then the DIMENSION statement is not required, for example:

```
COMMON NAME, CODE, SALES(9)
```

and there should be no DIMENSION declaration for that array in the program.

* Statements such as:

```
MAT SALES = 0
MAT NAMES = ""
```

are available to assign the same value to all the elements of an array, or

```
DIMENSION SALES(9), COPYSALES(9)
MAT COPYSALES = MAT SALES
```

to copy arrays like to like. A fatal error will result from this MAT assignment statement if the two arrays had different numbers of elements.

* There are no facilities - such as matrix arithmetic, matrix inversion, matrix input or matrix output - for manipulating the contents of a dynamic array.

* Any element of a dimensioned array may contain a numeric value or a string. If the string is a dynamic array, then the individual attributes of the dynamic array may be handled by references of the form:

```
SALES(1)<2,3,4>
```

which identifies the fourth subvalue of the third value of the second attribute of the dynamic array string held in element 1 of the dimensioned array.

You may even take a substring of this by means of the extreme notation:

```
SALES(1)<2,3,4>[5,6]
```

to return characters 5 to 10 of the data.

2.2 Dynamic arrays

As a data structure, the dynamic array offers great programming capabilities.

A dynamic array is simply a string containing the system delimiters:

* The attribute mark, ASCII character 254, and/or
* The value mark, ASCII character 253, and/or
* The subvalue mark, ASCII character 252.

Thus, the structure of a dynamic array is identical to that of a normal Pick item as it is held on a file.

A dynamic array may have zero, 1 or more attributes; any attribute may have zero, 1 or more values; any value may have zero, 1 or more subvalues.

A number of standard functions are supplied for use with dynamic arrays:

```
EXTRACT
INSERT and the modified form INS
REPLACE
DELETE and the modified form DEL
```

The following points are of interest:

* A dynamic array is held in an ordinary variable.
* Unlike a dimensioned array, a dynamic array is not declared explicitly. Only the way in which it is used determine whether or not it is to be regarded as a dynamic array.
* A dynamic array may be initialised by a statement such as:

```
DARRAY = ''
```

* A dynamic array may have any number of attributes, values and/or subvalues up to a total length of 32K characters.
* Elements may be inserted, changed and/or deleted from a dynamic array. The standard functions add or remove the necessary separator characters.
* References such as REC<1> or REC<1,2,3> can be used in exactly the same context as ordinary variables *except*:
  - they cannot be used in INPUT statements. Thus, the

statement:

```
INPUT REC<1,2,3>
```

is invalid and would be rejected by the compiler.

- they cannot be used in EQU statements. Thus, the statement:

```
EQU NAME TO REC<7>
```

would be rejected by the compiler.

- they cannot be used to *return* values from external subroutines. Thus, the statement:

```
CALL SUBR001(RATE,RECORD<3>)
```

would not work as expected if the subroutine changed the contents of the second argument. The reference RECORD<3> would, however, work correctly if this were only used to *pass data into* the subroutine.

These and other fundamental points are summarised in the *MB-Guide to the Basic language*.

2.3 Dynamic arrays versus dimensioned arrays

The following points may influence the decision as to whether to use dynamic arrays or a dimensioned array in your program design:

* When you are using dimensioned arrays to process records which have a large number of attributes, you may be hindered by fact that the maximum number of program variables (including those variables held in COMMON and elements of dimensioned arrays) is 3223.

* Each dynamic array is a simple variable and may be up to 32K bytes in length.

* You may append an unlimited number of attributes to the array. A dimensioned array is is static in size.

* You may insert new elements within the dynamic array.

* You may remove elements from dynamic array (and close up the gap).

  It is messy to insert new values within a dimensioned array (shifting all the following elements up one position) and to remove values from a dimensioned array (shifting all the following elements down one position).

  This and the previous points are ideally employed when a dynamic array is to be used to build up and maintain a list of data for use by the program.

  Care should be taken if this dynamic array list of data

consist of elements (such as other dynamic arrays and records) which themselves contain more than one attribute.

* The LOCATE function or the LOCATE statement enable you to locate and sort the elements of a dynamic array.

* MATREAD and MATWRITE statements must use the raw data from the disk (which is physically held as a dynamic array) and use this to build a dimensioned array. For this reason, these statements are slower than the corresponding READ and WRITE statements which handle dynamic arrays.

* After a READ statement, the DCOUNT may be used to determine how many attributes there are in the dynamic array read from the file. Most implementations do not have any facility for finding out the number of attributes read into a dimensioned array.

* The individual elements of a dimensioned array are accessed must faster than those of a dynamic array.

  This is because each element of a dimensioned array has a calculable start address; to find, say, element 66 of a dimensioned array, the run-time processor uses the fact that this is located 650 bytes (that is, 65 times the length of each element) from the start of the array. To find attribute 66 of a dynamic array, the run-time processor has to scan the entire string looking for the 65th and 66th attribute mark. The time difference is typically a factor of 10 or more.

* The use of the EQUATE statement to name each individual element of a dimensioned array makes it much easier to read and understand the program coding than with a dynamic array reference.

  Thus, in a sequence such as:

```
DIM RECORD(20)
EQUATE NAME TO RECORD(1), ADDRESS TO RECORD(7)
```

  statements such as:

```
INPUT NAME
PRINT ADDRESS
```

  are easier to read than statements such as:

```
PRINT DA.RECORD<1>
PRINT DA.RECORD<7>
```

* You cannot INPUT data directly into a dynamic array, you can INPUT directly into a dimensioned array:

```
INPUT RECORD(1) is valid
```

  but

```
INPUT DA.RECORD<1> is not valid
```

These and other fundamental points are summarised in the *MB-Guide to the Basic language.*

2.4 Descriptor table

The descriptor table is a list of all the variables which are used by a Basic program or a subroutine. It is produced by the compiler and used by the run-time interpreter to handle the variables. For each variable named in the program or subroutine (both those in COMMON and the local variables) and for each element of a dimensioned array, there is an entry in the descriptor table.

The maximum size of the descriptor table is 3223. This explains the restriction on the number of variables (and dimensioned array elements) which can be used in a program.

Each entry in the descriptor table is (normally) 10 bytes long. The first byte indicates the nature of the variable:

1) Unassigned: byte 1 contains hexadecimal 00.
2) Numeric: byte 1 contains 01.
3) String - direct: byte 1 contains 02.
4) String - indirect: byte 1 contains 82.
5) File variable: byte 1 contains 04.
6) Subroutine variable: byte 1 contains 40.

The remaining bytes, 2 to 10, contain data:

1) Unassigned: zeroes.

2) Numeric: held as a six-byte binary number.

3) String - direct: up to seven bytes of data. Eight bytes on Ultimate implementations.

4) String - indirect: a six-byte pointer to the location of the actual string in memory. We discuss this below.

5) File variable: the base FID (four bytes), the MOD (two bytes) and the SEP (two bytes) of the file.

6) Subroutine variable: a six-byte pointer to a catalogued subroutine, as used in a context such as CALL @ROUTINE

Within the table, COMMON variables usually appear first, followed by local variables, followed by dimensioned arrays. The organisation of the descriptor table can be seen by specifying the M (map) option on the BASIC command. These and several other features of the way in which data is organised within the program are illustrated by the following program:

```
COMMON NAME, CALC, DATE
EQU DROP TO CHAR(10), AST TO '*'
EQU FLAG TO SWITCH
DIMENSION ARRAY(20), STORE(5)
```

```
TOTAL=0
DIMENSION COPY(20)
FOR X=1 TO 20
    TOTAL=TOTAL+X
    ARRAY(X)=TOTAL
NEXT X
CALL SUB001(TOTAL,FLAG,VALUE)
PRINT DROP: AST,SWITCH,VALUE
END
```

If we compile this program with a command such as:

```
BASIC filename programname (M
```

the following map would be displayed:

```
C030  NAME      C040  CALC      C050  DATE        100  FLAG
 110  ARRAY      310  STORE      360  COPY        060  TOTAL
 070  X          080  VALUE      100  SWITCH
```

The following points are of interest:

* The names (or symbols) appear in the descriptor table in the sequence in which they are used (or declared) in the program.

* The number indicates the position (or the address) of that variable (or symbol) within the descriptor table;

* The addresses start at 030;

* A C before the address indicates that this is a COMMON variable;

* COMMON variables come first;

* The arrays come last;

* Each element of the array occupies 1 entry in the description table. Thus, ARRAY runs from 110 to 300, that is 20 places in the table.

If a string is seven bytes or less in length, then it is held explicitly as a type (3) variable in the descriptor table. But if the string increases in length, it is held in one of a series of buffers and the address of the buffer is held in the descriptor table as a type (4) variable. The buffers can have lengths of 50 bytes, 100 bytes, 150 bytes, 250 bytes, 500 bytes and so on. As the string increases in length, it is moved to the next larger buffer, but as it decreases in length it is not relocated. When a string is moved to a larger buffer, the previous buffer is released. If a new buffer is required, and there is not sufficient storage space available, then a process known as *garbage collection* is carried out and the buffers are reorganised to provide sufficient space for the new buffer requirements. If this

does not yield the required space, then the program will abort with a message such as:

```
[B28] LINE n OF program NOT ENOUGH WORK SPACE.
```

This explains two important points:

* The reason why some string operations, such as concatenation, are so expensive on time.

* A cause of the NOT ENOUGH WORK SPACE message when using very large strings such as data records and lists.

3 EQUATE statement

The EQUATE statement (and its abbreviation EQU) has several forms. Let's look at these in turn.

The first form:

```
EQUATE MESSAGE01 TO 'Enter YES or NO'
```

will result in all references to the name MESSAGE01 being replaced by the character string at compilation time. MESSAGE01 can be regarded as a constant (in the manner of constants in languages such as Pascal), and cannot be used on the receiving end of an assignment statement in that program.

The string or value to be assigned to the symbol MESSAGE01 must be a specific string and may not contain (or imply) an expression which can only be evaluated at execution-time. Thus, statements such as:

```
EQUATE INCREMENT TO 24
```

are valid, but statements such as:

```
EQUATE PRICE TO 30*100
EQUATE VALUE TO PRICE*QTY
EQUATE NAME TO TITLE:' ':SURNAME
EQUATE DROP TO CHAR(10):@(-3)
```

are invalid and would be rejected by the compiler.

A seeming exception to this is presented by statements of the second form:

```
EQUATE AMARK TO CHAR(254)
```

which will result in all references to the name AMARK being replaced by the actual character at *compilation* time. As with the previous form, AMARK is regarded as a constant and cannot be used on the receiving end of an assignment statement in that program.

The EQUATE statement is evaluated once only - at compilation time - and these two forms are preferable to assigning the values to variables by means of the equivalent assignment statements:

```
MESSAGE01 = 'Enter YES or NO'
INCREMENT = 24
AMARK = CHAR(254)
```

which would be less efficient at execution time because:

* The assignment statement itself must be executed every time the program is executed.

* All references to the variables MESSAGE01, INCREMENT and AMARK require the execution time processor to locate the variables and retrieve the contents each time those variables are used.

The third form:

```
EQUATE COST TO PRICE
```

allocates the names COST and PRICE to the same variable. In practice, this statement is of doubtful value since the use of two names for the same variable may confuse the reader. It would be much better to use the Editor to change all occurrences of the one name to the other.

The fourth form:

```
EQUATE DESCRIPTION TO STOCK(1)
```

is a convenient device for documentation purposes since it allows a meaningful name to be given to specific elements of a dimensioned array. If the array is to be used in MATREAD / MATWRITE statements, then the individual elements of the array can be assigned the names of the separate fields of the item, thus making the program easier to read.

4 EXECUTE statement

The format of the EXECUTE statement varies widely on the various implementations:

* Advanced Pick offers TCL as an alternative to EXECUTE.

* McDonnell Douglas offers PERFORM as an alternative to EXECUTE, or vice versa.

* The EXECUTE statement of Ultimate implementations is vastly different from that of the other versions, although it is understood that this will change in later releases.

Consult the reference manual for your implementation and make a note of the following points:

The form of the statement

How to capture the output from the TCL command

How to suppress the displayed output from the command

How to pass data to the TCL command

How to test for successful completion of the command

The following points are of interest:

* When an EXECUTE statement is carried out, the effect is the same as logging on to the account as a temporary user and issuing a TCL command to perform the required statement, and then logging the temporary user off the account when the action is complete. This means that, for each EXECUTE statement, the operating system must

allocate a new set of workspace, use it and then deallocate it.

This is an expensive process. Wherever possible, is it more efficient to use the

```
CHAIN
```

statement or, if the program is suitably constructed, one of the:

```
CALL
ENTER
```

statements instead of EXECUTE.

* Since a process which is invoked by an EXECUTE statement is carried out at a different processing level from the program which issued the statement, several differences and consequent difficulties arise. Typical of these are:

  + File / item locks are not applicable when a transfer is made to a Basic program.

  + A Basic program which is invoked by an EXECUTE statement will have different COMMON areas from the program which invoked it.

  + In an invoked Basic program, any PROCREAD statement will take the ELSE clause.

  + Data is most successfully *passed back* from an invoked Basic program back to the invoking program by means of a work-file.

  + An invoked EDIT command will have different TB tab-stops and pre-stored commands from another invoked EDIT command and these will also be different from those of an EDIT command which has been invoked at TCL.

* If you are capturing the output from an EXECUTE statement, this may be too large to accept into a Basic variable if it is the result of, say, a large Access report.

* Any stacked data will be submitted to the EXECUTEd command (whether it is needed or not) and will not be available for subsequent processing. Thus, in the example:

```
DATA 'ABC'
EXECUTE 'WHO'
INPUT CODE
```

the stacked data value ABC will be lost and cannot be accessed by the INPUT CODE statement.

5 Input statements

Depending upon which version of the operating system you are using, you will have a range of statements for accepting input data from the user as the program executes:

These include the INPUT statement, the INPUT @ statement (described below) and possibly also the statement:

IN VALUE
which will accept a single character from the keyboard and puts it ASCII decimal equivalent into the variable called VALUE.

This statement will accept any keyboard character, including the function keys, <BACK SPACE> and the <CURSOR CONTROL> arrow keys.

There is also an equivalent:

OUT VALUE
which will convert the decimal number in VALUE to the equivalent ASCII character and display this on the screen.

CALL GETBUF(RESPONSE,SIZE,TYPE,HOLD)
which will:

+ Move the cursor backwards from the current position a distance equivalent to the length of the data currently held in the variable RESPONSE.

+ Accept data from the user, as specified by SIZE (the maximum length of the input data), TYPE (a code specifying the format of the input data) and HOLD (0 if the cursor is to be held in position after the input, otherwise 1).

  The source subroutine (held as item GETBUF on file BP) may be amended to accept other data patterns, but the current standard values for TYPE are:

  0 alphanumeric characters only;
  1 numbers and numeric symbols (including + - $, .)
  2 numbers only.

  If any invalid data is entered, conflicting with the specified TYPE, the terminal will beep and ignore the character.

+ Put the user's response into the variable RESPONSE.

These and other fundamental points are summarised in the *MB-Guide to the Basic language*.

5.1 INPUT @ statement

The INPUT @ statement is one of a group of statements which offer very powerful facilities for data input on some implementations.

When you use the INPUT @ and associated statements,

* The program will position the cursor before accepting the input data.

* The program will display a default value according to a format-mask in the field where the data are to be input.

* The program will accept the user's input data value.

* The user may overwrite the default value, take the default value, or enter any of a set of *special action* codes.

* The program will redisplay the input value according to the format-mask.

* The program will display and control the erasure of messages at the foot of the screen.

Look at this sequence:

```
TAX=0
INPUT @(5,5):TAX "R12"
```

This will:

* Move the cursor to position (5,5).

* Display the current contents of the variable TAX (this is 0 in this particular sequence) according to the mask "R12", that is right-justified in a field of width 12.

* Accept a value for the variable TAX and allowing the user to type over the displayed value.

* Redisplay the input value according to the mask "R12".

* Interpret a null response as indicating that the previous contents of TAX are to be retained. Thus, the contents of TAX before the statement is executed can be thought of as a *default* value.

There are several statements to support the INPUT @ facility.

INPUTERR statement
: This controls the displaying and erasure of error-messages at the foot of the screen. For example:

INPUTERR "VALUE MUST BE LESS THAN 100"
: When this statement is encountered it will:

    * Clear the bottom line of the screen, if there is

already an INPUTERR error-message there.

* Display the text.

VALUE MUST BE LESS THAN 100

* Set an internal system flag to indicate that an error-message is present at the foot of the screen, indicating that this is to be cleared prior to subsequent INPUTERR error-messages.

Note that - despite the format - there is no *error-condition* associated with this statement. The INPUTERR statement is executed when it is encountered, exactly like a PRINT statement.

INPUTNULL statement
this statement allows the programmer to specify a character which is to be used to represent a truly null value. This statement is required since a null response at an INPUT @ statement will assume the default value. For example:

INPUTNULL "*"
This statement is not executed immediately, but must be issued prior to any INPUT @ statements to indicate that, if the user enters a response of * to subsequent INPUT @ statements, then this is to be interpreted as null. Any single character may be specified, and this null-indicator will stay in operation until changed by a subsequent INPUTNULL statement.

INPUTTRAP statement
This statement allows a set of *special operations* codes to be established according to certain single-character input by the user. Some examples of this statement are:

```
INPUTTRAP "?/.<>" GO 10,20,30,40,50
INPUTTRAP "?/.<>" GOSUB 10,20,30,40,50
```

Such statements are not executed immediately, but must be issued prior to INPUT @ statements to indicate that, in this example, a response of any one of the specific characters ? or ? or . or < or > to subsequent INPUT @ statements will cause the system to branch to the statements 10, 20, 30, 40 and 50 respectively.

This is a useful device which allows the analyst to design systems with conventions such as:

```
?    means display a help message.
/    means cancel the operation.
.    means abandon the job.
<    means return to the last input stage.
>    means skip to the next input stage.
```

and so on.

There may be any number of characters and a matching set of

destinations, and these INPUTTRAPs will stay in operation until changed by a subsequent INPUTTRAP statement.

6 PRINT and printing

All output which the user visualises as being sent to the printer is, in fact, intercepted and handled by the spooler. The spooler is discussed in detail in the *MB-Guide to the spooler.*

In this section, we assume that you are familiar with the fundamental features of the Basic PRINT / CRT / DISPLAY statement:

PRINT
: is used to output to the screen or to the spooler (if a PRINTER ON statement has been executed or the program has been invoked by the (P option).

  PRINT output can be controlled by the PAGE / HEADING / FOOTING statements.

CRT
: is used to output to the screen, regardless of whether or not a PRINTER ON statement has been executed. This statement does not affect the PAGE / HEADING / FOOTING controls.

DISPLAY
: is offered instead of CRT on Ultimate implementations.

OUT VALUE
: which will convert the decimal number in VALUE to the equivalent ASCII character and display this on the screen.

These and other fundamental points are summarised in the *MB-Guide to the Basic language.*

6.1 PRINTER ON / OFF / CLOSE statements

When the execution of a Basic program begins, any PRINT statements will normally send their output to the screen. To direct output to *the printer* (the spooler) you will use the:

```
PRINTER ON
```

statement. When this statement has been executed, all subsequent PRINT statements will send their output to the printer, or more correctly, to the spooler.

After you have done this, you may want to switch the PRINT output back to the screen, for example, to ask your users for some more information. You will do this by means of the statement:

```
PRINTER OFF
```

The use of the CRT statement, as discussed below, will

obviate the need to switch the PRINTER OFF and ON in such circumstances.

When the program execution has finished, then your report (currently being held by the spooler) will be printed as soon as the printer is free.

If you wish to cause your output to be printed BEFORE the end of your job - you may want to inspect some printed output before proceeding - then you will use the:

```
PRINTER CLOSE
```

statement. This closes the output file and sends the report to the printer immediately it is free.

If you have a program which is to produce all its output on the printer, you can do this much more simply by executing the program with the command:

```
RUN MYPROGS ADV003 (P)
```

The (P) has the effect of issuing a PRINTER ON statement before the program execution starts. In this situation, you will not need PRINTER ON statements in your program. Any PRINTER OFF or PRINTER ON statements will have their normal effect.

In general, all the forms of the PRINT statement which we have discussed - except those using the @ function and certain special print characters - will have the same effect on the terminal screen and on the printer.

If your system is to use any non-standard output devices, such as bar-code encoder, plotter, or telex, then these will be driven by data sent from such PRINT statements or by the OUT statement.

The CRT statement has exactly the same syntax as the PRINT statement. The differences are:

+ CRT *always* sends output to the screen, irrespective of whether a PRINTER ON statement has been issued or not.

+ CRT does not increment the line-counter details which are used to monitor the page-depth when the HEADING and FOOTING statements have been issued.

6.2 PRINT ON statement

The PRINT ON statement enables output to be sent to any number of separate spooler files during the execution of a program.

This allows you to generate several reports at the same time, in the same program, and then print them off separately. For example, an account update program might produce separate reports to:

* Print the transactions.

* Print the customers' statements.

* Print the management summary.

* Print the name and address labels for the statements.

And each of these might be produced on different stationery. We could achieve this by statements such as:

```
PRINT ON 1 DATE, CODE, VOUCH.NO, AMOUNT
PRINT ON 2 VOUCH.NO, VOUCH.DETAIL, AMOUNT, NEW.BALANCE
PRINT ON 3 CODE, NAME, NEW.BALANCE
PRINT ON 4 NAME
PRINT ON 4 STREET
PRINT ON 4 TOWN
```

At the end of job, all the separate spool files will be closed and then output in numerical order, or to separate spooler queues if you have used the SP-ASSIGN verb with parameters such as:

```
SP-ASSIGN F1 R4
```

which will assign all the PRINT ON 4 output to form-queue 1.

6.3 HEADING / FOOTING / PAGE statements

The HEADING / FOOTING / PAGE statements can be used to advantage when producing a siimple paginated report on the screen or the printer.

When a HEADING or a FOOTING statement is first executed, the run-time processor starts up a line count routine. Each subsequent PRINT statement (not PRINT statements which end with a colon nor CRT statements nor DISPLAY statements) will add 1 to this line count. When the line count reaches the page depth limit as shown by the terminal characteristics for the current output device (terminal or printer), then a page skip routine will be triggered:

* any footing will be output;

* if the output is to the terminal, then the output will halt until the user hits any key (or terminates the process by entering <CTRL> X);

* a skip is made to a new page (this will clear the terminal screen or skip to a new page on the printer);

* any heading will be output.

* the line count will be reset.

The PAGE statement forces this page skip routine.

The following points are of interest:

* The execution of a HEADING statement will generate a skip to a new page.

* A FOOTING statement has no immediate effect on the execution, other than to set the text to be used in subsequent footings and to trigger the line count routine.

* The simple:

```
HEADING " "
```

  statement will clear the screen.

* On some implementations, the statement:

```
HEADING ""
```

  will cancel the current HEADING specification, on others it will simply change the heading text to null.

* If the HEADING / FOOTING is changed during the execution of a program, the new specifications will only become effective when the next page skip is made.

* During testing, it is convenient to use a HEADING statement to paginate the output and (at the pause at the foot of each page) to allow the user to terminate the process by entering <CTRL> X.

* Care should be taken when using the HEADING/FOOTING statements other than in the production of simple reports. For example, you might use these statements to produce a report and then later in the same program use PRINT and INPUT statements for conversational data; all the PRINT statements will contribute to the line count and after, say, 23 PRINT/INPUT sequences, the program will pause (thinking that it has reached the foot of a page) until the user hits any key for it to continue.

* If you mix HEADING/FOOTING statements with PRINT ON for several output queues, the results will be unpredictable.

* The PAGE statement has no effect if there is no HEADING / FOOTING in operation.

6.4 @ function

The @ function is used for cursor-positioning when outputting to the screen and is commonly used in PRINT (or CRT or DISPLAY) statements:

```
PRINT @(0,5):
PRINT @(COL,ROW):MSG1
PRINT @(12,ROW):TEXT1:@(40):TEXT2
PRINT @(50):'?????':
```

It cannot be used in output to the printer.

Since it is simply a function which returns a value (the value being an escape sequence to trigger terminal display effects), the function can also be used to advantage in building strings which represent a screen image:

```
SCREEN = @(-1) : @(0,15) : TITLE
SCREEN = SCREEN : @(0,2) : 'Name: '
SCREEN = SCREEN : @(0,3) : 'Department: '
```

and then issuing a statement such as:

```
PRINT SCREEN
```

to display the entire screen. This is more efficient than having a series of PRINT statements each printing a line of the screen image.

It can also be used in a context such as:

```
ERROR.LINE = @(0,23) : @(-4) : 'Error: '

PRINT ERROR.LINE : 'Number must be an integer':
```

The action of the @ function is normally achieved by means of a file (typically called CURSOR). This holds, for each type of terminal, a string of characters which move the cursor and produce other effects on that particular terminal. This information is used at run-time, thereby enabling a compiled program to be used for any terminal.

7 File handling

File handling offers few surprises to anyone who is familiar with the fundamental Basic statements.

The file must first be opened by a statement of the form:

```
OPEN filename TO file-variable THEN/ELSE
OPEN 'DICT',filename TO file-variable THEN/ELSE
OPEN 'DICT filename' TO file-variable THEN/ELSE
```

The action of this statement is to find the base FID, the modulo and the separation of the file. This information is then placed in the file-variable. The contents of the file-variable cannot be used for any purposes other than in the statements shown below. It is convenient to think of *filename* as holding the external name of the file (as it will be used in TCL commands and Access sentences) and the *file-variable* as being the internal name of the file (as it will be used in this Basic program).

The *TO file-variable* element (and the *file-variable,* element of the statements shown below) is optional, and if this is omitted, then a default file-variable will be used. Only one such default file-variable is available. It is recommended that a file-variable is always specified.

If the item contents are to be read/written as a *dynamic*

array, then the READ and WRITE statements will be:

```
READ dyn-rec FROM file-variable, item-id THEN/ELSE
WRITE dyn-rec ON file-variable, item-id
```

If the item contents are to be read/written as a *dimensioned array*, then the READ and WRITE statements will be:

```
MATREAD dim-rec FROM file-variable, item-id THEN/ELSE
MATWRITE dim-rec ON file-variable, item-id
```

If a *single attribute* is to be read/written, then the READ and WRITE statements will be:

```
READV variable FROM file-variable, item-id, attr THEN/ELSE
WRITEV expression ON file-variable, item-id, attr
```

To delete a record, the statement is simply:

```
DELETE file-variable, item-id
```

There is no CLOSE statement on generic Pick, although Ultimate and Advanced Pick have a statement

```
CLOSE file-variable
```

which prevents any further READ / WRITE / DELETE statements being executed for the file.

7.1 File / item locking

According to the implementation which is being used, it may be possible to lock a single item (or the group which contains a specific item). When an item (or group) has been locked by one user, any other user who attempts to access that item (or group) will not be allowed to proceed until the first user has written the item back to the file and/or released the lock.

The item (or group) is locked by means of one of the statements:

```
READU dynarr FROM ...
MATREADU dimarr FROM ...
READVU variable FROM ...
```

is executed. The U at the end of the keyword indicates that the read is to be executed and the item (or group) locked for Update.

The lock on that item (or group) will be released only when that user executes a WRITE statement for that item, or when the user issues a:

```
RELEASE file-variable, item-id
```

statement.

If it is required to update that item on the file and still

retain the lock, then the format of the WRITE statement will be:

```
WRITEU dynarr ON file-variable, item-id
MATWRITEU dimarr ON file-variable, item-id
WRITEVU expression ON file-variable, item-id, attr
```

There is an extended form of the READ statements to allow the programmer to specify some action which is to be taken in the event of an item (or group) being found to be locked:

```
READU dynarr FROM fvar, id LOCKED/THEN/ELSE
MATREADU dimarr FROM fvar, id LOCKED/THEN/ELSE
READVU variable FROM fvar, id, attr LOCKED/THEN/ELSE
```

The LOCKED clause is constructed in exactly the same way as the THEN and the ELSE clauses and, like them, may be a multi-line structure.

A typical example of this statement might be:

```
100 READU RECORD FROM STOCK.FV, KEY LOCKED
        LOOP
            PRINT 'This record is locked. Wait/Abandon? ':
            INPUT RESP,1
        UNTIL RESP='W' OR RESP='A' DO
            IF RESP='W' THEN
                SLEEP 10;          ** Sleep for 10 seconds
                GO 100;            ** Try again
            END
        REPEAT
    END THEN
        GOSUB 1000; ** PROCESS RECORD
    END ELSE
        PRINT 'RECORD ':KEY:' NOT FOUND'
    END
```

7.2 SELECT / READNEXT statements

The general form of the Basic SELECT statement is:

```
SELECT {file-variable} {TO select-variable}
```

This will prepare to pass all the successive item-ids from the file which has been opened to the *file-variable* and make them available to a READNEXT statement of the form:

```
READNEXT key-variable {,position} {FROM select-variable}
    THEN/ELSE
```

via the *select-variable*.

The *position* parameter is used when the select-list was produced outside the program by means of a sentence such as:

```
SSELECT STOCK BY-EXP LOCATION
```

and represents the multi-value position of the various LOCATION fields in this instance.

The elements enclosed in braces {} are optional.

Note that there is no SSELECT statement in Basic, nor is there any facility for specifying selection criteria on the SELECT statement.

Unlike the TCL SELECT command, the Basic SELECT statement does not collect all the item-ids when it is executed. Instead, the action of the SELECT statement is to load into the select-variable, the item-ids of the items in the first group of the file. As the item-ids from this list are consumed by the READNEXT statement, the run-time processor places the item-ids in the next physical group into the select-variable. This continues until the end of the last group is reached; the ELSE clause on the READNEXT statement is then taken. One consequence of this is that, if the program creates new items on the same file, these too may be picked up when the SELECT/READNEXT action reaches the groups where the new items are held.

We illustrate some typical uses of these statements.

1) Example - omitting both the file-variable and the select-variable and using the defaults:

```
OPEN 'STOCK' ELSE STOP
SELECT
READNEXT STOCK.ID ELSE STOP
READ STOCK.REC FROM STOCK.ID ELSE STOP
```

2) Example - using the file-variable:

```
OPEN 'STOCK' TO STOCK.FV ELSE STOP
SELECT STOCK.FV
READNEXT STOCK.ID ELSE STOP
READ STOCK.REC FROM STOCK.FV,STOCK.ID ELSE STOP
```

3) Example - using the select-variable:

```
OPEN 'STOCK' ELSE STOP
SELECT TO STOCK.SV
READNEXT STOCK.ID FROM STOCK.SV ELSE STOP
READ STOCK.REC FROM STOCK.ID ELSE STOP
```

4) Example - using both the file-variable and the select-variable:

```
OPEN 'STOCK' TO STOCK.FV ELSE STOP
SELECT STOCK.FV TO STOCK.SV
```

```
READNEXT STOCK.ID FROM STOCK.SV ELSE STOP
READ STOCK.REC FROM STOCK.FV,STOCK.ID ELSE STOP
```

5) Example - using several select-variables:

```
*    OPEN FILES
OPEN 'STOCK' TO STOCK.FV ELSE STOP
OPEN 'INVOICES' TO INVOICE.FV ELSE STOP

*    SELECT FILES
SELECT STOCK.FV TO STOCK.SV
SELECT INVOICE.FV TO INVOICE.SV

*    MAIN PROCESSING
READNEXT STOCK.ID FROM STOCK.SV ELSE STOP
READ STOCK.REC FROM STOCK.FV,STOCK.ID ELSE STOP
READNEXT INVOICE.KEY FROM INVOICE.SV ELSE STOP
READ INVOICE.REC FROM INVOICE.FV,INVOICE.ID ELSE STOP
```

7.3 SELECT statement / SELECT sentence

The following point is important:

If there is an external select list produced before entry to the Basic program, then a SELECT statement in the program will be ignored and the READNEXT statement will gather the item-ids which were produced by the TCL SELECT (or SSELECT) sentence. This means that a program fragment such as:

```
OPEN 'STOCK' TO STOCK.FV ELSE STOP
SELECT STOCK
DONE=0
LOOP
     READNEXT STOCK.ID ELSE DONE=1
UNTIL DONE DO
     READ STOCK.REC FROM STOCK.FV,STOCK.ID THEN
```

can be executed without any preparatory selection at TCL and will select all the items on the STOCK file. However, if the program is executed in a TCL sequence such as:

```
>SSELECT STOCK WITH COLOUR "RED" BY PRICE

210 items selected

>RUN BP UPDATE.STOCK
```

then only the 210 items selected by the TCL command will be processed and not all the items on the file. Of course, the TCL sentence can include any sort specifications and/or sort criteria and the appropriate item-ids will be submitted in the appropriate sequence.

A similar effect can be achieved by using an EXECUTE statement to produce the select-list instead of the SELECT statement, as in this fragment:

```
OPEN 'STOCK' TO STOCK.FV ELSE STOP
EXECUTE 'SSELECT STOCK WITH COLOUR "RED" BY PRICE'
DONE=0
LOOP
     READNEXT STOCK.ID ELSE DONE=1
UNTIL DONE DO
     READ STOCK.REC FROM STOCK.FV,STOCK.ID THEN
```

Such a selected-list can be created by means of any of the verbs:

```
SELECT
SSELECT
QSELECT
GET-LIST
FORM-LIST
```

7.4 Several select-variables

With the Basic SELECT statement, it is possible to have several select lists active at one time, each with its own select variable. The EXECUTE statement of Ultimate and the PERFORM statement of McDonnell Douglas have facilities for loading the select-list from an EXECUTE (or PERFORM) statements into a select variable. Generic Pick, however, does not have such a facility, and it is only possible to use one such list at a time.

One solution is to build an internal list of the item-ids, and then use the items from this list for processing by the program. This is illustrated by the following program which processes all the items on all the files on an account: the list of file names is selected and collected into the dynamic array FILES, and the list of item-ids on each file is selected and processed directly.

```
* PROGRAM TO PROCESS ALL ITEMS ON ALL FILES
     EXECUTE 'SSELECT MD WITH *A1 "D"'
     FILES=''
     FILE.CNT=0
     DONE=0
     LOOP
         READNEXT FILE ELSE DONE=1
     UNTIL DONE DO
         FILE.CNT=FILE.CNT+1
         FILES<FILE.CNT>=FILE
     REPEAT
* FOR EACH FILE
     FOR F=1 TO FILE.CNT
         FILENAME=FILES<F>
         EXECUTE 'SSELECT ':FILENAME
         DONE=0
         LOOP
```

```
            READNEXT ITEMID ELSE DONE=1
         UNTIL DONE DO
*
* PROCESS ITEM 'ITEMID' ON FILE 'FILENAME'
*
         REPEAT
     NEXT F
     END
```

7.5 Handling backing storage

The Basic language has a number of statements which allow you to process data held on backing storage. The sequence of operations when using the backing storage with a Basic program is:

1) Issue the appropriate SET- command to indicate which device you are using and/or attach the device by means of the T-ATT command.

2) Indicate the record size which you will be using - by means of the T-ATT or the T-RDLBL command.

3) Mount the tape or diskette - with a write-permit ring if you are writing data to the device.

4) Invoke the Basic program.

5) Rewind the device - by means of the REWIND statement in the program or the TCL T-REW command.

6) Dismount the tape or diskette.

7) Issue a T-DET command to detach the device, releasing it for other users.

The SET- and T- commands may be invoked from within the Basic program by means of a series of EXECUTE/PERFORM commands:

```
EXECUTE 'SET-FLOPPY (AS'
EXECUTE 'T-STATUS' CAPTURING MESSAGE
EXECUTE 'T-ATT'
EXECUTE 'T-REW'    or REWIND ELSE ...
```

The Basic statements to manipulate the device are:

READT
to read a data record from the device:

```
READT TRECORD ELSE PRINT 'End of file reached'; STOP

READT TRECORD ELSE PRINT 'TAPE NOT ATTACHED'; STOP
```

WRITET
to write data to the device:

```
WRITET OPUTDATA ELSE GO 10
```

REWIND
to rewind the device:

```
REWIND ELSE PRINT 'TAPE NOT ATTACHED'; STOP
```

WEOF
to write an end-of-file marker on the device:

```
WEOF ELSE GO 400
```

As with all statements which offer a THEN / ELSE clauses, the THEN statements are processed after a successful operation. and the ELSE statements will be processed if the device has not been attached prior to running the program. The READT statement will also pass to the ELSE condition if the end-of-file marker is sensed.

The programmer should be completely aware of the *format* of the records when using the READT and WRITET statements.

If the program is to handle records which were written by the system software, then the following points are important:

* The individual items on a T-DUMP file are separated by ASCII character 251,

* A file-save or account-save cannot successfully be handled by a Basic program. This is because the delimiting characters used by these processes have a special significance to the Basic run-time processor and cannot be handled by a Basic program.

8 Subroutines

We shall assume that the reader is familiar with the concept and use of internal subroutines in Basic.

The following points are of interest:

* The subroutines are held within the program unit where they are used.

* They are reached by a GOSUB statement, and control is passed back to the main body of the program by a RETURN statement.

* They are usually isolated from the main body of the coding, protected by a GO TO statement, a STOP statement or some other device which will avoid their being executed other than by means of the GOSUB statement.

* The same variables and variable names are used in the main body of the program and in the subroutine.

The construction of a program as a set of *external* subroutines makes the development and testing of the program much simpler than if it were a single monolithic set of statements. This is particularly true if the program is very large and/or there are several programmers working on the

development simultaneously.

The following points are of interest:

* The item-id of the subroutine must be the same as that on the CALL statements which call the subroutine.

* The keyword SUB can be used instead of SUBROUTINE. The actual name specified on the SUBROUTINE statement is irrelevant on most implementations. Most programmers discover this by accident.

* The cataloguing of the subroutine and the main program varies between implementations:

  + Generic Pick requires the subroutines to be catalogued; it does not demand that the main program also be catalogued.

  + Some implementations [McDonnell Douglas] require the main program and the subroutines to be catalogued.

  + Some implementations do not require the subroutine to be catalogued if it is held on the same file as the calling program.

* If the subroutine has not been catalogued when it should have been, then a fatal error will result at run-time and the processing will abandon with the error-message:

  [B25] PROGRAM 'xxx' HAS NOT BEEN CATALOGED

* If the number of parameters on the CALL statement is fewer than on the SUBROUTINE statement, then a fatal error will result at run-time and the processing will abandon with the error-message:

  [B14] LINE n BAD STACK DESCRIPTOR.

* If the number of parameters on the CALL statement is greater than on the SUBROUTINE statement, then the run-time process may accept this without comment; the superfluous arguments will be ignored.

  I have also experienced the run-time error-message:

  [B27] LINE n RETURN EXECUTED WITH NO GOSUB

  under the same circumstances when the subroutine was otherwise quite sound.

8.1 Passing data to subroutines

Data may be passed to an external subroutine as a sequence of parameters (or arguments) in the CALL statement:

```
CALL SUB001(FILE,ITEM,SWITCH,RECORD)
```

or by means of the COMMON statement.

The following points may influence the decision as to how data is to be passed.

Data specified in a CALL statement (and in the corresponding SUBROUTINE statement) is physically copied from the variables used in the program to those used in the subroutine, and when the RETURN statement passes control back to the program, the data is physically copied from the variables used in the subroutine to those used in the program. This considerably slows up the processing if large amounts of data are passed to and from the subroutine. Some actual figures for the execution of various CALL statements are given in the section on timing.

Parameters in the CALL are used in situations where:

* The subroutine is to be used in several different and/or unrelated programs.

* The subroutine is to be used with different parameters at several places within the same program.

Data is passed by means of the COMMON statement in situations where:

* There is a considerable amount of data to be passed to and/or from the subroutine.

* The exact layout of the COMMON block is known and is the same for all programs and subroutines which will be used during the execution.

8.2 COMMON data

The COMMON statement allows you to arrange your data so that the variables are not held within the actual program or subroutine, but are held in a COMMON area which shared by a main program and all the external SUBROUTINEs which it calls. In order to indicate which variables are to be held in this common area, you will use a statement such as:

```
COMMON STOCK.F, STOCK.FV, ERROR
```

in the main program, and a corresponding

```
COMMON NAME, FVARIABLE, ERROR
```

in the subroutine.

Now, the first statement of the subroutine will be:

```
SUBROUTINE OPEN.FILES
```

without a parameter list, and the CALL statement will be:

```
CALL OPEN.FILES
```

also without the parameter list.

The COMMON statement must appear before any statements which use the common variables. As with the parameter list, the **order** in which the variables are declared is important, but not the names.

If you have written programs for any other computer system, you may have come across the concept of local and global variables. In the Pick operating system, COMMON variables are global variables, being available to the main program and all subroutines (which use a COMMON statement), whereas all other variables are local variables and only available to the program in which they are used.

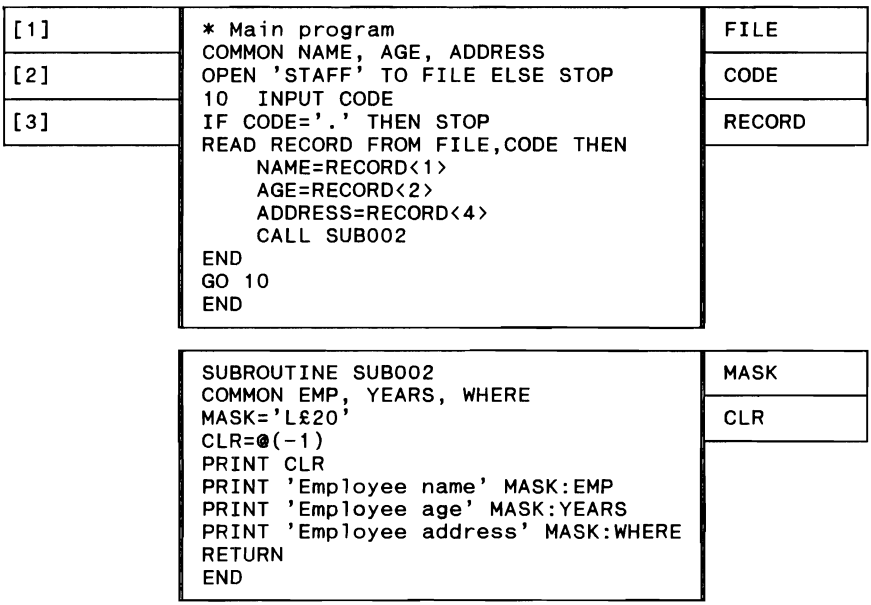
The following diagram depicts the way in which local variables are used in a Basic program and a subroutine. As the routine is compiled, the variables are assembled into a list (called the symbol table) and this list is used to store the contents of those variables at execution time. The variables in the list for one routine - the main program or a subroutine - cannot be accessed from another routine. Thus the variable FILE is inaccessible to the processing within the subroutine SUB001 and the contents of the variable CLR are inaccessible to the processing in the main program.

```
* Main program                          | FILE
OPEN 'STAFF' TO FILE ELSE STOP          |
10 INPUT CODE                           | CODE
IF CODE='.' THEN STOP                   |
READ RECORD FROM FILE,CODE THEN         | RECORD
    NAME=RECORD<1>                      |
    AGE=RECORD<2>                       | NAME
    ADDRESS=RECORD<4>                   |
    CALL SUB001(NAME,AGE,ADDRESS)       | AGE
END                                     |
GO 10                                   | ADDRESS
END                                     |
```

```
SUBROUTINE SUB001(EMP,YEARS,WHERE)      | EMP
MASK='L£20'                             |
CLR=@(-1)                               | YEARS
PRINT CLR                               |
PRINT 'Employee name' MASK:EMP          | WHERE
PRINT 'Employee name' MASK:YEARS        |
PRINT 'Employee address' MASK:WHERE     | MASK
RETURN                                  |
END                                     | CLR
```

When control is passed to the subroutine at the CALL SUB001 statement, the operating system copies the contents of NAME (from the main program) into the variable EMP (in the subroutine), AGE into YEARS and ADDRESS into WHERE. When the RETURN is executed, the data is copied back into NAME, AGE and ADDRESS.

The following diagram depicts the situation when COMMON variables are used to pass values between the main program and the subroutine SUB002. In this instance, there are three COMMON variables which we have numbered [1] to [3]. These COMMON variables are accessible from the main program - where they are known as NAME, AGE and ADDRESS, respectively - and also from the subroutine - where they are known as EMP, YEARS and WHERE. Any data which is placed into NAME in the main program goes into COMMON variable [1] and any data which is placed into (or taken from) EMP in the subroutine goes into (or comes from) COMMON variable [1]. The local variables FILE, CODE and RECORD are still inaccessible outside the main program, and MASK and CLR are still inaccessible outside the subroutine.



The following points are of interest:

* The variables are identified within the COMMON area by their position, so the sequence in which the names are specified on the COMMON statement is important.

  Thus, if a program had the statement:

  ```
  COMMON NAME, DEPT, AGE, TIME, DATE
  ```

  the name NAME would identify the first variable in the COMMON area, DEPT the second, AGE the third, and so on.

* The names of the variables in the various routines which use the COMMON area may be different (since they are identified only by their position), but it is usual for

them to be the same.

* The main program (which is loaded first during execution) must have the entire set of COMMON variables. The SUBROUTINEs need only have a subset of the COMMON variables, starting at the first COMMON variable.

  Thus, if a program and its subroutines requires five COMMON variables, the main program must declare all these:

```
COMMON NAME, DEPT, AGE, TIME, DATE
```

  whilst one subroutine may just use:

```
COMMON NAME, DEPT, AGE
```

  If one of the subroutines only needs to use the TIME and DATE variables in the COMMON area, then it would be a mistake (or at best, misleading) for that subroutine to have a statement of the form:

```
COMMON TIME, DATE
```

  since this subroutine would identify the first variable in the COMMON area by the name TIME, whilst the main program would identify this same variable by the name NAME. In this situation, there must be three dummy names given to pad out the TIME and DATE to the correct position. For example:

```
COMMON VAR1, VAR2, VAR3, TIME, DATE
```

* Advanced Pick has a facility for having *named* COMMON areas. The COMMON statement then has a form such as:

```
COMMON /PERSONNEL/ NAME, AGE, DEPT
COMMON /CALENDAR/ TIME, DATE
```

  using the names PERSONNEL and CALENDAR to identify two COMMON areas.

  A subroutine which only needs to use the TIME and the DATE variables would then include just the statement:

```
COMMON /CALENDAR/ TIME, DATE
```

* If a dimensioned array is to be declared as a COMMON area, then this will appear in a context such as:

```
COMMON NAME, DEPT, SALES(30)
```

  and will not be declared on a DIMENSION statement.

* A convenient technique is to use a COMMON block of the form:

```
COMMON PARAMS(100)
```

and then use the EQUATE statement to assign names to the individual elements of this array:

```
EQU NAME TO PARAMS(1)
EQU DEPT TO PARAMS(2)
EQU AGE TO PARAMS(3)
EQU TIME TO PARAMS(4)
EQU DATE TO PARAMS(5)
```

and so on. Then, a subroutine which only needs to use the TIME and the DATE variables would include just the statements:

```
COMMON PARAMS(100)
EQU TIME TO PARAMS(4)
EQU DATE TO PARAMS(5)
```

8.3 A technique for decomposing a basic program

It is frequently required to split a large program into a main program and several external subroutines. If the program has not been designed wisely, there may be a problem with passing variables between the various parts of the program. One solution to this is:

+ Create a file called BSYM if it does not already exist.

+ Compile the program using the M option. This put an entry on the BSYM file for every variable and label used in the program:

```
CLEAR-FILE DATA BSYM
BASIC PROGFILE MAINPROG (M
```

+ Select all the variable names from the BSYM file and use this to build a COMMON item on the program file.

```
SSELECT BSYM >='A' AND <= 'z'
SAVE-LIST X
COPY-LIST X
TO:(PROGFILE COMMON.ITEM
```

+ Put the word COMMON in front of each of the variable names in the COMMON item.

```
EDIT PROGFILE COMMON.ITEM
R999//COMMON /
FI
```

+ Split the program up into appropriate logical chunks and establish these as subroutines.

+ Create a new main program which comprises just a sequence of CALLs to the external subroutines.

+ Use the INCLUDE statement to include the COMMON item within the main program and each subroutine.

```
INCLUDE COMMON.ITEM
```

+ Compile and catalog the program and subroutines.

9 Passing control from one process to another

It is frequently required to pass control from one processing routine to another. In this section, we consider some techniques for doing this, and we see how data can be carried across during the transfer of processing.

9.1 Passing control: program to program

It is often required to pass control from one program to another, not as an external subroutine but to a completely independent program.

The ways of passing control from one program (we'll call this the *first program)* to another program (which we'll call the *second program*) include:

* Using the EXECUTE statement in a context such as:

```
     EXECUTE 'RUN PROGFILE PROGRAM2'
or
     CHAIN 'RUN PROGFILE PROGRAM2'
```

Data can be passed to the second program by means of DATA statements, and this can be picked up by means of INPUT statements in the second program.

FIRST PROGRAM

```
*
     DATA 'SMITH',32,'YES'
     EXECUTE 'RUN PROGFILE PROGRAM2'
*
```

PROGRAM 2

```
*
     INPUT NAME
     INPUT AGE
     INPUT YESNO.FLAG
*
*
* PROCESSING GOES HERE
*
     END
```

When the second program terminates, control will be returned to the first program if EXECUTE is used, but control will return to TCL (if appropriate) if CHAIN is used.

Generic Pick offers no means of passing data back from the second program to the first when using EXECUTE in this way.

Another way of passing data from one independent program to another is by means of the (I option on the RUN command. If

a program is executed by means of a command such as:

```
RUN PROGFILE PROG001 (I
```

or

```
PROG001 (I
```

(where the second form assumes that the program has been catalogued), then the descriptor table is not initialised and the contents will be exactly as they were left by the last Basic program which was executed.

9.2 Passing control: program to TCL

Any TCL command may be invoked from a program in by means of the EXECUTE/PERFORM statement. When the processing of the TCL process terminates, control will be returned to the program.

```
PROGRAM ──────────────────────────────
*
    EXECUTE 'LISTU'
*
    EXECUTE 'SORT ':FILE
*
──────────────────────────────────────
```

If the TCL process requires data, this may be passed via the DATA statement.

```
PROGRAM ──────────────────────────────
*
    DATA '(COPYSTOCK'
    EXECUTE 'COPY STOCK *'
*
──────────────────────────────────────
```

However, if the TCL process is a Proc, it is not normally possible to pass data in this manner. This is the situation with the current implementation of the ACCOUNT-SAVE verb (amongst others); these can be invoked from a program but conversational responses to the Proc's requests for data *cannot be fed from DATA statements within the Basic program.* The Proc must be rewritten (or some other solution devised) to accommodate this mode of invoking the process.

When passing data from the TCL command which invoked the program into the program, Advanced Pick offers the TCLREAD statement. This has the form:

```
TCLREAD variable
```

Ultimate implementations have the GET statements:

GET (ARG. {N}) v THEN/ELSE
: gets argument number *n* of any arguments specified on the RUN statement which invoked the program, and puts it into variable *v*. The ELSE clause is taken if there is no argument *n*.

GET (MSG. {N}) v THEN/ELSE
gets element number *n* of any message numbers and parameters resulting from the last EXECUTE statement, and puts it into variable *v*. The ELSE clause is taken if there is no element *n*.

Other implementations can use the solution of executing the program via a Proc, such as:

```
PQ
HRUN PROGFILE PROGRAM33
P
```

and then using the PROCREAD statement to pick up the contents of the Proc input buffer which contains the TCL command which invoked the Proc which invoked the Basic program.

The SYSTEM function call:

```
SYSTEM(15)
```

returns the options on the TCL command (that is, the RUN command or the catalogued program name) which invoked the program. Only alphabetic options are returned and these in ascending order.

Thus, if the program had been invoked by a command such as:

```
RUN PROGFILE TEST15 (QWSEDRFPTG12
```

or

```
RUN PROGFILE TEST15 (Q,W,S,E,D,R,F,P,T,G,12
```

then the statement:

```
PRINT SYSTEM(15)
```

within the program TEST15 would display:

```
DEFGPQRSTW
```

The options would still have their usual effect; in this instance, D would invoke the Basic debugger, P would send the output to the spooler, and so on.

9.3 Passing control: program to Proc

Control may be passed from a program to a Proc in the same manner as for any other TCL process. When the processing of the Proc terminates, control will be returned to the program.

PROGRAM
```
*
    EXECUTE 'MYPROC'
*
```

MYPROC
```
PQ
```

```
C PROCESSING COMES HERE
C
C
```

However, as mentioned earlier, conversational data cannot be passed via DATA statements since the Proc input statement(s) cannot access data from the input stack. A simple way of passing data into the Proc is via the Proc input buffer on the command which invokes the Proc:

PROGRAM
```
*
    EXECUTE 'MYPROC2 SMITH 32 YES'
*
```

MYPROC2
```
PQ
C PROC INPUT BUFFER CONTAINS
C  A1 = VERB
C  A2 = NAME
C  A3 = AGE
C  A4 = YES or NO
C
```

9.4 Passing control: Proc to program

Any process may be invoked from a Proc. A Basic program would be invoked by means of a Proc sequence such as:

PROC1
```
PQ
C Preliminary Proc processing here
HRUN PROGFILE PROGRAM3
P
C Follow up processing here
*
```

PROC2
```
PQ
C Preliminary Proc processing here
HRUN PROGFILE PROGRAM2
STON
HSMITH<
H32<
HYES<
P
C Follow up processing here
*
```

Data can be passed to the program via the stack and this will be picked up in the program by means of INPUT statements, exactly as we saw earlier.

A better means of passing data from a Proc to a program and back, is by means of statements of the form:

```
     PROCREAD variable THEN/ELSE
```
and
```
     PROCWRITE variable
```

statements. When the PROCREAD statement is executed, the contents of the primary input buffer of the Proc which invoked the program will be placed in the variable; if the program was not invoked from a Proc, then the ELSE condition will be taken. When the PROCWRITE statement is executed, the contents of the variable will be loaded into the primary input buffer of the Proc; this will normally be used when the Proc is to perform further processing when control is returned after the execution of the Basic program terminates.

9.5 Timing

The question of the speed and relative efficiency of the various Basic statements is of some importance. The table below shows some figures based upon a dedicated PC Pick implementation on an IBM PS/2 with a 286 processor.

In the table below, column 1 specifies the statement, column 2 gives the time (in milliseconds) for one iteration of the statement within the multi-line sequence:

```
FOR X=1 TO 1000
     statement
NEXT X
```

and column 3 gives the time for one iteration of the statement:

```
FOR X=1 TO 1000; statement; NEXT X
```

in which the entire loop is written on the same line.

Statement	Multi	Same
A=100	0.44	0.425
A=ARRAY(10)	0.51	0.495
A=ARRAY(Z) {Z = 10}	0.83	0.815
GO TO 100	0.265	0.255
GOSUB 100 and RETURN	0.55	0.535
A=1+6	0.6	0.55
A='1'+'6'	1.2	1.155
A=1*6	0.835	0.82
A=6*1	0.865	0.85
A=1/6	1.02	1.00
A=REPLACE(A,1,0,0,1)	1.8	1.79
A=REPLACE(A,1,0,0,'1')	1.53	1.51
A=A<1>	1.55	1.53
A=EXTRACT(A,1,0,0)	1.55	1.53
PRINT 3	1.57	1.56
PRINT 3 ON LPTR	1.65	1.63

A=@(5,5)	1.69	1.675
A=B:B:B:B {B is 10 characters long}	1.78	1.77
A=B:B:B:B {B is 20 characters long}	2.085	2.065
A=B:B:B:B {B is 30 characters long}	2.384	2.365
A=B:B:B:B {B is 40 characters long}	2.69	2.675
CALL SUB00	2.9	6.15
CALL SUB01(A)	3.545	6.98
CALL SUB02(A,B)	4.19	7.77
CALL SUB03(A,B,C)	4.84	8.58
OPEN 'MD' TO MD ELSE STOP	8.4	N/A
READ missingrec FROM 'BB' ELSE NULL	2.5	N/A
READ variable FROM 'EE' ELSE STOP	2.95	N/A
READ 250byterec FROM 'CC' ELSE STOP	5.05	N/A
READV VALUE FROM 'AA',1 ELSE STOP	3.5	N/A
READV VALUE FROM 'AA',5 ELSE STOP	3.9	N/A
READV VALUE FROM 'AA',30 ELSE STOP	5.8	N/A
READV VALUE FROM 'AA',50 ELSE STOP	7.45	N/A
WRITE variable ON 'EE'	4.3	4.3
WRITE 500byterec ON 'AA'	5.9	5.9
MATWRITE array10 ON 'EE'	6.00	6.00
MATREAD array10 FROM 'EE' ELSE STOP	6.75	N/A
READ 500byterec FROM 'DD' ELSE STOP	6.85	N/A
WRITEV variable ON 'AA',1	9.75	9.75
WRITEV variable ON 'AA',50	16.15	16.15
A=SQRT(46)	7.385	7.375
A=PWR(46,0.5)	16.91	16.91
A=EXP(LN(46)/2)	17.55	17.5

It is the *relative* timings for these various statements (not the actual time for any particular statement) which are of interest. Some are intuitively obvious, others less so.

The following points are of interest:

* The single-statement versions are generally faster. However, notable exceptions to this are the CALL statements.

* There is an anomalous difference between the timings for the CALL statements in the two different structures.

* Calling a subroutine with an argument list is affected by the number of arguments which are passed to and from the subroutine. This confirms that using COMMON variables is the best means of passing data to a subroutine.

* Statements which use variables are slower than those which use literals (or equated constants). Thus,

  ```
  PRINT 3
  ```

  executes faster than:

```
PRINT XYZ
```

if XYZ is a simple variable. If XYZ is equated to a constant:

```
EQUATE XYZ TO 3
```

there is no difference in the timings.

* Output to the spooler is slower than to the screen.

* Statements execute faster if the data is in the correct form (strings or numbers) for those statements. Thus,

```
REPLACE(A,1,0,0,'1')
```

which uses a string, executes faster then:

```
REPLACE(A,1,0,0,1)
```

which has to convert the number 1 to the string '1'. Similarly, the statement:

```
A = 1 + 6
```

is faster than:

```
A = '1' + '6'
```

* Concatenation is slow and the speed decreases as the length of the string increases.

* There is no great difference between the various dynamic array facilities. Compare A=A<1> with the EXTRACT function, and A<1>=1 with the REPLACE function.

* With the READ statement, the record from disk is read straight into the program where it is processed as a dynamic array.

* The MATREAD statement takes longer than the other READ statements because the incoming data record has to be parsed, each attribute of the dynamic array being interpreted as an element of the dimensioned array, and vice versa with the MATWRITE statement.

* Notice the relative speeds of READV versus READ, and of WRITEV versus WRITE. This is because the WRITEV has first to read in the original item, replace the attribute and write the entire record out.

* The assembler code for the REPLACE and the EXTRACT functions is identical to that for the corresponding A<n> references, and the timings are therefore the same.

Index

MB-Guides

013

The booklets in the *MB-Guide* series cover a range of fundamental topics of interest to users and those responsible for running Pick systems.

Each *MB-Guide* deals with a specific aspect of the operating system and the booklets represent an economical introduction to the various topics and the whole series forms an integrated presentation of the subject matter.

The booklets are intended to be a working document and, for this reason, space is provided for the user's notes, and the reader is encouraged to amend the booklet so that it applies to his/her own system.

It is anticipated that the series of *MB-Guides* will be of special interest to new users, and it should prove useful for training organisations, software houses and others who are responsible for the instruction of their clients and staff in the fundamental aspects of the Pick operating system.

Topics in Basic

Malcolm Bull

Training and Consultancy Publications