BASYS - A LANGUAGE FOR PROGRAMMING INTERACTION

B.R.Gaines and P.V.Facey

Summary - Over the past 10 years we have had extensive experience in the development and application of low-cost, interactive, minicomputer-based systems designed for close collaboration between people and computers. In papers elsewhere we have described and analysed these systems and the human factors involved. In particular we have suggested strategies for programming the interaction between the naive user and the computer. This paper concentrates on the software technology underlying these systems and analyses in some detail the types of string-handling facilities in particular that are needed to program interactive dialogue simply, flexibly and effectively. An outline is given of how these facilities are incorporated in the language BASYS that has been used in our applications.

1. Introduction

BASYS is a high-level language with extensive string handling facilities, whose syntax is similar to Dartmouth BASIC, and which has been implemented as a compact interpreter on a wide range of minicomputers. Over the past ten years BASYS-based minicomputer systems have been used for many different applications, including industrial and clinical instrumentation, psychological testing, hospital administration, and a variety of commercial systems for security and foreign exchange dealing. Previous reports have detailed the applications (Refs. 1-5) and analysed the problems of programming interactive dialogue for naive users (Refs. 6-7). This paper is concerned with the language itself, design considerations, facilities and implementation. It is not intended to promote BASYS as a 'standard language' (since it has never been such and varies from implementation to implementation), but rather to encourage the use of simple, interpretive 'kernel' languages as a key component of interactive systems, and to pass on our own experience to other designers of such systems.

2. Design Objectives

The key objectives established for the design of BASYS were:

(1) A 'high-level' language, simple to use and as readable as possible - because we envisaged programs undergoing rapid field development and modification, and needing to be as 'self-documenting' as possible - the syntax of Dartmouth BASIC was chosen as a model because experience had shown it to be simple to comprehend and very readable;

(2) Interactively modifiable - because the design of man-computer dialogues is an exercise in bringing man and machine into a close working relationship and we needed to be able to modify dialogue sequences rapidly in the light of user comment, and whilst the the user was still present - the interpretive implementation has made this relatively easy to achieve;

The authors are with the Man-Machine Systems Laboratory, Department of Electrical Engineering Science, University of Essex

(3) Readily extensible - because BASYS was intended as a 'kernel language' for a variety of specialist applications requiring special features as an integral part of the language - extensions incorporated in practice have included: command processors for remote microfilm terminals, speech synthesizers and mass spectrometers; and various data structure manipulation commands for speed and ease of understanding;

(4) Supporting natural interactive dialogue - because ease and flexibility in programming interaction with users was essential to all the applications envisaged, and we were determined to avoid the imposition of syntactic or semantic constraints on the form of the dialogue - in practice the 'pattern-matching' string processing operations of BASYS have now been widely used by both university and commercial programmers to generate a great variety of dialogue sequences that appear simple and natural to non-computer-oriented users;

(5) Allowing close and direct communication with the operating system - because many of the applications were 'database' orientated and efficient use of filing facilities was essential, and because on many minicomputer systems the operating system itself was so primitive that it was necessary to substantially extend it and the best way to do this was in BASYS itself - all implementations have the command structure of the operating system under which they run incorporated as an integral part of the language itself;

(6) Giving high-precision integer arithmetic - partly because many of the initial applications were financial where exact arithmetic is required, but also because even in, for example, medical database systems we found that the normal floating-point plus 1-word integer arithmetic normally available in minicomputer interpretive systems was inadequate - most implementations of BASYS offer variable-precision integer arithmetic up to some 19 digits;

(7) Compact implementation - we put this last not in importance but because it is still essential despite the other key requirements ! - for many of our applications the use of a high-level language has to be balanced against the extra storage required in what is intended to be a low-cost minicomputer, or microprocessor, system - in practice BASYS implementations seem to come out at about 4K machine words including all tables, editing phase, and workspace used by the language itself - user program partitions vary from application to application but are typically between 2K and 4K bytes.

3 Creating, editing and debugging BASYS programs

BASYS retains the syntactic structure of BASIC in that a program is a sequence of numbered lines ordered by their, not necessarily consecutive, line numbers, and statements consist of a meaningful key-word specifying a command processor followed by an expression, or sequence of expressions, that serve as parameters to it, e.g.:

```
 15 LET P=180
 46 PRINT 'The value of P is' P
137 INPUT X Y Z
240 PRINT (X+Y)/Z Z*Y P+(X-Y)/6
402 GOTO 15
```

The meaningful key-words seem to contribute much to BASIC's high readability, and the need to insert them itself prevents the vast, opaque syntactic constructions possible in ALGOL and mandatory in LISP.

The role of the line numbers and their use in transfers of control is more open to controversy, and we have debated it on many occasions over the years, particularly in relation to more meaningful labels. However, in the context of interactive programming where the programmer at a terminal cannot see all the text at a time, the use of numeric labels having the topology of the ordinary number system is itself an advantage. The programmer has an immediate 'picture' available to him of the layout of his program - going on to use the same line numbers as 'labels' for the transfer of control is then a minimal, natural construct, requiring the acquisition of no new concepts. We accepted the use of line numbers in BASIC when designing BASYS and have not regretted it. Indeed in BASYS it is extended to enable both string constants and dynamically-varying string data to be stored and referenced as part of a structure of numbered 'program'

In our class of applications creating a program at a terminal, or entering one sketched out off-line, or modifying one in use, are important activities to be performed simply and ergonomically. We wished to minimize the effort of program creation and documentation and maximize the clarity of the result. This led to the sub-objectives: (a) No unnecessary syntax on program entry - the programmer should be able to use the minimum string necessary to specify a statement; (b) Full clarity in program listings - the system should re-create the missing syntax on output and format it appropriately.

To achieve this we had to drop some BASIC conventions, notably the non-significance of spaces which are natural separators readily inserted with the space bar. This allowed a comma, or one or more spaces, or implicit separation, to be specified as optional separators. Command key-words could then be specified as a string of letters which matched, or partially matched, one of the standard commands. Hence any command could be shortened to its minimum unambiguous initial string. We chose command names so that the first two letters alone were always sufficient to resolve ambiguity, and a single letter, if ambiguous, was interpreted as the most frequently used of the possible commands, e.g.:

```
15L P=180
46P'The value of P is'P
137I X Y Z
240P(X+Y)/Z Z*Y P+(X-Y)/6
402G15
```

shows how the previous program might be entered - on listing it would be expanded to its full form.

The following example shows a sequence of actual program creation and debugging at a terminal. The '>' (go-ahead) is printed to indicate that BASYS is in edit mode ready to accept terminal program input. The '[' is a 'no-operation' command allowing the free entry of comments. When line 110 is entered a syntax error is immediately indicated because the initial command cannot be decoded. The programmer re-enters it correctly and terminates

```
>1[Created as a demo 24/10/76
>25P'DEMO
>80I X Y
>100P X X*X
>110LX=X+1:UN X>Z:G100
Syntax error
>110L X=X+1:UN X>Z:G100
 110 LET X=X+1 :UNLESS X>Z :GOTO 100
>LI
   1 [Created as a demo 24/10/76
  25 PRINT DEMO
  80 INPUT X Y
 100 PRINT X X*X
 110 LET X=X+1 :UNLESS X>Z :GOTO 100
>RUN
DEMO
:4 7
    4   16
Variable undefined at 110
 110 LET X=X+1 :UNLESS X>Z ! :GOTO 100
>X110/Z/Y
 110 LET X=X+1 :UNLESS X>Y :GOTO 100
>G100
    5   25
    6   36
    7   49
>P X Y
    7    7
>
```

it with an 'ESCAPE' rather than 'CARRIAGE-RETURN' causing an immediate listing of the entered line. He then asks for the program to be listed (note that 'L' alone becomes LET, but 'LI' becomes LIST) and then run. An error in the running program generates an informative message together with a listing of the offending line with a '!' showing where execution ceased (the variable 'Z' being undefined). The programmer edits line 110, changing Z to Y, and continues the program (with all variables unchanged). When it terminates and exits to the edit phase he is able to check the state of the variables, and could go on to enter more program and continue execution if he wished.

Note that there is no distinction in BASYS between 'stored' and 'directly-executed' commands - any can be either, and indeed commands such as LIST are often implemented as BASYS procedure calls on some 'hidden' BASYS programs that perform the required operation. The so-called 'edit-phase' is actually generated by all programs being linked to a final line of the form:

INPUT <string> :CODE <string> :LOOP

where <string> is a string-variable and CODE is a command that encodes its argument as a program statement, directly executing it if it has no line number. Thus the interpreter does not distinguish between 'edit-phase' and 'run-phase' and all commands and accesses to data structures are freely available in either.

Line 110 of the program above also makes apparent another feature of BASYS, that a number of commands may follow one another, separated by colons, on the same line. This is both convenient in grouping material together for readability, and

gives a powerful extension to the form of conditionals in BASIC. BASYS is a 2-dimensional language in which execution continues along a line until a conditional fails or the line terminates, and then passes to the next line. For example:

```
100 PRINT X X*X :LET X=X+1 :UNLESS X>Y :LOOP
```

(where LOOP transfers control back to the beginning of the same line) is equivalent to lines 100 plus 110 of the previous program.

Conditional tests are regarded as decisions to continue execution of the current line or go on to the next. Advantage is taken of this to generate implied conditionals in other commands particularly input-output statements which may fail for good reasons (end-of-file), and the pattern-matching string operations described later which may fail through lack of match. For example:

```
100 OPEN 'FILE' :GOTO 120
110 PRINT 'Cannot find FILE' :STOP
120
```

or better:

```
100 OPEN 'FILE
110 ELSE :PRINT 'Cannot find FILE' :STOP
120
```

each use the implied conditional in the file OPEN command to test its success. The second form shows how the use of implied conditionals together with the ELSE construction in BASYS (ELSE continues execution if the previous conditional failed) allows readable, GOTO-less programming using a few simple and natural constructs.

These examples and their rationale illustrate the 'flavour' of BASYS, its simplicity and the close integration of facilities for program creation, editing, debugging and listing. Hopefully they also indicate the high readability of BASYS, even greater than that of BASIC because of the 2-dimensional structure that encourages logical command groupings. The appendix gives a synopsis of the language - it will be seen that other obvious generalizations have been made: numerical expressions can occur wherever a number might; variable names are not restricted to a letter or a letter/digit, and so on. In general, these extensions in making the language simpler and more uniform for the programmer have also simplified implementation and led to a more compact interpreter.

4 String processing in BASYS

Our early experience in programming interactive dialogues had convinced us of the need for facilities to process character strings which went well beyond those of any commonly available language except SNOBOL. We were not aiming for 'natural language' with all its ambiguity and complexity but rather 'natural dialogue' for users already in a semi-formal situation, e.g. the acceptance of precisely those forms in which a clerk would have previously written information of a file-card - an acceptance unhindered by imposed syntactic constraints (e.g. commas being separators or certain characters have special meaning) and obvious in the very form of the language statements used to decode the input. Our initial aims were strongly influenced by the template-matching, contextual analysis, facilities necessary

to implement an ELIZA-type of program. Experience in the design and implementation of contextual editors also led to many features of the current system.

Our interpretive implementation of BASYS automatically gave us facilities for manipulating the character strings forming program lines, and it was natural to store string data in the same way. Any 'program' line beginning with a '$' 'command' is a string variable initially containing whatever characters follow the $. Such variables are referenced as $ <line number> where <line number> is a numeric expression evaluating to a line number, e.g.:

```
>10$Hi there
>PRINT $10
Hi there
>LET K=7 :PRINT $K+3
Hi there
>LIST
  10 $Hi there
>INPUT $10
:The end
>LIST
   10 $The end
>
```

Apart from its simplicity of implementation, this mechanism for string variables: (a) enables string variables and constants to be listed as part of the program; (b) enables string variables and constants to be placed in those parts of the program where they are used; (c) makes string arrays naturally available and, in particular, efficiently implements sparse string arrays.

String expressions containing string constants, variables and literals, and numeric variables converted to strings have a natural syntax given in the appendix, e.g. continuing from above

```
>PRINT 'What is' K ' times' K+1 $10
What is   7 times   8The end
>
```

Contextual string analysis is based on the concepts of a source string being analysed, a destination string to which output may be appended, and various commands for matching patterns against the source string and routing results to the destination string. The system variable QS contains the line number of the source string, QD that of the destination string, and QP contains a pointer to a character within $QS. The system variables are automatically set up by the string processing commands and need not be manipulated by the programmer. However access to them is useful in very complex string processing applications.

The command PUT <string expression> sets up a source string by assigning the value of the string expression to $QS and setting QP to zero (the command INPUT serves the same purpose but gets the string from the terminal). The command AS <string variable> sets up the string variable as a destination string, initially null, and assigns its line number to QD. The command WITH <string expression> is used to append the value of the string expression to $QD, typically to replace a pattern matched in the source string.

TO, FROM and SEEK are pattern search commands specifying a

pattern, or template, to be looked for in the source string $QS starting from the QP'th character. If the pattern is found then QP is updated to point beyond it, any appending to $QD is carried out, and execution proceeds to the next command in the current line. Otherwise, if the pattern is not found neither QP nor $QD are changed and execution drops through to the next line. FROM specifies an anchored search (for an initial pattern), and SEEK an unanchored search (for an imbedded template). TO specifies an unanchored search in which characters from the source string prior to the pattern matched are appended to the destination string. The forms of pattern template allowed are specified in the appendix and include string variables, literal strings, numeric strings (automatically converted and assigned to numeric variables), and a specified number of characters. The commands and templates are powerful enough to cover most requirements, but simple enough to enable the string analysis to be expressed comprehensibly, e.g.:

```
>1 $ABCDEF
>PUT $1 :AS $2 :FROM 'AB' :TO 'EF'
>PRINT $2
CD
>
```

The following example accepts the date in one of two formats:

```
>10 PRINT 'Date', :INPUT
>20 SEEK D '/' M '/' Y
>30 ELSE :SEEK D '/' M :LET Y=77
>40 ELSE :PRINT 'Date as 6/7/75, or 3/9, current year' :GOTO 10
>50 PRINT D M Y :GOTO 10
>RUN
Date:5/7
    5    7   77
Date:2/12/45
    2   12   45
Date:4-5-77
Date as 6/7/75, or 3/9, current year
Date:
```

And the following implements a simple calculator language:

```
>10 PRINT 'Calc', :INPUT :SEEK 'ADD' X Y :PRINT X+Y :LOOP
>20 SEEK 'SUB' X Y :PRINT Y-X :GOTO 10
>30 SEEK 'MUL' X Y :PRINT X*Y :GOTO 10
>40 PRINT 'That is a bit beyond me old boy' :GOTO 10
>RUN
Calc:ADD 3 5
    8
Calc:SUBTRACT 9 FROM 20
   11
Calc:Please could you MULtiply 4 by 3
   12
Calc:What is 8 DIVided by 4
That is a bit beyond me old boy
Calc:
```

String conversion filters are available, for example, to convert all characters to lower or upper case, and could have been used in this example.

These examples demonstrate the operation of the pattern-matching string analysis facilities in BASYS in simple situations. The

combined use of all the facilities together with computations with, and assignments to, QS, QD and QP, allows very complex string analysis to be carried out. What should be apparent, however, is the naturalness of this command family for use in the commonly required string analysis needed to support interactive data entry and dialogue. Our objective was not only to provide powerful string-processing facilities, but also to retain the readability, and the transparency of function, which are such important features of BASIC.

5 Procedures and the stack in BASYS

In early implementations of BASYS space for all numeric variables was allocated from a single level store. Once created variables names were static and retained their values between subroutine calls, program overlaying, and so on. Such a simple allocation scheme is adequate for small programs up to about 200 lines, but places an increasingly onerous memory load on the programmer as they become larger. For the suites of 30 or more 200 line programs that typified our commercial and medical systems, we found that the use of the same variable name for different purposes in different places was a frequent source of errors. We found ourselves generating elaborate cross-reference packages and losing many of the advantages of rapid development otherwise available with BASYS.

The obvious extension was to provide a simple block structure restricting the scope of names, and this has been done in recent implementations by assigning storage for simple variables and arrays from a system stack. The simple variable Q acts as a stack marker containing the line number of the command that has caused entry to a new block, and the command LOCAL is a form of LET which creates a new variable on the stack if there is not one above the topmost stack marker; variables above the current top stack marker are themselves called local to the current block. The command BEGIN X Y Z places a marker Q on the stack whose value is the current line number and sets up local variables X, Y and Z. The commands END, NEXT and BACK, are conditionals which can move the stack pointer below the topmost marker, i.e. remove local variables, and transfer execution to the line number following the value in Q. Between them they give the three possible combinations of these two operations and allow block structures, iterative loops, and procedure returns to be implemented.

The BASIC GOSUB command has been dropped as a means of implementing procedure calls. Instead, the command DO <line number> in BASYS causes execution of the specified line but does not pass control to it unless it is a BEGIN. When the line executed is a BEGIN then a marker Q with the calling line number is placed on the stack and control is paased to the executed block. The **local** variables declared in the BEGIN statement pick up any parameters listed after the line number in the DO statement, and may otherwise be assigned default values. Parameters may be passed by value or by reference, e.g.

DO 90 A K+3 45 %G

executed when A=5 K=4 G=-7 and line 90 is

90 BEGIN A B C=1 D=-50 E=100 F=E

leads to the setting up of local variables with the following values: A=5 (passed), B=7 (passed), C=45 (passed, default

ignored), D=-7 (passed as reference to G in outer block, default ignored), E=100 (default picked up), F=100 (default picked up - note that E is already available as local variable !). Note that A in the local block is quite separate from the A outside it which has become inaccessible, whereas D is a reference to G and hence gives direct access to it - in fact G in the outer block is available both as G and as D within the inner block. Thus the assignment A=0 D=2 is effective in the inner block and, on return, A=5 as before, whereas G=2 because of the assignment to D. Return is effected by the command BACK which deletes the local variables and stack marker Q, returning to the line whose number is next greater to Q. Note the availability of Q within the procedure enables return switches to be implemented.

A BEGIN command need not mark the beginning of a procedure but can just initiate a block with local variables terminated by an END. The ARRAY command in BASYS replaces the BASIC DIM and is executed at run time so that local arrays can be created on the stack and the space later returned. Reference variables are allowed not only in parameter passing but may also be created by assignment. This is very useful in setting up symbolic references to record elements of variable lengths stored in arrays, particularly since BASYS uses arrays for transferring arbitrary length, arbitrary structure, records to and from disc.

The facilities for variable name management, parameter passing and dynamic arrays in BASYS are again very simple and the data structures involved are readily comprehended. They are adequate however to remove problems of name conflict without undermining the simple mechanism of user program overlay communication in BASYS whereby the names and values of simple variables and arrays are retained between overlays. We have considered more elaborate schemes but concluded that to go further would give only minor advantage whilst reducing the simplicity of conception and use which is a feature of BASIC-like languages.

6 Conclusions

We hope that this brief paper gives enough of a feeling for BASYS and its rationale to be of value to other interactive language designers and users. Further information on applications will be found in Refs.1-5 and on implementation in Ref.8.

7 References

1 B.R.Gaines and P.V.Facey, Some experience in interactive system development and application, Proc.IEEE 63, 894, 1975.

2 B.R.Gaines, P.V.Facey and J.Sams, Minicomputers in security dealing, Computer 9, 6, 1976.

3 B.R.Gaines, P.V.Facey and J.Sams, An on-line fixed interest investment analysis and dealing system, Proc.EUROCOMP 74, 155, 1974.

4 P.V.Facey and B.R.Gaines, Real-time system design under an emulator embedded in a high-level language, Proc. BCS DATAFAIR 73, 285, 1973.

5 T.C.S.Kennedy and P.V.Facey, Experience with a minicomputer-based hospital administration system, Int. J. Man-Machine Studies, 5, 237, 1973.

6 B.R.Gaines and P.V.Facey, Programming interactive dialogues,

Proc. Conf. Computing and People, Leicester Polytechnic, 1976.

7 T.C.S.Kennedy, The design of interactive procedures for man-machine communication, Int. J. Man-Machine Studies 6, 309, 1974.

8 B.R.Gaines, Interpretive kernels for microcomputer software, Proc.Symp. Microprocessors at Work, University of Sussex, 56, 1976.

9 Appendix - Synopsis of BASYS

Arithmetic operators in BASYS

Precedence	Op	Meaning
1	←	Shift
1	↑	Exponentiation
2	-	Unary minus
3	/	Division
3	*	Multiplication
4	-	Subtraction
4	+	Addition
5	<	Less than) Arithmetic relations -
5	>	More than) any combination of these
5	=	Equal)
6	<	Alphabetically less than) String comparisons -
6	>	Alphabetically less than) any combination of
6	=	Equal) these
6	←	a←b true if string-a contains string-b
6	↑	a↑b true if string-a begins with string-b
7	'or"	quotes enclosing literal string
7	S	right-associative operator meaning string name
8	&	AND
9	!	OR

String expression fields in BASYS

A string expression <se> is anything that can be evaluated to yield a string in the same way that an arithmetic expression <ae> is anything that can be evaluated to give a number. An <se> is built up by concatenating the strings in its constituent fields. The main fields are:

$<ne>	String in dollar-line <ne>
'string'	The quoted string
"string"	The quoted string
<ne>	The value of <ne> converted to a numeric string
;	Carriage-return and line-feed
,	Null string - comma is used as field separator
%S<ne><ne'>	Substring<ne>of dollar-line<ne'>
%C<ne>	The ASCII character whose value is <ne>
%P<ne>	The program line <ne>
%R<ne>	Change output radix to <ne>
@<ne>	Change output format to <ne>

Storage commands - no effect on execution

$<characters> sets up character string for use in program

[<characters> sets up character string for comments

Editing and housekeeping

LIST <ne> <ne'> gives formatted listing of lines <ne> to <ne'>

ZERO <ne> <ne'> deletes program lines <ne> to <ne'>

CODE <se> analyses <se> as program input

GARB forces garbage collection - usually automatic

Assignment

LET <assignment$_1$> <assignment$_2$> etc, where an assignment is by value, <name> = <ne> , or by reference, <name>%<name'> , or both, <name>%<name'> = <ne>.

LOCAL <assignment$_1$> <assignment$_2$> etc., is similar to LET but only searches the local environment when setting up a new simple variable.

ARRAY <letter> <ne1> <ne2> <ne3> sets up the array, <letter>, of size <ne1> bytes, with elements <ne2> bytes long, and with a multiplier if 2-dimensional of <ne3> - <ne3> absent or zero signifies a 1-dimensional array.

Conditionals

IF <ne> continues execution of current line if <ne> non-zero

UNLESS <ne> continues execution of current line if <ne> zero

ELSE continues execution of line if previous conditional (except AND/ELSE) did not

AND continues execution of line if previous conditional (except AND/ELSE) did so

Transfer of Control.

DO <ne> executes program line <ne> - control returns to line following DO unless line executed commences with BEGIN (see next section)

GOTO <ne> transfers control to line <ne>

LOOP tranfers control to the beginning of the current line

RUN <ne> deletes simple variables and arrays, resets system parameters, and transfers control to line with next greater or equal number to <ne>

STOP stops execution and returns to keyboard edit phase

EXIT stops execution and CALLs standard system program

BYE stops execution and logs user off system

Procedures and Iteration

BEGIN <name1> = <ne1> <name2> = <ne2> etc

puts local variable Q on stack and sets it equal to current line number - sets up local variables, <name1>, <name2>, etc, and assigns them values, <ne1>, <ne2>, etc, unless parameters have been passed from a DO statement

DO <ne> <param1> <param2> etc

just executes line <ne> unless this commences with BEGIN when it sets Q on stack with value its line number and passes parameters by value or by reference to the local variables specified in the BEGIN

BACK <ne> if <ne> is omitted or non-zero, causes local

environment to be deleted and control to be passed to line at Q+1 or next greater - otherwise has no effect - this is the procedure return

NEXT <ne> if <ne> is omitted or non-zero this causes control to be passed to the statement at line Q+1 or next greater - otherwise deletes local environment and continues execution - this is iterative loop control

END clears the local environment and continues execution - this is termination of BEGIN block

String operations

PUT <se> sets up source string $QS containing <se> - sets QP=0

AS $<ne> <se> sets up destination string S<ne> and puts <se> in it - sets QD=<ne>

WITH <se> appends <se> to $QD

FROM <pattern> examines $QS from the QP'th character for the pattern - fails if pattern not found immediately - otherwise advances QP to point beyond pattern

SEEK <pattern> runs through $QS from QP'th character looking for pattern - fails if not found - otherwise advances QP to point beyond pattern

TO <pattern> same as SEEK but also if pattern is found it appends characters skipped over in $QS to $QD

A pattern consists of a succession of fields indicating matches or changes to the command parameters - it matches if and only if all its field match in sequence. The main fields are:

$<ne>	match string in $<ne>
or 'string' "string"	match quoted string
<variable>	any variable to which a numeric assignment is possible matches a field which is - [spaces] [+,-,null] [spaces] [digits] [decimal point] [digits] and any number thus matched is converted and assigned to the variable
@<ne>	matches <ne> characters
%F<ne>	changes format specification for numeric input

Peripheral transfers

As previously noted these are dependent on the operating system used but generally includes -

CALL and SAVE programs and overlays

ALLOCATE and RELEASE channels

OPEN, CREATE, RENAME, CLOSE and DELETE files

INPUT and PRINT character strings

READ and WRITE data blocks