The BCPL Programming Manual

BY TM 75-10

Martin Richards

Edited by

J.E.L. Peck and V.S. Manis

Technical Manual 75-10

DECEMBER 1977
DEPARTMENT OF COMPUTER SCIENCE
THE UNIVERSITY OF BRITISH COLUMBIA
VANCOUVER, BRITISH COLUMBIA V6T 1W5

Introduction

BCPL is a programming language designed primarily for non-numerical applications such as compiler-writing and general systems programming. It has been used successfully to implement compilers, interpreters, text editors, game playing programs and operating systems. The BCPL compiler is written in BCPL and runs on several machines including the IBM 370/168 at the University of British Columbia.

Some of the distinguishing features of BCPL are:

The syntax is rich, allowing a variety of ways to write conditional branches, loops, and subroutine definitions. This allows one to write quite readable programs.

The basic data object is a word (32 bits on the 370, 16 bits on many minicomputers) with no particular disposition as to type. A word may be treated as a bit-pattern, a number, a subroutine entry or a label. Neither the compiler nor the run-time system makes any attempt to enforce type restrictions. In this respect BCPL has both the flexibility and pitfalls of machine language.

Manipulation of pointers and vectors is simple and straightforward.

All subroutines may be called recursively.

This manual is not intended as a primer; the constructs of the language are presented with scant motivation and few examples. For a more tutorial presentation see the BCPL Primer (Dept. of Computer Science, UBC). To use BCPL effectively one must have a good understanding of how the machine works and be familiar with its operating system. To the experienced and disciplined programmer it is a powerful and useful language, but there are few provisions for the protection of naive users.

The main body of this manual describes the "official" standard subset of BCPL which will be supported at most BCPL installations. Many implementations provide extensions to the language and a summary of the extensions available on the 370 implementation can be found in Appendices D and E. Users are strongly recommended to remain within the standard subset unless there are exceptionally strong reasons for not doing so.

Acknowledgements

The overall layout and organisation of this manual is based on a manual written by J.H.Morris of the University of California, Berkeley, which itself was based on a well-written memorandum by R.H.Canaday and D.M.Ritchie of Bell Telephone

Laboratories.

The initial design and implementation of BCPL was done on CTSS at Project MAC in 1967 by M. Richards and since then the language has developed and been transferred to many machines around the world.

An OS machine code library was implemented for the 370 by J.K.M.Moody and many of the language extensions for the 370 were implemented with the assistance of H.C.M.Meekings. Many of the extensions were first designed and implemented by J.L.Dawson.

The language design was strongly influenced by the author's experience with CPL. This language is described by D.W.Barron et al. in "The Main Features of CPL", The Computer Journal, Vol. 6, p.134.

A number of extensions to BCPL have been made over the last few years. These extensions, which are found in a number of implementations, are flagged in this manual by placing a sharp (#) to the right of the text. #

The MTS version (BCPL-V)

The MTS adaption of BCPL is the work of V.S.Manis and M.DuMont at UBC, Vancouver, and the MTS machine code library is patterned after the OS version. Some extended features of the language and additional library routines available only in the MTS version are included in this revised document but are everywhere identified as belonging to BCPL-V. Users are warned that use of such features reduces portability of programs. At UBC, further work on BCPL has been done by John Peck, Kenny Wong, Stephen Ma, Ivor Ladd, Stephen Rand, Grant Weddell, Noel Kalicheran, David Lowe, David Mielke, Bill Webb and others.

Modifications made at UBC are denoted in this manual by a colon (:) to the right of the text. A stick (|) in the right margin indicates that the language feature is available in a number of other implementations, but perhaps in a different form. :#

Portability

One of BCPL's main goals is portability. For this reason, the compiler can produce an intermediate code which is the language of a hypothetical computer; a translator from this intermediate language to a specific machine language is relatively easy to construct. Interested readers are referred to "The Essence of Computer Science", by J.E.L. Peck, UBC Department of Computer Science TM 75-7.

A number of BCPL compilers exist at UBC, including one on a 32K Nova running RDOS at Electrical Engineering; one on a 56K Word PDP-11/40 running UNIX at Animal Resource Ecology; and

(soon) one on a 32K word Hewlett-Packard 21MX running RTE-II at Computer Science. For details, the reader is referred to the appropriate user's manual.

Part I

An overview of BCPL

This part is being revised.

Part II

The BCPL Language and Portable Library

Language Definition

Program

At the outermost level, a BCPL program consists of a sequence of declarations. To understand the meaning of a program, it is necessary to understand the meaning of the more basic constructs of the language from which it is made. We therefore choose to describe the language from the inside out starting with one of the most basic constructs, namely the

'element'.

Elements

```
<element> ::= <identifier> | <number> |
              <string constant |
              <character constant> |
              TRUE | FALSE |
                ?
```

An <identifier> consists of a sequence of letters, digits, points, and underlines, the first character of which must be a letter.

In sensible environments, the use of lower-case letters in identifiers is permitted (and encouraged!). Case is significant in identifiers, but not in system words. In other words, the identifiers 'FOO', 'foo', and 'Foo' are different, while the system words 'TRUE' and 'true' are the same.

Examples of identifiers are: GETBYTE, P1, BYTES_PER_CELL, but not 1P (does not begin with a letter) or GET BYTE (contains a space). (Because of the use of points to demarcate sections, an erroneous identifier beginning with a point will cause a fatal compilation error.)

A <number> is either an integer consisting of a sequence of decimal digits or an octal constant consisting of the sharp '#' followed by octal digits. Numbers may also be written in binary or hexadecimal, by preceding them with the warning sequences '#B' and '#X', respectively. For consistency, the warning sequence '#O' is equivalent to '#'.

Examples of numbers are

```
255, 0, #377
#B101001, #O477, #X3FACE
```

The reserved words TRUE and FALSE denote -1 and 0 respectively and are used to represent the two truth values.

A <string constant> consists of up to 255 (on the 370) characters enclosed in string quotes ("). The internal character set is EBCDIC (on the 370). The character " may be represented only by the pair *" and the character * can only be represented by the pair **. Other special characters may be represented as follows:

```
*N is newline,
*T is horizontal tab,
*S is space (also represented by itself),
*C is carriage return,
*E is escape,
*B is backspace,
```

*P is newpage.
*Xnn is the character whose hexadecimal
code is nn.

Within a string, a sequence consisting of an asterisk, some number of spaces, newlines, and tabs, and another asterisk is completely ignored. This permits a string constant to extend over a number of lines. For example, we may write

```
"THIS STRING *
*contains newlines*
* AND SPACES"
```

which is exactly equivalent to

```
"THIS STRING CONTAINS NEWLINES AND SPACES"
```

Example strings are "*NEND OF TEST*N" and "THIS IS A QUOTE*"".

The machine representation of a string is the address of the region of store where the length and characters of the string are packed. The fetching and storing of characters in strings may be done using the machine dependent library routines GETBYTE and PUTBYTE, or by the byte operator '%'.

A <character constant> consists of a single character enclosed in apostrophes ('). The apostrophe character ' can be represented in a character constant only by the pair *'. Other escape conventions are the same as for a string constant, except for '"'. A character constant is right justified in a word. Thus 'A' = 193 (on the 370). Examples of character constants are 'A', '0', '*'', '*N' and '"'.

The element '?' may be used in any context in which a variable is permitted. ? yields no particular value, and is thus useful for such purposes as leaving variables uninitialised, or filling a "hole" in a parameter list to a routine.

Expressions

Because an identifier has no type information associated with it, the type of an element (and hence an expression) is assumed to match the type required by its context.

All expressions are listed below. E1, E2 and E3 represent arbitrary expressions except as noted in the descriptions which follow the list, and K0, K1 and K2 represent constant expressions (whose values can be determined at compile time; see the section on Constant Expressions).

```
primary         <element>
                (E1)

fn call         E1()
                E1(E2,E3,...)

addressing      E1 !  E2     subscripting
                @E1          address generaton
                !E1          indirection
                E1 % E2      byte selection

arithmetic      E1 * E2
                E1 / E2
                E1 REM E2    integer remainder
                E1 + E2
                + E1
                E1 - E2
                - E1
                ABS E1

relational      E1 = E2
                E1 ¬= E2     not equal
                E1 < E2
                E1 <= E2
                E1 > E2
                E1 >= E2

shift           E1 << E2     l shift (logical) E2 bits
                E1 >> E2     r shift (logical) E2 bits
                             (E2 ≥ 0)

logical         ¬ E1         not (complement)
                E1 & E2      and
                E1 | E2      inclusive or
                E1 EQV E2    bitwise equivalence
                E1 NEQV E2   bitwise not-equivalence
                             (exclusive or)

conditional     E1 -> E2, E3

table           TABLE K0,K1,K2,...

VALOF           VALOF <command>
```

The relative binding power of the operators is as follows.

```
  (highest, most binding) function call
                          !    %
                          @ !
                          * / REM
                          + -
                          relationals
                          shifts
```

```
                             ¬
                             &
                             |
                             EQV NEQV
                             ->
                             TABLE
        (lowest, least binding) VALOF
```

Operators of equal binding power associate to the left. For example,

```
X + Y - Z
```

is equivalent to

```
(X + Y) - Z
```

In order that the rule allowing the omission of most semicolons should work properly, a dyadic operator may not be the first symbol on a line.

The function call is described in the section on function definitions, and the VALOF expression is described with the RESULTIS command.

Addressing operators

A powerful pair of operators in BCPL are those which allow one to generate and use addresses. An address may be manipulated using integer arithmetic and is indistinguishable from an integer until it is used in a context which requires an address. If the value of a variable X is the address of a word in storage, then X+1 is the address of the next word.

If V is a variable, then associated with V is a single word of memory, which is called a cell. The content of the cell is called the value of V and the address of the cell is called the address of V.

An address may be used by applying the indirection operator (!). The expression

```
!E1
```

has, as value, the content of the cell whose address is the value of the expression E1. (On the 370, only the low order 22 bits of E1 are used.)

An address may be generated by means of the operator @. The expression

@E1

is only valid if E1 is one of the following.
(1) An identifier (not declared by a manifest declaration), in which case @V is the address of V.
(2) A subscripted expression, in which case the value of @E1!E2 is E1+E2.
(3) An indirection expression, in which case the value of @!E1 is E1.

Case (1) is self-explanatory. Case (2) is a consequence of the way vectors are defined in BCPL. A vector of size n is a set of n + 1 contiguous words in memory, numbered 0, 1, 2, ..., n, as shown in Figure 8. The vector is identified by the address of word 0. If V is an identifier associated with a vector, then the value of V is the address of word 0 of the vector.

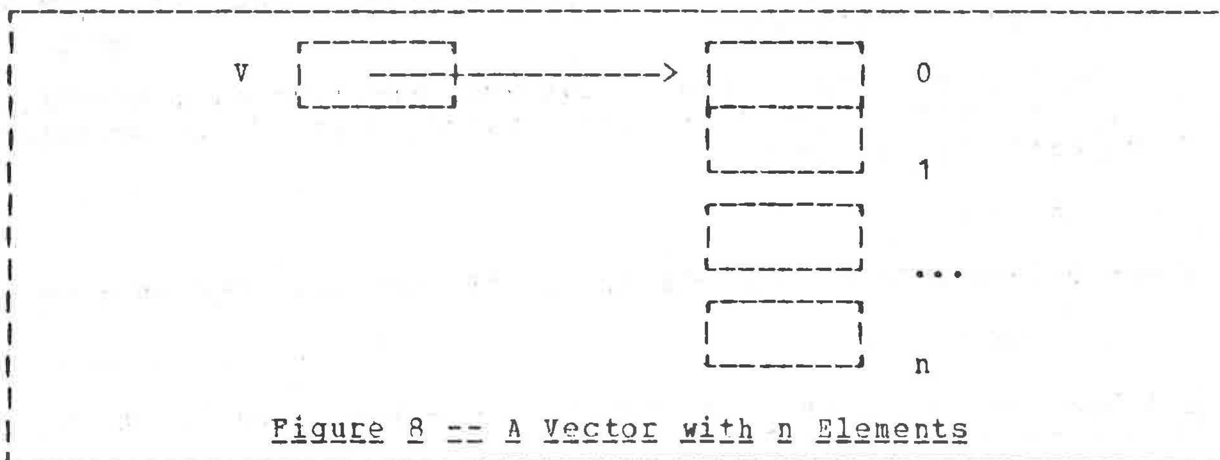

Figure 8 -- A Vector with n Elements

The value of the expression

V!E1

is the content of cell number E1 of vector V, as one would expect. The address of this cell is the value of

V + E1

hence

@(V!E1) = V + E1

This relation is true whether or not the expression

V!E1

happens to be valid, and whether or not V is an identifier.

Case (3) is a consequence of the fact that the operators @ and ! are inverse.

The interpretation of

```
!E1
```

depends on context as follows:

(1) If it appears as the left-hand side of an assignment statement, e.g.,
!E1 := E2 ,
E1 is evaluated to produce the address of a cell and E2 is stored in it.
(2) @(!E1) = E1 as noted above.
(3) In any other context E1 is evaluated and the content of that value, treated as an address, is taken.

Thus, ! forces one more content-taking than is normally demanded by the context.

As a summarising example, consider the four variables A, B, C and D with initial values @C, @D, 5 and 7, respectively. Then, after the assignment

```
A := B
```

their values will be @D, @D, 5, 7. If, instead, the assignment

```
A := !B
```

had been executed, then the values would have been 7, @D, 5, 7. And if the assignment

```
!A := B
```

had been executed, then the values would have been @C, @D, @D, 7. Note that

```
@A := B
```

is not meaningful, since it would call for changing the address associated with A, and that association is permanent.

The operator % has the same priority as !, and allows for the selection or assignment of a byte in a string (or vector). An example of this is

```
$( LET S = "ABCDEF"
   S % 1 := S % 5
   WRITES(S) $)
```

which would write the characters EBCDEF. Thus, the effect is the same as

```
$( LET S = "ABCDEF"
   PUTBYTE(S, 1, GETBYTE(S, 5))
   WRITES(S) $)
```

in which the procedures PUTBYTE and GETBYTE are library procedures discussed in the description of the portable library.

Arithmetic operators

The arithmetic operators *, /, REM, +, - and ABS (see appendix D) act on 32 bit quantities (on the 370) interpreted as signed integers.

The operators * and / denote integer multiplication and division. The operator REM yields the integer remainder after dividing the left hand operand by the right hand one. If both operands are positive the result will be positive, it is otherwise implementation dependent.

The operators + and - may be used in either a monadic or dyadic context and perform the appropriate integer arithmetic operations.

On many machines in which two's complement is used, the relation

```
A = (A / B) * B + A REM B
```

always holds, but if B is negative then A REM B is negative (number theorists may well shudder).

The treatment of arithmetic overflow is undefined.

Relations

A relational operator compares the integer values of its two operands and yields a truth-value (TRUE or FALSE) as result. The operators are as follows

```
= equal,
¬= not equal,
< less than,
<= less than or equal,
> greater than,
>= greater than or equal.
```

The operators = and ¬= make bitwise comparisons of their operands and so may be used to determine the equality of values regardless of the kind of objects they represent. The other tests are for signed arithmetic comparisons.

An extended relational expression such as

```
'A' <= CH <= 'Z'
```

is equivalent to

```
'A' <= CH & CH <= 'Z'
```

An example of an extended relational expression in use is in

```
WHILE '0' <= CH <= '9' DO
   SUM := SUM * 10 + CH - '0'
```

Shift operators

In the expression E1 << E2 (E1 >> E2), E2 must evaluate to yield a non-negative integer. The value is E1, taken as a bit-pattern, logically shifted left (or right) by E2 places. Vacated positions are filled with 0 bits.

Syntactically, the shift operators have lower precedence on the left than relational operators but greater precedence on the right. Thus, for example,

```
A << 10 = 14
```

is equivalent to

```
(A<<10) = 14
```

but

```
14 = A << 10
```

is equivalent to

```
(14=A) << 10
```

Logical operators

The effect of a logical operator depends on context. There are two logical contexts: 'truth-value' and 'bit'. The truth-value context exists whenever the result of the expression will be interpreted immediately as true or false. In this case each subexpression is interpreted, from left to right, in truth-value context until the truth or falsehood of the expression is determined. Then evaluation stops. Thus, in a truth-value context, the evaluation of

```
E1 | E2 & E3
```

is as follows.

The expression E1 is evaluated, and if true the whole expression is true, otherwise E2 is evaluated, and if false the whole expression is false, otherwise E3 is evaluated, and if false the whole expression is true, otherwise the whole expression is false. Lovers of side effects should beware. An example of a truth value context is in

```
WHILE CH = '*S' | CH = '*N' DO
   CH := RDCH()
```

In a 'bit' context, the operator ¬ causes bit-by-bit complementation of its operand. The other operators combine their operands bit-by-bit according to the table below. An example of a bit context is in CH := WORD & 255 .

OPERANDS		OPERATOR			
		&	|	NEQV	EQV
0	0	0	0	0	1
0	1	0	1	1	0
1	0	0	1	1	0
1	1	1	1	0	1

The logical operators & and | may also be represented by /| and |/ respectively (logicians may prefer this). The operators EQV and NEQV may be replaced by == and >< respectively (logicians may wince).

Operations combined with assignment

The assignment operator := may be preceded by any arithmetic, relational, shift, or logical operator.

The meaning of

```
E1 <op>:= E2
```

is the same as that of

```
E1 := E1 <op> E2
```

except that many implementations will often evaluate E1 only once, resulting in increased efficiency. Side effects may or

many not work as intended. #

Conditional operator

The expression

```
E1 -> E2, E3
```

is evaluated by evaluating E1 in truth-value context. If it yields true, then the expression has value E2, otherwise E3. E2 and E3 are never both evaluated.

An example use of a conditional operator is in

```
A := CH = 'N' -> '*N',
     CH = 'P' -> '*P',
     CH = 'T' -> '*T',
     CH = 'S' -> '*S',
     CH
```

Table

The value of the table expression

```
TABLE K0, K1, K2, ...
```

is the address of a static vector of cells initialised to the values of K0, K1, K2, ..., which must be constant expressions.

An example of a table is in

```
WRCH(N ! (TABLE '0', '1', '2', '3',
                '4', '5', '6', '7',
                '8', '9', 'A', 'B',
                'C', 'D', 'E', 'F'))
```

Note a possible ambiguity: if P is a routine, then P(3, TABLE 1, 4, 2) is is a call with two parameters. Other meanings may be obtained by using parentheses.

Constant expression

A constant expression is any expression involving only numbers, character constants, names declared by manifest declaration, TRUE, FALSE, and arithmetic, relational, shift, logical, and conditional operators (for ABS see Appendix E). An example of a constant expression is 'A' - '0'.

Many BCPL compilers will optimise conditional expressions and conditional commands in which the condition is a constant expression. This is quite convenient, as for example, in the expression #
#
#
#

```
CODE := PDP8 -> SIXBIT,
        IBM370 -> EBCDIC,
        U1108 -> FIELDATA,
        ASCII
```

If the manifest constants PDP8, IBM370, and U1108 are defined appropriately, the above command will compile into a simple assignment command. Note that no semantic difference occurs with this optimisation (although some incorrect programs may run--those with undefined identifiers in the parts that are optimised out).

Floating point arithmetic

A floating-point constant may have one of the following forms:

```
i.jEk
i.j
iEk
```

where i and j are unsigned integers and k is a (possibly signed) integer. The value is represented on the 370 as a 32 bit floating-point number. Note that .5 is not a floating-point constant--the point is taken as meaning the end of the program segment, and thus such a "constant" has a disastrous effect upon compilation.

The arithmetic and relational operators for floating-point quantities are as follows:

```
#* #/
#+ #- #ABS
#= # = #<= #>= #< #>
```

They have the same precedence as the corresponding integer operators. There are, also, two monadic operators FIX and FLOAT for conversions between integers and floating-point numbers. They have the same precedence as @.

The extension involving floating point may not be implemented on some minicomputers, due to the short word length. Observe that since BCPL does no type checking, the expression 1.0+2.0 may not yield the result expected when 1.0 #+ 2.0 was intended.

Field selectors

Field selectors allow quantities smaller than a whole word to be accessed with reasonable convenience and efficiency. A selector is applied to a pointer using the operator OF (or ::). It has three components: the size, the shift and the offset. The size is the number of bits in the field, the shift is the number of bits between the right-most bit of the field and the

right hand end of the word containing it, and the offset is the position of the word containing the field relative to the pointer.

The precedence of OF is the same as that of the subscription operator(!), but its left operand (the selector) must be a constant expression. A convenient way to specify a selector is to use the operator SLCT whose syntax is as follows:

```
<constant expression> :=
               SLCT <size>:<shift>:<offset> |
               SLCT <size>:<shift> |
               SLCT <size>
```

where <size>, <shift> and <offset> are constant expressions. Unless explicitly specified the shift and offset are assumed to be zero by default. A size of zero indicates that the field extends to the left hand end of the word.

Selectors are best defined using manifest declarations, e.g., MANIFEST $(FLAGS = SLCT 7:6:2 $).

A selector application may be used on the left hand side of an assignment and in any other context where an expression may be used, except as the operand of @. In the assignment

```
F OF P := E
```

the appropriate number of bits from the right hand end of E are assigned to the specified field. When

```
F OF P
```

is evaluated in any other context, the specified field is extracted and shifted so as to appear at the right hand end of the result.

On the 370, fields corresponding to half-words and bytes are treated efficiently. Field selectors are an exception to the earlier comments on the machine dependence of the extensions. Judicious use of field selectors rather than inline shifting (e.g., FLAGS OF WORD rather than (WORD ! 2) >> 6 & 127, using the declaration of FLAGS given above) can substantially decrease problems in transferring to a machine with a different word size, or in rearranging data structures. It will also increase readability and may improve the chances for the compiler to optimise.

<u>Section brackets</u>

Blocks, compound commands and some other syntactic constructions use the symbols $(and $) which are called opening and closing section brackets.

A section bracket may be tagged with a sequence of letters, digits and underlines (the same characters as are used in identifiers). A section bracket immediately followed by a space or newline is, in effect, tagged with null.

An opening section bracket can be matched only by an identically tagged closing bracket. When the compiler finds a closing section bracket with a non-null tag, if the nearest opening bracket (smallest currently open section) does not match, that section is closed and the process repeats until a matching opening section bracket is found. While the use of tagged command brackets is good programming practice, the use of the multiple bracket closure feature is not recommended.

It is impossible to write sections which are overlapping (not nested).

An example of the use of section brackets is in

```
LET WRITEOCT(N, D) BE
$( IF D > 1 THEN
       WRITEOCT(N >> 3, D - 1)
   WRCH(N & 7 + '0') $)
```

On ASCII terminals, section brackets may be represented by [and]. #

Commands

The complete set of commands is shown here, with E, E1, E2 and K denoting expressions, C, C1 and C2 denoting commands, and D1 and D2 denoting declarations.

```
routine call   E(E1, E2, ...)
               E()
assignment     <left hand side list> := <expr list>
conditional    IF E THEN C
               UNLESS E THEN C
               TEST E THEN C1 OR C2
repetitive     WHILE E DO C
               UNTIL E DO C
               C REPEAT
               C REPEATWHILE E
               C REPEATUNTIL E
               FOR N = E1 TO E2 BY K DO C
               FOR N = E1 TO E2 DO C
resultis       RESULTIS E
switchon       SWITCHON E INTO <compound command>
transfer       GOTO E
               FINISH
               RETURN
               BREAK
               LOOP
               ENDCASE
```

```
compound       $( C1; C2; ...  $)
block          $( D1; D2; ...; C1; C2; ...  $)
```

Discussion of the routine call is deferred to the section where function and routine declarations are described.

Assignment

The command

```
E1 := E2
```

causes the value of E2 to be stored into the cell specified by E1. E1 must have one of the following forms:

(1) the identifier of a variable,
(2) a subscripted expression E3!E4,
(3) an indirection expression !E3.
(4) if E1 is E3 % E4, then the rightmost byte of E2 is stored in the E4-th byte of the vector (or string) E3. :
:

In case (1) the cell belonging to the identifier is updated. Cases (2), (3), and (4) have been described earlier.

A list of assignments may be written thus:

```
E1, E2, ..., En := F1, F2, ..., Fn
```

where Ei and Fi are expressions. This is equivalent to

```
E1 := F1
E2 := F2
...
En := Fn
```

An example of an assignment is B := TRUE, or

```
C, D, E := 3, "HELLO", 'C' .
```

Conditional commands

```
IF E THEN C1
UNLESS E THEN C2
TEST E THEN C1 OR C2
```

Expression E is evaluated in truth-value context. Command C1 is executed if E is true, otherwise the command C2 is executed. An example of a conditional command is

```
IF CH = '-' THEN
$( B := TRUE;
   CH := RDCH() $)
```

__FOR command__

```
FOR N = E1 TO E2 BY K DO C
```

The N is the defining occurrence of an identifier and K must be a constant expression. This command will be described by showing an equivalent block.

```
$( LET N, t = E1, E2
   UNTIL N > t DO
   $( C
      N := N + K $) $)
```

If the value of K is negative the relation N > t is replaced by N < t. The declaration

```
LET N, t = E1, E2
```

declares two new cells with identifiers N and t, t being a new identifier that does not occur in C. Note that the control variable N is not available outside the scope of the command.

The command

```
FOR N = E1 TO E2 DO C
```

is equivalent to

```
FOR N = E1 TO E2 BY 1 DO C
```

An example of a FOR command is

```
FOR J = I+1 TO D DO
   WRCH('*S')
```

__Other repetitive commands__

```
WHILE E DO C
UNTIL E DO C
C REPEAT
C REPEATWHILE E
C REPEATUNTIL E
```

Command C is executed repeatedly until condition E becomes true or false as implied by the command. If the condition precedes the command (WHILE, UNTIL) the test will be made before each execution of C. If it follows the command (REPEATWHILE, REPEATUNTIL), the test will be made after each execution of C, and so C is executed at least once. In the case of

```
C REPEAT
```

there is no condition and termination must be by a transfer of control or RESULTIS command in C. C will usually be a compound command or block.

Within REPEAT, REPEATWHILE and REPEATUNTIL, C is taken as short as possible. Thus, for example

```
IF E THEN C REPEAT
```

is the same as

```
IF E THEN $( C REPEAT $)
```

and

```
E := VALOF C REPEAT
```

is the same as

```
E := VALOF $( C REPEAT $)
```

An example of a repeat command is

```
CH := RDCH()
   REPEATWHILE CH = '*S'
```

RESULTIS command and VALOF expression

The expression

```
VALOF C
```

where C is a command (usually a compound command or block), is called a VALOF expression. It is evaluated by executing the commands (and declarations) in C until a RESULTIS command

```
RESULTIS E
```

is encountered. The expression E is evaluated, its value becomes the value of the VALOF expression and execution of the commands within C ceases.

A VALOF expression must contain one or more RESULTIS commands and one must be executed. In the case of nested VALOF expressions, the RESULTIS command terminates only the innermost VALOF expression containing it.

An example of a VALOF expression is

```
VALOF
$( LET CH = RDCH()
   RESULTIS '0' <= CH <= '9' -> CH - '0',
            'A' <= CH <= 'F' -> CH - 'A' + 10,
            0 $)
```

SWITCHON command

```
SWITCHON E INTO <compound command>
```

where the compound command contains labels of the form

```
CASE <constant expression>:
```

or

```
DEFAULT:
```

The expression E is first evaluated and, if a case exists which has a constant with the same value, then execution is resumed at that label; otherwise, if there is a default label, then execution is continued from there, and if there is not, execution is resumed just after the end of the SWITCHON command. A common error is to forget an ENDCASE, causing control to flow to the next CASE.

The switch may be implemented as a direct switch, a sequential search or a binary search depending on the number and range of case constants and the whim of the particular compiler in use.

An example of a SWITCHON command is

```
SWITCHON CH INTO
$( CASE '*S': CASE '*T': CASE '*N': ENDCASE
   CASE '-':  NEG := TRUE
   CASE '+':  CH := RDCH() $)
```

Transfer of control

```
GOTO E
FINISH
RETURN
```

```
        BREAK
        LOOP
        ENDCASE
```

The command GOTO E interprets the value of E as an address, and transfers control to that address. (However, the design of BCPL is such that the command GOTO E is seldom needed. Moreover, the library routine LONGJUMP is often needed in those few places in which a GOTO command might be worthwhile). The command FINISH causes an implementation dependent termination of the entire program; usually it causes all currently opened files to be closed. RETURN causes control to return to the caller of a routine. BREAK causes execution to be resumed at the point just after the smallest textually enclosing repetitive command. The repetitive commands are those with the following key words:

UNTIL, WHILE, REPEAT, REPEATWHILE, REPEATUNTIL and FOR.

The command LOOP causes execution to be resumed at the point just before the end of the body of a repetitive command. For a FOR command it is the point where the control variable is incremented, and for the other repetitive commands it is where the condition (if any) is tested. ENDCASE causes execution to be resumed at the point just after the smallest textually enclosing SWITCHON command.

An example containing three transfers of control is

```
    $(CYCLE
          CH := RDCH()
          SWITCHON CH INTO
          $( CASE '*S': CASE '*N': CASE '*T': LOOP
             CASE '-': NEG := TRUE
             CASE '+': CH := RDCH()
                       ENDCASE
             CASE 'Z': BREAK $)
    $)CYCLE REPEAT
```

Compound command

A compound command is a sequence of commands enclosed in section brackets.

```
    $( C1; C2; ...  $)
```

The commands C1, C2, ... are executed in sequence. An example of a compound command is

```
    $( B := TRUE
       CH := RDCH()
       WHILE FOO() DO
          BAR() $)
```

Block

A block is a sequence of declarations followed by a sequence of commands enclosed together in section brackets.

```
$( D1; D2; ...; Dn; C1; C2; ...; Cm  $)
```

The declarations D1, D2, ... and the commands C1, C2, ... are executed in sequence. The scope of an identifier (i.e., the region of program where the identifier is known) declared in a declaration is the declaration itself (to allow recursive definition), the subsequent declarations and the commands of the block. Notice that the scope does not include earlier declarations or extend outside the block.

An example of a block is

```
$( LET A = READN() AND B = READN()
   RESULTIS A > B -> A, B $)
```

Observe that

```
LET A(N) BE TEST N > 0 THEN B(N-1) OR WRITES("A")
LET B(N) BE TEST N > 0 THEN A(N-1) OR WRITES("B")
```

is illegal, since the identifier B is unknown in the declaration of A; however, the (ridiculous) intent can be recovered by changing the second LET to AND, thereby making it one declaration (see below, "Simultaneous declarations").

The operator <>

The operator <> has a similar meaning to that of semicolon, but is syntactically more binding than DO, REPEAT, OR, etc. For example,

```
IF E DO C1 <> C2
```

is equivalent to

```
IF E DO $( C1; C2 $)
```

Declarations

Every identifier used in a program must be declared explicitly. There are 10 distinct declarations in BCPL:
global, manifest, static, dynamic, vector, function, routine, formal parameter, label and for-loop control variable.

Of these, for-loop control variables have been described above. The scope of identifiers declared at the head of a block is described in the previous section.

Global

A BCPL program need not be compiled in one piece. The sole means of communication between separately compiled segments of program is the global vector.

Alternatively, programs may communicate via external variables.

The declaration

```
GLOBAL $( Name : constant-expression $)
```

associates the identifier Name with the specified location in the global vector. Thus Name identifies a permanently allocated cell which may be accessed by Name or by any other identifier associated with the same global vector location.

Global declarations may be combined.

```
GLOBAL $( N1:K1; N2:K2; ...; Nn:Kn $)
```

is equivalent to

```
GLOBAL $( N1:K1 $)
GLOBAL $( N2:K2 $)
    ...
GLOBAL $( Nn:Kn $)
```

An example of a global declaration is GLOBAL $(WRCH:14 $) .

Note that in some implementations, particularly those with strange loaders, the globals may not be considered consecutive as in a vector. They are implemented instead as entries and externals to be linked by the loader. Thus, programs which deal with addresses of globals are implementation dependent. Furthermore, some implementations may require extra-lingual specifications of globals (e.g., during linking).

Externals

Globals may be replaced entirely by entries and externals, a concept which is perhaps cleaner and more easily handled by most loaders.

An entry declaration has the form

```
ENTRY $( name = string $)
```

and defines the name of a routine, function or variable which is

declared (or for a variable, is used and is to be stored) in this segment. The string is the name by which it is known to the loader. As a convenience, the form

```
ENTRY $( name $)
```

is permitted and means that the name used within the segment is the same as the name known to the loader.

An external declaration has the form

```
EXTERNAL $( name = string $)
```

and defines a name of a routine, function or variable, in another separately compiled segment (which may not even be written in BCPL). The string is the name by which it is known to the loader.

As a convenience, the form

```
EXTERNAL $( name $)
```

is also permitted with the same meaning as for ENTRY.

As an example, the following segment will define a function to be used by other segments.

```
ENTRY $( INVOLUTION = "INVOL" $)
LET INVOLUTION(F, X) = F(F(X))
```

The following program fragment is a segment which uses the function defined in the last example.

```
EXTERNAL $( INVOLUTION = "INVOL"
            WRITE = "WRNUM001" $)
LET G(T) = T*T + T + 4
AND START() BE
   FOR I = 0 TO 10 DO WRITE(INVOLUTION(G, I))
```

Manifest

An identifier may be associated with a constant by the declaration

```
MANIFEST $( Name = constant-expression $)
```

An identifier declared by a manifest declaration may only be used in contexts where a constant would be allowable. It may not, for instance, appear on the left hand side of an assignment. Like global declarations, manifest declarations may be combined.

```
MANIFEST $( N1=K1; N2=K2; ...; Nn=Kn $)
```

is equivalent to

```
MANIFEST $( N1=K1 $)
MANIFEST $( N2=K2 $)
...
MANIFEST $( Nn=Kn $)
```

An example of a manifest declaration is

```
MANIFEST $( CELL_SIZE = 32 $)
```

Note that <constant-expression> does not include <string>. For example, MANIFEST $(MSG = "HELLO" $) is illegal.

Static

A variable may be declared and given an initial value by the declaration

```
STATIC $( Name = constant-expression $)
```

The variable that is declared is static, that is it has a cell, in its segment, permanently allocated to it throughout the execution of the program (even when control is not dynamically within the scope of the declaration). Like global declarations, static declarations may be combined.

```
STATIC $( N1=K1; N2=K2; ...; Nn=Kn $)
```

is equivalent to

```
STATIC $( N1=K1 $)
STATIC $( N2=K2 $)
...
STATIC $( Nn=Kn $)
```

An example of a static declaration is

```
STATIC $( CHAR_POINTER = 0 $)
```

Dynamic

The declaration

```
LET N1, N2, ..., Nn = E1, E2, ..., En
```

creates dynamic cells and associates with them the identifiers N1, N2, ..., Nn. These cells are initialised to the values of E1, E2, ..., En. The space reserved for these cells is released when the block in which the declaration appears is left.

An example containing a dynamic declaration is

```
$( LET A = READN() AND B = READN()
   RESULTIS A > B -> A, B $)
```

Vector

The declaration

```
LET N = VEC K
```

where K is a constant expression, creates a dynamic vector (but not of dynamic size) by reserving K + 1 cells (K is a constant) of contiguous storage in the stack, plus one other cell in the stack which is associated with the identifier N. Execution of the declaration causes the value of N to become the address of the K + 1 contiguous cells. The storage allocated is released when the block is left. An example of a vector declaration is: LET S = VEC 255 . Note that LET S1, S2 = VEC n1, VEC n2 is not permitted. It can, instead, be written LET S1 = VEC N1 AND S2 = VEC N2.

Function and routine

The declaration

```
LET N(P1, P2, ..., Pm) = E
```

declares a function named N with m parameters. The parentheses are required even if m = 0. A parameter name has the same syntax as an identifier, and its scope is the expression E. A routine declaration is similar to a function declaration except that its body is a command and the equals sign is replaced by BE:

```
LET N(P1, P2, ..., Pm) BE C
```

If the declaration is within the scope of a global declaration for N or static declaration for N, then that cell will be initialised to the entry address of the function (or routine) before execution of the program. Thus the function may be accessed from any segment if global, or from anywhere in its own segment, if static. Otherwise, a static cell is created, is associated with the identifier N, and is initialised to the entry address.

The function or routine is invoked by the call

```
E0(E1, E2, ..., Em)
```

where expression E0 evaluates to the entry address. In particular, within the scope of the identifier N, the function

or routine may be invoked by the call

```
N(E1, E2, ..., Em)
```

provided the value of N has not been changed during the execution of the program.

Each value passed as a parameter is copied into a newly created cell which is then associated with the corresponding parameter name. The cells are consecutive in store and so the argument list behaves like an initialised dynamic vector. The space allocated for the argument list is released when evaluation of the call is complete. Notice that arguments are always passed by value; however, the value passed may, of course, be an address, e.g., UPDATE(@X).

A function call is a call in the context of an expression. If a function is being called, the result is the value of E, and if a routine is being called, the result is undefined. A routine call is a call in the context of a command and may be used to call either a function or a routine. A routine call has no result.

No dynamic (or vector or formal) variable that is declared outside the function may be referred to from within E. Thus, the only non-local variables permitted are those which are declared STATIC or GLOBAL. (This is a frequent source of error for beginners accustomed to more forgiving languages.) Naturally, manifest constants may be used in any context in which they are known.

An example of a routine declaration is

```
LET WRITEOCT(N, D) BE
$( IF D > 1 THEN WRITEOCT(N >> 3, D - 1)
   WRCH(N & 7 + '0') $)
```

and an example of a call of this routine is WRITEOCT(N, 6). Note that most BCPL implementations handle routine and function calls efficiently; thus, you should not hesitate to construct your program from many small routines or functions. (On the IBM 370/168, for example, BCPL routine calls are about twice as fast as those of Fortran!)

Label

A label may be declared by

```
Name:
```

A label declaration may precede any command or label declaration, but may not precede any other form of declaration in the same block. Exactly as in the case of a function or routine, a label declaration creates a static cell if it is not

within the scope of a global declaration of the same identifier. The local or global cell is initialised before execution with the address of the point in the program labelled, so that the command

```
GOTO Name
```

has the expected effect, which, of course, a good programmer should seldom use!

The scope of a label depends on its context. It is the smallest of the following regions of program:

(1) the command sequence of the smallest textually enclosing block,
(2) the body of the smallest textually enclosing VALOF expression or routine,
(3) the body of the smallest enclosing FOR command.

Labels may be assigned to variables and passed as parameters (dreadful thought, except with LONGJUMP!). It is, in general, not useful for them to be declared global, but they can be assigned to global variables (even worse, except for error recovery via LONGJUMP).

Using a GOTO command to transfer to a label which is outside the current function or routine will produce undefined (chaotic) results. Such transfers, if you insist on them, can only be performed using the procedures LEVEL and LONGJUMP which are described in the section on the portable library.

Simultaneous declaration

Any declaration of the form

```
LET ...
```

may be followed by one or more declarations of the form

```
AND ...
```

where any construct which may follow LET may follow AND. As far as scope is concerned, such a collection of declarations is treated like a single declaration. This makes is possible, for example, for two routines to know each other without recourse to the global vector.

Miscellaneous features

GET.

It is possible to include a file in the source text of a program using a GET directive of the form:

GET. "STRING"

This directive is replaced by the text of the file whose name is string. A get directive should appear on a line by itself.

Comments and spaces

The character pair // denotes the beginning of a comment. All characters from (and including) // up to but not including the character 'newline' will be ignored by the compiler. Blank lines are also ignored.

As a convenience, the character sequence '||' is treated as a similar comment delimiter. Also, it is possible to embed a comment within the warning marks "/* ... */", or "|* ... *|". Such a comment may appear anywhere that a space is permitted, and may contain newline characters. #

Space characters may be inserted freely except inside a basic symbol; a space character is required to separate identifiers or system words from adjoining identifiers or system words.

Optional symbols and synonyms

The reserved words DO and THEN are synonyms in BCPL. Most implementations of BCPL also allow other synonyms; a list of the synonyms for the 370 implementation can be found in Appendix A.

In order to make BCPL programs easier to read and to write, the compiler allows the syntax rules to be relaxed in certain cases. The word DO (or THEN) may be omitted whenever it is immediately followed by the keyword of a command (e.g., RESULTIS). Any semicolon occuring as the last symbol of a line may be omitted. This feature requires that no line may begin with a dyadic operator (see section 2.3). As an example, the following two pieces are equivalent.

```
IF A = 0 DO GOTO X;     | IF A = 0 GOTO X
A := A - 1;             | A := A - 1
```

Segments

Programs may be compiled in more than one segment. A segment, in BCPL, ends with a point ("."). This is useful for maintenance of large programs such as the BCPL compiler itself. An essential property of a segment is that identical global declarations within distinct segments refer to the same global cell at run time, whereas identical static declarations within distinct segments each refers to a distinct cell within its own

segment. Thus the only communication between segments is via the global cells, or externals.

A (stupid) example of the use of segments is

```
GLOBAL $( START:1; GREETING:101 $)
LET START() BE GREETING()
.
GLOBAL $( WRITES:65; GREETING:101 $)
LET GREETING() BE WRITES("HELLO")
.
```

A reasonable size for one segment is about 8 pages of BCPL source code.

<u>Segment headings</u>

A segment of BCPL program may start with a directive of the following form:

```
SECTION. "name"
```

where name is a module name acceptable to the loader. It defines the section name given to the object module corresponding to the control segment of program.

Implementations may provide a means for calling routines written in other languages. In order that these may be called, it may be necessary to force them to be loaded. If such a routine is called, the directive

```
NEEDS. "name"
```

may be used to force loading. NEEDS. directives follow the SECTION. directive. As an example, a segment might begin:

```
SECTION. "GRAPHICS"
NEEDS. "SINE"
NEEDS. "PLOT"
LET CIRCLE(R) BE ...
```

An object deck in another file may be required to be loaded together with the main program (e.g. the BCPL library). The directive

```
INCLUDE. "name"
```

may be used. INCLUDE. directives must follow the SECTION. directive, e.g.

```
SECTION. "MAINPROG"
INCLUDE. "CS:BCPLLIB"
LET START (PAR) BE ...
```

*** this may not work ***

It may be desirable to pass parameters to the compiler directly from the program. The PARAMETER. directive may be used to accomplish this, and all such directives must follow the SECTION. directive. On MTS the PARAMETER. directive does absolutely nothing at present.

The Run-time Library

This section summarises the library functions and routines that are available. Some of these routines may be simulated only with great difficulty on operating systems other than MTS. These are marked in the margin.

The input/output facilities of BCPL are quite simple, and are always invoked by means of function or routine calls. These facilities are based on the concept of character streams in which newline and newpage are also characters.

Elementary Input Functions

RDCH() is a function whose result is the next character from the currently selected input stream. If the stream is exhausted, it yields ENDSTREAMCH (=-1).

READN() is a function whose result is the next decimal integer from the current input stream. If the stream is exhausted, READN returns 0, and the global variable TERMINATOR is set to ENDSTREAMCH.

Elementary output routines

WRCH(ch) will write the character ch to the currently selected output stream. The effect of WRCH on special characters is as follows:

*N - write the output buffer and go to a new line,

*P - write the output buffer and go to a new page,

*T - go to the next tab position (no effect in BCPL-V),

*S - a space,

*B - back space (not for printing one character over another),

*C - write output buffer with carriage return, no line feed,

*E - write output buffer, no carriage return, no line feed,

*Xnn - write the character whose hexadecimal representation (EBCDIC or ASCII) is nn, e.g., *X15 is equivalent to *N. :

WRITES(s) writes the string s to the current output stream.

WRITED(n, d) writes the signed integer n to the current output stream right justified in a field of width d places. If d is too small the number is written correctly using as many characters as necessary.

WRITEZ(n, d) is as for WRITED, but generates leading zeroes.

WRITEF(format, a, b, ...) is a routine to output a,b, ... to the current output stream according to format. The format string is copied to the stream until the end is reached or the warning character '%' is encountered. The character following the '%' defines the format of the next value to be printed as follows:

- %% print '%',
- %S print as a string,
- %C print as a character,
- %N print as an integer (minimum width),
- %In print as an integer width n,
- %On print as an octal number of width n with leading zeroes,
- %Xn print as a hexadecimal number of width n with leading zeroes,
- %Zn prints as an integer width n with leading zeroes, where 0 ≤ n ≤ F (one hexadecimal digit).

The routine takes the format and up to 11 arguments.

Other input routines

UNRDCH() backspaces the current input stream by one character, in BCPL-V up to the last newline. Many implementations will not permit consecutive calls of UNRDCH.

READREC(v) is a function that will read the remainder of the MTS line (or the next line) from the current input stream into the vector v packing four characters per word. The result is the number of characters read (if limited by SETWINDOW the result is the negative amount). If a line of length 0 is read, READREC returns 0 and !v is set to 0. If the input stream is exhausted, READREC returns 0 and !v is set to ENDSTREAMCH.

SKIPREC() is a routine which causes the remaining characters in the current record of the currently selected input stream to be ignored.

Other output routines

NEWLINE() is equivalent to WRCH('*N').

NEWPAGE() is equivalent to WRCH('*P').

WRITEN(n) is equivalent to WRITED(n,0).

WRITEOCT(n, d) writes the d least significant octal digits of the unsigned integer n to the current output stream.

WRITEO(n) is equivalent to WRITEOCT(n, 8).

WRITEHEX(n, d) writes the d least significant hexadecimal digits of the unsigned integer n to the current output stream.

WRITEX(n) is equivalent to WRITEHEX(n, 8).

BINWRCH(ch) writes a character as is, i.e., without translating '*N' to newline etc.

WRITEREC(v,n) writes n characters from the vector v to the current output stream followed by a newline. The characters in v are packed four per word.

WRITESEG(v,n) writes n characters from the vector v to the current output stream. The characters in v are packed four per word.

Enquiries

TIMEOFDAY() is a function which yields a string of the form "hh:mm:ss" in BCPL-V.

DATE() is a function which yields a string of the form "yyyy mon dd" in BCPL-V.

ENDSTREAMCH is a manifest constant (=-1) which is produced by RDCH when the input stream is exhausted.

TIME() is a function whose result is the computation time in units of about 13.3 micro-seconds on an IBM 370/168.

INPUT() is a function that will return with the currently selected input stream (see SELECTINPUT).

OUTPUT() is a function that will return with the currently selected output stream (see SELECTOUTPUT).

BATCH() is a function that returns true if the program is running in batch mode in MTS, and false otherwise.

USERID() returns a string of length 4 which is the current MTS user id.

STACKBASE is a global variable which points to the base of the runtime stack.

STACKEND is a global variable which points to the end of the runtime stack.

STACKHWM() is a function which returns a pointer to the highest point on the runtime stack used so far.

LOADPOINT is a global variable which points to the base of the area in which the program is loaded.

ENDPOINT is a global variable which points to the top of the area in which the program is loaded.

PARMS is a global variable holding the address of a string which, in BCPL-V, was in the PAR= field from the MTS $RUN command.

TERMINATOR is a global variable that holds the character following the last digit of the most recent number read in by READN.

Input file manipulation

FINDINPUT(string) is a function taking a string which, in BCPL- V, is the name of an MTS file or device (e.g., "-A" or "**SOURCE**") and returning a stream-pointer (a machine address) to be used by SELECTINPUT. If the file or device does not exist, the result is zero.

FINDINPUTUNIT(string) is a function, in BCPL-V, taking a string which is the name of an MTS logical unit (e.g., "SCARDS" or "0") and returning a stream pointer to be used by the input routines.

SELECTINPUT(stream) is a routine which selects the specified input stream for future reading.

INCCNTROL(sw) causes the carriage control to be ignored on input if sw is false (in MTS the default is true).

REWIND() repositions the currently selected input stream to point to the first record, if possible.

ENDREAD() closes the currently selected input stream.

SETTRIM(sw) sets the control that specifies the treatment of trailing blanks in records read from the currently selected input stream. If sw is true trailing blanks will be skipped, if sw is false they will not (in MTS the default is

the MTS trim setting at the time that the program is run). :

SETWINDOW(w) limits the reading on the currently selected input stream with READREC to w characters or to the newline, whichever is shorter.

Output file manipulation

FINDOUTPUT(string) is a function taking a string (e.g., "-I" or "**TAPE**") as the name of an MTS file or device and returning a stream-pointer to be used by the routine SELECTOUTPUT. If the file or device does not exist, the result is zero. :

FINDOUTPUTUNIT(string) is a function, in BCPL-V, taking a string which is the name of an MTS logical unit (e.g., "SPUNCH" or "6") and returning a stream pointer to be used by the routine SELECTOUTPUT. :

SELECTOUTPUT(stream) is a routine which selects the specified output stream for future writing.

OUTCONTROL(sw) causes the carriage control to be suppressed on the currently selected output stream if sw is false (in MTS the default is true).

ENDWRITE() closes the currently selected output stream.

String manipulation

GETBYTE(s,i) is a function which returns the i-th byte of the string s. In BCPL-V it is equivalent to s % i. :

PUTBYTE(s,i,c) is a routine which inserts the character c into the i-th byte of the string s. In BCPL-V it is equivalent to s % i := c. :

PACKSTRING(v,s) is a function which packs the characters v!1 to v!n into s, where n = v!0 & 255. The result is the subscript of the highest element of s used (i.e., n/4 on the 370).

UNPACKSTRING(s,v) is a routine to unpack characters from the string s into v!1 to v!n when n is the length of the string, and set v!0 = n.

Programming aids

MAPSTORE() prints a map of the program area including function and routine names, and the values of all global variables used.

BACKTRACE(cd,addr) prints an error message using cd and addr followed by a summary of the dynamic stack giving the names of all functions and routines currently active and the values of the first few local variables of each. If cd = 0 the error message will be disabled.

FLUSH() empties all the input/output buffers (generally not useful).

ABORT(cd,addr,oldstack,data) is called automatically by the system after most faults. Here cd is an abort code, which is displayed to help identify the cause of the error; addr, oldstack and data are parameters used in internal calls to ABORT; they need not be supplied in user calls. In BCPL-V, ABORT operates in the following manner:

1) call USERABORT - initially a dummy routine,
2) dump a message,
3) in batch mode, call BACKTRACE and MAPSTORE, and then terminate,
4) in interactive mode, return to the operating system; if restarted, accept commands, which are single characters, with the following meanings:

B call BACKTRACE
C attempt to continue as though the error had never occurred
M call MAPSTORE
Q quit
R force an unconditional return from the function or routine in which the error occurred
S call START, to try to rerun the program.

Any other character is ignored. ABORT terminates by executing STOP(100).

Note that, by assigning the address of a procedure to USERABORT, the user may program any desired termination activity. USERABORT need not return; however, if it doesn't, then it should call CLEAR in order to reset the interrupt system, before calling LONGJUMP.

An example of USERABORT is

```
LET START() BE
$( STATIC $( REC.P:0; REC.L:0 $)
   LET MYABORT(CODE, ADDR) BE
   $( SELECTOUTPUT(SYSPRINT)
      WRITEF("PROGRAM FAULT, CODE %X4*N",
             CODE)
      CLEAR(); LONGJUMP(REC.P, REC.L) $)
   REC.P := LEVEL()
   USERABORT := MYABORT
LAB: REC.L := LAB
```

```
                    COMPUTE() $)
```

CLEAR() resets the program interrupt system, so that further interrupts are allowed.

STOP(n) will terminate the run, returning a completion code n.

FORTRAN interface in BCPL-V

SYMBOL(str) yields the address of the system routine with name str, if it has currently been loaded. If the routine cannot be found, SYMBOL yields 0.

CALL(f, a, b, c, ...) is a function taking, for f, a value yielded by SYMBOL, and, for a, b, c, etc., the parameters to be passed to the FORTRAN callable routine as in, e.g.,

```
SECTION. "TEST"
NEEDS. "MUNG"
GET "CS:BCPLLIB"
LET START() BE
$( LET MUNG = SYMBOL(MUNG)
   AND PAR1, PAR2 = 666, 4004
   LET I = CALL(MUNG, 4@PAR1, 4@PAR2)
   WRITEF("MUNG yielded %N*N", I) $)
```

(note the conversion to 370 addresses). The return code (contents of register 15) is stored in the global variable RETURNCODE.

If something goes awry inside the system routine, the symptom will generally be a program interrupt inside CALL.

FLOATCALL(f, a, b, c ...) is a function similar to CALL but should be used instead of CALL when the result yielded is in floating-point.

BCALL(f, a, b) is a routine similar to CALL, which takes exactly 2 additional parameters to f, and performs the assembler R-type calling sequence, loading the values of the 2 parameters into general registers 0 and 1.

Note that, for efficiency's sake, CALL, FLOATCALL, and BCALL do not check that f points to a valid routine. Thus a program fault will occur if f is invalid.

Miscellaneous

LEVEL() is a function whose result is the current value of the run-time stack pointer for use with LONGJUMP. The stack pointer changes only when a function or routine is entered

or left.

LONGJUMP(p,l) will cause a non-local jump to the label l at the activation level given by the stack pointer p. Most GOTO operations use this procedure.

APTOVEC(f,n) is a function which will apply f to two arguments v and n where v is a vector of size n. APTOVEC could (illegally) be defined in BCPL as follows:

```
LET APTOVEC(F,N) = VALOF
$( LET V = VEC N
   RESULTIS F(V,N) $)
```

It may be used for dynamic storage allocation.

GETSPACE(n) is a function in BCPL-V which requests n cells from the operating system and returns the address of the first cell.

FREESPACE(a) is a routine which returns to the operating system the cells pointed to by the address in cell a (which must have been yielded by GETSPACE previously).

COMMAND(s) passes the string s to the host operating system, which will attempt to execute it as a command.

SYSTEM() is a routine which returns control to the operating system without unloading the program.

ERROR() displays the message "ERROR RETURN", and then returns control to the operating system without unloading the program.

ERRORMESSAGE(string) prints the contents of string and then calls ERROR.

CATCHATTENTION(sw) is a routine which, if called with sw set to TRUE, will trap attention interrupts. The global flag ATTENTIONPENDING is initially set by CATCHATTENTION to FALSE. The first attention interrupt occurring after calling CATCHATTENTION merely sets ATTENTIONPENDING to TRUE, and the program continues to execute. If ATTENTIONPENDING is not reset to FALSE, the next attention interrupt will cause the program to stop executing. At this point it can be restarted with the MTS $RESTART command. If CATCHATTENTION is called with sw set to FALSE, the attention interrupt trap is disabled.

Part III

Using BCPL with MTS

Compilation and execution

The following represents a simple compilation and execution of a BCPL program under the MTS operating system. See later sections for compilation and execution options.

```
$RUN CS:BCPL SPUNCH=object
GET. "CS:BCPLHDR"
LET START() BE
$( ...
<source program>
...          $)
$ENDFILE
$RUN object+CS:BCPLLIB
...
<data>
...
$ENDFILE
```

Library declarations

The directive of BCPL-V

```
GET. "CS:BCPLHDR"
```

will insert the standard library declarations from the MTS file CS:BCPLHDR. The global numbers and loader names of some items declared in this file are shown below. By convention library variables are given global numbers in the range 1 to 199, and loader names beginning with a '#'. Users should avoid allocating globals in this region, or using loader names beginning with a '#', for their own purposes.

NAME	GLOBAL NUMBER	LOADER NAME
ABORT	20	#ABORT
APTOVEC	17	#APTOVEC
ATTENTIONPENDING	28	#ATTENTI

BACKTRACE	22	#BACKTRA
BATCH	90	#BATCH
BCPLSTART.	99	#BCPLSTA
BINWRCH	61	#BINWRCH
CALL	86	#CALL
CATCHATTENTION	27	#CATCHAT
CLEAR	26	#CLEAR
COMMAND	9	#COMMAND
DATE	94	#DATE
ENDPOINT	11	#ENDPOIN
ENDREAD	34	#ENDREAD
ENDTOINPUT	46	#ENDTOIN
ENDWRITE	44	#ENDWRIT
ERROR	7	#ERROR
ERRORMESSAGE	25	#ERRORME
EXIT.	6	#EXIT.
FINDINPUT	30	#FINDINP
FINDINPUTUNIT	31	#INPUTUN
FINDLOG	95	#FINDLOG
FINDOUTPUT	40	#FINDOUT
FINDOUTPUTUNIT	41	#OUTPUTU
FLOATCALL	87	#FLOATCA
FLUSH	5	#FLUSH
FREESPACE	19	#FREESPA
GETBYTE	80	#GETBYTE
GETSPACE	18	#GETSPAC
INCONTROL	35	#INCONTR
INPUT	33	#INPUT
LEVEL	15	#LEVEL
LOADPOINT	10	#LOADPOI
LONGJUMP	16	#LONGJUM
MAPSTORE	23	#MAPSTOR
NEWLINE	67	#NEWLINE
NEWPAGE	68	#NEWPAGE
OUTCONTROL	45	#OUTCONT
OUTPUT	43	#OUTPUT
PACKSTRING	82	#PACKSTR
PARMS	2	#PARMS
PUTBYTE	81	#PUTBYTE
RCALL	88	#RCALL
RDCH	50	#RDCH
READN	54	#READN
READREC	52	#READREC
READS	56	#READS
RETURNCODE	89	#RETURNC
REWIND	38	#REWIND
SAVEAREA	3	#SAVEARE
SELECTINPUT	32	#SELECTI
SELECTOUTPUT	42	#SELECTO
SETTRIM	36	#SETTRIM
SETWINDOW	37	#SETWIND
SKIPREC	53	#SKIPREC
STACKBASE	12	#STACKBA
STACKEND	13	#STACKEN

STACKHWM	14	#STACKHW
START	1	#START
STOP	4	#STOP
SYMBOL	85	#SYMBOL
SYSTEM	8	#SYSTEM
TERMINATOR	55	#TERMINA
TIME	92	#TIME
TIMEOFDAY	93	#TIMEOFD
UNPACKSTRING	83	#UNPACKS
UNRDCH	51	#UNRDCH
USERABORT	21	#USERABO
USERID	91	#USERID
WRCH	60	#WRCH
WRITED	71	#WRITED
WRITEF	66	#WRITEF
WRITEHEX	75	#WRITEHE
WRITEN	72	#WRITEN
WRITEO	74	#WRITEO
WRITEOCT	73	#WRITEOC
WRITEREC	62	#WRITERE
WRITES	65	#WRITES
WRITESEG	63	#WRITESE
WRITETOLOG	96	#WRITETO
WRITEX	76	#WRITEX
WRITEZ	70	#WRITEZ

Diagnostics

The BCPL compiler has three passes: parse, translate and code-generate. There are correspondingly three kinds of error diagnostic.

A parse diagnostic occurs when a relatively simple syntactic error is detected during the first pass of compilation and an error message is interleaved with the listing. The message includes a portion of the source program to give the context and a brief description of the probable error. The compiler usually skips to the end of the line before continuing the parse. Later error messages should be viewed with suspicion since the automatic recovery is often not very successful. Note that the command number is not the same as the line number.

Translation phase diagnostics occur in the second pass of compilation and report errors such as the use of an undeclared identifier. Each error is briefly described and some descriptive information may be printed.

Code-generation diagnostics are rare and usually result from table overflows or compiler errors.

See Appendix D for a list of compilation diagnostics and appropriate actions to take.

Compilation options

The compilation of a program under MTS can be controlled by various options passed by the MTS $RUN command

```
$RUN CS:BCPL SCARDS=sourceprogram SPRINT=listing -
     SPUNCH=object 0=ocode 1=assemblerlisting -
     PAR='CORE=c'ph1op/ph2op
```

where sourceprogram defaults to *SOURCE* and listing defaults to *SINK*. If 0=ocode is present, then the intermediate OCODE is sent to the file given. If 1=assemblerlisting is present, then the assembler listing of the object program is sent to the file given.

The option 'CORE=c' is an MTS interface directive, where c is n (bytes), or nP (pages) or nK (1028 bytes). It is almost never needed. The remaining options are directives to phase one and phase two of the compiler. The phase two options, if given, are separated from the phase one options by a solidus. Most options are specified by single letters and some are primarily debugging aids for the implementer.

The phase one (parse and translate) options (ph1op) are as follows:

Ln Set the size of work-space area used during compilation. The best value of n is usually between 6000 and 12000.
N Disable the GET directive.
S Print the source program.
Q (Quiet) Do not print the source program
T Print the parse tree of the source program.
O Punch the OCODE.
A Input is in ASCII.
Rn Terminate compilation if more than n error messages are generated (n defaults to 30).
I Suppress 370 code generation and punch MCODE with stack linkage of two cells instead of three.
U Do not translate lower case characters to upper case in the listing.
Z Suppress 370 code generation.

The phase two (code generator) options (ph2op) are as follows:

C Suppress generation of stack overflow checking code. If this option is given, programs will run slightly faster, but may end in mysterious ways.
K Compile instructions with each function and routine to count the number of times they are executed. The counts are printed by MAPSTORE.
P Compile instructions after labels and conditional jumps to accumulate execution counts. These counts are printed by MAPSTORE and allow one to make a detailed analysis of the execution of the program.
D List the object code.
L Output an assembly listing of the compiled program.

N Do not generate an object module for the program.
The default setting of the PAR= field is 'CORE=10P'L6000/K.

Execution options

The object module is invoked by the MTS command

$RUN object+CS:BCPLLIB linkage PAR='CORE=c'parm

where linkage is as usual, e.g., SCARDS=data, where 'CORE=c' is an optional MTS interface directive option as described in section 3.1.3 and parm is passed, as a string, to the routine START, as described below. The default setting for c is 10P and that for parm is the empty string.

Loading

In order to execute a BCPL program on MTS, the SYMTAB system parameter must be ON. If it is OFF, the message LOADER TABLES UNAVAILABLE is displayed.

When the complete program is loaded and executed, the machine code library initialises the run-time system and obtains space for the global vector and stack. The globals are initialised to their appropriate values and then control is passed to the BCPL program by calling the routine START (global 1) which must have been defined by the programmer. START is passed a string from the PAR= field of the MTS $RUN command that caused the program to be executed. If 'CORE=xx' is given, then that part will not be passed to START.

The size of the global vector is the smallest multiple of 100 words large enough to accommodate the highest global number actually used in any segment of the loaded program. The size of the run-time stack depends on the space available in the region in which the program is run. Some space is retained for input/output buffers and system use. The limits of the stack are held in STACKBASE and STACKEND.

When START is called, under MTS, the initial output selection is to SPRINT, and the initial input selection is from SCARDS.

Execution faults

In the event of an execution fault such as division by zero or a protection exception the routine ABORT is called when in MTS batch mode. This will print the fault number and the program address when the fault was detected, followed by a summary of the runtime stack (printed out by BACKTRACE) and a map of the program store and globals (printed out by MAPSTORE). This information is output to SERCOM.

When running from an interactive terminal under MTS, an execution error will halt execution with an error message. Upon issuing a $RESTART, the user is prompted for a letter. Giving B will display a backtrace, M will give a MAPSTORE, Q will terminate execution (Quit) and R will restart again. If there is a program interrupt in ABORT (or, more likely, USERABORT), the system displays the message RECURSIVE PROGRAM INTERRUPT, and halts.

The Profile Option

The profile option is a facility that helps the BCPL user to discover how often any statement in his program was obeyed when the program was run. The facility is invoked using the code generator option 'P'. It causes instructions to be compiled that will maintain execution counts at certain places in the compiled code and the locations and values of these counts can be printed at the end of a run using MAPSTORE. These counts can be related to the original source program with little difficulty and the rules for doing this are given below. For a typical program, the cost of the facility amounts to a 20% increase in program size and a similar increase in execution time for those sections of program which have been compiled with the option specified. Study of the profile output invariably leads to a greater understanding of the program and often indicates ways in which the program can be improved. If a private post mortem is set up to call MAPSTORE, the profile option may be useful as a debugging aid.

The profile option 'P' is best used with the 'K' option so that the output from MAPSTORE includes the print names and execution counts of functions and routines. The precise rule for where count instructions are inserted is as follows:

> A count instruction is inserted just before the first instruction following a label of a conditional jump.

A label in this context is either a programmer's label as in NEXT: A := B or a compiler inserted label such as the one appearing in the compiled code for IF X>0 THEN X:=0. By this rule, multiple labels at the same point give rise to only one count, and so in L:M: A := 3 it is not possible to determine directly the relative frequency of jumps to L and M, but the count of how often the assignment was obeyed is given. The detailed specification of where labels and conditional jumps are compiled is given below.

Each BCPL construction which compiles into code containing a label or conditional jump is given with an outline of its translation. The following notation is used:

E, E1, E2, E3	denote expressions,
N	denotes a name,
C, C1, C2	denote commands,
D	denotes a definition,
[Ei]	denotes an expression in a Boolean context,

```
L1, L2                  denote labels,
J1, J2                  denote unconditional jumps,
CJ1, CJ2                denote conditional jumps,
{L1}                    denotes an optional occurrence of L1,
                        compiled before the next  compiled
                        instruction.
```

Expressions in normal contexts:

```
VALOF C                       C L1 *
E1 -> E2, E3                  [E1] * E2 J2 L1 * E3 L2 *
```

Expressions in Boolean contexts:

```
VALOF C                       C L1 *
E1 -> E2, E3                  [E1] * [E2] * J2 L1 * [ E3] L2 *
E1 /| E2           )
E1 |/ E2           )        [E1] * [E2] {L1} *
(remember that E1<E2<E3 is equivalent to E1<E2/|E2<E3)
```

Commands:

```
IF E THEN C         )
UNLESS E THEN C     )         [E] * C L1 *
WHILE E DO C        )
UNTIL E DO C        )         J1 L2 * C L1 * [E] *
C REPEAT                      L1 C * J1
TEST E THEN C1 OR C2          [E] * C1 J2 L1 * C2 L2 *
FOR N = E1 TO E2 {BY E3} DO C
                              E1 E2 J2 L1 * C N:=N+1 L2 * CJ1 *
SWITCHON E INTO C             J1 C J2 L1 * E Ji L2 *
N:C                 )
CASE E: C           )         L1 * C
DEFAULT: C          )
LET D C                       if  D  contains a function or
                               routine definition then D * C *
                               otherwise D C
```

In practice these rules are easy to use since the labels and conditional jumps occur exactly where one would expect them and since, also, the profile counts contain sufficient redundancy for one to be confident that one is attaching the counts to the right source statements.

As an example, the following portion of profile output:

```
MAP AND COUNTS FROM 1310862(500238) TO 1315286
1310862 SECTION HANOI COMPILED ON 1976 DEC 10; LENGTH 118 WORDS
1310875/HANOI  1310876:   31  1310881:    16  1310884:    15
1310909:     0  1310915/START  1310916:     1  1310919:     2
1310937:     1  1310943:    1  1310953:     0
```

gives the counts for the following program when given the input 4 0.

```
SECTION. "HANOI"
GET. "CS:BCPLHDR"
LET HANOI(N, S, I, D) BE /* (31) */
 $( IF N<=0 THEN /* (16) */ RETURN
    /* (15) */
    HANOI(N-1, S, D, I)
    WRITEF("MOVE %N FROM %C TO %C*N", N, S, D)
    HANOI(N-1, I, S, D) $)
/* (0) */
LET START() BE /* (1) */
 $( LET N = 0
    WRITES("ENTER NUMBER*N")
    N := READN()
    WRITEF("NUMBER INPUT WAS %N*N", N)
    IF N<=0 THEN /* (2) */ $( MAPSTORE() ; FINISH $)
    /* (1) */
    HANOI(N, 'S', 'I', 'D')
    $) /* (1) */ REPEAT
/* (0) */
```

The counts have been inserted into the above program listing as comments in the appropriate places.

A complete job

The following is an example of a complete BCPL job.

```
$RUN CS:BCPL SPUNCH=-U
// THIS IS A DEMONSTRATION BCPL PROGRAM
GET. "CS:BCPLHDR"
// THIS INSERTS THE STANDARD GLOBAL DECLARATION

// START (GLOBAL 1) IS THE MAIN ROUTINE
LET START(PARM) BE $(1
GLOBAL $( TREE:100; TREEP:101; CH:102  $)
STATIC $( COUNT=0; MIN=0; MAX=0  $)
MANIFEST $(  // THE FOLLOWING NAMES WILL
             // BE USED AS SUBSCRIPT SELECTORS
             VAL=0; LEFT=1; RIGHT=2  $)
// THE FUNCTIONS PUT, LIST AND SUM(DEFINED BELOW)
// OPERATE ON A TREE STRUCTURE WHOSE ROOT IS HELD
// IN TREE.  IF T IS A BRANCH IN THIS TREE THEN
// EITHER T=0
// OR  T POINTS TO A TREE NODE AND VAL!T IS AN
// INTEGER(K SAY), LEFT!T IS A BRANCH CONTAINING
// NUMBERS <K AND RIGHT!T IS A BRANCH CONTAINING
// NUMBERS >=K.

// THE ROUTINE PUT WILL ADD A NODE TO THE
// TREE WHOSE ROOT IS POINTED TO BY P.
LET PUT(K, P) BE $(P
   UNTIL !P=0 DO $(
      LET T = !P
      P := K<VAL!T -> @LEFT!T, @RIGHT!T
```

```
      *)
      VAL!TREEP, LEFT!TREEP, RIGHT!TREEP := K, 0, 0
      !P := TREEP
      TREEP := TREEP + 3   $) P
   $) P

   // LIST THE NUMBERS HELD IN THE TREE T
   AND LIST(T) BE
      UNLESS T=0 DO $(
         LIST(LEFT!T)
         IF COUNT REM 10 = 0 DO NEWLINE()
         COUNT := COUNT + 1
         WRITEF(" %I6", VAL!T)
         LIST(RIGHT!T)
     $)

   AND SUM(T) = T=0 -> 0,
                VAL!T<MIN -> SUM(RIGHT!T),
                VAL!T>MAX -> SUM(LEFT!T),
                VAL!T+SUM(LEFT!T)+SUM(RIGHT!T)

   LET V = VEC 600
   TREE, TREEP := 0, V

   // THIS IS A CONVENIENT WAY TO
   // ORGANISE A TEST PROGRAM
   NXT: CH := RDCH()

   SW: SWITCHON CH INTO $(S
        // QUIT.
        CASE 'Q': CASE ENDSTREAMCH:
                  WRITES("*NEND OF TEST*N")
                  FINISH

        // PUT A NUMBER IN THE TREE
        CASE 'P':  PUT(READN(), @TREE)
                   CH := TERMINATOR
                   GOTO SW

        // LIST THE NUMBERS IN THE TREE
        CASE 'L': NEWLINE()
                  COUNT := 0
                  LIST(TREE)
                  NEWLINE()
                  GOTO NXT

        // COMPUTE THE SUM OF A RANGE OF NUMBERS
        // IN THE TREE
        CASE 'S': MIN := READN()
                  MAX := READN()
                  WRITEF("*NSUM OF NUMBERS BETWEEN ")
                  WRITEF("%N AND %N IS %N*N",
                         MIN, MAX, SUM(TREE))
                  CH := TERMINATOR
```

```
                  GOTO NXT

         // PRINT A STORE MAP
         CASE 'M':  MAPSTORE(); GOTO NXT

         // ZERO THE TREE
         CASE 'Z': TREE := 0; WRITES("*NTREE CLEARED*N")
                   GOTO NXT

         // IGNORE LAYOUT CHARACTERS
         CASE '*S': CASE '*N': GOTO NXT

         // FLAG INVALID CHARACTERS
         DEFAULT: WRITEF("*NBAD CH '%C'*N", CH); GOTO NXT
      $)S
     $)1   // END OF PROGRAM
     $RUN -U+CS:BCPLLIB
     P24 P13 P96 P46 P-12 P0 P45
     L S10 50
     Q
     $ENDFILE
```

When the object code is executed it will output the following:

```
         -12       0      13      24      45      46      96
     SUM OF NUMBERS BETWEEN 10 AND 50 IS 128
     END OF TEST
```

Part IV

Reference Material

Appendix A: Basic symbols and examples

The following list of words and symbols are treated as atoms by the syntax analyser. The name of the symbol or its standard representation on the 370 is given in the first column, and examples or synonyms are given in the second.

```
Basic symbol              Examples and synonyms

identifier                A  H1  PQRST  TAX_RATE  K.TRUE
number                    126  7249  #3771
string constant           "A"  "TEST*N"
character constant        'X'  ')'  '*N'  '"'
TRUE
FALSE
(                         [
)                         ]
@                         LV
!                         RV
%                         (BCPL-V only)
*
/
REM
+
-
ABS                       (see Appendix E)
=                         EQ
¬=                        NE
<=                        LE
>=                        GE
<                         LS
>                         GR
<<                        LSHIFT
>>                        RSHIFT
¬                         NOT
&                         LOGAND   (/| in BCPL-V only)
|                         LOGOR    (|/ in BCPL-V only)
EQV                       (== in BCPL-V only)
NEQV                      (>< in BCPL-V only)
->
,
TABLE
VALOF
;
:
$(                        $(AB  $(1   ( { BCPL-V only)
$)                        $)AB  $)1   ( } BCPL-V only)
VEC
BE
LET
AND
:=
BREAK
LOOP
ENDCASE
RETURN
FINISH
GOTO
RESULTIS
SWITCHON
```

INTO	
REPEAT	
REPEATUNTIL	
REPEATWHILE	
DO	THEN
UNTIL	
WHILE	
FOR	
TO	
BY	
TEST	
THEN	DO
OR	ELSE
IF	
UNLESS	
CASE	
DEFAULT	
SUBTITLE.	
TITLE.	
SECTION.	SECTION
GET.	GET
LIST.	
NOLIST.	
NEEDS.	NEEDS

Appendix B: BNF of BCPL

This appendix presents the Backus Naur Form of the syntax of BCPL. The whole syntax is given, with the following exceptions:

1. Comments are not included, and the space character is not represented even where required.

2. The section bracket tagging rule is not included, since it is impossible to represent in BNF.

3. The graphic escape sequences allowable in string and character constants are not represented.

4. No account is made of the rules which allow dropping of semicolon and DO in most cases. It seemed that these rules unnecessarily complicate the BNF syntax yet are easy to understand by other means.

5. BCPL has several synonymous system words and operators: for example, DO and THEN. Only a standard form of these symbols is shown in the syntax; a list of synonyms can be found in Appendix A.

6. Certain constructions can be used only in specific

contexts. Not all these restrictions are included: for example, CASE and DEFAULT can only be used in switches, and RESULTIS only in VALOF expressions. Finally, there is the necessity of declaring all identifiers that are used in a program.

7. There is a syntactic ambiguity relating to <repeated command> which is resolved in later.

The brackets { } implv arbitrary repetition of the categories enclosed.

1. Identifiers, strings, numbers

```
<letter> ::= A | B | ...  | Z
<octal digit> ::= 0 | 1 | ...  | 7
<digit> ::= 0 | 1 | ...  | 9
<string constant> ::= "<255 or fewer characters>"
<character constant> ::= '<one character>'
<octal number> ::= # <octal digit> { <octal digit} >
<number> ::= <octal number> | <digit> { <digit} >
<identifier> ::= <letter> { <letter> | <digit> | _ | .  }
```

1. Operators

```
<address op> ::= @ | !
<mult op> ::= * | / | REM
<add op> ::= + | -
<rel op> ::= = |  = | <= | >= | < | >
<shift op> ::= << | >>
<and op> ::= &
<or op> ::= |
<eqv op> ::= EQV | NEQV
```

3. Expressions

```
<element> ::= <character constant> | <string constant> |
              <number> | <identifier> |
              TRUE | FALSE
<primary E> ::= <primary E> (<expression list>) |
<primary E> ( ) |
(<expression>) | <element>
<vector E> ::= <vector E> !  <primary E> | <primary E>
<address E> ::= <address op> <address E> | <vector E>
<mult E> ::= <mult E> <mult op> <address E> |
             <address E>
<add E> ::= <add E> <add op> <mult E> |
            <add op> <mult E> | <mult E>
<rel E> ::= <add E> { <rel op> <add E> }
<shift E> ::= <shift E> <shift op> <add E> | <rel E>
<not E> ::=   <shift E> | <shift E>
<and E> ::= <not E> { <and op> <not E> }
<or E> ::= <and E> { <or op> <and E> }
```

```
<eqv E> ::= <or E> { <eqv op> <or E> }
<conditional E> ::=
     <eqv E> -> <conditional E> , <conditional E> |
     <eqv E>
<expression> ::= <conditional E> |
                 TABLE <const E> { <const E> } |
                 VALOF <command>
```

4. Constant expressions

```
<C element> ::= <character constant> | <number> |
                <identifier> | TRUE | FALSE |
                (<const E>)
<C mult E> ::= <C mult E> <mult op> <C element> | <C element>
<const E> ::=
      <constant expression> <add op> <C mult E> |
      <add op> <C mult E> | <C mult E>
```

5. Lists of expressions and identifiers

```
<expression list> ::= <expression> <REP, <expression> >
<name list> ::= <identifier> <REP, <identifier> >
```

6. Declarations

```
<manifest item> ::= <identifier> = <constant expression>
<manifest list> ::= <manifest item> { ; <manifest item} >
<manifest declaration> ::= MANIFEST $( <manifest list> $)
<static declaration> ::= STATIC $( <manifest list> $)
<global item> ::= <identifier> : <constant expression>
<global list> ::= <global item> { ; <global item} >
<global declaration> ::= GLOBAL $( <global list> $)
<simple definition> ::= <name list> = <expression list>
<vector definition> ::= <identifier> = VEC <constant expression>
<function definition> ::=
      <identifier> (<name list>) = <expression> |
      <identifier> ( ) = <expression>
<routine definition> ::=
      <identifier> (<name list>) BE <command> |
      <identifier> ( ) BE <command>
<definition> ::= <simple definition> | <vector definition> |
                 <function definition> | <routine definition>
<simultaneous declaration> ::=
      LET <definition> { AND <definition> }|
      <manifest declaration> | <static declaration |
      <global declaration>
```

7. Left hand side expressions

```
<LHSE> ::= <identifier> | <vector E> !  <primary E> |
           !  <primary E>
<left hand side list> ::= <LHSE> { <LHSE> }
```

8. Unlabelled commands

```
<assignment> ::= <left hand side list> := <expression list>
<simple command> ::= BREAK | LOOP | ENDCASE | RETURN | FINISH
<goto command> ::= GOTO <expression>
<routine command> ::= <primary E> (<expression list>) |
                      <primary E> ( )
<resultis command> ::= RESULTIS <expression>
<switchon command> ::=
       SWITCHON <expression> INTO <compound command>
<repeatable command> ::= <assignment> | <simple command> |
   <goto command> | <routine command> |
   <resultis command> | <repeated command> |
   <switchon command> | <compound command> |
   <block>
<repeated command> ::= <repeatable command> REPEAT |
   <repeatable command> REPEATUNTIL <expression> |
   <repeatable command> REPEATWHILE <expression>
<until command> ::= UNTIL <expression> DO <command>
<while command> ::= WHILE <expression> DO <command>
<for command> ::=
   FOR <identifier> = <expression> TO <expression>
         BY <constant expression> DO <command> |
   FOR <identifier> = <expression> TO <expression>
         DO <command>
<repetitive command> ::= <repeated command> | <until command> |
   <while command> | <for command>
<test command> ::= TEST <expression> THEN <command> OR <command>
<if command> ::= IF <expression> THEN <command>
<unless command> ::= UNLESS <expression> THEN <command>
<unlabelled command> ::= <repeatable command> |
   <repetitive command> | <test command> | <if command> |
   <unless command>
```

9. Labelled commands

```
<label prefix> ::= <identifier> :
<case prefix> ::= CASE <constant expression> :
<default prefix> ::= DEFAULT :
<prefix> ::= <label prefix> | <case prefix> | <default prefix>
<command> ::= <unlabelled command> |
   <prefix> <command> | <prefix>
```

10. Blocks and compound commands

```
<command list> ::= <command> { ; <command> }
<declaration part> ::= <declaration> { ; <declaration> }
<block> ::= $( <declaration part> ; <command list> $)
<compound command> ::= $( <command list> $)
<program> ::= <declaration part>
```

Appendix C: Character Sets

Table C-1 gives the graphics of the IBM TN character set (in hexadecimal), while Table C-2 shows (in octal) the current ASCII standard character codes.

Each table runs from left to right, with the digit above each character representing the low order digit of its code, and the digits at the left representing the high-order digits of the code. Thus, for example, the EBCDIC code for @ is 7C.

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
00						*T							*P	*C		
10						*N	*B									
20								*E								
30																
40	*S										¢	.	<	(	+	|
50	&										!	$	*	)	;	
60	-	/										,	%	_	>	?
70											:	#	@	'	=	"
80		a	b	c	d	e	f	g	h	i		{	≤	(	+	+
90		j	k	l	m	n	o	p	q	r		}	□	)	±	■
A0	–	°	s	t	u	v	w	x	y	z		└	┌	[	≥	•
B0	0	1	2	3	4	5	6	7	8	9		┘	┐	]	≠	—
C0		A	B	C	D	E	F	G	H	I						
D0		J	K	L	M	N	O	P	Q	R						
E0			S	T	U	V	W	X	Y	Z						
F0	0	1	2	3	4	5	6	7	8	9						

Table C-1

<table>
<tr><th></th><th>00</th><th>01</th><th>02</th><th>03</th><th>04</th><th>05</th><th>06</th><th>07</th></tr>
<tr><td>000</td><td>NUL</td><td>SOH</td><td>STX</td><td>ETX</td><td>EOT</td><td>ENQ</td><td>ACK</td><td>BEL</td></tr>
<tr><td>010</td><td>BS</td><td>HT</td><td>LF</td><td>VT</td><td>FF</td><td>CR</td><td>SO</td><td>SI</td></tr>
<tr><td>020</td><td>DLE</td><td>DC1</td><td>DC2</td><td>DC3</td><td>DC4</td><td>NAK</td><td>SYN</td><td>ETB</td></tr>
<tr><td>030</td><td>CAN</td><td>EM</td><td>SUB</td><td>ESC</td><td>FS</td><td>GS</td><td>RS</td><td>US</td></tr>
<tr><td>040</td><td>(1)</td><td>!</td><td>"</td><td>#</td><td>$</td><td>%</td><td>&</td><td>(2)</td></tr>
<tr><td>050</td><td>(</td><td>)</td><td>*</td><td>+</td><td>,</td><td>-</td><td>.</td><td>/</td></tr>
<tr><td>060</td><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td></tr>
<tr><td>070</td><td>8</td><td>9</td><td>:</td><td>;</td><td><</td><td>=</td><td>></td><td>?</td></tr>
<tr><td>100</td><td>@</td><td>A</td><td>B</td><td>C</td><td>D</td><td>E</td><td>F</td><td>G</td></tr>
<tr><td>110</td><td>H</td><td>I</td><td>J</td><td>K</td><td>L</td><td>M</td><td>N</td><td>O</td></tr>
<tr><td>120</td><td>P</td><td>Q</td><td>R</td><td>S</td><td>T</td><td>U</td><td>V</td><td>W</td></tr>
<tr><td>130</td><td>X</td><td>Y</td><td>Z</td><td>[</td><td>(3)</td><td>]</td><td>(4)</td><td>_</td></tr>
<tr><td>140</td><td>(5)</td><td>a</td><td>b</td><td>c</td><td>d</td><td>e</td><td>f</td><td>g</td></tr>
<tr><td>150</td><td>h</td><td>i</td><td>j</td><td>k</td><td>l</td><td>m</td><td>n</td><td>o</td></tr>
<tr><td>160</td><td>p</td><td>q</td><td>r</td><td>s</td><td>t</td><td>u</td><td>v</td><td>w</td></tr>
<tr><td>170</td><td>x</td><td>y</td><td>z</td><td>{</td><td>|</td><td>}</td><td>(6)</td><td>DEL</td></tr>
</table>

Table C-2

Notes

(1) Space

(2) Accent acute, or apostrophe

(3) Backslash

(4) Circumflex

(5) Accent grave

(6) Tilde (equivalent to ¬)

Appendix D: Compilation Diagnostics

The compiler produces a number of diagnostics when it translates an erroneous program. It should be emphasised that many diagnostic messages may be produced by the same error; therefore, one change to the program may remove many errors. Errors are identified by a number (which is primarily useful to the person who maintains the BCPL compiler), and some text. The messages, and a brief explanation, are listed below:

Error 6 '$(' expected
Error 7 '$)' expected
Error 8 Name expected
Error 9 Untagged '$)' mismatch
Error 15 ')' missing
Error 19 ')' missing
Error 30 ',' missing

```
Error 32  Error in expression
Error 33  Error in number
Error 35  Illegal floating-point operator
Error 40  Name expected
Error 42  '=' or 'BE' expected
Error 43  Name expected
Error 44  '=' or '(' expected
Error 45  ':' or '=' expected
Error 50  Error in label
Error 51  Error in command
Error 54  'OF' expected
Error 57  '=' expected
Error 58  'TO' expected
Error 60  'INTO' expected
Error 61  ':' expected
Error 62  ':' expected
Error 63  '/' or '|' missing
Error 64  Erroneous use of UNRESERVE.
Error 88  Input '...'  not provided for GET
Error 89  String expected
Error 91  '8', '(', or ')' expected
Error 94  Illegal character
Error 95  String too long
Error 97  String expected
Error 98  Program too large
Error 99  Incorrect termination
Error 101 Illegal use of CASE or DEFAULT
Error 104 Illegal use of BREAK, LOOP, or RESULTIS
Error 105 Illegal use of CASE or DEFAULT
Error 106 Two cases with same constant
Error 109 Ltype expression expected
Error 110 LHS and RHS do not match
Error 112 LHS and RHS do not match
Error 113 Ltype expression expected
Error 115 Name not declared
Error 116 Dynamic free variable declared
Error 117 Error in constant expression
Error 118 Error in constant expression
Error 119 Error in constant expression
Error 141 Too many cases
Error 142 Name declared twice
Error 143 Too many names declared
Error 144 Too many globals
Error 145 OCODE buffer overflow
Error 147 Error in expression
Error 150 Incorrect entry or external declaration
Error 151 Too many entries or externals declared
Error 152 Too many static cells
Error 153 Illegal use of character when producing MCODE
Error 199 Invalid selector expression
```

Any code generator messages should be reported to the person who maintains the BCPL compiler.

Appendix V: Common extensions

The extensions listed here are widely available in a number of BCPL implementations, and, although they are not in the standard language, they should be considered by other implementers of BCPL planning to extend the language. This appendix is provided in the hope that it will reduce needless incompatibilities between different implementations.

It must be stressed that these extensions should only be used where absolutely necessary, and then as sparingly as possible. They tend to decrease the efficiency and the understandability of the program and often indicate bad programming style.

The following features are included as "standard extensions":

- floating point (on machines with suitable word length),
- selectors,
- lower case letters in identifiers,
- the warning sequences #B, #O, and #X in constants,
- the sequence "...* <layout> *..." in string constants,
- the operators ABS and #ABS;
- the operator <>,
- assignments of the form "E1 <op>:= E2",
- the expression "?",
- comment forms other than "//",
- the SECTION and NEEDS directives.

BCPL-V

BCPL-V contains a number of extensions to the language. These are:

- entry and external declarations (found in a few other implementations),
- the byte operator and the representations,
- /\ for LOGAND, \/ for LOGOR, == for EQV and >< for NEQV,
- the special characters *Xnn and *E,
- conditional compilation,
- the provision for system keywords to be spelled in either upper or lower case,
- the directives INCLUDE, and PARAMETER.

BCPL-V provides a special directive "UNRESERVE." designed to ease difficulties in moving programs originally designed to run on other implementations. If such a program contains a BCPL-V system word as an identifier, either a massive edit must be performed, or the reserved word removed. Thus, for example, if the program uses the name FIX as an identifier, the directive

```
UNRESERVE. FIX
```

will cause the compiler to forget that FIX is a system word. This declarative should only be used in cases of dire emergency.

In addition, BCPL-V also includes an augmented run time library.

Appendix F: Compilation listing

*** warning--this section may change soon ***

The compiler may be coerced into providing a listing of the source program. In batch mode, such a listing defaults, while in terminal mode, no listing is normally produced. The listing is affected by the S and Q parameters. Each listing page contains a title and a subtitle line, followed by a number of program lines. On each program line appear:

- a card number (generally not useful),
- a command number (very useful for interpreting translation error messages),
- the text of the line,

- the MTS line number (occasionally, this line number is incorrect, e.g., with GET. In such a case, it is printed as a row of dashes).

The listing may be controlled by the following options

- TITLE. <string> sets the operand as title for succeeding pages,
- SUBTITLE. <string> sets its operand as the subtitle and skips to a new page,
- NOLIST. Turns off the listing,
- LIST. Enables listing if requested via the parameter S, or if running in batch mode.

Note that, in order to distinguish these declarations from a BCPL identifier, they end with a point. Such a declarative should appear on a line by itself.

Cross referencing

*** warning: the cross referencer is currently being redesigned ***

Cross referenceing of a BCPL program can be obtained by using the PAR=X option on the MTS $RUN command. The cross referencing routines will produce a table of all identifiers used in the program, where they were declared, and where they were referenced. They will also produce lists of unused globals and externals. Additional cross referencing options are available. They must be placed within a pair of parentheses immediately following the X in the MTS PAR field. These additional options are as follows:

In addition to other cross referencing information, produce a header of only those globals, externals, and manifests which are used. This header will be produced on MTS logical unit 2.

If logical unit 2 is not assigned, the default file is "-hdr#".

G do not include the lists of unused globals and externals in the cross referencing information.
H print no cross referencing information, and produce a header of only those globals, externals, and manifests which are used. This header will be produced on MTS logical unit 2. If logical unit 2 is not assigned, the default file is "-hdr#".
N (narrow) print the cross referencing information such that it will fit on a 8.5 inch wide sheet of paper.
S print the cross referencing information with single spacing, the default being double-spacing.

As an example, placing PAR=X(NG) on the MTS $RUN command would print a narrow cross reference table without the lists of unused globals, externals, and manifests. The cross referencing feature is very useful for debugging programs.

Appendix G: Miscellany

Streams in BCPL-V

A BCPL-V stream is represented as a pointer to a block described as follows:

```
MANIFEST
$( NEXT = 0
   UNITFLAG = 1
   WINDOW = 2
   POSITION = 3
   CURRENTLENGTH = 4
   BUFFERLENGTH = 5
   MODBITS = 6
   LINENUMBER = 7
   BUFFER = 8
   NLFLAG = 9
   EOF = 10
   NOCC = 11
   WINDOWSET = 12
   FDUB1= 13
   FDUB2= 14
   FILENAME = 15    $)
```

There are other fields in a stream block but they are of no interest to the user. Of most interest are the MODBITS and LINENUMBER fields; for the use of these, see relevant MTS documentation. Many of the other fields can change unpredictably, especially if a concatenated input file is used. Naturally, delving into a stream is highly machine dependent.

Maintaining large BCPL programs

A large program should be stored as a set of files; common declarations may be included with GET. If only one section of code need be recompiled, the object code may be placed into a temporary file, and the master object file updated via a utility program (*ROBJ may be used in MTS). Of course, proper care must be taken to ensure that modules do not get out of step in this updating process.

Calling Assembler-Language Routines on the 370

Many non-BCPL routines are called with a FORTRAN-compatible calling sequence, and can be invoked by using the routines CALL, FLOATCALL, and RCALL. Occasionally, this is difficult or inconvenient, and a special assembler interface must be used. In order to facilitate such a painful task, a macro-library is available. Dedicated users should study the source code for the I/O interface before attempting to code assembler routines.

Writing Re-enterable code

If a BCPL program is to be used re-enterably, the following constraints must be followed.

1) Static variables should not be used; instead globals or locals should be employed (this is because of base register problems on the 370).

2) The run-time library on the 370 should be modified slightly, since it is not currently re-enterable.

3) No program should ever exceed its stack space, for the results could be disastrous on implementations with inadequate memory protection.

Appendix H: The BCPL/370 Runtime Environment

This appendix contains information which is arcane and not terribly useful to the average BCPL programmer. It is, however, useful to those wishing to implement a function or routine in Assembler. Such programmers will need to know something of the runtime organisation used by BCPL programs.

BCPL programs, on the 370, live in a universe which contains a stack and a static area. A small amount of assistance is provided by a routine written in Assembly Language, which allocates and organises the stack.

The general structure of a BCPL module is illustrated by the following example,

```
SECTION "FOO"
GLOBAL
$( START: 1;
   BAR: 200;
   ZOT: 201 $)
LET START() BE ...
AND BAR(A, B) BE ...
AND ZOT(G, H, I) = ...
```

which translates into the following Assembly Language skeleton:

```
FOO    BCPLCS
START  BCPLNTR   1
       ...
       BCPLEX
BAR    BCPLNTR   200,ARGS=2
       ...
       BCPLEX
ZOT    BCPLNTR   201,ARGS=3
       ...
       BCPLEX
       BCPLCSND
       END
```

BCPLCS generates a prologue which is primarily useful for mapstore, while BCPLCSND produces a table used in initialising the global vector. The BCPLNTR macro generates the entry sequence at the beginning of a function or routine (there is no difference between the two at the Assembly Language level); its arguments are the global number of the routine, and the number of parameters. This number must be greater than or equal to the number of parameters used when you call the routine. BCPLEX causes the routine to return to its caller.

Before describing the usage of registers, it is necessary to mention the distinction between a BCPL address and a machine address. In BCPL's world, consecutive cells have addresses differing by 1. However it is unfortunately true that the 370 was (mis)designed to have word addresses differing by 4. Therefore, a BCPL address is equal to the machine address divided by 4. This is not particularly inefficient (no worse than Fortran!); for the code

```
Z := !I
```

the compiler produces something like:

```
        L       A3,I
        AR      A3,A3
        L       A2,0(A3,A3)
        ST      A2,Z
```

which causes no extra memory references. However, it is still a source of trouble when you code an Assembler subroutine to be

called from BCPL.

The registers are allocated in the following manner:

register number	mnemonic	use
0	R0	contains 0
1	R1	contains 4096
2	R2	contains 8192
3	R3	contains 12288
4	B	program base
5	SP	stack pointer
6	RET	procedure return address
7	A1	procedure parameter, or scratch
8	A2	"
9	A3	"
10	A4	"
11	S	subroutine library base
12	GLOB	points to global vector
13	R13	dsect base
14	R14	temporary, system usage
15	R15	"

The values in registers 0-3 are used in addressing. If you <u>must</u> touch these registers (e.g., by calling an external routine), be sure to restore them before exiting.

The other registers which will concern you are A1-A4, which are free for use as scratch. By a fortunate coincidence, the first four parameters to a function or routine are found in those registers (other parameters appearing on the stack: see below). Further, the value of a function is placed in A1 (even if it's a floating-point number) at exit time.

If you need to access a global, you may use register GLOB, which points to the global vector. Thus, if you need the contents of global 263, you may code

```
        L       A1,263*4(GLOB)
```

Register S points to a collection of support subroutines used for procedure calling and statistics collection. The subroutines are located at fixed offsets from the address given in S. You don't normally need to worry about S.

R13, R14, and R15 have the normal OS significance, but, in addition, R13 points to a dsect which contains internal information needed by the runtime routines, while r14 and r15 may be used freely, without any need to restore.

The three remaining registers are used for procedure linkage. B contains the program base, SP the current stack pointer, and RET the return address. A stack level consists of first these three registers, as they were at procedure entry, followed by the parameters, and then any data declared by the procedure, as in the following diagram:

(To be supplied)

<u>BCPL Trouble Report</u>

Reports of problems in the BCPL system are earnestly solicited, as are comments on this manual. In the latter case, please give not only the page references but also sufficient context. Please direct any such comments to

BCPL Maintainer
Department of Computer Science
University of British Columbia
2075 Wesbrook Place
Vancouver, B.C. V6T 1W5
Canada

Nature of Problem...............................

....

....

....

....

....

....

...........................Page Nos..............

Name..

Address...

....

...........................Postal Code...........

Document Update Request

Please return the form below if you wish to receive a revision of this manual to:

BCPL Maintainer
Department of Computer Science
University of British Columbia
2075 Wesbrook Place
Vancouver, B.C. V6T 1W5
Canada

Name..

Address...

....

..............................Postal Code...............

Table of Contents

Introduction 0
Acknowledgements 0
The MTS version (BCPL-V) 1
Portability 1

Part I--An overview of BCPL 2

Part II--The BCPL Language and Portable Library 2

Language Definition 2
Program 2
Elements 3
Expressions 4
Addressing operators 6
Arithmetic operators 9
Relations 9
Shift operators 10
Logical operators 10
Operations combined with assignment 11
Conditional operator 12
Table 12
Constant expression 12
Floating point arithmetic 13
Field selectors 13
Section brackets 14
Commands 15
Assignment 16
Conditional commands 16
FOR command 17
Other repetitive commands 17
RESULTIS command and VALOF expression 18
SWITCHON command 19
Transfer of control 19
Compound command 20
Block 21
The operator <> 21

Declarations 21
Global 22
Externals 22
Manifest 23
Static 24
Dynamic 24
Vector 25
Function and routine 25
Label 26
Simultaneous declaration 27
Miscellaneous features 27

GET. 27
Comments and spaces 28
Optional symbols and synonyms 28
Segments 28
Segment headings 29

The Run-time Library 30
Elementary Input Functions 30
Elementary output routines 30
Other input routines 31
Other output routines 32
Enquiries 32
Input file manipulation 33
Output file manipulation 34
String manipulation 34
Programming aids 34
FORTRAN interface in BCPL-V 36
Miscellaneous 36

Part III--Using BCPL with MTS 38

Compilation and execution 38
Library declarations 38
Diagnostics 40
Compilation options 41
Execution options 42
Loading 42
Execution faults 42
The Profile Option 43
A complete job 45

Part IV--Reference Material 47

Appendix A: Basic symbols and examples 47

Appendix B: BNF of BCPL 49

Appendix C: Character Sets 52

Appendix D: Compilation Diagnostics 54

Appendix E: Common extensions 56
BCPL-V 56

Appendix F: Compilation listing 57
Cross referencing 57

Appendix G: Miscellany 58
Streams in BCPL-V 58
Maintaining large BCPL programs 59
Calling Assembler-Language Routines on the 370 59
Writing Re-enterable code 59

Appendix H: The BCPL/370 Runtime Environment 59