A PROPOSED DEFINITION OF THE LANGUAGE

B C P L

This document sets out a proposal for the definition of a standard BCPL and was prepared by M.D. Middleton in collaboration with

R. Firth (HPAC Ltd.)

M. Richards (University Cambridge)

I. Willers (CERN Geneva)

who are members of the BCPL Standards Committee elected at the BCPL User Meeting on 7th March 1979 in Cambridge.

M.D. Middleton
Das RZ der Universitaet Regensburg
Universitaetsstrasse 31
D-8400 Regensburg
W.-Germany

CONTENTS

BCPL Standard

1 Introduction

1.1 Scope and Purpose

This document has been produced in response to requests at the inaugural meeting of the BCPL Users Group on 7th March 1979. It is a revised version of a previous attempt at standardization (Middleton 1979) which was in turn based on the original BCPL manual (Richards 1969, 1973). It is intended as a specification of the language and its runtime system which will be adopted as a standard at a subsequent BCPL User Group meeting.

1.2 Language Extensions

The language and runtime system described in the body of this paper is BCPL-level 0 and is intended to form the basis of a standard to which all implementations should adhere.

The Appendix gives a number of possible extensions. These are grouped into 'Packets'. None of the packets is mandatory but if any facilities from any given packet described in the appendix are included then the whole packet should be implemented in the form described there.

1.3 Meta-Language

For the syntactic definition Backus Naur Form with the following extension is used:

[] are metabrackets which mean that the categories between the brackets must occur at least n times but not more than m times. If n or m is omitted the default values are n = 0 and m = .

For the semantic definition the following conventions are used unless explicitly qualified in a particular section.

E, E1, E2,... stand for arbitrary expressions

K, K1, K2,... stand for arbitrary constant expressions and

C, C1, C2,... stand for arbitrary commands.

2 Data

There are no explicit data types in BCPL. The only unit of data is the BCPL word which for any given implementation is a bit string of fixed length not less than 16 bits. A BCPL word may be used to represent values of many different types including integers, characters and truth values. As there are no explicit data types the compiler performs no type checking and always assumes that operands have the required type.

The basic unit of storage is a cell which is large enough to hold a BCPL word. Each available cell has an integer address which can be operated upon and stored in the same way as any other BCPL word. Adjacent cells have integer addresses that differ by one.

2.1 Data Storage

A BCPL program has at least three available areas of storage: dynamic, static and global.

2.1.1 Dynamic

This area is normally a single block of contiguous storage which is used for storage of temporary results and dynamic variables.

2.1.2 Static

This is a not necessarily contiguous area of store in which static variables can be stored.

2.1.3 Global

This is a block of contiguous store which is used for communication between seperately compiled sections of a BCPL program. The cells of this block are numbered from zero up to some limit and all are available to each section.

2.1.4 Other storage areas

Other storage areas may (but do not have to) be made available to the program by means of procedure calls. See section A7 in the Appendix.

3 Lexical considerations

The machine representation of a BCPL program is dependent on the character set used.

3.1 Character set

The character set is divided into the following:

<layout char>	::=	<space char> | <newline char>
<space char>		includes at least 'space' and 'tab'
<newline char>		is any 'carriage control character' such as linefeed, new page etc.
<letter>		includes all upper case letters and, if the implementation allows, lower case letters. Lower case letters have the same meaning as upper case letters except in strings and character constants where they stand for themselves.
<digit>	::=	0|1|2|3|4|5|6|7|8|9
<special>		a set of special characters which are used to build BCPL basic symbols.
<other>		all other characters in the character set. These may occur only in strings, character constants and comments.

3.2 Basic contructions of the language

The basic syntactic unit in BCPL is the <symbol>

<symbol> ::= <element> | <basic symbol>.

With the exception of a restriction on <newline character> any number of <layout character>s may appear between <symbol>s, but <layout character>s may not appear within a <symbol>. It will also be seen from the syntax that <layout character>s are sometimes necessary to separate <symbol>s which would otherwise elide, for example <name> and <dec number>.

3.2.1 Tag, Name, Identifier

<tag> ::= [<tag character>]

where <tag character> is one of <letter>, <digit> or '.' (dot). Some implementations also allow the underline character in tags.

<name> ::= <letter> <tag>

<identifier> is a <name> which is not a <basic symbol>.

3.2.2 Basic Symbol

A <basic symbol> is a symbol made up of either a sequence of letters or a sequence of <special>s.

The machine representation of <basic symbol>s is implementation dependent and in this document a canonical representation of the <basic symbol>s is used. The following

table gives the set of canonical <basic symbol>s together with a list of alternative representations.

basic symbol	alternative representation(s)		
TRUE			
FALSE			
?	NIL		
(	[		
)	]		
@	LV		
!(monadic)	RV		
!(dyadic)			
*			
/			
REM			
+			
-			
=	EQ		
~=	=	NE	\=
<	LT		
<=	LE		
>	GT		
>=	GE		
<<	LSHIFT		
>>	RSHIFT		
NOT	~		\
&	/\	LOGAND	
|	\/	LOGOR	
EQV			
NEQV			
->			
,			
TABLE			
VALOF			
;			
:			
VEC			
BE			
LET			
AND			
:=			
BREAK			
LOOP			
ENDCASE			
RETURN			
FINISH			
GOTO			
RESULTIS			

```
SWITCHON
INTO
REPEAT
REPEATUNTIL
REPEATWHILE
UNTIL
WHILE
FOR
TO
BY
IF
UNLESS
CASE
DEFAULT
DO                  THEN
ELSE                OR
ABS
GET
MANIFEST
GLOBAL
STATIC
```

Further Symbols

The following character combinations are used in certain BCPL constructions and are given here for completeness.

character combination		construction
$(	)	section brackets
$)	)	
'(single quote)		character constant
"(double quote)		string constant
#		literal
//	)	comments
/*	)	
*/	)	

Extended Basic Symbols

The following list gives basic symbols and character combinations used in recommended extensions.

symbol	alternative representations		
#+			
#-			
#*			
#/			
#=	#EQ		
#~=	#NE	# =	#\=
#<	#LT		
#<=	#LE		
#>	#GT		
#>=	#GE		
#ABS			
FIX			
FLOAT			
SECTION			
NEEDS			
EXTERNAL			
%			
$$			
$<			
$>			

3.2.3 Element

```
<element>          ::= <identifier> | <literal>
```

An <identifier> represents either a variable which at runtime will be bound to some particular cell or a <manifest constant> which is treated as a literal. A <literal> is a direct representation of a constant.

```
<literal>          ::= <number> | <character constant> |
                       <string constant> | <logical value> |
                       <undefined>
```

Number

```
<number>           ::= <based number> | [<digit>]

<based number>     ::= #[O]  [<octal digit>]  |
                       #B[<binary digit>]  |
                       #X[<hex digit>]
```

<octal digit> ::= 0 | 1 | ... 7

<binary digit> ::= 0 | 1

<hex digit> ::= 0|1|2|3|4|5|6|7|8|9||A|B|C|D|E|F

Semantics

A number has some machine representation which is chosen such that the BCPL operators have the expected results. Binary, octal and hexadecimal numbers may be written with leading zeros suppressed and, so long as the cell length of the implementation is not exceeded, the normal rules for conversion between positive numbers in these bases hold.

Character constant

<character constant> ::= '<string character>'

Any single character may appear within the single quotes except *, ' or <newline character>. <string character> may also be any of the following special representations:

special representation	represents
**	*
*'	'(single quote)
*"	"(double quote)
*S or *s	space
*T or *t	tab
*N or *n	newline
*P or *p	newpage
*Xnn or *xnn	the hexadecimal character nn
*Onnn or *onnn	the octal character nnn

Semantics

A character constant is the representation of the given character in an implementation dependent internal code. It occupies a complete BCPL word and is right justified and padded on the left with zeros.

String constant

<string constant> ::= "<up to K string characters>"

Within a string any character represents itself except *, " and <newline character>. The special representation of character constants (see above) may be used within a string. K is an implementation constant whose value is not less than 127; in most implementations it is 255.

In a string the sequence

* [<layout char>] *

is ignored.

Semantics

A string constant is represented by the BCPL-address of the zeroth of a set of contiguous cells which at runtime hold the actual characters of the string together with its length packed. Each cell is divided into a number of 'bytes' (not necessarily of the same size). Byte zero of cell zero of a

string contains the length and subsequent bytes contain the characters packed contiguously and in the given sequence. The unused bytes of the last cell are filled with binary zeros. See section 10.3 for information about the string handling library procedures.

Logical value

<logical value> ::= TRUE | FALSE

The meaning of the values is self explanatory. The machine representation is such that the operators

NOT, &, |, EQV, NEQV

have the expected results, see section 4.3.1. Thus the representation of FALSE is a bit pattern of all zeros and the representation of TRUE is a bit patern of all ones.

Undefined

<undefined> ::= ?

The value of this is undefined.

3.3 Omission of symbols

The symbols ; (semicolon), DO and THEN can be omitted when the compiler can tell by context whether one is required or not. There are, however, a few cases where ambiguities can arise and these are resolved by the compiler according to the following two rules:

a) The first symbol after a newline character may not be a dyadic operator, or -> or a comma.

b) The compiler reads the source program sequentially symbol for symbol without backtracking and while parsing a given construction accepts all symbols which (with the exception of rule (a) could possibly belong to that construction. The first symbol which could not belong to the given construction is taken as the start of a new construction.

With these rules in mind the following recommendations are made to programmers over omission of symbols.

The following syntactic conventions are safe.

a) Semicolon should only be omitted if it is the last non-comment symbol on a line.

b) DO or THEN should only be omitted when the following symbol is one that can only start a command.

3.4 Tagged brackets

The section brackets $(and $) may be written as $(<tag> and $)<tag>. Each opening section bracket must be matched by an identically tagged closing bracket. However, when the compiler finds a closing section bracket with a non-null tag, if the nearest opening section bracket does not match, that section is closed and the process is repeated until a matching opening section bracket is found. Thus syntactically a tagged opening section bracket is the same as a null-tagged (untagged) one and a tagged closing bracket is the same as one or more untagged closing section brackets. Some implementations generate a warning message if a tagged section bracket is not explicitly closed by a matching one.

3.5 Comments

Comments may be introduced by one of the two symbols

// or /*

The sequence

// [<character other than <newline char>>] <newline char>

has the syntactic significance of a <newline>

The sequence

/* <any character sequence that does not contain*/> */

has the syntactic significance of a <layout char>.

3.6 GET

It is possible to include a file in the source text by using the directive

GET <file identity>

where <file identity> is a machine dependent identification of a file. This is usually a string.

The GET directive must appear on a line by itself and the effect is to replace this line with the text of the given file.

4 Expressions

4.1 Syntax of expression

The following syntax defines the form of BCPL expressions. This syntax must be used in conjunction with the binding power of the operators to give an unambiguous parsing of an expression.

```
<expression>      ::= <element> | (<expression>) |
                      <function call> |
                      <expression> [<dyop> <expression>] |
                      <monop> <expression> |
                      <conditional exp> | VALOF <command> |
                      TABLE <expression list>

<monop>           ::= ! | @ | ABS | + | - | NOT

<dyop>            ::= ! | * | REM | / | + | - | = | ~= |

                      < | > | <= | >=| << | >> | & | | |

                      EQV | NEQV

<function call>   ::= <procedure call>

<procedure call>  ::= <expression>(<expression list>) |
                      <expression>()

<conditional exp> ::= <expression> -> <expression>,
                      <expression>

<expression list> ::= <expression> [,<expression>]
```

<command> is defined in section 5.

4.2 Operator precedence

Ambiguities in the above syntax are resolved by the following order of binding power.

(highest, most binding)	() (bracketed expression)
	Procedure call
	! (dyadic)
	! (monadic) @
	* / REM
	+ - (monadic and dyadic)
	= ~= < <= > >=
	<< >>
	NOT
	&
	|
	EQV NEQV
	-> , (conditional comma)
	TABLE , (comma in a table)
(lowest, least binding)	VALOF

Operators of equal precedence associate to the left.

VALOF is different from the other operators in that it operates on a <command> rather than on an expression. In so far as this <command> can terminate with an <expression>, VALOF is the least binding operator.

4.3 Semantics of expression

There are five context dependent modes of evaluation of expressions

R-mode
L-mode
assignment mode
truth value mode
constant.

4.3.1 R-mode expressions

The normal mode is R-mode and unless something to the contrary is stated all expressions are evaluated in this mode. All operators are valid in R-mode.

Brackets

```
(E)
```

Brackets serve only to affect the grouping of operands of an expression.

Function call

```
E(E1, E2, ...)
```

See procedure call section 6.7

Vector application

```
E1!E2
```

The BCPL vector application is a symetrical operation such that:

```
E1!E2 = !(E1+E2)
```

Note that this implies E1!E2 = E2!E1. One interpretation of the expression E1!E2 is that E1 is a pointer to a set of contiguous cells (a vector) and E2 is an index. The result of the operation is the E2th cell of the vector.

Indirection

```
!E
```

The ! operator acts as an indirection operator. The expression E is evaluated and is interpreted as the address of a cell whose content is then the result of the whole expression.

Address of

```
@E
```

The operator @ causes E to be evaluated as an address. The expression E is evaluated in L-mode and the result of the whole expression is then this value.

Arithmetic operators

The arithmetic operators

```
* / REM + -
```

operate on values as if they were integers. REM is the remainder (modulus) operator:

```
if   m/n = q
and  m REM n = r
then q * n + r = m
```

for m, n (except n=0), if m and n are both positive then q is the largest integer which satisfies the above equations for

positive r. The direction of rounding is undefined for the operator / if either of its operands are negative.

For all arithmetic operations the effect of integer overflow is ignored.

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
```

These operators make arithmetic comparisons of their operands. An extended relational expression such as

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

is equivalent to

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

Shift operators

In the expression E1<<E2 (E1>>E2), E2 should evaluate to yield a non-negative integer. The value is E1, taken as a bit pattern, shifted left (or right) by E2 places. Vacated positions are filled with 0 bits. If the value of E2 is an integer larger than the word length on the implementation number of bits in a word then the then the result of the

shift operations is undefined.

Logical operations

These operate on values considered as bit patterns: the operator NOT causes bit by bit complementation of its operand. The other operators combine their operands bit by bit according to the following table.

Operands		Operator			
		&	|	EQV	NEQV
0	0	0	0	1	0
0	1	0	1	0	1
1	0	0	1	0	1
1	1	1	1	1	0

Conditional operations

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

E1 is evaluated in truth-value mode. The result of the expression is the value obtained by either evaluating E2 or E3 depending on whether E1 yields TRUE or FALSE, respectively. If E1 yields neither then the result is undefined.

Table

The value of the expression

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

is the address of the zeroth element of a static vector of n + 1 cells initialized to the values K0, K1, K2, ..., Kn which must be constant expressions.

VALOF expression

The expression

VALOF C

where C is a command, is evaluated by executing C until a RESULTIS command is encountered. The value of the VALOF expression is then the value of the expression contained in the RESULTIS command and execution of C finishes.

4.3.2 L-mode expressions

L-mode expressions occur in two contexts:

a) as operand of the @ operator, or

b) on the left hand side of an assignment command.

An expression is evaluated in L-mode to give the address of a cell. Therefore only the following constructions are allowed:

N	where N is an identifier which is the name of a dynamic, static or global cell. The value is the address of the given cell.
!E	where E is any expression. The value is E
E1!E2	where E1 and E2 are any expressions. The value is E1 + E2.

4.3.3 Assigment mode

An assignment mode context occurs only on the left hand side of an assignment command. Here the expression is evaluated to give the address of a cell in which a value will be stored. For further details of the assignment command, see section 5.2.

4.3.4 Truth-value mode

A truth-value context occurs whenever the result of the expression will be interpreted immediately as TRUE or FALSE. The expression is evaluated only so long as is necessary to determine whether it is true or false - more precisely: if after parsing (i.e. taking account of brackets and binding power) the least binding operator is &, | or NOT it is evaluated from left to right according to the following rules:

Op	Form of Expression	Evaluation as truth value
NOT	NOT E1	E1 is evaluated in truth-value mode and if this is TRUE the result is FALSE, otherwise the result is TRUE.
&	E1&E2	E1 is evaluated in truth-value mode. If this result is FALSE, the whole expression is FALSE, otherwise the expression has the result E2 evaluated in truth-value mode.
|	E1|E2	E1 is evaluated in truth-value mode. If this result is true then the whole expression is true, otherwise the whole expression has the value of E2 evaluated in truth-value mode.

In all other cases the expression is evaluated in R-mode and the result is interpreted as TRUE or FALSE. If the result of this evaluation does not actually yield either of the values TRUE or FALSE then the result is undefined.

4.3.5 Constant expressions

A constant expression is one which must be evaluated at compile time and may include only:

<identifiers> declared as MANIFEST constants

<number>

<character constant>

<logical value>

<undefined>

ABS

+ - * / REM

<< >>

NOT & | EQV NEQV

5 Commands

The complete set of commands is given in this section

```
<unlabelled command>  ::= <routine call> | <assignment> |
                          <conditional> | <repetitive> |
                          <for command> | <resultis> |
                          <switchon> | <transfer> | <block> |
                          <compound>
```

A definition of compound commands and blocks is given in section 7.2. The definition of the remainder of the commands is given in the following subsections.

5.1 Routine call

```
<routine call>    ::= <procedure call>
```

See procedure call section 6.7.

5.2 Assignment

```
<assignment>      ::= <expression list> <assop>
                      <expression list>

<assop>           ::= :=
```

Semantics

There are two basic forms of the assignment command:

a) Simple assignment command

```
E1 := E2
```

The expression E2 is evaluated in R mode to give a BCPL-word and the expression E1 is evaluated in

assignment mode to give the identity of a place where this should be stored. If E1 is an L-mode expression this is simply a cell in which the result is to be stored.

b) Multiple assignment

L1, L2, ... := E1, E2, ...

The expressions L1, L2, ..., E1, E2, ... are evaluated and assigned to the cells defined by the assignment mode expressions L1, L2, ... in an undefined order. Some assignment may take place before all left and right hand expressions have been evaluated.

5.3 Conditional

Syntax

```
<conditional>     ::= IF <expression> THEN <command> |
                      UNLESS <expression> THEN <command> |
                      TEST <expression> THEN <command> ELSE
                      <command>
```

Semantics

The semantic forms of the command are

IF E THEN C1

UNLESS E THEN C2

TEST E THEN C1 ELSE C2

E is evaluated in truth-value mode and if the result is true C1 is executed otherwise C2 is executed.

5.4 Repetitive commands

Syntax

```
<repetitive>      ::= WHILE <expression> DO <command> |
                      UNTIL <expression> DO <command> |
                      <command> REPEAT |
                      <command> REPEATWHILE <expression> |
                      <command> REPEATUNTIL <expression>
```

Semantics

The command is executed repeatedly until the condition (<expression>) becomes true or false as implied by the command. If the condition preceeds the body (WHILE, UNTIL) the test will be made before each execution of the body. If it follows the body (REPEATUNTIL, REPEATWHILE) the test will be made after execution of the body which is therefore executed at least once. The rule that as much as possible is included in the construction being parsed (see section 3.3), applies here. Thus for example

```
    WHILE E1 DO C REPEATUNTIL E2
```

is the same as

```
    WHILE E1 DO $( C REPEATUNTIL E2 $)
```

and

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

is the same as

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

5.5 FOR command

Syntax

```
<for command>     ::= FOR <indentifier> = <expression>
                      TO <expression> [BY <expression>]  DO
                      <command>
```

Semantics

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

If the constant K is positive this is equivalent to

```
        $( LET N, d = E1, E2
           WHILE N <= d DO
           $( C
              N := N + K
           $)
        $)
```

If the value of K is negative N <= d is replaced by N >= d. If 'BY K' is omitted 'BY 1' is assumed. The declaration

```
        LET N, d
```

declares two new variables N and d; d being a new identifier which does not occur in C. On some implementations (particularly 16 bit word addressed machines capable of addressing a segment of more than 32k words) the test is for N-d <= 0 or N-d >= 0 as the case may be.

5.5 RESULTIS command

Syntax

<resultis> ::= RESULTIS <expression>

Semantics

This command gives the result of the smallest textually enclosing VALOF expression (see section 4.3.1). It may occur only in the body of a VALOF expression.

5.6 Switchon

Syntax

<switchon> ::= SWITCHON <expression> INTO <compound>

where the compound command contains <case label>s.

Semantics

A case label has the form

CASE K:

or DEFAULT:

where K is a constant expression.

The switchon command is executed by first evaluating the expression and if a case exists which has a constant with this value, then execution is continued from that label; otherwise if there is a default label execution is continued

from there; otherwise execution is continued from the point just after the end of the switchon command.

5.7 Transfer

Syntax

<transfer> ::= GOTO <expression> | FINISH | RETURN | BREAK | LOOP | ENDCASE

Semantics

a) GOTO E

The expression is evaluated and interpreted as an address to which control is transferred. The only meaningful result of the expression is the value associated with a label (see section 6.8). GOTO may occur anywhere in the program where a command is allowed. The label to which control is transferred must be in the same procedure body as the GOTO command.

b) FINISH

may occur anywhere in the program where a command is allowed and causes an implementation dependent termination of the entire program.

c) RETURN

causes control to be returned to the caller of the current procedure.

d) BREAK

causes execution of looping command to be terminated. Control is resumed just after the end of the smallest textually enclosing looping command. The resumption point must be in the same procedure body as the BREAK command. A looping command is either a repetitive command or a FOR command.

e) LOOP

causes execution of a looping command to be repeated. Control is transferred to the point just before the end of the body of the looping command. For a FOR command this is the point where the control variable is incremented and for the repetitive commands it is the point where the condition (if any) is tested. The resumption point must be within the same procedure body as the LOOP command.

f) ENDCASE

causes control to be transferred to the point just after the end of the smallest textually enclosing switchon command. The resumption point must be within the same procedure body as the ENDCASE command.

6 Declarations

Every identifier used in a BCPL program must be declared explicitly. There are 10 distinct declarations in BCPL which fall into two groups.

a) Static declaration:

Global, Static, Manifest, Function, Routine, Label.

b) Dynamic declarations:

Dynamic, Vector, Formal-parameter, For-command-control.

The declaration of formal parameters is described in section 6.7. and the for-command is described in section 5.5. All other declarations except label declarations occur at the head of a block and are known as block-head-declarations.

```
<block head declaration>  ::= <global dec> | <static dec> |
                              <manifest dec> | <dynamic dec> |
                              <vector dec> | <function dec> |
                              <routine dec>
```

6.1 Scope and extent of block-head-declarations

The scope of an identifier (i.e. the region of the program in which it is known) is the declaration in which the identifier is declared (to allow for recursive definition), the subsequent declarations and commands up to the end of the smallest textually enclosing block, or the end of the program if there is no textually enclosing block; but for dynamic declarations excluding any textually nested procedure bodies. This restriction on dynamic declarations means that no

reference may be made in a procedure to a dynamically declared identifier which is declared outside the procedure (such quantities are called dynamic free variables).

Identifiers declared at the same level must have different names, but an identifier may be declared with the same name as one declared a different level. In this case the two identifiers are normally distinct and the identifier declared at the textually outer level is not directly accessible within the scope of the identifier declared at the inner level. (For exceptions see rule for initializing global cells in sections 6.7 and 6.8).

The extent of an identifier (that is the duration at run time when a cell is actually assigned to the identifier) depends on the type of the declaration. There are two possible extents in BCPL.

a) Static

   The identifier is permanently associated with one particular cell and this cell remains available throughout the run.

b) Dynamic

   On each entry to the block containing the declaration a cell is associated with the identifier. This cell remains available until control passes to the end of the block containing the declaration. Note that if the declaration is invoked recursively there may be more than one instance

of the same identifier existing at any one time.

6.2 Global declaration

The means of communication between separately compiled segments of a program is the global vector.

Syntax

```
<global dec>      ::= GLOBAL <glob defs>
<glob defs>       ::= $( <glob def> [;<glob def>] $)
<glob def>        ::= <identifier>:<expression>
```

Semantics

The declaration

```
GLOBAL $( <glob def1>; <glob def2>; ... <glob defn> $)
```

is a syntactic abbreviation of

```
GLOBAL $( <glob def1>  $)
GLOBAL $( <glob def2>  $)
...
GLOBAL $( <glob defn>  $)
```

The declaration

```
GLOBAL $( N : K $)
```

(where K is a constant expression) associates the identifier N with the Kth cell of the global vector. Thus N identifies a static cell which may be accessed by N or any other identifier associated with the same global vector cell.

6.3 Static Declaration

Syntax

```
  <static dec>      ::= STATIC <sm defs>

  <sm defs>         ::= $( <sm def> [; <sm def>] $)

  <sm def>          ::= <indentifier> = <expression>
```

Semantics

The declaration

```
 STATIC $( <sm def1>; <sm def2>; ... <sm defn> $)
```

is a syntactic abbreviation of

```
 STATIC $( <sm def1> $)

 STATIC $( <sm def2> $)

 ...

 STATIC $( <sm defn> $)
```

The declaration

```
 STATIC $( N = K $)
```

where K is a constant expression causes permanent allocation of a cell to the identifier N. This cell will be initialized to the value K prior to execution of the program.

6.4 Manifest

Syntax

<manifest dec> ::= MANIFEST <sm defs> sp

<sm defs> is defined in section 6.3.

Semantics

The declaration

```
MANIFEST  $( <sm def1>; <sm def2>; ... <sm defn> $)
```

is a syntactic abbreviation of

```
MANIFEST  $( <sm def1> $)
MANIFEST  $( <sm def2> $)
...
MANIFEST  $( <sm defn> $).
```

The declaration

```
MANIFEST $( N = K $ )
```

causes the name N to be associated with the value given by the constant expression K. This association takes place at compile time and no storage cell is involved at run time. Thus the value associated with the name cannot be changed and the name cannot be used in a L-mode or assignment-mode contexts.

6.5 Dynamic declaration

Syntax

```
<dynamic dec>      ::= LET <identifier list> =
                       <expression list>

<identifier list> ::= <identifier>[, <identifier>]
```

Semantics

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

declares identifiers N1, N2, ... Nn. At run time a cell is allocated to each of these identifiers dynamically when control passes through the declaration, and these cells are initialized in an undefined order with the values E1, E2, ... En. The initial value may also be given as undefined (?) in which case no initialization takes place.

6.6 Vector declaration

Syntax

```
<vector dec>       ::= LET <identifier> = VEC <expression>
```

Semantics

LET N = VEC K

declares a variable N which points to a vector of K+1 cells where K is a constant expression. At run time K+1 contiguous

cells numbered from 0 to K are allocated dynamically and a further cell is allocated with which the identifier is associated. This latter cell is initialized with the address of the zeroth cell of the vector. The cells of the vector are not initialized.

6.7 Procedure

There are two types of procedure in BCPL: the function and the routine.

Syntax

```
<function dec>    ::= LET <identifier> <par list> =
                      <expression>

<routine dec>     ::= LET <identifier> <par list> BE
                      <command>

<par list>        ::= () | (<identifier list>)
```

Semantics

Routines and functions are equivalent except that a function yields a result whereas a routine does not. A function may be called as a routine and a routine may be called as a function (returning an undefined value).

The declarations

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

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

declare a function (routine) named N with n parameters. The brackets are required even if n=0. A parameter has the scope of the expression E (command C).

If the procedure declaration is in the scope of a global declaration with the same name, then the global cell will be initialized with the entry address of the procedure before execution of the program. Otherwise, a static cell is created, is associated with the identifier N, and is initialized with the entry address.

A procedure is invoked by the call

E0(E1, E2, ... En)

where E0 is evaluated to give the entry address. In particular, within the scope of the identifier N, the procedure N may be invoked by the call

N(E1, E2, ... En)

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

Arguments are passed by value. Each argument (Ei) is evaluated and the value is copied into a newly created cell which is then associated with the parameter Pi. The order of evaluation of arguments is undefined. These cells are consecutive in store so that the argument list behaves like an initialized vector. The space allocated to parameters is

released when evaluation of the procedure is complete. Notice that although arguments are always passed by value, this value may be an address.

6.8 Labels and Prefixes

Syntax

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

Semantics

Case label and default label are described in section 5.6 (Switchon).

The declaration

```
N:
```

declares the label N. Exactly as in the case of a procedure declaration a label causes a static to be declared if it is not within the scope of a global declaration of the same identifier. The local or global cell is initialized before execution of the program with the address of the point in the program labelled, so that the command

```
GOTO N
```

has the expected effect.

The scope of a label is the smallest of the following regions:

a) the command sequence of the smallest textually enclosing block, or

b) the body of the smallest textually enclosing VALOF expression routine or for-command.

Using a goto command to transfer control to a label which is outside the current procedure will produce undefined (chaotic) results. Such transfers can be performed by using the library procedures LEVEL and LONGJUMP (see section 10.2).

6.9 Simultaneous declarations

Any declaration of the form

```
LET ...
```

may be followed by one or more declarations of the form

```
AND ...
```

where any construction which may follow LET may also follow AND. As far as the scope is concerned, such a collection of declarations is treated as a single declaration. The order in which declarations are evaluated is undefined.

7 The program constructions

A BCPL program consists of a number of separately compilable sections (modules) which will be loaded together with a runtime system to produce a complete program. The individual sections communicate with one another by means of the global vector.

7.1 Section

Syntax

```
<BCPL section>    ::= <declarations>

<declarations>    ::= <block head declaration>
                      [; <block head declaration>]
```

Semantics

Although in principle all block-head-declarations are allowed in a BCPL section, only global, static, manifest and procedure declarations are meaningful.

7.2 Block and compound command

Syntax

```
<block>            ::= $( <declarations> ;
                       <command sequence> $)

<compound>         ::= $( <command sequence> $)

<command sequence>     ::= <command> [; <command>]
```

Semantics

A block or compound command is syntactically equivalent to a single command. The commands in a command sequence are executed in the order in which they occur.

8 Standard Header File

In order to simplify the language and allow for efficient interfacing with the operating system the procedures described in sections 9 and 10 are not declared as standard in the compiler. The declarations for these routines are contained in a standard header file which may be incorporated into each program by means of a GET directive. This process costs a little at compile time but has significant advantages for cross-compilation and transportation of BCPL programs.

Global cells 0 to n, where n is implementation dependent, are reserved for system procedures and these are defined in the standard header, together with a number of manifest constants. On most implementations, n is at least 99.

8.1 Globals in the Standard Header File

All Procedures names (including START) described in sections 9 and 10 of this document are allocated cells in the global vector. All further procedures from the appendix and implementation dependent routines which are implemented as part of the runtime system of the installation are also included, as is the global cell:

RESULT2

A general working cell which may be used by functions to

return a 'second result'.

8.2 Manifest Constants in the Standard Header

The following manifest constants are defined in the standard header.

ENDSTREAMCH

The result returned by RDCH when a stream is exhausted. On most implementations ENDSTREAMCH=-1.

BYTESPERWORD

The number of 'bytes' (=characters) packed in a cell in strings and in PUTBYTE and GETBYTE.

FIRSTFREEGLOBAL

The number of the first global cell which is available to the user.

BITSPERWORD

The number of bits in a BCPL word.

MAXINT

The most positive integer which may be held in a BCPL word.

MININT

The most negative number which may be held in a BCPL word.

9 Input/Output

Input/output facilities in BCPL are always invoked by procedure calls. The basic form of I/O always takes place via streams. A stream is basically an ordered sequence of normal characters, intermixed with newline, space, and other format characters, which are accessed sequentially. Each stream is identified by a stream identity the form of which is implementation dependent. Normally this is either an integer in a given range or the address of a control block. On entry to a BCPL program an input stream (the standard input for the operating system), and an output stream (standard output = printer or terminal) may be set up and selected.

Many of the procedures associated with input do not include a stream identity in the call. These procedures operate on the 'Current Input Stream' (CIS) which is defined by the routine SELECTINPUT. Likewise many output procedures operate on the 'Current Output Stream' (COS).

9.1 Standard Stream Organization Procedures

STREAM := FINDINPUT(S)

This function initializes a stream for reading. S is a string which identifies the stream to the operating system. The content of the string is implementation dependent. The result of the function is the a value which represents the stream and is used SELECTINPUT. If the stream cannot be opened for some reason the value zero is returned and an implementation dependent error code is given in RESULT2.

STREAM := FINDOUTPUT(S)

As for FINDINPUT but for output streams.

SELECTINPUT(N)

CIS:=N. All calls of RDCH will operate on stream N until changed by a further call of SELECTINPUT. This routine may be called with a given stream indentity even if this stream is already the CIS. N=0 is allowed and causes CIS to be undefined in which case future calls of RDCH will cause an error.

SELECTOUTPUT(N)

COS:=N. As SELECTINPUT.

ENDREAD()

This routine closes CIS and sets the input stream selection to undefined.

ENDWRITE()

This routine closes COS and sets the output stream selection to undefined.

REWIND()

The CIS is rewound. Not all streams are rewindable. The CIS is closed and re-opened for input.

STREAM := INPUT()

The stream identity of the CIS is returned. If the result is zero there was no currently selected input stream.

STREAM := OUTPUT()

The stream identity of the COS is returned. If the result is zero there was no currently selected output stream.

9.2 Standard Input/Output Procedures

CH := RDCH()

This function delivers the next character from CIS. If the stream is exhausted it yields the value ENDSTREAMCH which is a manifest constant.

UNRDCH()

This routine backspaces the CIS so that the next call of RDCH will have the same effect as the last call. UNRDCH() may be applied to many streams independently:

each stream has its own one-character "unread" state. The effect of successive calls of UNRDCH is undefined. If UNRDCH is called when no characters have been read the effect is undefined.

N := READN()

This function reads and yields the value of a decimal number [<format>][+|-] [<digit>] where <format> is a sequence of layout characters ('*S', '*N', '*P', '*T') and <digit> is a decimal digit. READN will read all characters up to the first non digit following the digit string. The terminating character is returned to the input stream using UNRDCH. If there were no decimal digits the number returned is zero. All numbers capable of being represented in an integer (including the largest negative number on a twos complement machine) will be read correctly. The result of attempting to read a number which cannot be represented as an integer is undefined.

WRCH(CH)

This routine writes CH to COS.

WRITES(S)

This routine writes the string S to COS using WRCH.

WRITED(N,W)

This routine writes N to COS in decimal using WRCH, right justified and with sign if negative in a field width W. If the given field is not big enough to hold the value then it is output in the minimum possible field length.

WRITEN(N)

This routine writes N to COS in minimum field width using WRCH.

WRITEOCT(N,D)

This routine writes D least signigicant octal digits of N to COS using WRCH.

WRITEHEX(N,D)

This routine writes D least significant hexadecimal digits of N to COS using WRCH.

WRITEF(F,A1,A2,...,A11)

This routine writes the arguments A1,A2,... to COS according to the format string F. The format string F is copied character for character to COS until the end is reached in which case the procedure is terminated or until a warning character '%' is encountered in which case the action depends on the next character(s) as

follows:

S Write next arg.(Ai) by WRITES(Ai)

C Write next arg.(Ai) by WRCH(Ai)

N Write next arg.(Ai) by WRITEN(Ai)

In Write next arg.(Ai) by WRITED(Ai,n)

On Write next arg.(Ai) by WRITEOCT(Ai,n)

Xn Write next arg.(Ai) by WRITEHEX(Ai,n)

% Write '%'

$ Skip next arg.(Ai)

The letters S,C,N,I,O,X in the above table may be in upper or lower case. The field width n is a single hex digit (0-9, A-F). After outputing the argument the whole process is repeated starting at the next character of the format string.

NEWLINE()

This routine writes a newline to COS using WRCH.

NEWPAGE()

This routine writes a newpage to COS using WRCH.

10 BCPL Runtime System

The runtime system includes code necessary for such things as initialization, procedure entry and exit etc. together with the standard I/O and other routines described in this chapter and accessible via the global vector.

10.1 Start and Stop

START(ARG)

A BCPL program is invoked by calling the (user written) function START which is by convention global number one. On many implementations the argument ARG is a string which is passed from the operating system to the user program. On some implementations parameters may be passed to the user program via a stream which is accessed via FINDINPUT with an implementation dependent argument. On entry to START a standard input stream and a standard output stream may exist and be selected. Return from START is exactly equivalent to FINISH.

STOP(N)

This routine causes termination of the program. N is the completion code that is passed back to the operating system. STOP(0) is equivalent to FINISH.

10.2 Stack organization routines

P := LEVEL()

This function gives a representation of the current procedure activation level for use with LONGJUMP.

LONGJUMP(P,L)

This routine causes a non-local jump to the label L at the activation level P.

RES := APTOVEC(F, N)

This function applies the procedure F to two arguments V and N where V is a vector of size N. The result is the value (if any) returned by the call of F. APTOVEC could be described in (illegal) BCPL as:

```
LET APTOVEC(F,N) = VALOF
$( LET V = VEC N // illegal because N not constant
   RESULTIS F(V,N)
$)
```

10.3 String handling

In BCPL strings are packed with more than one character per word so as to ecomonize space. The exact manner of packing is implementation dependent and is described in section 3.2.3.

It is however sometimes convenient in BCPL to operate on individual characters each in a separate cell. In such a representation the characters are stored in cells 1 to n of a

vector U and the length (i.e. the number of characters in the string) is stored in U!0. These two forms of a string are known as a 'packed string' (or simply string where no confusion will arise) and 'unpacked string'.

UNPACKSTRING(S, U)

This routine unpacks the characters from S into U!1 to U!N where N is the length of the string and sets U!0=N. The effect of UNPACKSTRING is undefined if S and U overlap.

I := PACKSTRING(U, S)

This function packs N and the characters U!1 to U!N into S where N = U!0. The result is the subscript of the highest element of S used. The effect of PACKSTRING is undefined if U and S overlap

CH := GETBYTE(V, N)

This function yields the Nth byte from vector V.

PUTBYTE(V, N, B)

This routine puts the byte B in the Nth byte position of vector V.

Acknowledgements

The criticisms and suggestions of Benedict Heal (Newcastle University) and Mike Jordan (Fendragon, Cambridge) are gratefully acknowledged together with discussions with many

BCPL users (too many to mention by name here).

References

Middleton M.D. (1977) A Proposed Definition of the Language BCPL, das RZ der Universitaet Regensburg, Regensburg

Richards, M. (1969) BCPL: A tool for Compiler Writing and Systems Programming, Proceedings of the Spring Joint Computer Conference, Vol 34.

Richards, M. (1973) The BCPL Programming Manual, The Computer Laboratory, Cambridge

Richards, M. and Whitby-Strevens, C. (1979) BCPL - The Language and its Compiler Cambridge University Press.

Appendix - Extensions

This appendix contains a list of recommended extension packets. None of these packets is mandatory, but if any one feature of a packet is implemented then the whole packet should as far as possible be implemented. Variations of packets should not be implemented. Before an implementor introduces any change in the language he should consider carefully if the proposed changes are really necessary as non-standard language elements can lead to severe problems with portability. Extensions to the standard procedures are much easier to cope with and these should always be the preferred method of extending the language.

A1 Character Constants

The following additional special representations are allowed:

special representation	represents
*C or *c	Carriage return
*B or *b	Backspace
*E or *e	Causes output of currently buffered characters on the stream without a carriage control character if this is possible.

N.B.

Each of the special representations will be mapped onto a distinct values in the internal code of the machine. The actual effect of the characters will however depend on the I/O medium used and the effect of a given character is not necessarily uniquely defined for every medium.

A2 Character Operator

The dyadic operator % has binding power just less than the operator ! and may occur in two contexts.

a) In an R-mode expression

   E1 % E2 is equivalent to the standard meaning of GETBYTE(E1, E2).

b) In an assignment mode expression

   E1 % E2 := E3 is equivalent to the standard meaning of PUTBYTE(E1, E2, E3).

A3 Field Selectors

Syntax

<expression> ::= SLCT [<expression> :] <expression>

SLCT and : have lower binding power than any other expression operator.

Semantics

SLCT K1:K2:K3

where K1, K2 and K3 are constant expressions. This operation defines a field within a vector. K1 is the size of the field in bits. K2 is the number of bits between the right most bit of the field and the right hand end of the cell containing it and K3 is the subscript of the element of the vector containing the field. If the size (K1) or the shift (K2) are

not explicitly specified zero is assumed. A size (K1) of zero implies that the field extends to the left most bit of the cell containing it.

The effect of the SLCT operator is to pack the three values into a BCPL word for use later by the operator OF. There are implementation dependent limits on the possible sizes of K1, K2 and K3. The exact method of packing field selectors is implementation dependent except that, subject to limits of size

SLCT 0:0:n = n

SLCT K1:K2:K3 is a constant expression.

Field selector application

Syntax

<dyop> ::= OF

Semantics

The field selector operator OF may occur in two contexts:

a) R-mode expression

K OF E

K is a constant expression which is interpreted as a field selector. E is any expression and is interpreted as the address of the zeroth cell of a vector. The effect of the OF operator is to extract the field defined by K from the vector defined by E and shift this so that the result is

right justified.

b) Assignment mode

K OF E1 := E2

K is interpreted as a field selector and the appropriate number of bits from the right hand end of the result of E2 are assigned to the specified field.

A4 Optional compilation

A directive of the form:-

$$tag

will set the value of $$tag to the complement of its previous value. Such a directive can occur anywhere in the program. A tag retains its value until the end of the program text unless complemented explicitly. All tags have initial value FALSE. The text enclosed between

$<tag and $>tag

will only be compiled if the value of $$tag is TRUE. The tag complementing directive is only executed if it is encounted in a region of text that is not being skipped.

A5 Compound Assignment

Syntax

```
<assignment>      ::= <expression list> <assop>
                      <expression list>

<assop>           ::= *:= | /:= | REM:= | +:= | -:= | &:= |
                      |:= | EQV:= | NEQV:=
```

Although <assop> is a basic symbol and as such may not have embedded layout characters it is nevertheless made up out of the two symbols <op> and := and all the synonyms for a given <op> are also allowed.

Semantics

There are two forms of compound assignment:

a) E1 <assop> E2

   where <assop> has the form <op>:=. This has the same effect as

   E1 := E1 <op> E2

b) Compound multiple assignment

   L1, L2, ... <op>:= E1, E2, ...

   This has the same effect as

   L1, L2, ... := L1 <op> E1, L2 <op> E2, ...

A6 Section and needs

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

SECTION "<name>"

when <name> is a module name acceptable to the linker. It defines the section name to be given to the generated object module. A new block-head-declaration is defined:

<needs dec> ::= NEEDS "<name>"

is a <block head declaration>, where <name> is also a name acceptable to the linker. This directive causes an external reference to be set up in the object module, so that the specified object module will be included automatically by the linker.

A7 Store Allocation

In addition to the normal BCPL stack there may be another area of dynamically allocated storage called the heap. This area may be used by the I/O system for maintaining buffers etc. but is also available to the user. The procedures for accessing the heap are as follows.

SIZE := MAXVEC()

This function returns the size of the largest vector that there is room for in the heap. Note that on some implementations the heap and stack will grow towards each other an this (as well as the fact that the heap

may be used for I/O buffers) must be allowed for when interpreting the result of MAXVEC. Further complications may arise in a real-time system.

V := GETVEC(SIZE)

This function allocates a contiguous area of store of SIZE + 1 words from the heap and returns its vector address (so that V!0 to V!SIZE are available). If this is not possible the value 0 is returned instead.

FREEVEC(V)

This routine returns a vector to the heap. V must be a vector allocated by GETVEC, and only entire vectors may be returned. The heap mechanism is defined to coalesce a returned vector with any contiguous free store, and to return store released to the common stack/heap pool.

SIZE := STACKSIZE()

This routine returns an approximation to the amount (in words) of stack space free. Note that on systems where the heap and stack grow towards each other the stack space available may be affected by calls of GETVEC, FREEVEC, and I/O routines (which may involve allocation or deallocation of buffers).

A8 Scaled arithmetic

RES := MULDIV(A,B,C)

This function calculates (A*B)/C, holding the intermediate product (A*B) as a double length integer. If the result does not fit in a normal integer the action of MULDIV is undefined. The remainder from the division is left in global variable RESULT2.

A9 Block I/O

Data can be accessed from normal input/output streams block by block. Depending on the implementation, a 'block' implies either a physical or logical block or more usually a logical record. The result of mixing block I/O and normal character I/O in a stream is undefined and it is recommended that the two types of access are not both used on the same stream. The size of a block is measured in units, normally bytes or words, which are implementation dependent.

RES := RDBLOCK(B, L)

This function reads the next 'block' from CIS into buffer B which is of size L. The result is the number of units actually read. A result of zero indicates that the 'block' was too large to fit into the buffer - in this case the first L units of the block are read into the buffer and the next call of RDBLOCK will attempt to

deliver the rest. A result of less than zero indicates an error or that the stream is exhausted - further details in RESULT2.

RES := WRBLOCK(B,L)

This routine writes a 'block' of size L from buffer B to the stream COS. A result of zero indicates a successful transfer. A non-zero result indicates an error condition - further details in RESULT2.

A10 Binary I/O

Binary I/O is inherently machine dependent and therefore very difficult to standardize. For this reason two alternative schemes are put forward. Whichever scheme is used, the effect of mixing binary and character I/O on a given stream is undefined.

a) Via stream definition

Streams are defined on creation as binary by means of suitable implementation dependent arguments to FINDINPUT. and FINDOUTPUT. Binary I/O then takes place via the routines RDCH and WRCH.

b) Via special binary I/O procedures

The following two routines may be used on normal streams to read/write in binary:

RDBIN() gives the next binary character on the CIS

WRBIN(x) writes x in binary to the COS.

Further Facilities for both schemes

In both schemes a means of specifying record sperators is necessary:

i) The standard header contains a manifest constant ENDRECORDCH (usually = -2) which is returned by RDBIN or RDCH on binary streams and indicates the end of a record.

ii) The routine ENDRECORD() causes an end of record to be output to the COS.

A11 Direct access I/O

The facilities for direct access of data offered by different operating systems vary considerably, and the primitives offered in BCPL are deliberately kept simple so as to encompass as many variations as possible. The term 'file' here means a logical collection of data such is a data set which is organized as a number of 'blocks' which may be accessed in a random order. The term 'block' means the physical or logical block or record which is transferred by the operating system. It may be possible on some operating systems to access some files both by means of direct access I/O and stream oriented I/O. This is however not mandatory. The size of a block is measured in units, normally bytes or words, which is implementation dependent.

D := FINDDIRECT(S,K)

This function is the direct access analogue for FINDINPUT and FINDOUTPUT. The 'file' identified by the string S is located and opened for reading and/or writing. The key K specifies the type of access. This should be defined by a manifest constant which can include the following values:

```
DA.IN  - read access only
DA.OUT - write access only
DA.IO  - read and write access
```

The result D is a 'file identifier' which is used in other direct access I/O procedures. A result of zero indicates that the file could not be opened - further information is obtainable in RESULT2.

RES := READDIRECT(D, N, B, L)

This function reads block N from the file with identifier D into buffer B which is of size L. The block identifier N will normally be an integer (block number) but could, in some implementations, be a string which is used as a key. The result is number of units actually read. A negative result indicates an error - further details in RESULT2.

RES := WRITEDIRECT(D, N, B, L)

This function writes a buffer B of size L to block N of the file with file identifier D. Block identifier N is

as in READBLOCK. A zero result indicates success and a negative result indicates an error - further details in RESULT2.

CLOSEDIRECT(D)

Closes file D.

A12 System Services

It is recommended that two procedures are provided to give access to the implementation dependent system services, one for the filing system and one for the other services.

RES := FILESYS(OP, A1, A2,...)

OP is a constant specifying a filing system service (such as 'deletefile' or 'renamefile'). A1, A2,... are appropriate arguments for the specified service and the result RES will usually indicate whether the operation was successful.

RES := OPSYS(OP, A1, A2,...)

OP is a constant specifying an operating system service and A1, A2,... are appropriate arguments for that service. RES is an implementation dependent result.

A13 Floating point

There are two possible schemes for a floating point package in BCPL: on implementations where the cell size is big enough to hold the machine representation of a floating point number the Floating Point Language Packet may be implemented and where this is not possible the Floating Point Procedure Packet may be implemented. In either case the Floating Point I/O Procedures should be implemented.

a) Floating Point Language Packet

Floating point constants

A floating point constant is syntactically and semantically equivalent to <number> and may have one of the forms

i.jEk

i.j

iEk

where i and j are unsigned integers and k is a (possibly signed) integer. The value is the machine representation of a floating point number.

Floating point operators

The floating point operations are formed by prefixing the corresponding integer operation with the character #. See section 3.2.2 for the complete list. The precedence of a

floating point operator is the same as its integer equivalent. In addition there are the monadic operators FIX and FLOAT, which have the same precedence as @, for conversion between integer and floating point representation. Note that the omission of # in a negative floating point constant such as #-1.2 will produce an unexpected value.

b) Floating Point Procedure Packet

A floating point number is represented by a pointer to a vector of FP.LEN cells where FP.LEN is a manifest constant declared in the standard library header. The arithemetic procedures are functions which operate on these representations and require 1 or 2 arguments as normal operands plus a further argument as destination. The result of the function is in this case the destination.

Arithmetic Functions

```
FPLUS(A,B,C)     C:=A#+B      resultis C
FMINUS(A,B,C)    C:=A#-B      resultis C
FNEG(A,B)        B:=#-A       resultis B
FMULT(A,B,C)     C:=A#*B      resultis C
FDIV(A,B,C)      C:=A#/B      resultis C
FABS(A,B)        B:=#ABS A    resultis B
```

Any number of the arguments in the above functions may be identical.

Other Functions

```
FFIX(A)                     resultis FIX A
FFLOAT(A,B)     B:=FLOAT A  resultis B
FCOMP(A,B)                  resultis (  0 A = B
                                     ( -1 A < B
                                     (  1 A > B
```

c) Floating Point I/O Procedures

WRITEFP(A,F,N)

This routine writes the FP number A to the stream COS in the form m.n where F is the total field width and N the number of places after the decimal point. If N is zero or cannot be represented in the given format it is output in the form m.nE[+|-]dd in a field width of F.

RES := READFP(A)

This function reads a FP number from stream CIS and converts it to its internal representation. For implementations with the FP Procedure Packet it stores this in the vector A and yields A as the result. For implementations with the FP Language Packet it returns the explicit internal representation of the FP number and does not use A.

A14 Time and Date

Information about time and date is provided as follows:

DATE(V)

This routine packs the date as a string into vector V and returns as result the pointer V.

TIMEOFDAY(V)

This routine packs the time of day as a string into vector V and returns a pointer to V.

T := TIME()

The result T is an integer representing the CPU time used in implementation dependent units. The unit is defined in the standard library header by TICKSPERSEC.

A15 External procedure

The facilities offered by linkers vary considerably from machine to machine. For this reason it is difficult to specify a mechanism that is generally applicable for interfacing BCPL programs with routines written in other languages. It is therefore recommended that the following scheme be adopted if possible, but if for some reason variations are necessary different system words should be used.

External declaration

Syntax

```
<blockhead declaration> ::= <ext dec>

<ext dec>          ::= EXTERNAL <ext defs>

<ext defs>         ::= $( <ext def> [; <ext def>] $)

<ext def>          ::= <name> : <string>
```

Semantics

The declaration

```
EXTERNAL $( N : S $)
```

where S is a string constant causes permanent allocation of a cell named N. This cell will be initialized to the value of the external reference S prior to execution of the program.

Calling non-BCPL procedures

Standard procedures are available for calling procedures defined in other languages. It is recommended that the names of these standard procedures are of the form CALLlang where lang specifies which language, e.g. CALLFORT.

CALLlang(RTN,N,A1,A2,...,An)

RTN is the procedure to be called which should be declared by an EXTERNAL declaration. N is the number of arguments passed to RTN and may be omitted in some implementations. A1,A2,...,An are the arguments passed

to RTN. The form of these arguments is implementation dependent. If RTN is a function, CALLlang yields the result.

INDEX of non-terminals