UNIVERSITY OF ESSEX
DEPARTMENT OF
COMPUTER SCIENCE
COMPUTING SERVICE

The BCPL Reference Manual

2nd Edition
P. Gardner
Revised January 1978

STEPHEN MURRELL

THE BCPL REFERENCE MANUAL

2nd Edition
September 1976
Revised January 1978

Pete Gardner
University of Essex

This manual describes the language as implemented in compiler version 3F.

CONTENTS

1.0 BRIEF DESCRIPTION OF BCPL

1.1 HISTORY OF ESSEX BCPL

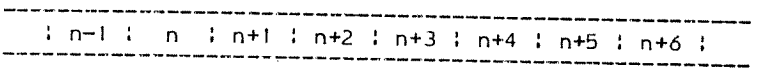
BCPL is a (recursive) programming language, originally developed and implemented by Martin Richards at MIT project MAC, for general non-numerical problems and systems work. It employs sensible and efficient control constructs which largely eliminate the need for "goto"s, and achieves both simplicity and generality by providing a single data type - the word. A word may function as several different conceptual types, although of course it is not possible to check that it is consistently so used.

Richards' original compiler produced object code (OCODE) for an idealised stack machine and was itself available in OCODE. Hence to transport the compiler to another machine, it was only necessary to write a code generator to translate OCODE to the required machine code. By this method Richards took BCPL to Cambridge (England), from which version the original Essex ICL 1900 BCPL was developed via a code generator written by Bernard Sufrin in 1969. It was transported onto the PDP-10 in 1970 via a code generator written in BCPL by Brian O'Mahoney and David Eyres, with help from Bernard Sufrin.

By 1973 BCPL had developed into the major software language in use by the Department of Computer Science for both research (e.g. in AI) and teaching (e.g. Compiler writing), and so with the intention of improving the compiler, both in performance and by adding new language features, a new compiler was developed by Pete Gardner for release in 1974. The performance, as measured by the kilo-core second load on the system, has been improved by a factor 3 or 4; the new features included are documented in this manual. The main contributory factor was the removal of the intermediate OCODE stage which required in effect two compilations, one from BCPL to OCODE and one from OCODE to machine code.

1.2 THE BCPL MACHINE

BCPL is based on a simple storage model which consists of a set of consecutively addressed cells arranged thus:-



All cells are of uniform size, and hold a binary bit pattern called a value (usually between 16 and 60 bits, depending on the word size of the machine; on the PDP-10 the word size is 36 bits).

A value is the only primitive data type in BCPL, but it is used to represent an integer, character, truth value, address and (of course) a bit pattern. Used as an address a value can refer to (point to) a vector, a character string, a table, a function or a routine.

To facilitate the representation of all these conceptual data types a large set of useful operators have been provided. For example + - * / assume that their operands represent integers and the result is consistent with this representation; ROTL and ROTR assume their relevant operand is a bit pattern; A!I assumes that A points to a group of consecutively addressed cells (a vector) and selects the Ith (counting from Ø). However no type checking is performed by these operations and it is possible, for example, to add an integer to a character (e.g. 4+'Ø' equivalent to '4' on the PDP-1Ø). Such operations are sometimes desirable although the user should guard against losing machine independence by using too many tricks of this kind. For example, any assumption about the precise number of bits in a word could be dangerous.

1.3 INTRODUCTION TO DATA TYPES

1.3.1 VALUES

Conceptual values can be represented in the written program by canonical symbols, thus:-

Type	Example	Machine dependent bit representation on PDP-1Ø (36 bit words)
OCTAL	#3777	ØØØØØØØØ3777
INTEGER	1Ø3	ØØØØØØØØØ147
REAL	1.Ø	172Ø4ØØØØØØØ
CHARACTER	'Q'	ØØØØØØØØ121
BOOLEAN	TRUE	777777777777

1.3.2 VARIABLES

A variable is a name which is associated with a storage cell - by means of a declaration, e.g.

LET A, B = 1, 2;

declares two variables (A and B) and initialises them to

(the bit pattern equivalent of) the integers 1 and 2.

1.3.3 MANIFEST CONSTANTS

A manifest constant is the direct association at compile time of a name with a value, e.g.

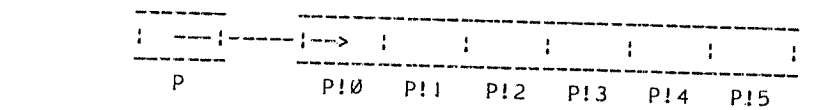
```
MANIFEST £( TEN = 10; S.COMMA = #54 £)
```

1.3.4 VECTORS

A vector is the only form of data structure which can be formally declared, e.g.

```
LET P = VEC 5
```

sets up the structure:-



P!<E> where <E> is an expression refers to the <E>th cell of the vector. There is no index bound checking.

1.3.5 OTHER DATA STRUCTURES

Other data structures can be modelled using 'pointer' values to construct arrays, lists, trees, etc. Vectors can also be dynamically acquired from free storage.

1.3.6 SCOPE AND EXTENT

LET data is allocated in the main stack (as in Algol60). It is also possible to declare STATIC data which exists throughout the duration of the program. Programs are block structured (as in Algol60) each block opening a new level of nomenclature for variables. Variables can also be declared EXTERNAL (as in PL/1) to enable communication between separately compiled segments of a program. (The original GLOBAL vector, which served the same purpose; is still available).

1.3.7 FUNCTIONS AND ROUTINES

Parameters are called by value. Names of variables are communicated by using their addresses as value parameters, this can be done by using @ and ! operators. The value of the expression @A is the address of the variable A. If this is substituted for formal parameter P then !P in the body of the called routine (or function) refers to the original variable A. Routines and functions may be called with a variable number of parameters. Return from a routine is by RETURN or the textual end. Return from a function is upon completion of the evaluation of the expression which the function denotes.

The declaration of a function or routine creates a STATIC data item which points to the body of the function or routine. This data item can be manipulated in the same way as any other STATIC data item and obeys the same SCOPE and EXTENT restrictions.

1.4 FURTHER READING

The BCPL Programming Manual, by Martin Richards. University of Cambridge, Computing Laboratory, Corn Exchange Street, Cambridge. CB2 3QG. England.

INTCODE - An Interpretive Machine Code for BCPL, by Martin Richards. University of Cambridge, Computing Laboratory, Corn Exchange Street, Cambridge. CB2 3QG. England.

The BCPL User Guide, by Pete Gardner et al. Department of Computer Science, Computing Service, University of Essex, Wivenhoe Park, Colchester, Essex. CO4 3SQ. England.

2.0 EXPRESSIONS

2.1 SYNTAX OF EXPRESSIONS

```
<E>       ::= <P> <<binop> <P>>* ¦ <E> -> <E>, <E> ¦
              SELECTOR <E>:<E>:<E> ¦ BYTE <E>:<E> ¦
              TABLE <E-list>
<P>       ::= <name> ¦ <integer> ¦ <real> ¦ <octal> ¦
              <charconst> ¦ <truthvalue> ¦ £constant ¦
              NIL ¦ <string> ¦ (<E>) ¦ VALOF <C> ¦
              <E>() ¦ <E>(<E-list>) ¦ <unop><E> ¦
              <literal>
<binop>   ::= %<name> ¦ ! ¦ !! ¦ && ¦ * ¦ / ¦ REM ¦ #* ¦
              #/ ¦ ** ¦ + ¦ - ¦ #+ ¦ #- ¦ << ¦ >> ¦
              ROTL ¦ ROTR ¦ ALSHIFT ¦ ARSHIFT ¦ LS ¦
              LE ¦ EQ ¦ GE ¦ GR ¦ NE ¦ #LS ¦ #LE ¦
              #EQ ¦ #GE ¦ #GR ¦ #NE ¦ /\ ¦ \/ ¦ EQV ¦
              NEQV ¦ BITAND ¦ BITOR
<unop>    ::= NOT ¦ @ ¦ ! ¦ + ¦ - ¦ ABS ¦ #+ ¦ #- ¦
              #ABS ¦ FIX ¦ FLOAT
<E-list>  ::= <E> <,<E>>*
```

Alternative representation of operators are listed in Appendix B.

2.2 SEMANTICS OF EXPRESSIONS

Expressions are used to calculate values, which on the PDP-10 are bit-patterns of length 36. Unlike languages such as Fortran and Algol, there are no explicit notions of type in BCPL. Values are used by the programmer to model objects of many different kinds, (e.g. truth values, strings, vectors, data structures, bit-patterns) and there are a large number of basic operation on values which have been provided in order to model the transformation of these objects.

The simplest kinds of expression are names (see 2.4) and constants (see 2.5).

2.3 PRIORITY OF OPERATORS

In an expression which contains more than one operator, some scheme must exist to define the priority of evaluations of one operator over another. The table below gives the priority of operators in BCPL.

The top of the list is the highest priority.

1. name, constant, bracketed expression, VALOF
2. function application
3. monadic ops. NOT @ ! + - ABS #+ #- #ABS FIX FLOAT
4. %name
5. !
6. :: &&
7. * / REM #* #/ **
8. + - #+ #-
9. << >> ROTL ROTR ALSHIFT ARSHIFT
10. LS LE EQ GE GR NE #LS #LE #EQ #EQ #GE #GR #NE
11. /\ BITAND
12. \/ BITOR
13. EQV NEQV
14. conditional expression
15. TABLE SELECTOR BYTE

Operators of the same priority level are evaluated from left to right (i.e. the left most operation performed first), except for :: and && which are evaluated from right to left. The relational operators are considered to be of equal priority.

Examples.

A \/ B + C * D is evaluated as	A \/ (B + (C * D))
A * B * C	(A * B) * C
A :: B :: C	A :: (B :: C)
A LE B LE C	A LE B LE C

The order of evaluation of the operands of any operator (> priority 3) is undefined, except for those of priority 10, and /\ and \/, whose left operand is evaluated before its right operand.

The remainder of this chapter deals with each of the elements of expressions in turn.

2.4 NAMES

Names are used to identify objects, these objects may either be a storage cell (i.e. a variable) or a constant (i.e. a MANIFEST - see 3.6). All objects referred to by a name are full word size.

The name itself may consist of from 1 to 127 alphanumeric characters or period; the first character must be alphabetic.

Examples.

TOTAL i J Version25 A.MUCH.LONGER.NAME

2.5 CONSTANTS

2.5.1 SEMANTICS OF CONSTANTS

All the following forms are shorthand ways of introducing a bit pattern into the program.

2.5.2 INTEGER CONSTANTS

Unsigned decimal no. in the range Ø to 3435938367, it may (like all other constants) be negated by a preceding minus sign. If an integer greater than 3435938367 is encountered by the PDP 1Ø compiler, it uses only the last 36 bits of the value and gives no error message.

Examples.

345 8388608

2.5.3 REAL CONSTANTS

These consist of an integral part, a fractional part and an optional exponent part. The integral and fractional parts both consist of any number (>= 1) of digits separated by a decimal point. The accuracy of the fractional part on the PDP-1Ø is 8+ digits. The exponent part consists of an E followed by an optionally signed decimal integer in the range -38 to +38. If an exponent out of this range is given, underflow or overflow will occur with no warning given by the compiler.

Examples.

Ø.Ø 3.14159 1Ø.ØE6 Ø.17E-13

2.5.4 OCTAL CONSTANTS

These consists of any number of octal digits preceded by a #. The last 36 bits of the octal number are used by the compiler.

Examples.

#77 #77ØØØØ77 #4323717

2.5.5 CHARACTER CONSTANTS

A character constant is a sequence of BCPL characters between single quote characters - '. (See Appendix C). Up to the first 5 characters are packed (right aligned) in the word.

Examples.

'A'	equivalent to #101
'*C' (carriage return)	equivalent to #15
'FOO'	equivalent to #4323717
IF 'A' LE CH LE 'Z' DO	...

2.5.6 TRUTH VALUE

The logical representations of true and false are represented by a word with all bits set and a word with all bits clear respectively. The reserved words TRUE and FALSE are used to represent these values.

2.5.7 '£' CONSTANTS

These are entirely machine dependent, a brief description of them appears below.

£DAY is the value of the PDP-10 DATE CALLI UUO at the time of compilation.

£TIME is the value of the PDP-10 MSTIME CALLI UUO at the time of compilation.

£TRACE is the value TRUE or FALSE according to whether the program is being compiled with postmortem code enabled or not. (See descriptions of /F and /O switches in Appendix E).

£SIXBIT <string> is the PDP-10 sixbit representation of the first 6 characters in the BCPL string <string>.

£ASCII <string> is the PDP-10 left aligned ASCII representation of the first 5 characters in the BCPL string <string>.

£ASCIZ <string> is the PDP-10 left aligned ASCIZ representation of the first 4 characters in the BCPL string <string>.

£STRING <string> is the value of the first word of the BCPL string <string>.

£AZ <string> is the PDP-10 multiword ASCIZ representation

of the BCPL string <string>. It is of type <literal>. Beware of the length limit on all BCPL strings (127 characters on the PDP-10).

£EXP and £XWD have the value zero, see Appendix D for examples of use.

£opcode is the 9 bit value of the standard PDP-10 mnemonic for all opcodes in the range #40 to #677. For all the extended ITCALL UUOS, the extended CALLI UUOS and the JRST and JFCL extensions, it is a 36 bit value, the form of which is described in Appendix D

2.6 NIL

Is an expression which can be used as the initialisation expression in STATIC or LET declarations. It is used as a "don't care" value. In fact, it eliminates any initialisation of the STATIC or LET variable.

Examples.

```
STATIC £( X = NIL £)

LET A, B, C = NIL, TRUE, NIL
```

A and C are defined but uninitialised, B is defined and initialised to TRUE.

2.7 STRINGS

BCPL strings consist of up to 127 characters bracketed by the double quote character ". The full set of ascii characters is permitted in the string and the way of representing them can be found in appendix C. Strings are packed 5 characters to a word, the first character position in the string contains the character count and any character positions left over in the last word are zero filled. The value used in the BCPL expression is the address where the string is stored. Strings may be continued on more than one source line if necessary by ending the line to be continued with a '*', and restarting the string on a subsequent line with another '*'. The '*' characters and all layout characters (spaces, newlines etc.) are ignored.

Examples.

```
"A"   "THE SUM IS*T:N*C*L"
"A VERY SIMPLE STRING"
"A MULTI-LINE*
    * STRING"
```

2.8 BRACKETED EXPRESSIONS

Brackets in bracketed expressions serve no other purpose than that of overriding the normal priority of evaluation of an expression.

Example.

A - B + C - D is evaluated as [[A - B] + C] - D
(where [] indicate the priority of evaluation)

but

A - (B + C) - D is evaluated as [A - [B + C]] - D
which may yield a very different result.

2.9 VALOF

VALOF <C>

The value of a VALOF expression is determined by executing the command <C> (usually a section or a block) until a RESULTIS command (see 4.22) is executed, which causes execution of <C> to cease. The value of the VALOF expression is the value of the <E> expression in the RESULTIS command.

Example.

```
FIRSTZERO := VALOF £(FZ // FINDS FIRST ZERO WORD IN VEC V
                        LET I = 0
                        UNTIL V!I = 0 DO I := I + 1
                        RESULTIS I
                    £(FZ
```

2.10 FUNCTION APPLICATION

<E>() or <E>(<E1>, ... <En>)

The function application is evaluated by first evaluating the expressions <E1>, <En> and assigning the value of <E1> to <En> to the first n formal parameters of the function <E>. <E> is then entered. The result of the function application is the value of the expression in the function definition.

Examples.

```
CH := INCH()
LET SINH(X) = (E(X) #- E(#-X)) #/ 2.0
X := (REAL -> RDF, RDNO)(INPUT)
```

2.11 UNARY OPERATORS

2.11.1 NOT

This treats its operand as a binary bit pattern and gives the logical negation of the operand.

Examples.

```
NOT  7        is equivalent to  #777777777770
NOT #12345    is equivalent to  #777777765432
```

2.11.2 @

The address of an expression which represents a storage cell may be obtained via use of the @ operator. Such expressions are ! expressions and non MANIFEST names.

Examples.

```
@A       is the address of storage cell A
@(V!6)   is equivalent to V + 6
@!E      is equivalent to E
```

2.11.3 MONADIC !

The value of a monadic ! expression is the value of the storage cell whose address is the operand of the !. Thus @!E = !@E = E, (providing E is an expression of the class described in 2.11.2).

Examples.

```
!X := Y    Stores the value of Y into the storage cell
           whose address is the value of X.
P := !P    Stores the value of the cell whose address
           is the value of P, as the new value of P.
```

2.11.4 MONADIC PLUS AND MINUS

Monadic plus is in effect a null operation, it is provided for completeness. Monadic minus provides the negative of its operand. The # versions are used for real operands.

2.11.5 ABS AND #ABS

These provide the absolute (i.e. magnitude) of their operand. The # version is used for real operands.

Examples.

```
ABS -5    is equivalent to 5
ABS 73    is equivalent to 73
```

2.11.6 FIX

This gives the integral value of its real operand, rounded to the nearest integer. (Rule for rounding is as for ALGOL 60).

Examples.

```
FIX 1.432  is equivalent to 1
FIX 73.78  is equivalent to 74
```

2.11.7 FLOAT

This gives the real value of its integer operand.

2.12 INFIX FUNCTION APPLICATION

This provides a way of intoducing a binary operator which is defined as a BCPL function and then invoking it as a function. The form is %<name>.

Example.

```
LET ADD(A,B) = A + B  // function definition
X := Y %ADD Z         // same as Y + Z
```

2.13 VECTOR APPLICATION

Provides a way of selecting an element of a vector. A vector is any set consecutive storage cells, one way of introducing such a set is described under vector declaration (see 3.4.3).

The basic form of a vector application is E1 ! E2.

Notes:-

E1 ! E2 is the same as ! (E1 + E2)

```
E1 ! 0      is the same as ! E1
E1 ! E2     is the same as E2 ! E1
@(E1 ! E2)  is the address of cell E2 in vector E1.
@(E1 ! E2)  is the same as E1 + E2
```

!E is interpreted as monadic (See 2.11.3).

2.14 SELECTORS AND BYTES

2.14.1 SELECTOR AND BYTE CONSTRUCTION

SELECTOR and BYTE are used to construct a special kind of pointer, used to extract a portion of a given word (see 2.14.2 and 2.14.3).

[On the PDP-10 these are PDP 10 byte pointers
(BYTE 6:30) is the bit position field of a byte pointer.
(BYTE 6:24) is the byte size field of a byte pointer.
(BYTE 18:0) is the address field (SELECTOR only).]

A byte size of -1 in a SELECTOR or a BYTE construction gives a full word size byte.

SELECTOR E1:E2:E3

constructs a pointer to extract a byte of size E1 bits, E2 bits from the right hand end of the word whose address is given by adding E3 to the value of the right hand operand of any selector application in which it is used.

BYTE E1:E2

constructs a pointer as above, but without the address field. BYTE operators act directly on the word indicated as the object of the BYTE application.

Note.

(SELECTOR E1:E2:0)::@E3 is equivalent to (BYTE E1:E2)&&E3

2.14.2 SELECTOR APPLICATION

SELECTOR application is the process of applying a SELECTOR (constructed as in 2.14.1), to perform a byte extraction on the given data structure. SELECTOR application is denoted by ::.

Example.

```
(SELECTOR 7:0:3)::V // refers to a byte 7 bits wide
                    // in the least significant 7 bits
                    // of the third word of vector V
```

2.14.3 BYTE APPLICATION

BYTE application is the process of applying a BYTE pointer (constructed as in 2.14.1), to extract a byte from a given word. BYTE application is denoted by &&.

Example.

```
(BYTE 18:18)&&X      // left half of X.
```

2.15 ARITHMETIC OPERATORS

2.15.1 HIGH PRIORITY OPERATORS

*	is integer multiply.	e.g. 2 * 3 = 6
/	is integer divide.	e.g. 7 / 5 = 1
REM	is integer remainder.	e.g. 7 REM 5 = 2
#*	is rounded real multiply.	e.g. 2.5 #* 3.25 = 8.125
#/	is rounded real divide.	e.g. 8.125 #/ 2.5 = 3.25
**	is real raised to integer power.	e.g. 3.5 ** 2 = 12.25

The integer (real) operators above interpret the values of their operands as integers (reals) and yield integer (real) results.

2.15.2 LOW PRIORITY OPERATORS

+	is integer add.	e.g. 2 + 3 = 5
-	is integer subract.	e.g. 2 - 3 = -1
#+	is rounded real add.	e.g. 2.5 #+ 3.25 = 5.75
#-	is rounded real subtract.	e.g. 2.5 #- 3.25 = #-0.75

The integer (real) operators above interpret the values of their operands as integers (reals) and yield (real) results.

2.16 SHIFT OPERATORS

<<	is logical left shift. Zero bits shifted in on the right, bits lost on the left.
>>	is logical right shift. Zero bits shifted in on the left, bits lost on the right.
ROTL	is left rotate. Bits shifted off the left are inserted on the right.
ROTR	is right rotate. Bits shifted off the right are inserted on the left.
ALSHIFT	is arithmetic left shift. Zero bits shifted in on the right, bits lost on the left.
ARSHIFT	is arithmetic right shift. Sign bit shifted in on the left, bits lost on the right.

2.17 RELATIONAL OPERATORS

These give the value true or false according to whether the condition is satisfied. The # versions are for real value relations. Relations have equal left to right priority, but the operands are evaluated in left to right order.

```
E1 relop1 E2 relop2 E3
is equivalent to
E1 relop1 E2 /\ E2 relop2 E3
except that E2 is only evaluated once.
```

LS less than	LE less than or equal to
#LS less than	#LE less than or equal to
EQ equal to	GE greater than or equal to
#EQ equal to	#GE greater than or equal to
GR greater than	NE not equal to
#GR greater than	#NE not equal to

2.18 LOGICAL OPERATORS

These treat their operands as binary bit patterns.

/\	specifies the AND operation.
\/	specifies the inclusive OR operation.
EQV	specifies the equivalence operation.
NEQV	specifies the non equivalence operation.
BITAND	specifies a bitwise AND operation.
BITOR	specifies a bitwise inclusive OR operation.

Examples.

```
3 /\ 5 = 1                      3 \/ 5 = 7
3 EQV 6 = #777777777772         3 NEQV 6 = 5
3 BITAND 5 = 1                  3 BITOR 5 = 7
```

Note.

The difference between BITAND and /\ (and similarly BITOR and \/), is that /\ and \/ are only supposed to be used on truth values, whereas BITAND and BITOR are used on any bit pattern. The effect may best be demonstrated by the following example:-

```
LET A, B = 1, 2
IF A /\ B THEN       // The body will be executed.
IF A BITAND B THEN   // The body will not be executed.
```

In the first IF, A is interpreted as TRUE (i.e. not FALSE) also B is interpreted as TRUE (i.e. not FALSE) and so the body of the IF will be executed. In the second IF, the operation A BITAND B will yield Ø (FALSE) and so the body of the IF will not be executed. This difference of

interpretation only applies in the interpretation of conditional expressions and conditional commands. (See 2.19 and section 4).

2.19 CONDITIONAL EXPRESSIONS

<E1> -> <E2>, <E3>

The expression <E1> is evaluated and if it yields the result not FALSE then the expression <E2> is evaluated as the result of the conditional, if <E1> however yields FALSE then the expression <E3> is evaluated as the result of the conditional.

Example.

```
A > B -> A, B // gives as result the larger of A and B.
```

2.20 TABLES

TABLE <L-list>

The value of a TABLE is a pointer to a STATIC list which contains a set of initial values. It can be used as a vector. All the elements of the table must be load time constants or compile time constants (see 2.21).

Examples.

```
TABLE 1, "ONE", @ONE, 2, "TWO", @TWO, 3, "THREE", @THREE
DAY!(TABLE "MON","TUE","WED","THU","FRI","SAT","SUN")
```

2.21 CONSTANT EXPRESSIONS

2.21.1 SEMANTICS

The BCPL compiler attempts to reduce all expressions, or parts of expressions, by evaluating the constant part. A compile time constant is one in which the expression can be reduced totally to a single constant. There are some places (e.g. vector declarations) where this is required, and others where the compiler can shorten the code it produces if it finds such a constant (see 5.7).

2.21.2 COMPILE TIME CONSTANTS

Consist of any expression containing only:-

<name>	(manifest only)
<integer>	
<real>	
<octal>	
<charconst>	
<truthvalue>	
£constant	
<unop>	(except @ !)
<binop>	(except %name ! ::)
<E1> -> <E2>, <E3>	(providing E1 is a compile time constant and that either E1 yeilds not FALSE and E2 is a compile time constant or E1 yields FALSE and E3 is a compile time constant).

2.21.3 LOAD TIME CONSTANTS

Consist of any expression containing only:

Any compile time constant.

<string>	
<literal>	
TABLE <E-list>	
@ <name>	(which is of STATIC data type)
<E1> -> <E2>, <E3>	(providing E1 is a compile time constant and that either E1 yields not FALSE and E2 is a load time constant or E1 yields FALSE and E3 is a load time constant).
VEC <K>	which gives a pointer to a STATIC uninitialised vector of K words long.

3.0 DECLARATIONS

3.1 SYNTAX OF DECLARATIONS

```
<D>      ::=   LET <decl> <AND <decl>>* ! STATIC <body> !
                MANIFEST <body> ! EXTERNAL <E>** <body> !
                GLOBAL <body>
<decl>   ::=   <name>(<name list>**) < = <E> ! BE <C>> !
                <name> = <E> ! <name list> = <E-list>
<body>   ::=   £( <def> <! <def>> £)
<def>    ::=   <name> = <E> ! <name> ! <E>
```

3.2 SEMANTICS OF DECLARATIONS (SCOPE AND EXTENT)

Declarations are used to associate names with values, these values are either constant (MANIFEST) or the addresses of storage cells.

The SCOPE of a name N is the textual region of the program in which that name can be used to refer to the same data item. The EXTENT of an item is its period of existence. There are two underlying types of data item, DYNAMIC and STATIC. DYNAMIC data items are allocated space on a STACK when they come into existence. The space which they are allocated becomes available for reuse when they cease to exist. Thus if a function or routine is called recursively, there may be more than one instance of a dynamic data item on the stack. STATIC data items exist for the whole execution of the program, there is only one copy of a static item.

A DYNAMIC data item may only be referred to within the body of the routine or function in which it is declared. It may not be referred to in any other function or routine.

Two data items with the same name may not be declared (either explicitly or implicitly) in the same block or formal parameter list.

Example.

```
LET A, B = 1, 2
LET F(X) = A * X + B
```

is illegal, but may be rewritten as

```
STATIC £( A=1; B=2 £)
LET F(X) = A * X + B
```

3.3 STATIC DECLARATIONS

(STATIC, LABELS, ROUTINES AND FUNCTIONS)

3.3.1 STATIC

STATIC £(<name> <:¦=> <L> < ; <name> <:¦=> <L>>* £)

This declares static data items whose scope is the rest of the block body in which the declaration appears, and whose extent is the entire program execution. It declares the <name>s to be associated with the data items which are initialised at start of execution of the program to the value(s) (<L>) given. The value(s) may be any load time constant or compile time constant (see 2.21). Exceptions:- NIL and VEC give uninitialised static areas.

Examples.

```
STATIC £( X = ø £)
STATIC £( ENDROUTINE = @ENDR; MESSAGE = "TERMINATOR" £)
STATIC £( IOVEC = VEC 65ø; INPUT = NIL; OUTPUT = NIL £)
```

3.3.2 LABEL

<name>: <C>

This declares a static data item whose scope is the command sequence of the BLOCK in which the label appears and whose extent is as for all static data. It declares the <name> to be associated with a data item which is initialised at start of execution of the program to the address of the point of code at which the label appears. Because a label is a static data item, it may be used in any expression, or be assigned to. If its value is altered during execution of the program, then any further use of the label (even in GOTOs) will refer to the new value.

Example.

```
GOTO !((TABLE @ADD, @SUB, @MUL, @DIV)!OP)
ADD: RES := A + B ; GOTO END
SUB: RES := A - B ; GOTO END
MUL: RES := A * B ; GOTO END
DIV: RES := A / B
END:
```

3.3.3 ROUTINES

```
LET <name>(<name list>) BE <C>
LET <name>() BE <C>
```

These define a routine, and an associated static data item, whose scope is the declaration itself and the rest of the block body in which the declaration appears, and whose extent is as for all static data. It defines the <name> to be associated with the data item which is initialised at the start execution to point to the code of the routine. Because a routine is thus a static data item it can be used in any expression and assigned to. If the value of the data item is altered during execution of the program, then any further invokation of the routine will refer to the new value.

A routine does not yield a value and hence should not be used in expressions, but only as a command.

Example.

```
LET OUTPN (N) BE
£( IF N GR 9 DO OUTPN (N/10)
   OUTCH (N REM 10 + '0')
£)
```

3.3.4 FUNCTIONS

```
LET <name>() = <E>
LET <name>(<namelist>) = <E>
```

Functions have the same semantics as ROUTINES (see 3.3.3); except that functions yield a value and thus may appear in expressions.

Examples.

```
LET FACTORIAL(N) = N = 0 -> 1, N * FACTORIAL (N-1)

LET ALPHA(CH) = VALOF
£( IF 'A' LE CH LE 'Z' DO RESULTIS TRUE
   WARN(CH)
   RESULTIS FALSE
£)
```

3.4 DYNAMIC DECLARATIONS (FORMAL PARAMETERS AND LET)

3.4.1 THE STACK

The STACK is used for all DYNAMIC data items. Space is allocated to the formal parameters of a routine or function when the routine or function is called, and subsequently space is allocated to local variables as each block in turn is entered. The space used for a) formal parameters and b) local variables is returned automatically to the stack for reuse when a) the function or routine is terminated or b) the block is exited. All dynamic data items thus have the SCOPE of the declaration itself and all enclosed blocks, their EXTENT is the duration of execution of the block (or in the case of formal parameters - the routine or function).

3.4.2 FORMAL PARAMETERS

The appearance of a formal parameter in the namelist in a routine or function definition (see 3.3.3 and 3.3.4) defines a dynamic data item on the stack (see 3.4.1). Formal parameters may or may not be initialised according to whether or not a corresponding actual parameter appears in the call of the routine or function. Formal parameters are allocated consecutive cells on the stack, thus the address of the second formal parameter is the address of the formal parameter + 1, etc. All parameters in BCPL are passed by value, thus changing the value of a formal parameter will not change the value of the corresponding actual parameter. "Call by reference" can be achieved by using the @ operator in the call of a routine or function and the ! operator when referring to the parameter in the body of the routine or function.

Examples. See 3.3.3 and 3.3.4

3.4.3 (DYNAMIC) LET

LET <namelist> = <E-list>
LET <namelist> = VEC <K> <, VEC <K>>*

These forms of the LET declaration are used to define dynamic data items on the stack (see 3.4.1). The simple (first) form is used to define individual data items. When execution reaches the declaration on entry to the enclosing block, then the data item comes into existence and is initialised to the value <E> (unless <E> is NIL, in which case the data item is uninitialised).

The vector (second) form is similar to the simple form,

except that the data item is initialised to the address of a vector (set of consecutively addressed cells) on the stack, whose subscript range is from zero to <K> (a compile time constant - see 2.21.2). The vector itself has the same EXTENT as the dynamic declaration in which it is defined. Simple LETs and vector LETs can be mixed together in ESSEX BCPL.

Examples.

```
LET IOV = VEC 650
LET IOVP, IOVT = IOV, IOV + 650
LET V, P, X = VEC 3, NIL, 0
```

3.4.4 WHERE

<C> WHERE <decl>

WHERE is used to associate a declaration with a particular command, it thus makes the declaration local to the command. The SCOPE of a WHERE declaration is the command which the WHERE follows, the EXTENT is as for other dynamic declarations. A WHERE can introduce any declaration normally introduced by LET. (See 3.3.3, 3.3.4 and 3.4.3).

Examples.

```
OUTS(DAY!TAB) WHERE TAB = TABLE
  "Mon","Tue","Wed","Thu","Fri","Sat","Sun"

N := F(X) WHERE F(N) = N = 0 -> 1, N * F(N-1)
```

3.5 SIMULTANEOUS DECLARATIONS (AND)

AND joins together any declarations normally introduced by LET or WHERE, and is used to common the scope of the declarations, usually so that the body of one or more of the declarations may contain references to the name(s) of the data item(s) declared in the other declaration(s).

Examples.

```
LET A, AV = @BV, VEC 6
AND B, BV = @AV, VEC 6
LET F(X) BE G(X + 1)
AND G(X) BE F(X - 1)
```

3.6 MANIFEST DECLARATIONS

MANIFEST £(<name> = <K> <; <name> = <K>>* £)

This declares each <name> as a manifest constant whose value is the expression <K> which must be a compile time constant (see 2.21.2). Wherever the <name> appears, it is used as if it were a numeric constant. Manifest constants serve three very useful functions. Firstly they enable the programmer to give mnemonic names to constants, which aid readability of the program. Secondly they can be used for constants which may change in further compilations, such as vector sizes. Thirdly, they can be used to write machine independent programs that depend on machine dependent factors, such as the number of characters held in a word etc. The scope of a MANIFEST declaration is the rest of the block body in which the MANIFEST is declared.

Examples.

```
MANIFEST £( WORKSIZE = 1024; BELLCHARACTER = #7 £)
LET V = VEC WORKSIZE
MANIFEST £( MAXSTRLENGTH = 127; CHARSPERWORD = 5 £)
LET PACKVEC = VEC MAXSTRLENGTH/CHARSPERWORD + 1
```

3.7 INTER-FILE AND LIBRARY COMMUNICATION (EXTERNAL AND GLOBAL)

3.7.1 USE OF EXTERNAL AND GLOBAL

EXTERNAL and GLOBAL provide a mechanism by which static data items within a file can be referenced from within another file. This serves two purposes. Firstly, the programmer can split his complete program into small manageable pieces (individual files), which can be seperately compiled and then loaded together by the system loader and executed. This means that if a "bug" is detected, and one piece (file), requires alteration, then only that piece (file) need be recompiled. This results in a considerable saving in machine time. Secondly, a standard library of pre-compiled BCPL files containing routines and functions can be made available to all users. Such a library, containing a complete Input/Output system and many other useful routines, exists for ESSEX BCPL. It (and the way of communicating with it via a standard set of EXTERNAL declarations) is described in section 6 of this manual.

3.7.2 THE EXTERNAL DECLARATION

```
EXTERNAL <prefix>** £( <definition> < ; <definition>>* £)
<prefix>      ::= <name> ! <string>
<definition>  ::= <inname> ! <inname> = <extname> !
                  <inname> : <extname>
<extname>     ::= <name> ! <string>
<inname>      ::= <name>
```

The internal name <inname> is the name by which the item is known WITHIN the scope of the EXTERNAL declaration (which is the remainder of the block body in which the declaration appears).

The external name <extname> is the name by which the item is known OUTSIDE the scope of the EXTERNAL declaration. If the <extname> part is missing, <inname> is used. On the PDP-10, because of a system loader restriction, only the first 6 characters of the external name are used for inter-file communication.

The <prefix> part, if it appears, is prefixed to the external name. It is intended for use mainly with library names in order to help make the external names within the library unique and protected.

EXTERNALS are used by putting the same EXTERNAL declaration at the top of each file in which the data is referenced (including the file in which the item is defined). All references to the internal name of a data item within the scope of the EXTERNAL declaration cause referance to be made to its external name. A static declaration (either implicit - labels, functions, routines - or explicit - STATIC) within the scope of the EXTERNAL declaration defines the static as normal and makes the external name available to other files.

Examples.

```
EXTERNAL £( READPHASE £)
EXTERNAL "%" £( OUTNO ; WRITENO : WRNO £)
```

3.7.3 THE GLOBAL DECLARATION

```
GLOBAL £( <name> : <K> < ; <name> : <K>>* £)
```

GLOBAL has been retained in ESSEX BCPL for historical reasons, and because programs obtained from other sources may rely on them. EXTERNAL provides a much more powerful mechanism for interfile communication.

Now GLOBAL items are treated as if they were EXTERNAL and the rules for EXTERNAL now apply to GLOBAL. GLOBAL declarations are processed as follows:-

For positive global offsets.
GLOBAL £(name:n £) is equivalent to
EXTERNAL ".G" £(name:"n" £)

For negative global offsets.
GLOBAL £(name:n £) is equivalent to
EXTERNAL ".N" £(name:"n" £)

where "n" is expanded to 4 digits with leading zeros if necessary.

Example.

GLOBAL £(PRINT:37 £)
equivalent to EXTERNAL ".G" £(PRINT:"0037" £)

Unless all GLOBALs in a program have associated static data items, the segments that use GLOBAL should have
£LIBRARY "BCL:GLOBAL"
in them to satisfy GLOBAL references between -200 and +400 with no associated STATICs.

Note. The assumption that global items appear next to each other in a consecutively addressed vector at run time will probably cause failure in this implementation.

3.7.4 SPLITTING UP A BCPL PROGRAM

This is done by putting routines and functions that can be seperately compiled into different files. Communication is then achieved between files using the EXTERNAL facility. It is usually advisable to put such EXTERNAL declarations into a GET file (see 5.1), so that if it is necessary to change the external name of an EXTERNAL then only one alteration need be made to the source files and there is less chance of error occuring.

4.Ø COMMANDS

4.1 SYNTAX OF COMMANDS

```
<C> ::= <IF¦UNLESS¦WHILE¦UNTIL> <E> DO <C> ¦
        TEST <E> THEN <C> OR <C> ¦
        SWITCHON <E> INTO <C> ¦
        FOR <name> = <E> TO <E> <BY <E>>** DO <C> ¦
        <name>:<C> ¦ CASE <E><...<E>>**:<C> ¦
        DEFAULT :<E>...<E>>**:<C> ¦ <C> <> <C> ¦
        <C> REPEAT ¦ <C> <REPEATWHILE¦REPEATWHILE> <E> ¦
        <C> WHERE <D> <AND <D>>* ¦
        GOTO <E> ¦ RESULTIS <E> ¦ BREAK ¦ LOOP ¦
        RETURN ¦ FINISH ¦ ENDCASE ¦
        £( <D>** <C>** £) ¦ £[ <codeblock> £] ¦
        <E>() ¦ <E>(<E-list>) ¦
        <E-list> := <E-list> ¦
        <E-list> <binop>:= <E-list> ¦
        TRACE <name>() ¦ TRACE <name>(<E-list>)
```

4.2 SECTIONS

£(<D>** <C>** £)

Sections are either blocks or compound statements depending on whether they contain any declarations or not. This is only semantically important in that a block is scanned for labels and a compound statement is not. Syntactically they both provide a mechanism for grouping statements together. A machine code block is a special case, it is described in Appendix D.

4.3 SIMPLE ASSIGNMENT

E1 := E2

E2 is evaluated then the result is stored in the item denoted by E1. E1 should be either an expression which represents a storage cell or a selector or byte expression which indicates that the specified byte is to be overwritten with the value E2 (truncated if necessary).

Examples.

```
X := Y
!V := Ø
V!1 := 1
(BYTE 6:12)&&Z := Q
```

4.4 MULTIPLE ASSIGNMENT

<E-list> := <E-list>

Example.
Same as in 4.3, but order of evaluation undefined.

X, !V, V!1, (BYTE 6:12)&&Z := Y, Ø, 1, Q

4.5 LEFT HAND SIDE FUNCTION APPLICATION

If a function application appears on the left hand side of an assignment then the function is called as a routine with an extra parameter. Viz the value on the right side of the assignment. A routine invoked in this way will get the reply TRUE from the library function LHS (see 6.6.3). Useful for modelling arrays.

Example.

ARRAY(X,Y) := 2
is equivalent to the command
ARRAY(X,Y,2)
but LHS() will return TRUE in the first case and FALSE in the second.

4.6 UPDATE ASSIGNMENT

<E-list> <binop>:= <E-list>

This represents an update assignment (<binop> can be any binary operator except %<name>). <binop>:= can be used instead of := in all cases discussed in 4.3, 4.4 and 4.5.

E1 <binop>:= E2 is equivalent to E1 := E1 <binop> E2
E1(E5,E6) <binop>:= E2 is equivalent to
E1(E5, E6) := E1(E5, E6) <binop> E2

Examples.

X +:= 1 updates X by 1
V !:= Ø assigns Ø!V to V

4.7 ROUTINE APPLICATION

<E>() or <E>(<E-list>)

This is similar to function application (see 2.10) except that no result is returned.

Example.

```
PRINT("PRINT CALLED")
```

4.8 RETURN

This command is used to return control from the currently active routine to its point of invokation.

4.9 GOTO

GOTO <E>

The expression is evaluated to yield a value (assumed to be a label in the program) to which control is passed. Control should not be passed to labels outside the function or routine in which the GOTO appears. Such transfers should only be performed by the library routines LEVEL/LONGJUMP or LABEL/JUMP. (See 6.6.5 to 6.6.8).

Examples.

```
GOTO NEXT
GOTO X = Y -> EQUALS, NOTEQUALS
```

4.10 IF

IF <E> DO <C>

<E> is evaluated and if the result is not FALSE <C> is executed.

Example.

```
IF TRACING DO PRINT("ENTERED")
```

4.11 UNLESS

UNLESS <E> DO <C>

<E> is evaluated and if the result is FALSE, <C> is executed.

Example.

```
UNLESS N=0 DO PRINT(N)
```

4.12 TEST

TEST <E> THEN <C1> OR <C2>

<E> is evaluated and if the result is not FALSE, <C1> is executed, otherwise if the result of <E> is FALSE, <C2> is executed.

Example.

```
TEST REAL THEN RES := A #+ B OR RES := A + B
```

4.13 WHILE

WHILE <E> DO <C>

A) <E> is evaluated
B) if the result is not FALSE, <C> is executed and control returned to A), otherwise the WHILE command is terminated.

Example.

```
WHILE 'A' LE CH LE 'Z' DO CH := INCH()
```

4.14 UNTIL

UNTIL <E> DO <C>

A) <E> is evaluated
B) if the result is FALSE, <C> is executed and control returned to A), otherwise the UNTIL command is terminated.

Example.

```
UNTIL FINISHED DO ROUTINE()
```

4.15 FOR

FOR <name> = <E1> TO <E2> <BY <K>>** DO <C>

The FOR command is best described by giving the semantic equivalent which is:-

```
£( LET <name> = <E1>
   LET    ?    = <E2>
   UNTIL <K> > Ø -> N GR ?, N LS ? DO
   £( <C>
      <name> +:= <K>
   £)
£)
```

If <BY <K>> is omitted <K> is assumed to be 1. ? is a compiler generated local variable, note that <name> is implicitly declared as a local (dynamic) variable by the FOR command and has only the scope shown in the semantic equivalent above.

Examples.

```
FOR I = Ø TO 1Ø DO V!I := Ø
FOR I = START() TO END() - 1 BY - 5 DO STEP(I)
```

4.16 REPEAT

<C> REPEAT

<C> is repeated indefinitely. <C> is usually a section containing a BREAK command to terminate the repetition.

Example.

```
£( IF !V = Ø DO BREAK
   V !:= Ø
£) REPEAT
```

4.17 REPEATWHILE

<C> REPEATWHILE <E>

A) <C> IS EXECUTED.
B) <E> is evaluated and if the result is not FALSE control is returned to A), otherwise the REPEATWHILE command is terminated.

This differs from WHILE in that in this case <C> is executed at least once.

Example.

```
CH := INCH() REPEATWHILE 'A' LE CH LE 'Z'
```

4.18 REPEATUNTIL

<C> REPEATUNTIL <E>

A) <C> is executed.
B) <E> is evaluated and if the result is FALSE control is returned to A), otherwise the REPEATUNTIL command is terminated.

This differs from UNTIL in that this case <C> is executed at least once.

Example.

```
ROUTINE() REPEATUNTIL FINISHED
```

4.19 LOOP AND BREAK

These commands can appear in the body (<C>) of any of the loop commands. (i.e. those described in sections 4.13 to 4.18). BREAK transfers control to immediately after the loop command. LOOP transfers control to :-

The start of the command for WHILE, UNTIL and REPEAT.
The test for REPEATWHILE and REPEATUNTIL.
The increment and test for FOR.

Example.

```
FOR I = 1 TO 100 DO
£( IF V!I = 0 DO BREAK        //END OF VECTOR
   IF V!I < 0 DO LOOP         //IGNORE NEGATIVE
   POS(V, I)
£)
```

4.20 SWITCHON, CASE, DEFAULT AND ENDCASE

SWITCHON <E> INTO <C1>

Where <C1> is usually a compound statement containing labelled commands of the form:-

CASE K: <C2>
DEFAULT: <C2>

<E> is evaluated and compared with the values <K> of the CASE labels, if a match is found then control is passed to the command <C2> following the CASE label. If no match is found then control is passed to the DEFAULT case. IF there is no DEFAULT case an automatic ENDCASE is performed. <K> must be a compile time constant (see 2.21.2).

An ENDCASE command causes control to pass immediately to a point just following the command <C1>, that is it effectively ends the execution of the SWITCHON command.

Two further CASE and DEFAULT constructions are available, they are:-

CASE <K1> ... <Kn>: <C2>
which is equivalent to
CASE <K1>: CASE <K2>: etc to CASE <Kn>: <C2>

and

DEFAULT <K1> ... <Kn>: <C2>
which tells the compiler that <E> in the SWITCHON will always lie in the range <K1> to <Kn>. (This may enable the compiler to produce faster code under certain circumstances).

Beware, the construct <K1> ... may be confused by the compiler as a badly written real number if A) <K1> is an integer and B) no space is left between <K1> and the '...'. Remember that '.' is also a vaild character in a BCPL name (see 2.4), so the compiler will read 'A...B' as a single name and not a CASE range.

Example.

```
MANIFEST £( CHA='A'; CHZ='Z' £)
SWITCHON NEXT() INTO
£(  CASE CHA ... CHZ: ALPHA(); ENDCASE
    CASE '0' ... '9':
    CASE '.': NUMBER(); ENDCASE
    CASE #177: CASE 0: ENDCASE
    DEFAULT 0 ... #177: ERROR()
£)
```

4.21 FINISH

This command causes execution of the program to cease.

4.22 RESULTIS

RESULTIS <E>

This command is used to return the value of the expression <E> as the result of a VALOF expression (see 2.9).

4.23 LABELS

<name>: <C>

This makes the <name> known as a static data item, which is initialised at start of execution of the program to the address of the point of code at which the labels appears. (see 3.3.2).

4.24 VERY BINDING SEMICOLON

<C1> <> <C2>

This is used to join commands together as if they were a section. It is equivalent to writing

£(<C1>; <C2> £)

but is more convenient to write and often more readable.

Example.

```
IF LIST = 0 DO LIST := NEWVEC(3) <> LIST!0 := 0
```

4.25 TRACE

TRACE <name>()
TRACE <name>(<E-list>)

This command creates a pseudo routine call to a trace package. It is only compiled if the program is being compiled with postmortem code. The <E-list> expressions together with the <name> (which is used to identify the particular TRACE command) are passed as parameters to the trace package. The normal call format is a similar parameter list to that expected by OUT (see 6.4.8).

In the PDP-10 implementation TRACE points are always handled by the default library when the compiled program is running in a batch mode. If the compiled program is running interactively then the user is queried on the first encounter of each TRACE point as to whether he requires the trace information or not.

Example.

```
TRACE BREAK1("TOP=%8, CHAIN=%8*C*L", LH&&PTR, RH&&PTR)
```

5.0 PROGRAMMING AIDS

5.1 GET

GET "PDP-10 FILESPEC"

This directive may occur anywhere in a BCPL file on a line by itself. The GET directive is replaced by the text of the specified file.

It is used to read in a common piece of BCPL source text, and as such is usually used for common MANIFEST, EXTERNAL and GLOBAL declarations. These can be kept in a single file, which is compiled with all the files of a complete program. As described in 3.7, equivalent EXTERNAL (and GLOBAL) declarations should appear at the top of each file in which the EXTERNALs (or GLOBALs) are used or defined. GET provides the mechanism for doing this economically without the risk of writing the declarations differently in separate files.

The EXTERNALS used for the BCPL library described in section 6 are declared in a single GET file "BCL:BCPLIB.GET".

The default device, extension and ppn are
DSK:filename.GET[user]
or failing that
DSK:filename.BCL[user].

5.2 THE START ROUTINE

By convention execution of a BCPL program starts at the routine named START which should be supplied by the user and declared to be EXTERNAL. (Such a declaration exists in BCL:BCPLIB.GET - see section 6).

5.3 LAYOUT CONVENTIONS

5.3.1 SEMICOLONS

Semicolons between commands (declarations) in a section may be omitted provided that the commands (declarations) are each written on separate lines.

5.3.2 CONTINUATION OF EXPRESSIONS

Expressions may be written on more than one line, provided that each line ends with a dyadic operator, or in the case of an <E-list>, a comma.

Examples.

```
A + B +
C

TABLE 0, 0, 0,
      0, 0, 0
```

5.3.3 OMISSION OF DO OR THEN

The keywords DO and THEN are interchangeable, and may be omitted (when no ambiguity occurs from doing so).

Example.

```
IF X=0 X:=5
```

but beware of

```
IF X=0 !X:=3
```

5.3.4 TAGGING OF SECTION BRACKETS

Section brackets may be tagged with either a BCPL name or a positive integer. The tag must be connected to the bracket with no space between. Tags serve two purposes, one is to name a section to give readability, the second is to ensure section brackets are closed. A tagged closing bracket closes all unclosed brackets up to the matching named bracket.

Example.

```
£(A
    £(1
        £(2    £)A
is the same as
£(A
    £(1
        £(2    £)2    £)1    £)A
```

5.4 COMMENTS

There are three ways of introducing comments into a BCPL segment.

Comments introduced by

// continue up to the end of the line
/* continue up to a matching */

The sequence

). and any characters up to a (
is equivalent to a comma.

Example.

```
LET F(S). APPLIED TO (M) BE
/* TESTS M TO SEE IF POSITIVE
   AND THEN APPLIES S TO IT */
£(F IF M>=ø DO S(M) <> RETURN
     IF S=SQRT DO ERROR(M) // SQRT OF NEG NO. FORBIDDEN
£)F
```

5.5 INPUT FORMAT

The Input lines to the BCPL compiler may contain up to 150 characters and should be terminated by a carriage return-line feed sequence. All input files are read in ascii line mode, editor line numbers are ignored, but listed if listing enabled.

5.6 LISTING CONTROL

The following escape words are used only to control the compilers listing of the source program.

£NOLIST	Turns listing off, unless /L or /C compiler switch is on.
£LIST	Turns listing back on.
£EJECT	Skips listing to the top of a new page.
£SPACE n	Skips n lines in the listing.

5.7 CONDITIONAL COMPILATION

If the expression which controls the execution of a command is a constant, and the value of the constant is such that it would inhibit the execution of the command, then the compiler will not plant the code for that command, unless it contains an alternative way of entering it (e.g. via a label). Commands that fall into this category are IF, UNLESS, TEST, WHILE, UNTIL, REPEATWHILE and REPEATUNTIL. Conditional expressions are treated in the same way.

Example.

```
IF DEBUGGING DO OUT("SUBROUTINE ENTERED")
```

If DEBUGGING is a manifest constant and if its value is FALSE then no code at all would be planted for the above. If its value is TRUE then the code planted would only be for

```
OUT("SUBROUTINE ENTERED")
```

During program development debugging code can be enabled with a MANIFEST constant set to TRUE; when the program is complete and working the debugging code can be effectively removed just by recompiling with the same MANIFEST constant set to FALSE.

Two further kinds of conditional compilation are available.
a) £TRACE returns TRUE or FALSE according to whether the program is being compiled with postmortem code enabled or not, and can be used in conjunction with the feature described above.
b) Comments of the form
//? comment
are normally ignored by the compiler. If, however, the /U compiler switch is turned on then the //? itself is ignored and the rest of the line compiled as if it were not a comment.

5.8 USER LIBRARY AIDS

The following are produced for aiding the preparation of libraries of user programs. They cause entries to be made in the LINK loader tables type 4, 16 and 17, prospective users of these should familiarise themselves with the functions of these tables.

£ENTRY "<name>"	Puts <name> in the ENTRY block.

£NEEDS "PDP 1Ø FILE SPEC"	Causes the specified file to be loaded with this segment and prior to searching any libraries.
£LIBRARY "PDP 1Ø FILESPEC"	Causes the specified file to be searched at the end of normal loading but prior to the system library search, for any unsatisfied external references.
£LDTEXT "STRING"	Causes the string to be put out as it appears into the code file.

Any EXTERNALS defined within a file that do not appear in a £ENTRY will be of type INTERNAL. I.E. they will satisfy any references if the file is loaded, but a reference will not cause the loader to load the file.

5.9 VERSION NUMBER

£VERSION n will set the version field to n in the loaded program. n should be an octal constant.

6.0 THE BCPL LIBRARY

6.1 THE STRUCTURE OF THE LIBRARY

It is possible to compile files of a complete program separately and link them together at load time using EXTERNAL or GLOBAL declaration (see 3.7).

The BCPL library consists of a collection of BCPL routines in a system file (SYS:BCPLIB.REL), which is searched by the loader after all the user supplied files have been loaded; any EXTERNAL names referred to, but not defined in the user files, are searched for in the library and, if found, the correct code is loaded.

The file BCL:BCPLIB.GET contains EXTERNAL declarations for the more widely used routines, which are described in this section.

A standard header file (such as BCL:BCPLIB.GET) should always be used for communication with a library, as the loader names of the library routines may be changed from time to time but the 'internal' names, used below, will always remain the same.

There are many more routines in the library than are listed below. The other routines, and some other useful libraries are described in the BCPL USER GUIDE.

6.2 INPUT AND OUTPUT SPECIFICATION

6.2.1 STREAMS AND STREAM CONTROL BLOCKS

The basic primitive of the BCPL input/output system is the STREAM. Within a BCPL program, a stream corresponds to a file on an external medium (such as a disc or a teletype). Routines for reading and writing manipulate data structures called STREAM CONTROL BLOCKS, which contain all the information necessary to communicate with the external media (with the help of the PDP-10 monitor). Stream control blocks are sometimes known as SCBs. Further details of SCBs can be found in Appendix H.

In general, communication with external media follows the schema below:-

1. Create a stream which corresponds to the external file, (usually by one of the functions which follow), assign the SCB corresponding to this stream to some variable.

2. Perform the I/O functions required.

3. Inform the I/O system that the stream is no longer required.

6.2.2 FINDFILE

Finds a monitor input stream. This takes the following parameter list:-

FINDFILE(DEV,FILE,EXT,PPN,ERROR,MODE,BUFFS,PROT,TIME,DATE)

the parameter list may be abbreviated to cover as few parameters as the user wishes to supply. Any which are omitted will be defaulted to zero. If a completely null parameter list is given then the standard teletype stream (TTY) is returned.

DEV is the device in sixbit or BCPL string form. Zero implies "DSK".

FILE is the filename in sixbit or BCPL string form.

EXT is the file extension in sixbit or BCPL string form.

PPN contains in the left half word the project number and in the right half word the programmer number.

ERROR should be an error label constructed via the LABEL routine (see 6.6.7) this is JUMP'ed to if an I/O error occurs.

MODE is the PDP 10 I/O mode.

BUFFS the number of buffers. Zero implies the default for the device.

PROT as defined for the monitor LOOKUP/ENTER block.

TIME as defined for the monitor LOOKUP/ENTER block.

DATE as defined for the monitor LOOKUP/ENTER block.

FINDFILE returns a SCB for use with the routines described in section 6.3. The space for the SCB and any required buffer space is obtained via a call to NEWVEC (see 6.5.4), so INITIALISEO (see 6.5.2) should have been called before any calls to FINDFILE are made.

[The size of the SCB allocated by FINDFILE is controlled by the static SCBSIZE (see BCL:SCB.GET) which may be increased by the user before calling FINDFILE if extra space in the SCB is required.]

The LOOKUP/ENTER block used is the 4-word block, as

described in the DECsystem-10 Monitor Calls Manual.

Example.

```
INPUT := FINDFILE("DSK", "TEST", "SRC")
```

6.2.3 CREATEFILE

Finds a monitor output stream. It has the same parameter specifications as FINDFILE. The SCB returned can be used with any of the routines described in Section 6.4.

6.2.4 UPDATEFILE

Finds a monitor update stream, which can be used for both input and output. It has the same parameter specifications as FINDFILE. The SCB returned can be used with any of the routines described in sections 6.3 and 6.4.

6.2.5 FINDTTY

This function which takes no parameters returns a pointer to the SCB for the jobs controlling stream (TTY). This corresponds to the user's console for interactive users, or the command/log file for batch users. The SCB returned can be used with any of the routines described in sections 6.3 and 6.4.

Example.

```
OUTPUT := FINDTTY()
```

6.2.6 TTY

This is a static in the library which is set initially to point to the SCB for the the jobs controlling stream (see 6.2.5). This stream can be used for both input and output.

6.2.7 INPUT, OUTPUT, MONITOR AND CONSOLE.

These are four statics in the library which are initially unset but which are conventionally used for holding I/O SCB pointers.

INPUT - conventionally used for current main data input stream, all the functions with names prefixed by IN described in section 6.3, assume that the SCB pointed to by INPUT is the stream on which their action is to apply.

OUTPUT - conventionally used for current main data output stream, all the routines with names prefixed by OUT described in section 6.4, assume that the SCB pointed to by OUTPUT is the stream on which their action is to apply.

CONSOLE - conventionally used for current command or control input stream.

MONITOR - conventionally used for current monitoring stream for error messages, progress reports and tracing information.

6.2.8 ENDREAD, ENDWRITE AND CLOSE

All these routines take an SCB as their parameter and invoke the closing routine of the SCB. For FINDFILE, CREATEFILE and UPDATEFILE the default action is to empty the buffers, release the monitor channel and give the SCB and any buffer space back to the freespace system. For FINDTTY the default action is to empty the teletype buffer.

6.3 INPUT ROUTINES

6.3.1 STRUCTURE OF INPUT ROUTINES

For each input function there are three standard formats available.

INroutine which is a function which returns a value read from the stream INPUT (see 6.2.7)

RDroutine which is a function which returns a value read from the stream which is its first parameter

READroutine which is a routine which assigns a value, read from the stream which is its first parameter to the location which is its second parameter.

Example.

All the following have the same action

```
CH := INCH()
CH := RDCH(INPUT)
READCH(INPUT, @CH)
```

6.3.2 INCH RDCH AND READCH

READCH is the basic input routine, all the other input routines described in section 6.3 invoke READCH. READCH invokes the read routine for the stream which is its first parameter. The streams set up by FINDINPUT or FINDUPDATE behave according to the 'mode' of the file:-

Modes 0-1 return a 7 bit ascii character, ignoring line numbers, returns '*E' end-of-stream character at end of file.

Modes 2-14 return a 36 bit (full word) byte, jumps to error label at end of file.

Modes 15-17 READCH only - the second and subsequent parameters are a command list, each element of which is an 'IOWD', the left half of which is the negative of the word count, and in right half of which is the address-1 of the data area for the transfer. If there are not enough words left in the file to satisfy the command list then a jump is made to the error closure. The last word of the command list must be zero.

The read routine in the TTY SCB returns 7 bit ascii characters.

For example see 6.3.1

6.3.3 INNO RDNO AND READNO

Reads an (optionally signed) decimal number. All characters up to + - or a digit are ignored. The number is terminated by any non digit character, which will be the next character read from the stream.

Example.

All the following have the same action.

```
NO := INNO()
NO := RDNO(INPUT)
READNO(INPUT, @NO)
```

6.3.4 INF RDF AND READF

Reads an (optionally signed) real number. All characters up to + - or a digit are ignored. The number read should conform to BCPL real number syntax. The character following the number will be the next one read from the stream.

Example.

All the following have the same action.

```
F := INF()
F := RDF(INPUT)
READF(INPUT,@F)
```

6.4 OUTPUT ROUTINES

6.4.1 STRUCTURE OF OUTPUT ROUTINES

Most of the output routines have two standard formats:-

OUTroutine which writes its value(s) to the stream OUTPUT (see 6.2.7)

WRITEroutine which expects as its first parameter a stream to which it writes its value(s).

NEWLINE, NEWLINES, SPACE and SPACES write to the stream OUTPUT.

Example.

Both the following have the same effect.

```
OUTCH(CH)
WRITECH(OUTPUT,CH)
```

6.4.2 OUTCH AND WRITECH

WRITECH is the basic output routine, all other output routines described in section 6.4 invoke WRITECH. WRITECH invokes the write routine for the stream which is its first parameter. WRITECH is the direct counterpart of READCH (see 6.2.2 for discussion of modes).

For example see 6.4.1

6.4.3 OUTNO AND WRITENO

Write a (possibly negative) decimal number in as few character positions as are necessary.

Example.

Both the following have the same effect.

```
OUTNO(NO)
WRITENO(OUTPUT,NO)
```

6.4.4 OUTF AND WRITEF

Write a (possibly negative) real number in standard BCPL real number syntax, if the decimal exponent lies in the range -1 to +6 then the exponent is suppressed.

Example.

Both the following have the same effect.

```
OUTF(F)
WRITEF(OUTPUT,F)
```

6.4.5 OUTS AND WRITES

Write a BCPL string.

Example.

Both the follwing have the same effect.

```
OUTS("STRING")
WRITES(OUTPUT,"STRING")
```

6.4.6 OUTO, WRITEO, OUTOCT AND WRITEOCT

Write the specified number of rightmost (least significant) octal digits of the number. OUTO and OUTOCT are synonymous, as are WRITEO and WRITEOCT.

Example.

Both the following have the same effect.

```
OUTO(OCTVAL,PLACES)
WRITEO(OUTPUT,OCTVAL,PLACES)
```

6.4.7 OUTI AND WRITEI

Write the decimal number to the number of places specified, padded on the left with spaces if necessary.

Example.

Both the following have the same effect.

```
OUTI(DECVAL,PLACES)
WRITEI(OUTPUT,DECVAL,PLACES)
```

6.4.8 OUT AND WRITE

```
OUT(FORMAT, P1, P2 ... P26)
WRITE(STREAM, FORMAT, P1, P2 ... P26)
```

Write the parameters P1 to P26 according to the format list in the BCPL string FORMAT. The FORMAT string is written out character by character until the escape character '*' is found. If the character following the '*' is one of the following then the next parameter is written out by the appropriate routine.

```
*C   Use WRITECH to write the next parameter
*N    "  WRITENO  "   "      "    "      "
*F    "  WRITEF   "   "      "    "      "
*S    "  WRITES   "   "      "    "      "
*On   "  WRITEO   "   "      "    "      "      to n places
*In   "  WRITEI   "   "      "    "      "       "  "    "
**   write the character '*'.
```

The action is undefined for other characters.

Example.

Both the following have the same effect.

```
OUT("*C*L*N ERROR*C DETECTED", EC, EC > 1 'S', ' ')
WRITE(OUTPUT,
    "*C*L*N ERROR*C DETECTED", EC, EC > 1 'S', ' ')
```

6.4.9 NEWLINE, NEWLINES, SPACE AND SPACES

```
NEWLINE() writes a newline.
NEWLINES(n) writes n new lines.
SPACE() writes a space.
SPACES(n) writes n spaces.
```

On the stream held in OUTPUT.

.5 FREE STORAGE ROUTINES

6.5.1 PURPOSE

The free storage system is used to maintain a vector which is used by the I/O system for SCB and buffer space. It also provides a mechanism by which the user obtains dynamically (and frees dynamically) vectors of arbitrary size. In BCPL systems 3F and later, it is not strictly necessary to call INITIALISEIO or FREESPACE when running on a virtual memory system.

6.5.2 INITIALISEIO

INITIALISEIO(IOVECTOR, SIZE)

This routine is used to initialise the I/O system and must be called before any of INNO, RDNO, READNO, INF, RDF, READF, FINDFILE, CREATEFILE and UPDATEFILE. The IOVECTOR is passed to FREESPACE (see 6.5.3). Approximately 350 words of IOVECTOR space are used for normal input and output streams set up by FINDFILE and CREATEFILE, and 650 words for update streams set up by UPDATEFILE.

6.5.3 FREESPACE

FREESPACE(VECTOR, SIZE)

Gives the vector to the free storage system to be allocated by NEWVEC (see 6.5.4). If input and output via FINDFILE, CREATEFILE or UPDATEFILE is required then INITIALISEIO should be called instead (see 6.5.2).

6.5.4 NEWVEC

NEWVEC(SIZE)

This function returns a vector of SIZE+1 elements (i.e. having subscripts ZERO to SIZE), from the free space vector supplied to INITIALISEIO or FREESPACE. If NEWVEC is called (either directly or indirectly from FINDFILE, CREATEFILE or UPDATEFILE) before INITIALISEIO or FREESPACE have been called - then the message

?BCLNFS NEITHER INITIALISEIO NOR FREESPACE HAVE BEEN CALLED

will be printed and the job terminated.

If NEWVEC is called and there is no space left in the free

space vector then the message

?BCLFSE BCPL I-O FREESPACE EXHAUSTED

will be printed and the job terminated.

NOTE: newvec(n)!-1 = (n+2,,(525252$_8$))

6.5.5 FREEVEC AND FREE

FREEVEC(VECTOR)

FREEVEC and FREE are synonyms for the same routine, which is used to return a vector previously allocated by NEWVEC back to the free storage system.

6.6 OTHER UTILITY ROUTINES

6.6.1 UNPACKSTRING

UNPACKSTRING(S, V)

This routine unpacks the BCPL string S into the vector V, one character per word, V!0 will hold the length of the string. It returns V as its result.

6.6.2 PACKSTRING

PACKSTRING(V, S)

Packs the n characters in V+1 to V+n into the vector S in standard BCPL string format (see 2.7). V!Ø contains the value n. It returns the packed vector S as its result.

6.6.3 LHS

This function returns the value TRUE only if the routine that calls it, was itself invoked as a result of appearing on the left hand side of an assignment statement (see 4.5). LHS otherwise returns the value FALSE. [This routine will not work if the LHS/ NUMBARGS suppress switch to the compiler is used.]

6.6.4 NUMBARGS

This function returns the number of actual parameters that were passed to the routine that invoked NUMBARGS. It is used to write routines which can have varying numbers of parameters. (E.G. FINDFILE).

WARNING: The BCPL library relies on NUMBARGS for its variable parameter routines such as PRINT, FINDFILE etc. The NUMBARGS code should not be suppressed (via the appropriate compiler switch) if the library is to be used.

6.6.5 LEVEL

This function returns the value of the current environment, (stack level), which can be used with LONGJUMP (see 6.6.6).

6.6.6 LONGJUMP

LONGJUMP(LEVEL, LABEL)

Jumps to the label LABEL, re-setting the environment to that held in LEVEL (obtained from LEVEL function - see 6.6.5). The LEVEL and LONGJUMP routines used together give a completely machine independent way of breaking out of a set of nested routine invokations.

Example.

```
LET R() BE
£(R
  STATIC £( LL=Ø;LB=Ø £)
  LET S(N) BE
  £(S TEST N = 0 THEN LONGJUMP(LL,LB)  // JUMPS OUT IN
                      //ONE ACTION TO LAB, RESETTING THE
                      //ENVIRONMENT (STACK) TO THAT OF R
      OR S(N-1)
  £)S
  LL := LEVEL()       //RETURNS THE ENVIRONMENT (STACK)
                      //OF THE INVOKATION OF R
  LB := LAB           //USE A STATIC IN SCOPE OF S
  S(4Ø)
  LAB:
£)R
```

6.6.7 LABEL

LABEL(LAB)

This function returns an object consisting of the label LAB and the current environment (stack level); the value can be used with JUMP (see 6.6.8). Values returned from LABEL are often referred to as CLOSURES.

6.6.8 JUMP

JUMP(CLOSURE)

Jumps to the label component CLOSURE, resetting the environment to that indicated by CLOSURE (obtained from LABEL function - see 6.6.7). LABEL and JUMP can be used as a way of breaking out of a set of nested routine invokations. A LABEL closure is used for the ERROR parameter to the FINDFILE, CREATEFILE and UPDATEFILE functions, so that when an error occurs a jump can be made to a predefined point in the program, and the environment re-set.

Example.

This has the same effect as the example in 6.6.6.

```
LET R() BE
£(R
  LET S(N,E) BE
  £(S IEST N=0 THEN JUMP(E)     // JUMPS OUT IN ONE ACTION
                                // TO LAB, RESETTING THE
                                // ENVIRONMENT TO THAT OF R
      OR S(N-1, E)
  £)S
  S(40, LABEL(LAB))             //LABEL RETURNS THE ENVIRONMENT
                                //(STACK) OF THE INVOKATION OF
                                //R, WITH THE VALUE OF LAB
  LAB:
£)R
```

Example.

```
INPUT := FINDFILE("DSK", "INPUT", "SRC", 0, LABEL(ERR))
 .
 .
 .
ERR: WRITES(TTY,"*C*LCAN*'T FIND INPUT.SRC")
FINISH
```

7.0 RUNNING A BCPL PROGRAM

7.1 COMPILER COMMAND FORMAT

The compiler is run by the command

.R BCPL

when the compiler is ready it responds with

*

The form of the command string is

```
*REFILESPEC,LISTFILESPEC=INPUTFILELIST
OR
*REFILESPEC=INPUTFILELIST
OR
*INPUTFILELIST
```

= CAN BE REPLACED BY _.

INPUTFILELIST
is optional GETFILESPECS followed by INPUTFILESPEC

Compiler switches (listed in Appendix E) can appear after any file spec, in the form

```
/SWITCH:N    (:N is optional)
or
(SWITCH:N,SWITCH:N ... SWITCH:N)    (:N is optional)

Default INPUTFILESPEC is DSK:filename.BCL[user]
                      or DSK:filename.BCP[user]
Default RELFILESPEC is DSK:filename.REL[user]
                    or DSK:filename.INT[user] (INTCODE)
                    or DSK:filename.OUT[user] (TREE)
Default LISTFILESPEC is DSK:filename.LST[user]
Default GETFILESPEC is DSK:filename.GET[user]
                    or DSK:filename.BCL[user]
```

Any filename missing is defaulted to that of the input file.

A relfile is always produced (unless inhibited by the 'G' switch) and a listfile is not normally produced (unless required by the 'L' or 'C' switch).

If the 'D' switch is enabled then a symbol file DSK:relfilename.SMB[user] is produced for use by the debugger.

If LISTFILESPEC appears then a list file is always produced.

7.2 COMPILER OUTPUT

The RELFILE normally produced by the compiler is a standard relocatable binary suitable for loading with LINK loader. The compiler compiles TWOSEG code, but the 'P' switch forces it to produce low code only. The TWOSEG code is restartable and re-entrant, unless the 'V' switch is used in which case it may not be restartable.

The LISTFILE normally produced lists the source line number and the block nesting level in two columns on the left side of the page and the source lines on the right. Erroneous lines are listed even if listing is temporarily suppressed (see 5.6). The line numbers are indented one space for each nesting level.

INTCODE output is in ascii text form, described in section 8.

The symbol file, if produced, contains BCPLDT debugging information in ascii text form.

7.3 COMPILER ERROR MESSAGES

Error messages are of two levels, ERROR (E) level and WARNING (W) levels. The latter being errors from which the compiler thinks it can recover. The errors are two kinds, Syntactic Errors and Semantic Errors. Syntax errors are detected during the first scan of the program; the erroneous source line is listed on the error medium with a pointer to the next symbol after the error. (If the error is at the end of a line, the pointer may point to the start of the next line). If the error has occurred in a GETfile, this is indicated in the error message.

Semantic errors are detected during the second scan of the program; the part of the parse tree that is erroneous is printed.

The final ERROR count is printed on the Teletype (LOG file for batch users).

Error messages are normally put on the Teletype (LOG file for batch users), unless the compiler is running on a BATCH job with LISTFILE present in which case the LISTFILE is used as well. Error messages on the Teletype are abbreviated. They are preceded by a code in brackets consisting of a letter (E for ERROR, W for WARNING) and the error message number (see Appendix G) followed by the line no on which the error was detected. If the error was in a getfile this is indicated by (G:getfilename).

7.4 PROGRAM SIZE

7.4.1 NON VIRTUAL MEMORY SYSTEMS

The compiler has workspace to handle up to approximately 200 source lines. It is advisable to split files larger than this into separate parts. If, however, you wish to increase the workspace of the compiler, this can be done with the 'S' switch. An indication that the workspace is near full is when the % figure (printed by the compiler at the end of the compilation) approaches 100%.

7.4.2 VIRTUAL MEMORY SYSTEMS

The compiler acquires workspace for compiling source programs until it reaches the limit imposed by the Monitor. It is, however still advisable to split large files into separate parts, as this makes better use of the separate compilation facility, and optimises use of machine time during program development.

7.5 INTERACTIVE USE

The compiler is best run interactively by use of COMPIL class commands, see 5.5 for description of input files.

.COMPILE
.LOAD
.EXECUTE
.DEBUG

which all recognise .BCL and .BCP extensions as requiring the BCPL compiler, and generate the correct command string for the compiler. See the time-sharing manual for a complete description of COMPIL and how to pass switches to the compiler via COMPIL. Source errors detected by the compiler are usually reported to the controlling teletype.

The postmortem, described below (see 7.7) can be obtained at any point during the execution of the program by "control-C"ing out of the program and then typing

.REENTER

This can also be performed upon completion of execution.

The program can be debugged either using DDT or BCPLDT. The user is advised to look at some BCPL generated code (use the /M switch) before attempting to use DDT. All the statics in the program are put into the symbol table for the segment. [For a description of the generated code see Appendix F.]

7.6 BATCH USE

The following deck will list, compile and execute a two file BCPL program with a common GET file, and provide a postmortem if the program fails:-

```
£SEQUENCE nnnnn              // get this from open shop
£JOB name [proj,prog]        // program name and user no.
£DECK header.GET
        .
        .                    // "GET" header file
        .
£EOD
£BCPL name.BCL
        .
        .                    // first BCPL file
        .
£EOD
£BCPL name.BCL
        .
        .                    // second BCPL file
        .
£EOD
£DATA                        // must appear to
        .                    // force execution
        .                    // data (if any -
        .                    // can be null)
        .
£EOD
%ERR:                        // these cards ensure clean
%FIN:                        // termination and postmortem
.REENTER                     // under all circumstances
end-of-file card             // from open shop.
```

To get access to the data following the £DATA card use INPUT:=FINDFILE("CDR"), see 6.2.2.

To create lineprinter output files use OUTPUT:=CREATEFILE("LPT"), see 6.2.3.

Source errors are listed on the source listing file and the Teletype.

7.7 THE POSTMORTEM

When BCPLDT has not been loaded the postmortem consists of three parts:-

The STACK BACKTRACK gives the hierarchical nesting of routines, I.E. who called who called who etc. The last routine entered is printed first.

The HISTORY gives the last 64 interesting events (routine entries and exits, labels and trace point calls). Exits

have '=' after them labels have ':' after them and trace points are indicated by '*'.

The PROFILE gives the number of times each routine was activated, each label passed and each trace point activated.

The library routines do not feature in the POSTMORTEM. The postmortem slows down program execution slightly. The postmortem can be removed from working programs by recompiling with the compiler switch 'P' set.

'.8 BCPLDT

BCPLDT is a purpose built BCPL debugger, it is designed primarily for interactive debugging, but can also be used under batch. Compiling programs with the 'D' compiler switch set creates a symbol file which can be used by the debugger.

Text and/or BCPLDT commands can be put into the symbol file (if one is being created) by puting lines of the form:
£SYMBOL "BCPL string"
into the source file. The strings are appended in the order of occurrence to the normal BCPLDT symbol file.

The debugger is described fully in the BCPL USER GUIDE.

8.0 INTCODE

8.1 THE PURPOSE OF INTCODE

INTCODE was designed by Martin Richards as a simple machine language with a primitive underlying machine structure, and as such, suitable for interpretation. Its uses are a) to allow BCPL programs to be run on small machines and b) permit transportability of BCPL programs (especially the BCPL compiler).

The INTCODE generator running at ESSEX is a version written by Vasiliki Kollias as part of an M.Sc. project, and was incorporated into the standard running BCPL compiler by Dave Lyons and Pete Gardner. It produces slightly different INTCODE from the original Martin Richards specification, and this section describes the new INTCODE. This section is paraphrased from part of Vasiliki Kollias' M.Sc. Thesis, and also from the original description of INTCODE by Martin Richards. All the features of the language are inplemented except for floating point operators, exponentiation, static and dynamic selectors and bytes, and the TRACE command.

INTCODE is obtained by using /P:2 compiler switch, the compiler tells you the maximum local label used in the segment, so that the local label vector in your assembler can be adjusted accordingly.

8.2 THE INTCODE MACHINE

8.2.1 WORDS

The basic unit of information is the word. According to the specification of the language the word size is implementation dependent, but it should ideally contain at least 18 bits in order to hold all the fields of an instruction.

8.2.2 REGISTERS

There are 5 registers in the INTCODE machine:

A & B: The Accumulator and Auxiliary accumulator.

C: The Control register giving the location of the next instruction to be executed.

D: The address register used to hold the effective address of an instruction.

P: A pointer used to address the local work area and

function arguments.

8.2.3 INSTRUCTION FORMAT

The instruction format comprises four fields: the function part F (3 bits - used to store one of the 8 machine instructions). The I bit (the Indirection bit) which when set then the D register is replaced by the contents of the location addressed by D. The P bit specifies whether P is to be added into D. The A field (remaining bits) stores the word address.

8.2.4 MACHINE INSTRUCTIONS

Opcode	Mnemonic	Meaning
Ø	L	B := A; A := D
1	S	Location (D) := A
2	A	A := A + D
3	J	C := D
4	T	IF A \= Ø DO C := D
5	F	IF A = Ø DO C := D
6	K	P := P + D; Location (P), Location (P + 1) := D, C; C := A
7	X	See below

The X instructions (the A field indicates one of):

X1	A := Location (A)
X2	A := -A
X3	A := \A
X4	This causes return from the current function; by convention the result is left in the accumulator. C := Location (P + 1); P := P - Location (P)
X5	A := B * A
X6	A := B / A
X7	A := B REM A
X8	A := B + A
X9	A := B - A
X10	A := B = A
X11	A := B \= A
X12	A := B < A
X13	A := B >= A
X14	A := B > A
X15	A := B <= A
X16	A := B << A
X17	A := B >> A

```
X18   A := B /\ A
X19   A := B \/ A
X20   A := B NEQV A
X21   A := B EQV A
X22   FINISH
X23   Switch on the value of A using data in the
      locations addressed by C, C+1 etc.
      B, D := Location (C), Location (C+1);
      UNTIL B = 0 DO
      £( B, C := B - 1, C + 2
         IF A = Location (C) DO
         £( D := Location (C + 1)
            BREAK
         £)
      £)
      C := D
X24   This causes an escape from intcode into the
      interpreter to perform the function
      indicated by the value in the accumulator.
```

As well as these instructions specified in the original INTCODE, ESSEX INTCODE defines the following:

```
X26   Selector apply. The parameters to this function
      are held in the 3 words following the instruction.
      A := A + Location (C)
      A := Location (A)
      A := A >> (- Location (C + 1))
      A := A /\ Location (C + 2)
      C := C + 3
X27   Selector store. The parameters to this function
      are held in the 3 words following the instruction.
      A := A + Location (C)
      D := Location (A)
      D := D /\ \Location (C + 2)
      B := B << (- Location (C + 1))
      B := B /\ Location (C + 2)
      Location (A) := B + D
      C := C + 3
X37   Byte apply. The parameters to this function
      are held in the 2 words following the instruction.
      A := A >> (- Location (C))
      A := A /\ Location (C + 1)
      C := C + 2
X38   Byte store. The parameters to this function
      are held in the 2 words following the instruction.
      D := Location (A)
      D := D /\ Location (C + 1)
      B := B << (- Location (C))
      B := B /\ Location (C + 1)
      Location (A) := D + B
      C := C + 2
X39   A := B ROTL A
X40   A := B ROTR A
X41   A := B ALSHIFT A
X42   A := B ARSHIFT A
```

```
X43   A := B BITOR A
X44   A := B BITAND A
X45   A := ABS A
```

8.3 THE INTCODE ASSEMBLY LANGUAGE

8.3.1 GENERAL LAYOUT

The assembly language for INTCODE has been designed primarily to be compact and simple to assemble, but some care has also been taken so that it can be read and modified with reasonable ease by the programmer. The text of the assembly language consists of letters, digits, spaces, newlines and the characters '//' '£' '#' '"' and '.'.

'//' is used as a continuation symbol; it is skipped and the remaining characters of the line up to and including the next newline are ignored. Its main purpose is to simplify the use of card images as a medium for transfering INTCODE programs. The maximum length of an INTCODE line up to a '//' is 72 characters.

'£' marks the entry point of a function or routine, and is followed by the internal name of the routine. This can be ignored unless the interpreter is to include run time tracing facilities.

'#' marks the end of a routine or function and is matched to '£'. It can be used if an assembler is attempting to record the stack high water mark reached in a routine, it should otherwise be ignored.

'.' occurs in the names of routines and functions and in the external names. It should be treated as an alphanumeric character for this purpose.

'Z' is used to indicate the end of the segment. Local labels can be unset.

An example assembler/interpreter which runs on the PDP-10 can be seen in BCL:ASSINT.MAS.

8.3.2 INSTRUCTIONS

The assembly form of an instruction consists of the mnemonic letter for the machine function, optionally followed by 'I' if indirection is specified, optionally followed by 'P' if the P modification is specified, followed by the address which is either a decimal integer or a label.

8.3.3 LABELS

There are 3 forms of labels, local, GLOBAL and EXTERNAL.

Local labels are referenced by 'L' followed by a decimal integer; they refer to a label local to this segment.

GLOBAL labels are referenced by 'G' followed by a decimal integer.

EXTERNAL labels are referenced by 'E' followed by a BCPL string in string quotes '"'.

A number not preceded by a letter is interpreted as a local label; it should be set to the address of the next location to be loaded.

GLOBAL labels are initialised during assembly when a directive is found of the form 'G' followed by a GLOBAL number (decimal integer) followed by 'L' followed by a local label number. This can be interpreted in one of two ways. Either a) The address of the location referred to by the local label should now be known globally by the GLOBAL number, referable to in other segments, or b) A global vector, at some point in the work area (P), is established and the GLOBAL number is interpreted as pointing into this area. To set the GLOBAL, the value of the location referred to by the local label is copied into the GLOBAL location.

EXTERNAL labels are intialised during assembly when a directive is found of the form 'E' followed by an EXTERNAL name (in quotes - '"') followed by 'L' followed by a local label number. This should be handled similarly to GLOBAL format a) above.

It is usually easiest, in the early stages of implementing an assembler/interpreter to permit intersegment communication via GLOBALs only, and to implement them via format b) above.

8.3.4 DATA

Data may be assembled by a statement consisting of 'D' followed by a signed decimal integer for constant values, or 'DL' followed by a local label number for pointers.

Characters may be packed and assembled using character statements of the form 'C' followed by the integer value of the character. On the PDP-10 the characters are ASCII codes and the first character is usually a character count. The characters should be packed up in the manner most suitable for the interpreting machine.

'Y' followed by an integer count indicates a static vector, whose subscripts are from 0 to the count. The designated number of words should either be skipped or set to zero.

Example program.

```
GLOBAL $( START:1 $)
EXTERNAL $( CREATEFILE; INITIALISEIO
            WRITE; OUTPUT $)
STATIC $( V = VEC 10 $)
STATIC $( OUTPUT = 0 $)

LET START() BE
$( LET IOV = VEC 650
   LET FACT(N) = N = 1 -> 1, N * FACT(N-1)
   INITIALISEIO(IOV, 650)
   OUTPUT := CREATEFILE("TTY")
   FOR I=1 TO 10 DO WRITE(OUTPUT,
      "FACT(:N) IS :N*C*L", I, FACT(I))
$)
```

Produces the following INTCODE.

```
 JL6
$START 4 LP3 SP2 JL9
$FACT 7 LIP2 L1 X10 FL10 L1 JL11 10 LIP2 L1 X9 SP5 LIL8 K/
3 LIP2 X5 11 X4
#
 1 Y10
 2 DL1 3 D0 5 DL4 8 DL7
 9 LIP2 SP656 L650 SP657 LIE"INITIALISEIO" K654 LL12 SP65/
6 LIE"CREATEFILE" K654 SL3 L1 SP654 JL13 14 LIL3 SP657 LL/
15 SP658 LIP654 SP659 LIP654 SP662 LIL8 K660 SP660 LIE"WR/
ITE" K655 LIP654 A1 SP654 13 L10 X15 TL14 X4
#
 12 C3 C84 C84 C89 15 C16 C70 C65 C67 C84 C40 C58 C78 C41/
 C32 C73 C83 C32 C58 C78 C13 C10
 6 X22

 E"OUTPUT"L3
 G1L5
Z
```

APPENDIX A

FULL SYNTAX OF BCPL

```
PROGRAM         ::= <segment>*
<segment>       ::= <rblock>
<rblock>        ::= <block> ¦ <compound>
<block>         ::= <D> <;<D>* <; <C>>*
<D>             ::= LET <declaration> <AND <declaration>>* ¦
                    EXTERNAL <ebody> ¦
                    MANIFEST <mgbody> ¦
                    GLOBAL   <mgbody> ¦
                    STATIC   <sbody>
<ebody>         ::= <E><ebody2> ¦<ebody2>
<ebody2>        ::= £( <edef> <; <edef>>* £)
<edef>          ::= <name> ¦ <name> = <E> ¦ <name> : <E>
<mgbody>        ::= £( <mgdef> <;<mgdef>>* £)
<mgdef>         ::= <name><K> ¦ <name> = <K> ¦ <name> : <K>
<sbody>         ::= £( <sdef> <; <sdef>>* £)
<sdef>          ::= <name> <L> ¦ <name> = <L> ¦ <name> : <L>
<declaration>   ::= <name>(<namelist>**) < = <E> ¦ BE <C>> ¦
                    <name> = <VE> ¦
                    <namelist> = <VE list>
<C>             ::= <C1> ¦ <C2>
<C1>            ::= IF <E> DO <C> ¦
                    WHILE <E> DO <C> ¦
                    UNLESS <E> DO <C> ¦
                    UNTIL <E> DO <C> ¦
                    TEST <E> THEN <C> OR <C> ¦
                    SWITCHON <E> INTO <C> ¦
                    FOR <name>=<E> TO <E> <BY <K>>** DO <C> ¦
                    <name> : <C> ¦
                    CASE <K> <... <K>>**: <C> ¦
                    DEFAULT <<K> ... <K>>**: <C>
<C2>            ::= <C3> ¦
                    <C2> <> <C1> ¦
                    <C2> <> <C3> ¦
                    <C2> REPEAT ¦
                    <C2> REPEATWHILE <E> ¦
                    <C2> REPEATWHILE <E> ¦
                    <C2> WHERE <declaration>
                    <AND <declaration>>*
<C3>            ::= GOTO <E> ¦
                    RESULTIS <E> ¦
                    BREAK ¦
                    LOOP ¦
```

```
                    RETURN ¦
                    FINISH ¦
                    ENDCASE ¦
                    £( <rblock> £) ¦
                    <E>() ¦
                    <E>(<Elist>) ¦
                    <Elist> ¦¦= <Elist> ¦
                    <Elist> <binop> ¦¦= <Elist> ¦
                    TRACE <name>() ¦
                    TRACE <name>(<Elist>)
<binop>         ¦¦= <op1> ¦ <op2> ¦ <op3> ¦ <op4> ¦
                    <op5> ¦ <op6> ¦ <op7> ¦ <op8> ¦ !
<namelist>      ¦¦= <name> <, <name>>*
<VElist>        ¦¦= <E> <, <E>>*
<Llist>         ¦¦= <L> <, <L>>*
<L>             ¦¦= <VE>
<K>             ¦¦= <E>
<VE>            ¦¦= VEC <K> ¦ <E>
<E>             ¦¦= <Sexpr> ¦ TABLE <Llist>
<Sexpr>         ¦¦= <Cexpr> ¦ SELECTOR <E> ¦ <E> ¦ <E> ¦
                    BYTE <E> ¦ <E>
<Cexpr>         ¦¦= <logical1> ¦
                    <logical1> -> <Sexpr>, <Sexpr>
<logical1>      ¦¦= <logical2> ¦ <logical1> <op1> <logical2>
<op1>           ¦¦= EQV ¦ NEQV
<logical2>      ¦¦= <logical3> ¦ <logical2> <op7> <logical3>
<logical3>      ¦¦= relational> ¦
                    <logical3> <op8> <relational1>
<relational>    ¦¦= <shift> <<op2> <shift>>*
<op2>           ¦¦= LS ¦ LE ¦ EQ ¦ GE ¦ GR ¦ NE ¦ #LS ¦ #LE ¦
                    #EQ ¦#GE ¦ #GR ¦ #NE
<op7>           ¦¦= \/ ¦ BITOR
<op8>           ¦¦= /\ ¦ BITAND
<shift>         ¦¦= <arith> ¦ <shift> <op3> <arith>
<op3>           ¦¦=  LSHIFT ¦ RSHIFT ¦ ROTL ¦ ROTR ¦
                    ALSHIFT ¦ ARSHIFT
<arith1>        ¦¦= <arith2> ¦ <arith> <op4> <arith2>
<op4>           ¦¦= + ¦ = ¦ #+ ¦ #-
<arith2>        ¦¦= <appl> ¦ <arith2> <op5> <appl>
<appl>          ¦¦= <appl2> ¦ <appl2> <op6> <appl1>
<op6>           ¦¦= ¦¦ ¦ &&
<appl2>         ¦¦= <appl3> ¦ <appl2> ! <appl3>
<appl3>         ¦¦= <unary> ¦ <appl3> %<name> <unary>
<unary>         ¦¦= <primary> ¦ <op7> <unary>
<op7>           ¦¦= NOT ¦ @ ¦ ! ¦ + ¦ - ¦ ABS ¦ #+ ¦ #- ¦
                    #ABS ¦ FIX ¦ FLOAT ¦
<primary>       ¦¦= <name> ¦ <constant> ¦ (<E>) ¦
                    <primary>() ¦ <primary>(<Elist>) ¦
                    NIL ¦ VALOF <C>
<constant>      ¦¦= <real> ¦ <integer> ¦ <octal> ¦
                    <truthvalue> ¦ <string> ¦ <charconst> ¦
                    £constant1 ¦ <literal>
<real>          ¦¦= <fraction> <exponent>**
<fraction>      ¦¦= <integer>.<integer>
<exponent>      ¦¦= E< + ¦ ->** <integer>
<integer>       ¦¦= <d><d>*
```

```
<octal>       ::= <od><od>*
<truthvalue>  ::= TRUE | FALSE
<charconst>   ::= '<c>*'
<string>      ::= "<c>*"
<literal>     ::= £constant2
<d>           ::= <od> | 8 | 9
<od>          ::= 0 | 1 | ... | 7
<name>        ::= <l><<l> | . | <d>>*
<l>           ::= A | B | ... | Z | a | b | ... | z
<c>           ::= any BCPL character - see Appendix C.
```

Notes.

The syntax is expressed in near BNF. The symbol '*' following a non terminal symbol (i.e. on in <> pairs), indicates indefinate repetition (i.e. from zero to infinity). Similarly, the symbol '**' following a non terminal symbol indicates optional occurrence (i.e. zero or one occurrence).

The syntax expressed at the start of the other sections of this manual is not necessarily the actual syntax recognised by the compiler. It is merely intended to give a reminder or flavour of the syntactic constructs being described in that section.

APPENDIX B

RESERVED WORDS AND SYMBLOLS

Reserved word	Equivalent symbol	Reserved word	Equivalent symbol
ALSHIFT		LS	<
AND		LSHIFT	<<
ARSHIFT		LV	@
BE	IS	MANIFEST	
BITAND		NE	\=
BITOR		NEQV	
BREAK		NIL	
BY		NOT	\ ~
BYTE		OF	:: ^
CASE		OR	
DEFAULT		REM	
DO	THEN	REPEAT	
ENDCASE		REPEATUNTIL	
EQ	=	REPEATWHILE	
EQV		RESULTIS	
EXTERNAL		RETURN	
FALSE		ROTL	
FINISH		ROTR	
FIX		RSHIFT	>>
FLOAT		RV	!
FOR		SELECTOR	
FROM	&&	STATIC	
GE	>=	SWITCHON	
GET		TABLE	
GLOBAL		TEST	
GOTO		THEN	DO
GR	>	TO	
IF		TRACE	
INTO		TRUE	
IS	BE	UNLESS	
LE	<=	UNTIL	
LET		VALOF	
LOGAND	/\ &	VEC	
LOGOR	\/ !	WHERE	
LOOP		WHILE	
#ABS		¦	\/ LOGOR
#EQ	#=	¬	\ NOT
#GE	#>=	~	¬ NOT
#GR	#>	&&	FROM

LE	#<=	^	:: OF
LS	#<	::	^ OF
NE	#\=	_	:=
=	#EQ	:=	_
>=	#GE	<<	LSHIFT
>	#GR	>>	RSHIFT
<=	#LE	<>	::
<	#LS	;;	<>
\=	#NE	£(	(
+		(	£(
-		£)	)
*		)	£)
/		(	[
:	EQ	[	(
:=	GE	)	]
>	GR	]	)
<=	LE	£[	
<	LS	£]	
\=	NE	'	
"		"	
.		:	
,		,	).any string(
'		;	
**		!	
/\	& LOGAND	@	LV
{	/\ LOGAND	%	
\	: LOGOR	...	
->			

// introduces a comment up to the end of line
/* introduces a comment up to a matching */
//? introduces a "user debugging" comment (see 5.7).

APPENDIX C

CHARACTER CODES.

Characters are in general represented by themselves in a character or string constant. The exceptions are *, line ending characters, and ' (for a character constant) and " (for a string constant). Layout characters (tab, space etc. except for newline and newpage) may be represented by themselves, but are usually represented by one of the escape forms listed below.

The following are the most common escape forms:-

**	stands	for	*
*'	"	"	'
*"	"	"	"
*C	"	"	Carriage return
*L	"	"	Line feed
*T	"	"	Tab
*S	"	"	Space
*P	"	"	Form feed (newpage)
*E	"	"	End of stream (file)

In order to help when a string spreads over one line, * followed by spaces, tabs or newlines up to a matching * is ignored.

Examples.

"A NEWLINE IS USUALLY REPRESENTED BY *C*L"

"HE* *LLO" is equivalent to
"HELLO".

Below is a table showing how to represent all the characters in the PDP-10 character set.
Top line is the ASCII (PDP-10) character.
Bottom line is the BCPL character (string) representation.

Strings are limited to a maximum 127 characters.

x=	0	1	2	3	4	5	6	7
0x	Null *0	^A *^A	^B *^B	^C *^C	^D *^D	^E *^E	^F *^F	^G *^G
1x	Bksp *B	Tab *I	Line *L	VT *^K	FF *P	CR *C	^N *^N	^O *^O
2x	^P *^P	^Q *^Q	^R *^R	^S *^S	^T *^T	^U *^U	^V *^V	^W *^W
3x	^X *^X	^Y *^Y	Eofile *E	Escape *£	^\ *^\	^] *^]	^^ *^^	^_ *^_
4x	Space *S	! !	" *"	# #	£ £	% %	& &	' *'
5x	((	))	* **	+ +	, ,	- -	. .	/ /
6x	0 0	1 1	2 2	3 3	4 4	5 5	6 6	7 7
7x	8 8	9 9	: :	; ;	< <	= =	> >	? ?
10x	@ @	A A	B B	C C	D D	E E	F F	G G
11x	H H	I I	J J	K K	L L	M M	N N	O O
12x	P P	Q Q	R R	S S	T T	U U	V V	W W
13x	X X	Y Y	Z Z	[[	\ \	]]	^ ^	_ _
14x	` `	a a	b b	c c	d d	e e	f f	g g
15x	h h	i i	j j	k k	l l	m m	n n	o o
16x	p p	q q	r r	s s	t t	u u	v v	w w
17x	x x	y y	z z	{ {	| |	} }	~ ~	Delete *177

APPENDIX D

MACHINE CODE BLOCKS AND OPERATORS

```
Syntax:-
    £ [ <macinstr> < ; <macinstr>>* £]
<macinstr> ::= <label> : <instfield> ! <instfield>
<instfield>::= <opcode> !
               <opcode> <addrpart> !
               <opcode> <accpart>, <addrpart> !
               £EXP <addrpart> !
               £XWD <lhvalue>, <addrpart> !
               £( <block> £)
<opcode>   ::= <K>
<accpart>  ::= <K>
<lhvalue>  ::= <K>
<addrpart> ::= @ <indpart> ! <indpart>
<indpart>  ::= ! <litpart> ! <litpart>
<litpart>  ::= <addr> ! <addr> (<modpart>)
<addr>     ::= <K> ! <name> ! <string> ! <literal>
<modpart>  ::= <K>
```

Opcode is expected to be a constant either in the range Ø to #777, in which case it is put into bits 27 to 35, otherwise it is expected to be a full word value. To this end the standard opcodes in the range #40 to #677 are provided in the form £opcode, evaluating to a 9 bit value. Extended opcodes for all of the CALLI and ITCALL UUOs, JRST and JFCL evaluate to a full word value.

Examples.

```
£MOVE      evaluates to  #2ØØ
£JRST      evaluates to  #254
£JRSTF     evaluates to  #254ØØØØØØØØ
£SKPINC    evaluates to  #Ø515ØØØØØØØØ
£MSTIME    evaluates to  #ØØ1ØØØØØØØ23
```

@ <indpart> sets the indirect bit.

! <litpart> causes <litpart> to be put in the literal table and the pointer substituted for <addr>. <litpart> should be a compile time constant in this case.

The instruction word is constructed as follows.

The <addr> part is put into the (BYTE 18:ØØ) field.
(BYTE 18:18) is zeroed.
A 9 bit <opcode> is put into the (BYTE 9:27) field, a full word <opcode> is added into the instruction word.
The <accpart> is ORed into the (BYTE 4:23) field.
The indirect bit is ORed into the (BYTE 1:22) field.
The <modpart> has its left and right halves swapped and is ORed into the instruction word.
Finally - if <addr> was <name> the correct address, and modifier are put into the instruction word.
The £EXP and £XWD forms are used for setting data words.

Labels in a machine code segment are manifest labels (ie they do not create a data item like normal BCPL labels).
A GET file (BCL:ACS.GET) has all the register assignment mnemonics for use in machine code blocks. See the BCPL USER GUIDE for a description of BCL:ACS.GET.

Commas between fields in machine code statements are treated as spaces, hence:-

```
    £POPJ   P,
and £POPJ   P
```

are the same, and probably not the instruction intended by the programmer, who should write:-

```
    £POPJ   P,Ø
```

Example.

```
GET "BCL:ACS"               // GET REGISTER ASSIGNMENTS
LET ALPHA(L) = VALOF        // SETS UP ENTRY AND POSTMORTEM
£[ £SETZ AC, 0              // FALSE INTO RESULT REG
   £MOVE  B, L              // LOAD PARAMETER
   £CAIG  B, 'Z'            // SKIP IF GREATER THAN Z
   £CAIGE B, 'A'            // SKIP IF GR/EQ TO A
   £( RETURN £)             // NOT ALPHA, RETURN FALSE
   £SETO  AC, 0             // TRUE INTO RESULT REG
£]                          // RETURN TRUE
```

APPENDIX E

COMPILER SWITCHES

Can be listed with /H compiler switch.

Valid switches are:-

/A Allocate undefined names as 'EXTERNAL'.
/B Batch mode compilation forced if job run interactively, messages are listed on listing file if listing on or TTY if listing off.
/C Cross reference listing.
/D Debugger symbol tables required.
/E Error abort. Abort on single error.
/E:n Error abort if more than n errors. If n = 0 (Default case) then aborts on 30 errors for interactive run, 200 for batch run.
/F Fast code, contains no postmortem code.
/G Generation of code suppressed.
/H Help text printed on TTY.
/H:n Helpfull BCPL news printed on TTY.
/I Interactive mode compilation forced if job run as batch, messages are listed on TTY.
/J Just syntax check.
/K Keywords in lower and upper case.
/K:n Keywords in lower case if n > 0.
/L List all source program regardless of any £NOLISTs found.
/M Machine code listed on message medium.
/N No real GETfiles, ignore all GET commands.
/O Optimise trace code, no postmortem or stackcheck code.
/P Pure/Impure code selector (Pure default).
/P:n Alternative code generator selector. 2=INTCODE, 3=Parse tree.
/Q Include GETfile declarations in cross reference listing.
/R Reset all switches to zero or false (default condition).
/S Size of workspace increased by 1K.
/S:n Size of workspace set to nK, default is 3K.
/T Tree from parser listed on message medium.
/U User debugging code enabled.
/V Non reinitialised data.
/W Warning messages suppressed.

/X Processor type. Ø=default. 1=KA1Ø. 2=KI1Ø.
/Y Yank out NUMBARGS/LHS code.
/Z Generate library type rel file.

n is read as octal unless preceded by
+ or - in which case it is read as decimal.

Switches start at zero or false, and values are retained until reset or compiler is reloaded.

Repeated occurrences of a switch without a value increments the previous value by 1. For this purpose non zero switches are interpreted as true.

APPENDIX F

CODE CONVENTIONS

Register Allocation:

```
Ø        - JRST to a special routine to catch jump to zero.
1        - result register, results from VALOF and function
           calls.
2 to 12  - working registers.
13       - stack shadow register.
14       - link register.
15       - the constant 1.
16       - stack register.
17       - system pushdown list
```

Registers 1 to 12 can be assumed free for programmer use in any subroutine or machine code block.
All other registers may be used in a subroutine, but Ø,15,16 and 17 should be reset before returning control to BCPL code. Register 13 is only used if stackchecking is enabled.

The subroutine entry code is

```
     ADDI   16,@0(14)
     MOVEM  14,0(16)
```

This is followed by (if postmortem enabled):

```
     MOVEI  13,n(16)  where n is extent of stack used.
     JSP    14,.TE
     XWD    Pointer to profile count,,
                Pointer to BCPL name of routine.
```

or the following if only stack check enabled

```
     MOVEI  13,n(16)
     CAMLE  13,.SL
     JSP    14,.SO
```

The subroutine exit code (if postmortem enabled):

```
     JSP    14,.TX
```

otherwise it is

```
     JRSTF  @0(16)
```

```
Subroutine call when NUMBARGS/LHS code is enabled is
        JSP   14,subroutine
        SUBI  16,@word
        MOVEI 13,n(16) if stackcheck or postmortem enabled.
        [where word is of the form
        (BYTE 1:35) set if LHS true
        (BYTE 12:23) count of NUMBARGS
        (BYTE 5:18) zero
        (BYTE 18:0) stack increment]

otherwise it is
        JSP   14, subroutine
        SUBI  16, stack increment
        MOVEI 13,n(16) if stackcheck or postmortem enabled.

A source error causes the following code to be planted
        JSP   14,.ER
        XWD   Lineno, Pointer to BCPL file name.
        XWD   Errno, Pointer to BCPL error message.

A label trap, planted if postmortem enabled is
        JSP   14,.TL
        XWD   Pointer to profile count,,
                  Pointer to BCPL name of label.

A TRACE point trap, planted if postmortem enabled is
        Move parameters to stack
        JSP   14,.TI
        SUBI  (see subroutine call)
        MOVEI 13,n(16)
        CAIA  Pointer to BCPL name of trace point.
        CAI   Pointer to profile count.

The finish code is
        JSP   14,.FN

If a debugging symbol file is being produced the
segment is headed by:
        JSP   14,.SY
        XWD   Pointer to name of symbol file,,
                  Continuation address.
```

APPENDIX G

COMPILATION ERROR MESSAGES.

Syntax error messages:-

1. UNMATCHED '£)' TAG
   There is a tagged closing section bracket whose tag name does not match any tag name on the unclosed opening section brackets in your segment. The top level is closed.

2. UNMATCHED '£(' TAG
   The compiler has encountered a closing section bracket which has no tag name, but appears to close an opening section which has a tag name. The top level is closed.

3. 'BYTE' OR 'SELECTOR' OUT OF PLACE
   The BYTE and SELECTOR construction operators must be either an expression on their own, or within brackets, because they have such low priority.

4. COMMAND EXPECTED
   The compiler has either a) found a declaration while scanning commands, in which case it will process the declaration, giving the names within it the scope of the remainder of the current block or b) found something incomprehensible while scanning commands.

5. UNBRACKETED 'TABLE' IN EXPRESSION LIST

6. ')' EXPECTED AFTER EXPRESSION LIST
   In a function or routine call.

7. EXPRESSION MALFORMED
   The compiler, when reading what it believes to be an expression (or sub expression), cannot understand the start of the expression (or sub expression).

8. ':' MISSING
   One of the fields is missing in a SELECTOR or BYTE construction, it is assumed to be zero.

9. ',' MISSING IN CONDITIONAL
   After the true branch.

10. UNMATCHED '(' IN EXPRESSION
The closing bracket on a sub expression appears to be missing.

11. 'LET' EXPECTED
An 'AND' has been found without preceding 'LET', it is treated as 'LET'.

12. '£(' MISSING
Before EXTERNAL, STATIC, MANIFEST or GLOBAL list.

13. ')' MISSING AFTER NAMELIST
The closing bracket is missing in the () or (<namelist>) in a function or routine declaration.

14. '=' OR 'BE' MISSING
After function or routine header.

15. '=' MISSING
In what appears to be a simple LET declaration, the = has been left out or := used instead.

16. DECLARATION EXPECTED
LET or WHERE have been found without a normal LET type declaration following.

17. NAME EXPECTED
The compiler expects to find a name as part of a declaration list. Caused usually by missing the closing section bracket on an EXTERNAL, STATIC, MANIFEST or GLOBAL, or using a reserved word where a name is intended.

18. 'OR' MISSING AFTER 'TEST'

19. '=' MISSING IN 'FOR'
After the FOR loop control variable.

20. 'TO' MISSING IN 'FOR'

21. 'INTO' MISSING AFTER 'SWITCHON'

22. ':' MISSING AFTER 'CASE' OR 'DEFAULT'

23. RANGE BADLY FORMED AFTER 'DEFAULT'
The range after DEFAULT should have both lower and upper values.

24. NAME EXPECTED AS LABEL
The compiler has found what it believes to be a label, indicated by ':', but the expression is not simply a name.

25. ASSIGN OPERATOR MISSING
The compiler thinks it has seen an expression list on the left hand side of an assignment. This error message sometimes is caused by misspelling a keyword.

26. END OF PROGAM FOUND
Before the end of the file; this is usally caused by a mismatch of £(and £) pairs.

27. STRING MISSING AFTER 'GET'
There should be a BCPL string after GET, indicating which file the compiler is to GET.

28. '#' OUT OF CONTEXT
is used to indicate an octal number, or to indicate floating point interpretation of operators. It must be adjacent to the number or operator it qualifies.

29. STRING TOO LONG OR " OR ' MISSING
A string exceeds the maximum permitted length. This is usually caused by using " or ' or * in a string instead of *" or *' or **.

30. OCTAL DIGIT EXPECTED
In octal constant.

31. CHARACTER FOUND OUT OF PLACE
A character which is meaningless to BCPL has been found in the input file. It is ignored.

32. '*N' INTERPRETED AS '*C*L'
Some BCPL compilers on machines with a single new line character, accept *N for newline. The compiler converts this to *C*L.

33. FRACTION MISSING
A fractional part must always appear in a real number constant.

34. EXPONENT MISSING
If E is given it must be followed by a valid exponent in a real number constant.

35. END OF FILE FOUND
Before logical end of program; this is usually caused by a mismatch of £(and £) pairs.

36. UNABLE TO GET GETFILE
There is either a) some error in the file specification for the getfile requested or b) the file is inaccessible.

37. END OF LINE FOUND WHILE READING STRING
A string is not permitted to span more than one line unless specifically indicated. Often caused by using " or ' or * in a string instead of *" or *' or **.

38. TRACE ROUTINE CALL EXPECTED
The syntax of the TRACE command has been infringed.

39. MONADIC OPERATORS ARE MORE BINDING THAN VECTOR APPLY
This warning is intended to advise that an expression of

the form @A!B is interpreted as (@A)!B and not @(A!B).

Semantic error messages:-

40. 'BREAK' NOT INSIDE A LOOP
BREAK can only be used inside a loop command such as FOR, UNTIL, WHILE and the REPEATs.

41. 'LOOP' NOT INSIDE A LOOP
Same as 40.

42. 'ENDCASE' NOT INSIDE A 'SWITCHON'

43. 'RESULTIS' NOT INSIDE A 'VALOF'

44. 'CASE' NOT INSIDE A 'SWITCHON'

45. 'DEFAULT' NOT INSIDE A 'SWITCHON'

50. NAME NOT DECLARED
Name is treated as EXTERNAL. Repetitions of this error for the same name can be suppressed by use of the /A switch.

51. MULTIPLY DECLARED NAME
A name may be declared by using it as a routine name, or a label or as a FOR loop variable; all names with identically the same scope must be unique.

52. DYNAMIC FREE VARIABLE
Local variables declared by simple or vector LET declarations, may only be referenced in the routine in which they are declared, not even in any routines embedded in the declaring one.

53. NON 'MANIFEST' NAME IN CONSTANT
Any names appearing in a constant expression must have been previously declared as MANIFEST. This message may occur if the name is undeclared altogether.

54. LOCAL OR 'MANIFEST' NAME USED WITH @ IN LOAD TIME CONSTANT
The address of a local or MANIFEST can not be used in the constant expression for a SIATIC or TABLE element.

55. ADDRESS OF 'MANIFEST' CONSTANT USED
Since a MANIFEST is merely a way of attaching a mnemonic name to a constant, the address of such a constant is meaningless.

56. VALUE ASSIGNED TO 'MANIFEST' CONSTANT
See 55. Such an assignment is meaningless.

57. 'GLOBAL' OUT OF RANGE
must be in range -9999 to +9999.

60. 'CASE' VALUES OVERLAP

Two CASE values are the same or overlap in a single SWITCHON command body.

61. MORE THAN ONE 'DEFAULT'
In a single SWITCHON command body.

70. UNMATCHED LHS/RHS
In a LET declaration or an assignment there are too many or too few values on the right of the declaration or assignment for the number of locations named on the left.

71. ADDRESS EXPECTED
The expression (either operand of @ or on the left of an assignment) is expected to indicate an address.

72. LOAD TIME CONSTANT EXPECTED
The expression which is the value of a STATIC or TABLE element must be a load time constant.

73. NAME EXPECTED IN @ EXPRESSION
The name of a static (inplicit or explicit) is the only expression permissible as the operand of @ in a load time constant.

74. CONSTANT EXPECTED
The expression must be a compile time constant.

75. STRING OR NAME EXPECTED AS EXTERNAL NAME
The equivalent name of an EXTERNAL (that on the right of = or :) must be a BCPL name or a BCPL string.

76. STRING OR NAME EXPECTED AS 'EXTERNAL' PREFIX
The prefix to an EXTERNAL list should be a BCPL name or a BCPL string.

INTCODE version error messages:-

85. STATIC SELECTORS/BYTES ARE NOT IMPLEMENTED IN THIS VERSION

86. DYNAMIC SELECTORS/BYTES ARE NOT IMPLEMENTED IN THIS VERSION

87. EXPONENTIATION IS NOT IMPLEMENTED IN THIS VERSION

88. TRACE IS NOT IMPLEMENTED IN THIS VERSION

89. FLOATING POINT OPERATORS ARE NOT IMPLEMENTED IN THIS VERSION

General error messages:-

96. LINE TOO LONG
Source lines are limited to 150 characters. The rest of the line is ignored.

97.

These errors are machine dependent and the text describes the error.

8. COMPILER ERROR - PLEASE REPORT TO SOFTWARE STAFF
Retain all the documentation/cards/listings you have of your program. Make a note of the time and date of the compilation run and take all that information to your software representative.

9. COMPILER WORKSPACE FULL, RECOMPILE WITH MORE SPACE
The compiler utilises a limited workspace. If you should get this message either a) split the segment into two or more pieces (better solution) or b) increase the compilers workspace via the compiler switches. While developing a program, indication that this error is impending will be apparent if the % figure printed by the compiler after each compilation approaches 100%.

APPENDIX H

STREAM CONTROL BLOCKS.

The format of the SCB is described in terms of SELECTORS in the file BCL:SCB.GET, and users should consult this for the current allocation of the fields in the SCB.

Words Ø to 1Ø (Octal) are the same for all SCB's. They are

```
SC.FLAGS     Word Ø : flag word
SC.READER    Word 1 : read routine for this stream
SC.WRITER    Word 2 : write routine for this stream
SC.CLOSER    Word 3 : close routine for this stream
SC.ERROR     Word 4 : error "LABEL" closure for this stream
SC.OLDREAD   Word 5 : putback word
SC.PUTSTACK  Word 5 :    "       "
SC.USER6     Word 6 : reserved for user
SC.USER7     Word 7 :     "      "    "
SC.USER1Ø    Word 1Ø:     "      "    "
```

The flag word (word Ø) has the following bit allocations.

```
SC.MONITORSTREAM (BYTE 1:35) set for Monitor SCB
SC.SIZE          (BYTE 6:12) size of SCB.
SC.SCB           (BYTE 1:Ø)  always set in SCB.
```

APPENDIX I

EXAMPLE BCPL PROGRAM

```
GET "BCL:BCPLIB"

/* THIS PROGRAM CREATES A LINEPRINTER FILE WHICH SHOWS
   WHICH CHARACTERS IN THE SOURCE FILE (DSK:DEMO.BCL)
   ARE IN UPPER CASE. IT DOES THIS BY OVERPRINTING ALL
   UPPER CASE CHARACTERS.
*/

MANIFEST £( IOSIZE=650; LINESIZE=150 £)

STATIC £( V = VEC LINESIZE £)

LET START() BE
£(MAIN
    LET IOV = VEC IOSIZE
    AND C, OVERPRINT, VC = NIL, NIL, NIL
    INITIALISEIO(IOV, IOSIZE)
    INPUT := FINDFILE("DSK","DEMO","BCL")
    OUTPUT := CREATEFILE("LPT")

    C := INCH()

    £(A
        OVERPRINT, VC := FALSE, 0
        UNTIL LINECH(C) \/ C = '*E' DO
        £(B // READS AN INPUT LINE INTO V
            VC +:= 1
            V!VC := C
            C := INCH()
        £)B
        PRINTLINE(VC)
        FOR I = 1 TO VC DO // REMOVE NON INTERESTING CHARS
            SWITCHON V!I INTO
            £(C
                CASE 'A' ... 'Z': OVERPRINT := TRUE
                CASE '*T': ENDCASE
                DEFAULT 0 ... #177: V!I := '*S'
            £)C
        IF OVERPRINT DO
            FOR I=1 TO 2 DO OUTCH('*C')<>PRINTLINE(VC)
        IF C = '*E' BREAK
        £( OUTCH(C); C:=INCH() £) REPEATWHILE LINECH(C)
```

```
    £)A REPEAT
    CLOSE(INPUT)
    CLOSE(OUTPUT)
£)MAIN

AND PRINTLINE(CT) BE
    FOR I = 1 TO CT DO OUTCH(V!I)

AND LINECH(CH) = '*L' LE CH LE '*C'
```